

NEAR EAST UNIVERSITY



**GRADUATE SCHOOL OF APPLIED
AND SOCIAL SCIENCES**

**OBJECT DATA MODELING AS STRUCTURING
APPROACH IN DATABASE DESIGN**

Anwar Mahmoud Dawoud

MASTER THESIS

Department of Computer Engineering

Nicosia - 2006

Anwar M M Dawoud :

Object Data Modeling As Structuring Approach In Database Design

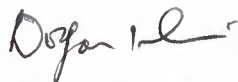
**Approval of the Graduate School of Applied and
Social Sciences**

Prof. Dr. Fakhraddin Mamedov
Director



**We certify this thesis is satisfactory for the award of the
Degree of Master of Science in Computer Engineering**

Examining Committee in charge:



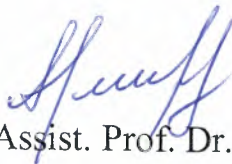
Prof. Dr. Doğan Ibrahim, Chairman, Chairman of
Computer Engineering Department, NEU



Assist. Prof. Dr. Firudin Muradov, Member, Computer Engineering
Department, NEU



Assist. Prof. Dr. Murat Tezer, Member, Computer & Education
Department, NEU



Assist. Prof. Dr. Adil Amirjanov, Supervisor, Computer Engineering
Department, NEU

ACKNOWLEDGEMENTS

I could not have prepared this thesis without the generous help of my supervisor, friends, and family.

I would like to express my gratitude to Assist. Prof. Dr. Adil Amirjanov for providing invigorating environment in which I could write this thesis.

My deepest thanks are to Assoc. Prof. Dr. Rahib Abiyev for his help and answering any question I asked him.

Finally, I could never have prepared this thesis without the encouragement and support of my father, mother, and brother Khaled Dawoud (Mechanical Engineer-MSc.).

ABSTRACT

Object-Oriented Data Modeling is a modern approach for structuring any problem by using object-oriented programming languages. For solving the problem that involves the database design this approach as well can be useful for reduction of abstractions during the mapping of real life problem. It can be said that Object-Oriented Database System is a hybrid of database and object-oriented technique. This thesis describes the steps of transforming and implementation of object-oriented techniques to the database software that is now implemented as Object-Relational Database System with extensions in SQL (Structured Query Language). In this thesis the Object Data Modeling is used as structuring approach for database design which shows that this approach improve readability and maintainability of software. Also, it reduces the design time by reduction of the steps of abstractions and the time for executing an application.

CONTENTS

ACKNOWLEDGEMENTS	i
ABSTRACT	ii
CONTENTS	iii
INTRODUCTION	1
CHAPTER ONE OBJECT ORIENTED DATA MODELING	4
1.1 Overview	4
1.2 What is Object-Oriented Programming (OOP)?	4
1.3 What's an Object?	4
1.4 What is A Database system?	5
1.5 Object-Oriented Databases	6
1.6 Basic Object-Oriented Modeling	7
1.6.1 Complex Objects	7
1.6.2 Object Identity	8
1.6.3 Classes and Types	9
1.6.4 Attributes	11
1.6.5 Behaviors	11
1.6.6 Encapsulation	11
1.6.7 Overriding Behaviors and Late Binding	13
1.6.8 Inheritance	13
1.6.9 Naming	15
1.7 Summary	16
CHAPTER TWO OBJECT ORIENTED DATABASE CONCEPTS	17
2.1 Overview	17
2.2 OODB SYSTEMS Perspectives	17

2.3 Architecture	21
2.3.1 Client-server	21
2.3.2 Storing and executing methods	22
2.4 Integrity	24
2.4.1 Relationships Integrity	24
2.4.2 Nulls Integrity	24
2.4.3 Referential Integrity	25
2.4.4 Entity integrity	26
2.5 Concurrency Control	26
2.6 Recovery	27
2.7 Transactions	28
2.8 Persistence	29
2.9 Security	29
2.10 Summary	30
 CHAPTER THREE OBJECT-RELATIONAL DB SYSTEMS	 31
3.1 Overview	31
3.2 Introduction to Object-Relational Database System	31
3.3 SQL3	33
3.3.1 Object identity	34
3.3.2 Row types	34
3.3.3 User-Defined Types (UDTs)	34
3.3.4 User-Defined Routines	35
3.3.5 Relations and inheritance	36
3.3.6 Polymorphism	36
3.3.7 Subtypes and supertypes	37
3.3.8 Persistent stored modules	37
3.3.9 Large Objects	37

3.4 Comparison of ORDBMS and OODBMS	39
3.5 Summary	40
CHAPTER FOUR OBJECT RELATIONAL ALGEBRA	41
4.1 Overview	41
4.2 Introduction to the Original Algebra	41
4.3 Semantics	43
4.3.1 Union	43
4.3.2 Intersect	44
4.3.3 Difference	44
4.3.4 Product	45
4.3.5 Restrict	47
4.3.6 Project	48
4.3.7 Join	50
4.3.8 Divide	53
4.4 Associativity and Commutativity	55
4.4 Summary	55
CHAPTER FIVE OBJECT RELATIONAL SQL	56
5.1 Overview	56
5.2 The Object Definition Language (ODL)	56
5.2.1 Tables	57
5.2.2 Views	60
5.2.3 Types	61
5.2.4 Procedures	61
5.2.5 Functions	63
5.2.6 Triggers	64
5.3 The Object Query Language (OQL)	65

5.3.1 Select statement	65
5.3.2 Insert statement	67
5.3.3 Update statement	67
5.3.4 Delete statement	68
5.4 Summary	68
CHAPTER SIX DATABASE DESIGN WITH OBJECT DATA MODELING	69
6.1 Overview	69
6.2 Object Relational DB Application	69
6.3 The Object Definition Language	76
6.3.1 Create objects in the database	76
6.3.2 Create tables in the database	77
6.3.3 Create views in the database	81
6.3.4 Create procedures in the database	82
6.3.5 Create functions in the database	84
6.3.6 Create triggers in the database	85
6.4 Object Query Language	85
6.4.1 Insert a row into the table	85
6.4.2 Update a row into the table	86
6.4.3 Delete a row into the table	86
6.4.4 Select rows from the table	87
6.5 Summary	90
CONCLUSION	91
REFERENCES	92
APPENDIX A	I-1
APPENDIX B	II-1

INTRODUCTION

The database management system (DBMS) is now the underlying frame-work of the information system and has fundamentally changed the way many organizations operate. The database system remains a very active research area and many significant problems have still to be satisfactorily resolved. The database approach emerged to resolve the problems with the file-based approach. A database is a shared collection of logically related data, designed to meet the information needs of an organization.

As applications become large even the usual approach for solving problem needs to be modified for simplification of the database design. One of the approach for simplification of the database design is to use object-oriented modeling of data that was implemented before the object-oriented programming languages.

Object-oriented database technology is a marriage of object-oriented modeling and database technologies. Perhaps the most significant characteristic of object-oriented database technology is that it combines object-oriented modeling with database technology to provide an integrated application development system.

Until recently, the choice of DB System seemed to be between the relational DB System and the object-oriented DB System. However, it would be useful to share the same basic relational structure of Relational DB System (RDBS), and incorporates some concept of 'object' in data modeling.

The concept of the Object Relational DB System (ORDBS) as a hybrid of the RDBS and Object-Oriented DB System (OODBS) is very appealing, preserving the wealth of knowledge and experience that has been acquired with the RDBS.

The ORDBMS (Object Relational DBMS) provide Object Definition Language(ODL), which allows users to define the database, and a Object Query Language(OQL), which allows users to insert, update, delete, and retrieve data from the database.

The ORDBMS has promising potential advantages such as increased productivity, improved data integrity and improved security.

The first chapter shows some basic concepts related with this thesis such as Object-oriented programming, objects, database system, and object-oriented database. Also it explains the basic object-oriented modeling such as complex objects, object identity, classes, attributes, behaviors, encapsulation, inheritance, overriding behaviors and late binding, and naming.

The second chapter explains OODB SYSTEMS perspectives; also it explains the two architectures of OODBMS, client-server architecture, and the storage of methods. Also it explains some concepts such as Integrity, Concurrency Control, Recovery, Transactions, Persistence, and Security.

The third chapter discusses some concepts related to Object-Relational Database System such as Object identity, Row types, User-Defined Types (UDTs), User-Defined Routines, Polymorphism, subtypes and supertypes, persistent stored modules, and Large Objects, also the end of this chapter we will make Comparison between ORDBMS and OODBMS.

The fourth chapter shows some information about Relational Algebra operators such as Restrict, Project, Product, Union, Intersect, Difference, Join, Divide.

The fifth chapter shows the main syntax of the Object Structured Query Language that includes Object Definition Language to create objects, tables, views, procedures, functions, and triggers, and also Object Query Language to implement the statements: select, update, insert, and delete.

The last chapter applies the syntax shown in chapter (5). At the same time comparison is made to show the main differences between using object modeling approaches, and without using it.

CHAPTER ONE

OBJECT ORIENTED DATA MODELING

1.1 Overview

This chapter shows some basic concepts such as Object-oriented programming, objects, database system, and object-oriented database. Also there is some explanations of the basic object-oriented modeling such as complex objects, object identity, classes, attributes, behaviors, encapsulation, inheritance, overriding behaviors, late binding, and naming.

1.2 What is Object-Oriented Programming (OOP)?

Object-oriented programming (OOP) is a method of creating a software application from a group of software components called objects. The application itself is made up of messages that are passed between these objects. OOP contrasts with procedural programming. In procedural programming data and the functions that manipulate the data are separate from one another, because much of the data is available to multiple functions, this can make debugging difficult. Particularly as applications become large, the procedural model begins to break down. By contrast, because data and functions (called methods in OO-speak) are contained within objects, the data is more protected and the application can grow with fewer problems. Objects also model real-world objects more accurately, so conceptually even complex problems can be simpler to address through software.

1.3 What's an Object?

Objects are at the heart of OOP, Objects are self-contained software components that are used to build object oriented applications, also objects contain data (usually referred to as attributes) and ways of working with the data (methods).

Another term associated with this is class, which could be thought as a generic version of the object.

1.4 What is A Database System?

A Database system is basically just a computerized record keeping system. The database itself can be regarded as a kind of electronic filing cabinet, i.e., it is a repository or container for collection of computerized data files [5]. Users of the system can perform a variety of operations on such files for example:

- Add new, empty files to the database.
- Inserting data into existing files.
- Retrieving data from existing files.
- Changing data in existing files.
- Deleting data from existing files.
- Removing existing files from the database.

So the definition for a database system is basically a computerized record keeping system. i.e., it is a computerized system whose overall purpose is to store information and to allow users to retrieve and update that information on demand. The information in question can be anything that is of significance to the individual or organization concerned. In other words, that is needed to assist in the general process of running the business of that individual or organization.

Figure 1.1 is a simplified picture of a database system. It is meant to show that a database system involves three major components: data, hardware, and users. There are three components briefly considered as below [5].

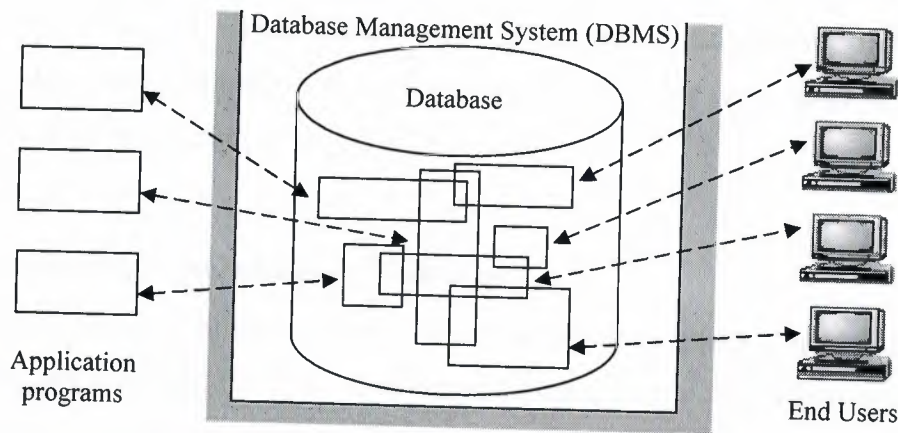


Figure 1.1 Simplified picture of a database system

1.5 Object-Oriented Databases

Object-oriented database technology is a marriage of object-oriented modeling and database technologies. Figure 1.2 illustrates how these programming and database concepts have come together to provide what we now call object-oriented databases

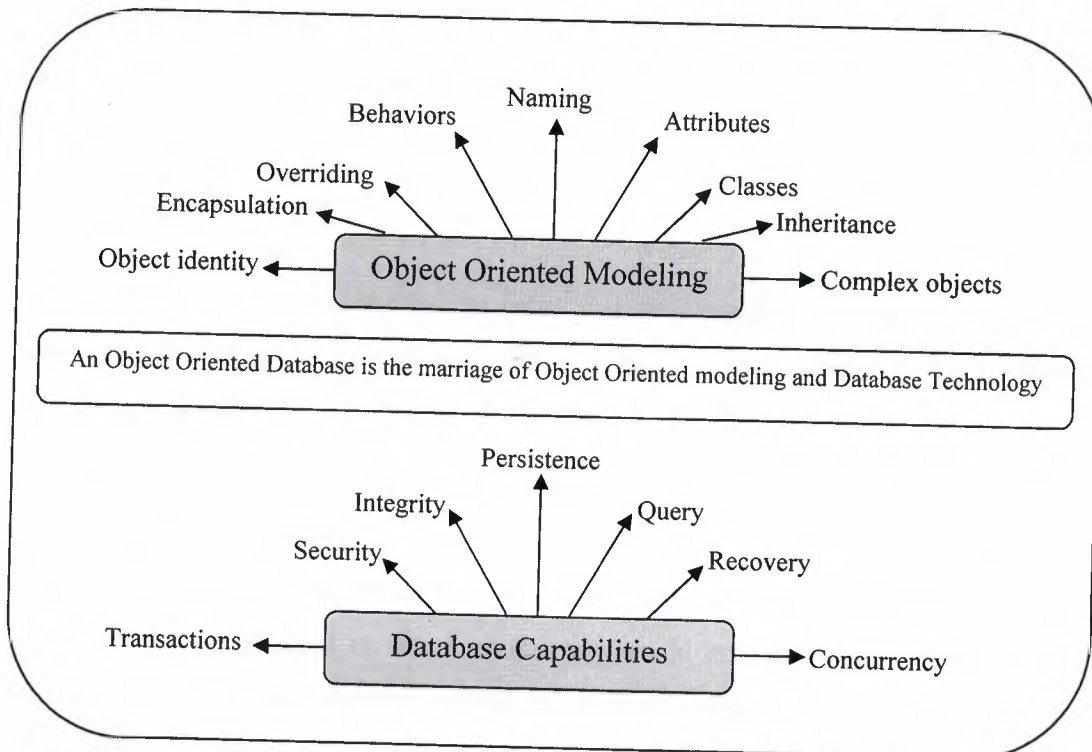


Figure 1.2 Makeup of an Object-Oriented Database

Perhaps the most significant characteristic of object-oriented database technology is that it combines object-oriented modeling with database technology to provide an integrated application development system.

1.6 Basic Object-Oriented Modeling

An object-oriented database system must satisfy two criteria: it should be a DBMS, and it should be an object-oriented system, i.e., to the extent possible, it should be consistent with the current crop of object-oriented programming languages. The first one translates into ten features:

- 1- Complex Objects.
- 2- Object Identity.
- 3- Classes and Types.
- 4- Attributes.
- 5- Behaviors.
- 6- Encapsulation.
- 7- Overriding Behaviors and Late Binding.
- 8- Inheritance.
- 9- Naming

The second criterion translates into some features: integrity, concurrency control, recovery, transactions, persistence, and security. In this chapter the first criteria will be discussed.

1.6.1 Complex Objects

Object oriented systems and applications are unique in that the information being maintained is organized in terms of the real-world entities being modeled. This differs from relational database applications that require a translation from the real-world information structure to the table formats used to store data in a

relational database. Normalizations upon the relational database tables result in further perturbation of the data from the user's perceptual viewpoint. Object oriented systems provide the concept of complex objects to enable modeling of real-world entities. A complex object contains an arbitrary number of fields, each storing atomic data values or references to other objects (of arbitrary types). A complex object exactly models the user perception of some real-world entity. Complex objects are built from simpler ones by applying constructors to them. The simple objects are objects such as integers, characters, byte strings of any length, booleans and floats (one might add other atomic types). There are various complex object constructors such as tuples, sets, bags, lists, and arrays. The minimal set of constructors that the system should have is set, list and tuple. Sets are critical because they are a natural way of representing collections from the real world. Tuples are critical because they are a natural way of representing properties of an entity. Of course, both sets and tuples are important because they gained wide acceptance as object constructors through the relational model. Lists or arrays are important because they capture order, which occurs in the real world, and they also arise in many scientific applications, where people need matrices or time series data [1, 6].

1.6.2 Object Identity

Object oriented databases (and programming languages) provide the concept of an object identifier (OID) as a means of uniquely identifying a particular object. OIDs are system generated. A database application does not have direct access to the OID. The OID of an object never changes, even across application executions. The OID is not based on the value stored within the object. This differs from relational databases, which use the concept of primary keys to identify a particular table row (i.e., tuple). Primary keys are based upon data stored in the identified row. The concept of OIDs makes it easier to control the storage of objects (e.g., not based on value) and to build links between objects (e.g., they are based on the

never changing OID). Complex objects often include references to other objects, directly or indirectly stored as OIDs.

The size of an OID can substantially affect the overall database size due to the large number of inter-object references typically found within an OO application. When an object is deleted, its OID may or may not be reused. Reuse of OIDs reduces the chance of running out of unique OIDs but introduces the potential for invalid object access due to dangling references. A dangling reference occurs if an object is deleted, and some other object retains the deleted object's OID, typically as an inter-object reference. This second object may later use the OID of the deleted object with unpredictable results. The OID may be marked as invalid or may have been re-assigned. Typically, an OODBMS will provide mechanisms to ensure dangling references between objects are avoided [1,6].

1.6.3 Classes and Types

OO modeling is based on the concept of a class. A class defines the data values stored by, and the functionality associated with, an object of that class. One of the primary advantages of OO data modeling is this tight integration of data and behavior through the class mechanism. Each object belongs to one, and only one, class. An object is often referred to as an instance of a class. A class specification provides the external view of the instances of that class.

A class has an extent (sometimes called an extension), which is the set of all instances of the class. Implementation of the extent may be transparent to an application, but minimally provides the ability to visit every instance of the class. Within an OODBMS, the class construct is normally used to define the database schema. Some OODBMS use the term type instead of class. A type, in an object-oriented system, summarizes the common features of a set of objects with the same characteristics. It corresponds to the notion of an abstract data type. It has two parts: the interface and the implementation (or implementations). Only the interface part is visible to the users of the type, the implementation of the object is

seen only by the type designer. The interface consists of a list of operations together with their signatures (i.e., the type of the input parameters and the type of the result). The type implementation consists of a data part and an operation part. In the data part, one describes the internal structure of the object's data. Depending on the power of the system, the structure of this data part can be more or less complex. The operation part consists of procedures which implement the operations of the interface part.

In programming languages, types are tools to increase programmer productivity, by insuring program correctness. By forcing the user to declare the types of the variables and expressions he/she manipulates, the system reasons about the correctness of programs based on this typing information. If the type system is designed carefully, the system can do the type checking at compile-time, otherwise some of it might have to be deferred at compile time. Thus types are mainly used at compile time to check the correctness of the programs. In general, in type-based systems, a type is not a first class citizen and has a special status and cannot be modified at run-time.

The notion of class is different from that of type. Its specification is the same as that of a type, but it is more of a run-time notion. It contains two aspects: an object factory and an object warehouse. The object factory can be used to create new objects, by performing the operation new on the class, or by cloning some prototype object representative of the class. The object warehouse means that attached to the class is its extension, i.e., the set of objects that are instances of the class. The user can manipulate the warehouse by applying operations on all elements of the class. Of course, there are strong similarities between classes and types, the names have been used with both meanings and the differences can be subtle in some systems. It do not feel that can be should choose one of these two approaches and it can be consider the choice between the two should be left to the designer of the system. It can be require, however, that the system should offer

some form of data structuring mechanism, be it classes or types. Thus the classical notion of database schema will be replaced by that of a set of classes or a set of types [1,2,6].

1.6.4 Attributes

Attributes represent data components that make up the content of a class. Attributes are called data members in the C++ programming language. Instance attributes are data components that are stored by each instance of the class. Class attributes (static data members in C++) are data values stored once for all instances of the class. Attributes may or may not be visible to external users of the class. Attribute types are typically a subset of the basic data types supported by the programming language that interfaces to the OODBMS. Typically this includes enumeration types such as characters and booleans, numeric types such as integers and floats, and fixed length arrays of these types such as strings. The OODBMS may allow variable length arrays, structures (i.e. records) and classes as attribute types [1, 2, 6].

1.6.5 Behaviors

Behaviors represent the functional component of a class. A behavior describes how an object operates upon its attributes and how it interacts with other related objects. Behaviors are called member functions in the C++ programming language. Behaviors hide their implementation details from users of a class [1].

1.6.6 Encapsulation

The idea of encapsulation comes from (i) the need to cleanly distinguish between the specification and the implementation of an operation and (ii) 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. There are two views of encapsulation: the

programming language view (which is the original view since the concept originated there) and the database adaptation of that view.

The idea of encapsulation in programming languages comes from abstract data types. In this view, an object has an interface part and an implementation part. The interface part is the specification of the set of 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 a procedural part. The data part is the representation or state of the object and the procedure part describes, in some programming language, the implementation of each operation.

The database translation of the principle is that an object encapsulates both program and data. In the database world, it is not clear whether the structural part of the type is or is not part of the interface (this depends on the system), while in the programming language world, the data structure is clearly part of the implementation and not of the interface.

Consider, for instance, an Employee. In a relational system, an employee is represented by some tuple. It is queried using a relational language and, later, an application programmer writes programs to update this record such as to raise an Employee's salary or to fire an Employee. These are generally either written in an imperative programming language with embedded DML statements or in a fourth generation language and are stored in a traditional file system and not in the database. Thus, in this approach, there is a sharp distinction between program and data, and between the query language (for ad hoc queries) and the programming language (for application programs).

In an object-oriented system, the employee can be defined as an object that has a data part and an operation part, which consists of the raise and fire operations and other operations to access the Employee data. When storing a set of Employees,

both the data and the operations are stored in the database. Thus, there is a single model for data and operations, and information can be hidden. No operations, outside those specified in the interface, can be performed. This restriction holds for both update and retrieval operations [6, 10].

1.6.7 Overriding Behaviors and Late Binding

OO applications are typically structured to perform work on generic classes (e.g., a vehicle) and at runtime invoke behaviors appropriate for the specific vehicle being executed upon. Applications constructed in such a manner are more easily maintained and extended since additional vehicle classes may be added without requiring modification of application code. Overriding behaviors is the ability for each class to define the functionality unique to itself for a given behavior. Late binding is the ability for behavior invocation to be selected at runtime based on the class of an object (instead of at compile time) [1, 12].

1.6.8 Inheritance

Inheritance in the object model is a means of defining one class in terms of another. This is common usage for most of us. For example, a conifer is a type of tree. There are certain characteristics that are true for all trees, yet there are specific characteristics for conifers [1].

Note that in an object model, there is no distinction in usage between system predefined types and user-defined types. This is known as extensibility. So it is possible to define a type as a sub-type of a system type or as a sub-type of a user-defined type. So the inheritance is a way to construct new classes from existing classes. It defines what attributes and methods are available in the new class. Classes may inherit from one or more classes. As seen in figure 1.3. A class that inherits from exactly one class is said to use single inheritance (sometimes called simple inheritance).

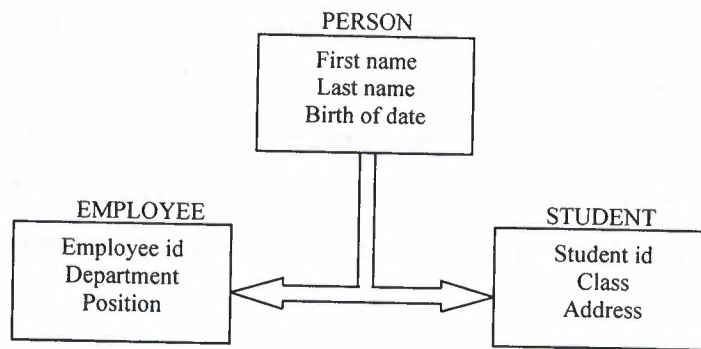


Figure 1.3 Simple Inheritance Process

In this example, a Student is a type of Person. Likewise, an employee is a type of Person. Both Student and Employee inherit all the attributes and methods from Person. Student has a locally defined student ID attribute. Employee has a locally defined employee ID attribute. So, if you would look at a Student object, you would see attributes of name, date of birth, and student ID.

Multiple inheritance means that inheriting from more than one class. Shown in figure 1.4.

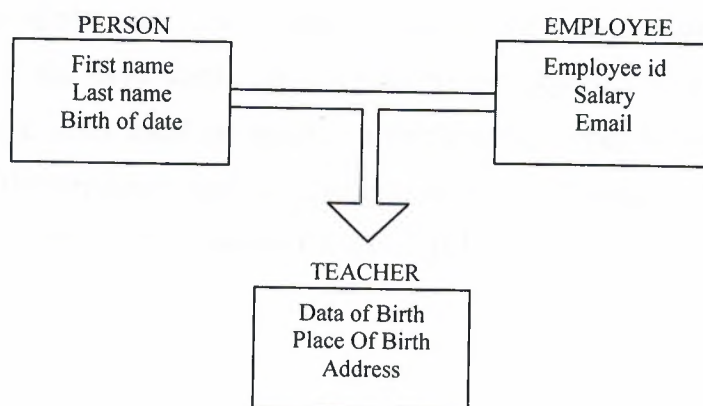


Figure 1.4 Multiple Inheritance Process

In this example, a Teacher is a type of Person. Also a teacher is a type of employee. So Teacher inherits all the attributes and methods of Person and Employee. Inheritance is a powerful object oriented modeling concept that supports reuse and extensibility of existing classes. The inheritance relationships between a group of classes define a class hierarchy. Class hierarchies improve the ability of users to understanding software systems by allowing knowledge of one class (a superclass) to be applicable to other classes (its subclasses) [6].

1.6.9 Naming

OO applications are characterized as being composed of a network of interconnected objects. An application begins by accessing a few known objects and then traverses to additional objects via relationships from the known objects. As objects are created they are linked (i.e. related) to other existing objects. Given this scenario, the database must provide some mechanism for identifying one or more objects at application start-up without using relations from existing objects. This is typically accomplished by allowing objects to be named and providing a retrieval mechanism based upon name. An application begins by loading one or two 'high-level' objects that it knows by name and then traverses to other reachable objects.

Object names apply within some name scope. Within a given scope, names must be unique (i.e. the same name can not refer to two objects). The simplest scope model is for the entire database to act as a single name scope. An alternative scope model is for the application to identify name scopes. Using multiple name scopes will reduce the chance for name conflicts [1,6,10].

1.7 Summary

Object-oriented programming (OOP) is a method of creating a software application from a group of software components called objects. Objects are at the heart of OOP. Objects are self-contained software components that are used to build object oriented applications, also objects contain data (usually referred to as attributes) and ways of working with the data (methods).

The basic Object-Oriented modelings are Complex Objects, Object Identity, classes, attributes, behaviors, encapsulation, inheritance, overriding behaviors and late binding, naming, classes or type hierarchies.

CHAPTER TWO

OBJECT ORIENTED DATABASE CONCEPTS

2.1 Overview

This chapter explains OODB SYSTEMS perspectives; also it explains the two architecture of OODBMS, client-server architecture, and the storage of methods. Also it explains some concepts such that integrity, concurrency control, Recovery, transactions, persistence, and security.

2.2 OODB SYSTEMS Perspectives

Database systems are primaries concerned with the creation and maintenance of large, long-lived collections of data. Modern database systems are characterized by their support of the following features:

- **A data model:** A particular way of describing data, relationships between data, and constraints on the data.
- **Data persistence:** the ability for data to outlive the execution of a program, and possibly the lifetime of the program itself.
- **Data sharing:** The ability for multiple applications (or instances of the same one) to access common data, possibly at the same time.
- **Reliability:** The assurance that the data in the database is protected from hardware and software failures.
- **Scalability:** The ability to operate on large amount of data in simple ways.
- **Security and integrity:** The protection of the data against unauthorized access, and the assurance that the data conforms to specified correctness and consistency rules.
- **Distribution:** The ability to physically distribute a logically interrelated collection of shared data over a computer network, preferably making the distribution transparent to the user.

In contrast, traditional programming languages provide constructs for procedural control and for data and functional abstraction, but lack built-in support for many of the above database features. While each are useful in their respective domains, there exist an increasing number of applications that require functionality from both database system and programming languages. Such applications are characterized by their need to store and retrieve large amounts of shared, structured data.

In the last two decades, there has been considerable effort invested in developing systems that integrate the concepts from these two domains. However, the two domains have slightly different perspectives that have to be considered and the differences addressed [8].

Perhaps two of the most important concerns from the programmers perspective are performance and ease-of-use, both achieved by having a more seamless integration between the programming language and the DBMS than that provided with traditional database systems. With a traditional DBMS:

- It is the programmer's responsibility to decide when to read and update objects (records)
- The programmer has to write code to translate between the application's object model and the data model of the DBMS (for example, relations) which might be quite different. With an object-oriented programming language, where an object may be composed of many sub-objects represented by pointers, the translation may be particularly complex. In fact, it has been claimed that a significant amount of programming effort and code space is devoted to this type of mapping, possibly as much as 30% as noted above. if this mapping process can be eliminated or at least reduced, the programmer would be freed from this responsibility, the

resulting code would be easier to understand and maintain, and performance may increase as a result.

- It is the programmer's responsibility to perform additional type-checking when an object is read back from the database. For example, the programmer may create an object in the strongly-typed object-oriented language java and store it in a traditional DBMS. However, another application written in a different language may modify the object, with no guarantee that the object will conform to its original type.

These difficulties stem from the fact that conventional DBMS have a two-level storage model: the application storage model in main or virtual memory, and the database storage model on disk, as illustrated in figure (2.1). In contrast, an OODBMS tries to give the illusion of a single-level storage model, with a similar representation in both memory and in the database stored on disk, as illustrated in figure (2.2).

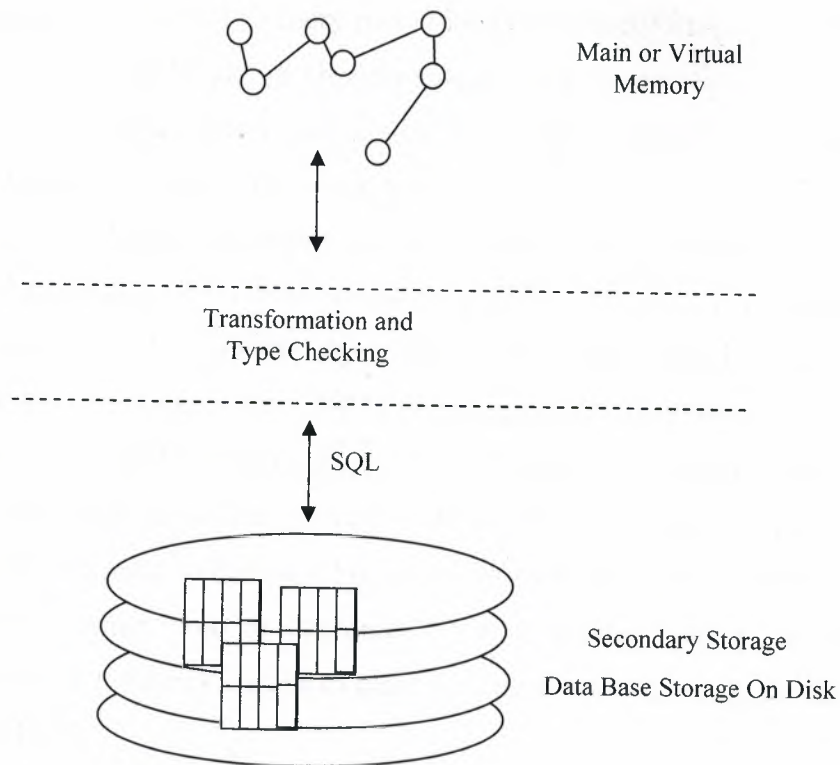


Figure 2.1 Tow Level Storage Model

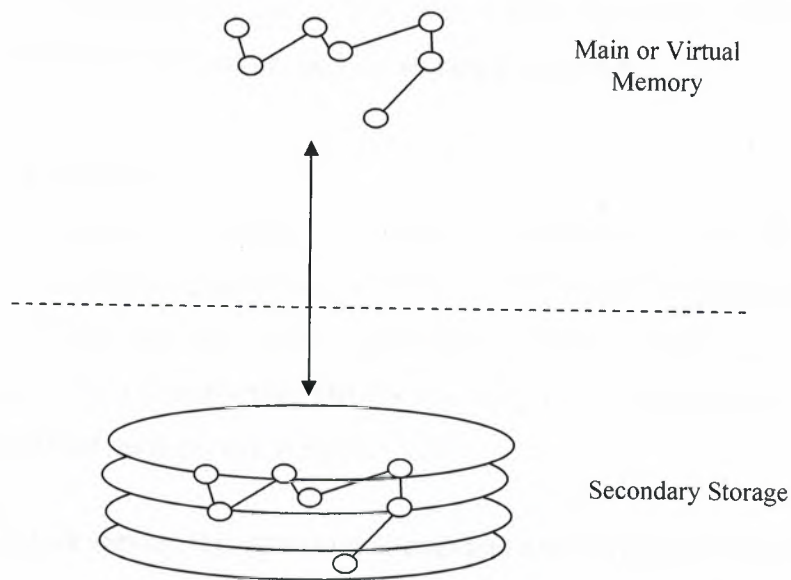


Figure 2.2 Single-Level Storage Model

Although the single-level memory model looks intuitively simple, to achieve this illusion the OODBMS has to cleverly manage the representations of objects in memory and on disk objects, and relationships between objects, are identified by object identifiers (OIDs). There are two types of OIDs: logical OIDs that are independent of the physical location of the object on disk, and physical OIDs that encode the location. In the former case, a level of indirection is required to look up the physical address of the object on disk. In both cases, however, an OID different in size from a standard in-memory pointer that need only be large enough to address all virtual memory. Thus, to achieve the required performance, an OODBMS must be able to convert OIDs to and from in-memory pointers. This conversion technique has become known as 'pointer swizzling' or 'object faulting', and the approaches used to implement it have become varied, ranging from software-based residency checks to page faulting schemes used by the underlying hardware [7,8].

2.3 Architecture

There are two architecture issues: how best to apply the client-server architecture to the OODBMS environment, and the storage of methods.

2.3.1 Client-server

Many commercial OODBMSs are based on the client-server architecture to provide data to users, applications, and tools in distributed environment. However, not all systems use the same client-server model. There are three basic architectures for a client-server DBMS that vary in the functionality assigned to each component, as depicted in figure (2.3).

- **Object server:** this approach attempts to distribute the processing between the two components. Typically, the server process is responsible for managing storage, locks, commits to secondary storage, logging and recovery, enforcing security and integrity, query optimization, and executing stored procedures. The client is responsible for transaction management, and interfacing to the programming language. This is the best architecture for cooperative, object-to-object processing in an open, distributed environment.
- **Page server:** in this approach, most of the database processing is performed by the client. The server is responsible for secondary storage and providing pages at the client's request.
- **Database server:** in this approach, most of the database processing is performed by the server. The client simply passes requests to the server, receives result, and passes them on to the application. This is the approach taken by many relational DBMSs.

In each case, the server resides on the same machine as the physical database. The client may reside on the same or different machine. If the client needs access to

database distributed across multiple machines, then the client communicates either a server on each machine. There may also be a number of clients communicating with one server: for example, one client for each user or application [3, 8].

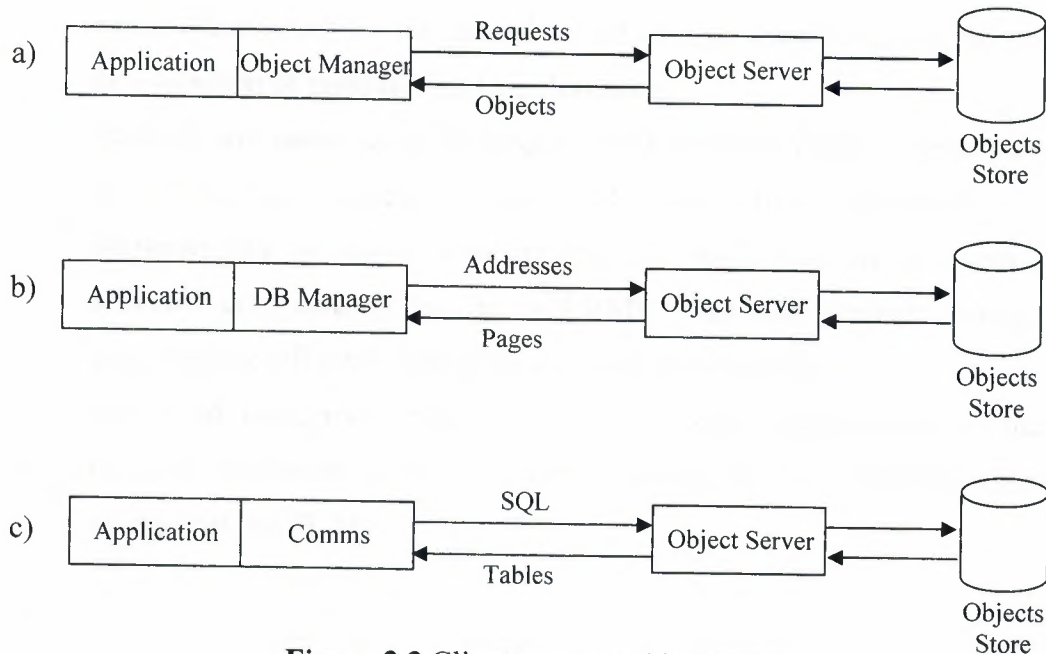


Figure 2.3 Client-server architectures:
(a) object server (b) page server (c) database server.

2.3.2 Storing and executing methods

There are two approaches to handling methods: (1) to store the methods in external files, as shown in figure 2.4 (a); and (2) to store the methods in the database, as shown in figure 2.4 (b). The first approach is similar to function libraries or application programming interfaces (APIs) found in traditional DBMSs, in which an application program interface with a DBMS by linking in functions supplied by the DBMS vendor. With the second approach, methods are stored in the database and are dynamically bound to the application at runtime. The second approach offers several benefits:

- **It eliminates redundant code** Instead of placing a copy of a method that accesses a data element in every program that deals with that data, the method is stored only once in the database.
- **It simplifies the modifications** of methods which require to be changed into one place only. All the programs automatically use the updated method-Depending on the nature of the change, rebuilding, testing, and redistribution of programs may be eliminated.
- **Methods are more secure** Storing the methods in the database gives them all the benefits of a security provided automatically by the OODBMS.
- **Methods can be shared concurrently** Again concurrent access is provided automatically by the OODBMS. This also prevents multiple users making different changes to a method simultaneously.
- **Improved integrity** Storing the methods in the database means that integrity constraints can be enforced consistently by the OODBMS across all applications [3, 8].

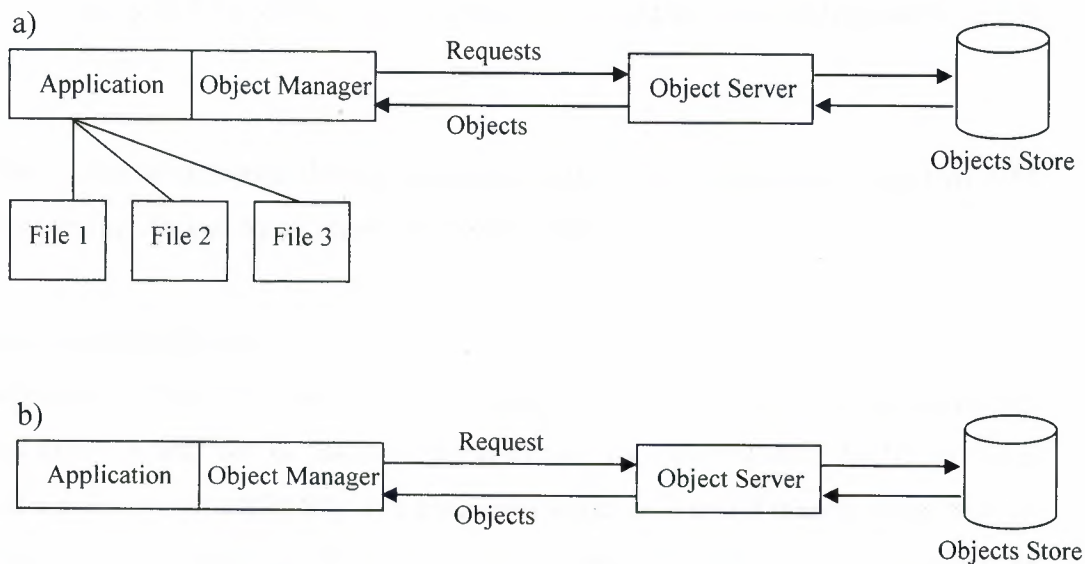


Figure 2.4 Strategies for handling method:
 (a) Storing method outside database; (b) Storing method in database.

2.4 Integrity

2.4.1 Relationships Integrity

Relationships are presented in an object data model using reference attributes. There are three types of the relationships: one-to-one (1:1), one-to-many (1: M), and many-to-many (M: M).

- **1:1 relationships:** A 1:1 relationship between objects A and B is presented by adding a reference attribute to object A and, to maintain referential integrity, a reference attribute to object B.
- **1:M relationships:** A 1:M relationship between objects A and B is presented by adding a reference attribute to object B and an attribute containing a set of references to A.
- **M:N relationships:** A M:N relationship between objects A and B is presented by adding an attribute containing a set of references to each object.

The relational database design decomposes the M:N relationship into two 1:M relationships linked by an intermediate entity [8].

2.4.2 Nulls Integrity

Represent a value for an attribute that is currently unknown or is not applicable for this tuple. A null can be taken to mean the logical value 'unknown'. It can mean that a value is not applicable to a particular tuple, or it could merely mean that no value has yet been supplied. Nulls are a way to deal with incomplete or exceptional data. However, a null is not the same as a zero numeric value or a text string filled with spaces; zeros and spaces are values, but a null represents the absence of a value. Therefore, nulls should be treated differently from other

values. Some authors use the term 'null value'. In fact, a null is not a value but represents the absence of a value and so the term 'null value' is deprecated.

Without nulls, it becomes necessary to introduce false data to represent this state or to add additional attributes that may not be meaningful to the user.

Null can cause implementation problem. The difficulty arises because the relational model is based on first-order predicate calculus, which is a two-valued or Boolean logic- the only values allowed are true or false. Allowing nulls means that we have to work with a higher-valued logic, such as three- or four-valued logic.

2.4.3 Referential Integrity

The existence of relationships gives rise to the need for referential integrity. There are several techniques that can be used to handle referential integrity:

- Do not allow the user to explicitly delete objects. In this case, the system is responsible for 'garbage collection'; in other word, the system automatically deletes objects when they are no longer accessible by the user.
- Allow the user to delete objects when they are no longer required. In this case, the system may detect invalid references automatically and set the reference to NULL (the null pointer) or disallow the deletion. The versant OODBMS uses this approach to enforce referential integrity.
- Allow the user to modify and delete objects and relationships when they are no longer required. In this case, the system automatically maintains the integrity of objects. Inverse attributes can be used to maintain referential integrity [3, 8].

2.4.4 Entity integrity

The first integrity rule applies to the primary keys of base relation. Thus, nowadays, it could be defined as a base relation which is the relation that corresponds to an entity in the conceptual schema.

Entity integrity in a base relation, no attribute of a primary key can be null. By definition, a primary key is a minimal identifier that is used to identify tuples uniquely. This means that no subset of the primary key is sufficient to provide unique identification of tuples. If it allows a null for any part of a primary key, we are implying that not all the attributes are needed to distinguish between tuples, which contradicts the definition of the primary key [8].

2.5 Concurrency Control

ODBMS provide concurrency control mechanisms to ensure that concurrent access to data does not yield inconsistencies in the database or in applications due to invalid assumptions made by seeing partially updated data. The problems of lost updates and uncommitted dependencies are well documented in the database literature. Relational databases solve this problem by providing a transaction mechanism that ensures atomicity and serializability. Atomicity ensures that within a given logical update to the database, either all physical updates are made or none are made. This ensures the database is always in a logically consistent state, with the DB being moved from one consistent state to the next via a transaction. Serializability ensures that running transactions concurrently yield the same result as if they had been run in some serial order. Relational databases typically provide a pessimistic concurrency control mechanism. The pessimistic model allows multiple processes to read data as long as none update it. Updates must be made in isolation, with no other processes reading or updating the data. This concurrency model is sufficient for applications that have short transactions, so that applications are not delayed for long periods due to access conflicts [5, 6].

For applications being targeted by OODBMS (e.g. multi-person design applications), the assumption of short transactions is no longer valid. Optimistic concurrency control mechanisms are based on the assumptions that access conflicts will rarely occur. Under this scenario, all accesses are allowed to proceed and, at transaction commit time, conflicts are resolved. OODBMS have incorporated the idea of optimistic concurrency control mechanisms for building applications that will have long transaction times. Handling of conflicts at commit time cannot simply abort a transaction, however, since one designer may be losing days or weeks of work. OODBMS must provide techniques to allow multiple concurrent updates to the same data and support for merging these intermediate results at an appropriate time (under application control).

An alternative policy is to allow reading and a single update to occur in parallel. Readers are made aware that the data they are reading may be in the midst of an update. Thus readers may be viewing slightly outdated information. Implementation of this approach fits well in the client-server architecture typical of an OODBMS. Each client application gets its own local copy of the data. If an update is made to the data, the server does not permanently store it until all concurrent read transactions are completed. Thus, all read transactions execute seeing a consistent data set, albeit one that is in the process of being updated. Once all readers have completed, the write transaction is allowed to complete modifying the permanent copy of the data. Some OODBMS may, at transaction commit, inform reading clients that the data they just read is in the process of being updated [3, 5, 6].

2.6 Recovery

Recovery is the ability for a database to return to a consistent state after a software or hardware failure. Similar to concurrency, the transaction concept is used to implement recovery and to define the boundaries of recovery activity. One or more forms of database journaling, backup, checkpointing, logging, shadowing,

and/or replication are used to identify what needs to be recovered and how to perform a recovery. Databases must typically respond to application failures, system failures, and media failures. Application failures are typically trapped by the transaction mechanism and recovery is implemented by rolling back the transaction. System failures, such as loss of power, may require log and/or checkpoint supported rollback of uncommitted transactions and roll forward of transactions that were committed but not completely flushed to disk. Media failures, such as a disk head crash, require restoration of the database from a backup version, and replaying of transactions that have been committed since the backup.

The ability of a database to recover from failures results in a heavy processing and storage overhead. In the process of evaluating an OODBMS, its ability to recover from faults, and the overhead incurred to provide that recovery capability, must be carefully considered. Applications envisioned for OODBMS often do not have the same strict recovery requirements as do relational database applications (e.g. banking systems). In addition, the amount of data stored in such systems may result in unacceptable storage overheads for many forms of recovery. For these reasons, an OODBMS evaluation effort must carefully select the recovery capabilities needed based on both the functional and performance requirements of the application [3, 5, 6].

2.7 Transactions

Transactions are the mechanism used to implement concurrency and recovery. Different transaction policies have been described in Section 2.5, Concurrency, under the topics of pessimistic, optimistic, and multiple readers/single write concurrency control policies. Within a transaction, data from anywhere in the (distributed) database must be accessible. A feature found in many OODBMS products is to commit a transaction but to allow the objects to remain in the client cache under the expectation that they will soon be referenced again.

Some OODBMS have incorporated the concept of long and/or nested transactions. A long transaction allows transactions to last for hours or days without the possibility of system generated aborts (due to lock conflicts for example). System generated aborts must be avoided for applications targeting OODBMS since a few hours or days of work cannot be simply discarded. Long transactions may be composed of nested transactions for purposes of recovery [5, 6].

2.8 Persistence

Persistence is the characteristic that makes data available across executions. The objective of an OODBMS is to make objects persistent. Persistence may be based on an object's class, meaning that all objects of a given class are persistent. Each object of a persistent class is automatically made persistent. An alternative model is that persistence is a unique characteristic of each object (i.e., it is orthogonal to class). Under this model, an object's persistence is normally specified when it is created. A third persistence model is that any object reachable from a persistent object is also persistent. Such systems require some way of explicitly stating that a given object is persistent (as a means of starting the network of inter-connected persistent objects) [2, 3, 5].

2.9 Security

Secure OODBMSs protect their data from malicious misuse. Security requirements are similar to data integrity requirements that protect data from accidental misuse. Secure databases typically provide a multi-level security model where users and data are categorized with a specific security level. Mandatory security controls ensure that users can access data only at their level and below. Discretionary security controls provide access control based on explicit authorization of a user's access to data. Applications targeted by OODBMS often do not require strict security controls, although discretionary access controls seem desirable for work-group design type applications [3, 5].

2.10 Summary

There are two architecture of OODBMS environment, the client-server architecture, and the storage of methods.

Relationships are presented in an object data model using reference attributes. There exist several kinds of integrity such as relationships integrity, nulls integrity, referential integrity, and entity integrity. ODBMS provide some mechanisms, such as integrity, concurrency control, recovery, transactions, persistence, and security.

CHAPTER THREE

OBJECT-RELATIONAL DBMS

3.1 Overview

This chapter discusses some concepts related to the object-relational database system such as object identity, row types, user-defined types (UDTs), user-defined routines, polymorphism, subtypes and supertypes, persistent stored modules, and large objects, also the end of this chapter there exist comparison between ORDBMS and OODBMS.

3.2 Introduction to Object-Relational Database System

Until recently, the choice of DBMS seemed to be between the relational DBMS and the object-oriented DBMS. However, vendors of RDBMS products are still conscious of the threat and promise of the OODBMS. They agree that their systems are not currently suited to the advanced applications, and that added functionality is required. However, they reject the claim that extended RDMSs will not provide sufficient functionality or will be too slow to cope adequately with the new complexity.

The examining of the advanced database applications that are emerging, due to find extensive use of many object-oriented features such as a user-extensible type system, encapsulation, inheritance, polymorphism, dynamic binding of method, complex objects including non-first normal form objects, and object identity. The most obvious way to remedy the shortcomings of the relational model is to extend the model with these types of features. This is the approach that has been taken by many prototype extended relational systems, although each has implemented different combinations of features. Thus, there is no single extended relational model; rather, there are a variety of these models. However all the models do share the same basic relational tables and query language, all incorporate some

concept of 'object', and some have the ability to store methods (or procedures or triggers) as well as data in the database.

Various terms have been used for systems that have extended the relational data model. The original term that was used to describe such systems was the extended relational DBMS (ERDBMS). However, in recent years the more descriptive term Object-Relational DBMS has been used to indicate that the system incorporates some notion of 'object', and more recently the term Universal Server or Universal DBMS (UDBMS) has been used. It stands for Object-Relational DBMS (ORDBMS). Three of the leading RDBMS vendors (Oracle, Informix, and IBM) have all extended their systems to become ORDBMSs, although the functionality provided by each is slightly different. The concept of the ORDBMS, as a hybrid of the RDBMS and OODBMS, is very appealing, preserving the wealth of knowledge and experience that has been acquired with the RDBMS. Some analysts predict the ORDBMS will have a 50% larger share of the market than the RDBMS [8,15].

The main advantages of extending the relational data model come from reuse and sharing. Reuse comes from the ability to extend the DBMS server to perform standard functionality centrally, rather than have it coded in each application. For example, applications may require spatial data type that represent points, lines, and polygons, with associated functions that calculate the distance between two points, the distance between a point and a line, whether a point is contained within a polygon, and whether two polygonal regions overlap, among others. If it is possible to embed this functionality in the server, it saves having to define them in each application that needs them, and consequently allows the functionality to be shared by all applications. These advantages also give rise to increased productivity both for the developer and for the end-user.

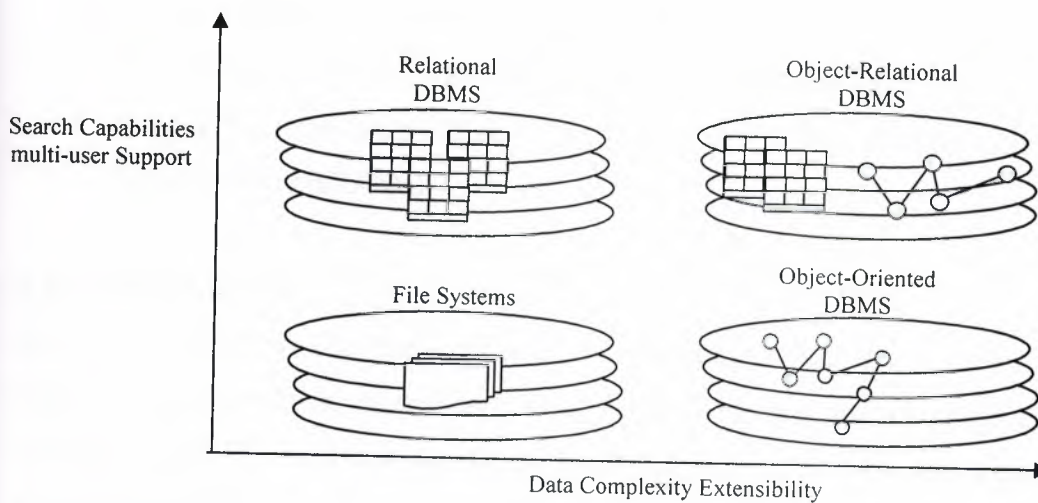


Figure 3.1 Classification of DBMSs

Another obvious advantage is that the extended relational approach for each serves. The significant bodies of knowledge and experience have been gone into developing relational applications. This is a significant advantage, as many organizations would find it prohibitively expensive to change. If the new functionality is designed appropriately, this approach should allow organizations to take advantage of the new extensions in an evolutionary way without losing the benefits of current database features and functions. Thus, an ORDBMS could be introduced in an integrative fashion, as proof-of-concept projects. The forthcoming SQL standard is designed to be upwardly compatible with the current SQL standard, and so any ORDBMS that complies with SQL3 should provide this capability [8,15].

3.3 SQL3

The object-oriented features proposed in the next SQL standard, SQL3, covering:

- Type constructors for row types and reference type.
- User-defined types (distinct types and structured types) that can participate in supertype/subtype relationships.

- User-defined procedure, functions and operators.
- Type constructors for collection types (arrays, sets, and lists).
- Support for large objects Binary Large Objects (BLOBs) and Character Large Objects (CLOBs).

3.3.1 Object identity

Each relation has an implicitly defined attribute named OID that contains the tuple's unique identifier, where each OID value is created and maintained by postgres. The OID attributes can be accessed but not updated by user queries. Among other users, the OID can be used as a mechanism to simulate attribute types that reference tuples in other relation. The relation name can be used for the type name because relations, types, and procedures have separate name spaces [8].

3.3.2 Row types

A row type is a sequence of field name/date type pair that provides a data type that can represent the types of rows in tables, so that complete rows can be stored in variables, passed as arguments to routines and returned as return values from function calls. A row type can also be used to allow a column of a table to contain row values [1, 8].

3.3.3 User-Defined Types (UDTs)

It refers to user-defined types as Abstract Data Types (ADTs), that may be used in the same way as the built-in types (for example CHAR ,INT,FLOAT). UDTs are subdivided into two categories: distinct types and structured types. The simplest type of UDT in SQL3 is the distinct type, which allows differentiation between the same underlying base types. In its more general case, a UDT definition consists of one or more attribute definitions. It has also been proposed that a UDT definition consist additionally of routine declarations. If this proposal is not accepted, these declarations form part of the schema. In what follows, it can be assumed that UDT definition may contain routine declarations. It stands to

routines and operators generically as routines. In addition, within the UDT definition it can be also define the equality and ordering relationships for the UDT.

There is still some discussion in the SQL3 drafting teams whether attributes and routines should be further protected using the tags public, private, or protected, as in C++, with the following interpretations:

- Only public components are visible to authorized users of the UDT.
- Private components are visible only within the definition of the UDT that contains them.
- Protected components are partially encapsulated, begin visible both within their own UDT and within the definitions of all subtypes of that UDT.

If no tag is specified, the last specified tag is assumed. The default for the first tag is public. The value of an attribute can be accessed using a modified dot notation [1, 8].

3.3.4 User-Defined Routines

User-defined routines (UDRs) define methods for each manipulating data and are an important adjunct to UDTs. An ORDBMS should provide significant flexibility in this area, such as allowing UDRs to return complex values that can be further manipulated (such as tables), and support for overloading of function names to simplify application development. In SQL3, UDRs may be defined as a part of a UDT or separately as part of a schema. An SQL-invoked routine may be a procedure, function, or iterative routine. It may be externally provided in a standard programming language such as C/C++, or defined completely in SQL using extensions that make the language computationally complete. An SQL-invoked procedure is invoked from an sql CALL statement. It may have zero or more parameters, each of which may be an input parameter(IN), an output

parameter (OUT), or both an input and output parameter (INOUT), and it has a body if it is defined fully within SQL. An SQL-invoked function returns a value; any specified parameter [1, 2, 8].

3.3.5 Relations and inheritance

A relation inherits all attributes from its parents unless an attribute is overridden in the definition. Multiple inheritance is supported, however, if the same attribute can be inherited from more than one parent and the types of the attributes are different, the declaration is disallowed. Key specifications are also inherited [2, 8].

3.3.6 Polymorphism

Different routines may have the same name, that is routine names may be overloaded, for example to allow a UDT subtype to redefine a method inherited from a supertype, subject to the following constraints:

- No two function in the same schema are allowed to have the same signature, that is, the same number of arguments, the same data types for each argument, and the same return datatype.
- No two procedure in the same schema are allowed to have the same name and the same number of parameters.

The current draft SQL3 proposal uses a generalized object model, so that the types of all arguments to a routine are taken into consideration when determining which routine to invoke, in order from left to right. Where there is not an exact match between the data type of the argument and the data type of the parameter specified, type precedence list are used to determine the closest match. the exact rules for routine determination for a given invocation are relatively complex [8].

3.3.7 Subtypes and supertypes

SQL3 allows UDTs to participate in a subtype/supertype hierarchy. A type can have more than one supertype (that is, multiple inheritance is supported), and more than one subtype. A subtype inherits all the attributes and behavior of its supertypes and it can define additional attributes and functions like any other UDT and it can override inherited function [8].

3.3.8 Persistent stored modules

A number of new statement types have been added in SQL3 to make the language computationally complete, so that object behavior (methods) can be stored and executed from within the database as SQL statements. Statements can be grouped together into a compound statement (block), with its own local variables. Some of the additional statements provided in SQL3 are:

- An assignment statement that allows the result of an SQL value expression to be assigned to a local variable, a column, or an attribute of a UDT.
- An IF...THEN...ELSE...END IF statement that allows conditional processing.
- A CASE statement that allows the selection of an execution path based on a set of alternatives.
- A set of statements that allows repeated execution of a block of SQL statements. The iterative statements are FOR, WHILE, and REPEAT.
- A CALL statement that allows procedures to be invoked and RETURN statement that allows an SQL value expression to be used as the return value from an SQL function [15].

3.3.9 Large Objects

A Large Object is a table field that holds a large amount of data as a long text file or a graphics file. There are three different types of large object data types defined in SQL3:

- Binary Large Object (BLOB), a binary string that does not have a character set or collation association.
- Character Large Object (CLOB) and National Character Large Object (NCLOB), both character strings.

The SQL large object is slightly different from the original type of DLOB that appears in many current database systems. In such systems, the BLOB is non-interpreted byte stream, and the DBMS does not have any knowledge concerning the content of the BLOB or its internal structure. This prevents the DBMS from performing queries and operations on inherently rich and structured data types, such as images, video, word processing documents, or web pages. Generally, this requires that the entire BLOB be transferred across the network from the DBMS server to the client before any processing can be performed. In contrast, the SQL3 large object does allow some operations to be carried out in the DBMS server.

The standard string operators, which operate on characters strings and return character strings, also operate on character large object string, such as:

- The concatenation operator, (string1|| string2), which returns the character string formed by joining the character string operands in the specified order.
- The character substring function, SUBSTRING (string FROM startops FOR length), which returns a string extracted from a specified string from a start position for a given length.
- The fold function, UPPER (string) and LOWER (string), which convert all characters in a string to upper/lower case.
- The length function, CHAR+LENGTH (string), which return the length of the specified string.

- The position function, POSITION(string1 IN string2), which returns the start position of string1 within string2.

However, CLOB strings are not allowed to participate in most comparison operations, although they can participate in a LIKE predicate, and a comparison or quantified comparison predicate that uses the equal (=) or not equal(<>) operators.

3.4 Comparison of ORDBMS and OODBMS

It can be conclude the treatment of Object-Relational DBMS and Object-Oriented DBMS with a brief comparison of the two types of system. it can be assumed that future ORDBMSs will be compliant with SQL3 [8].

Table 3.1 Comparison Between ORDBMS and OODBMS.

Feature	ORDBMS	OODBMS
Encapsulation	Supported through UDTs	Supported and broken for queries
Inheritance	Supported (separate hierarchies for UDTs and tables)	Supported
Polymorphism	Supported (UDF invocation based on the generic function)	Supported as in an object oriented programming model language.
Relationships	Strong support with user-defined referential integrity constraints	Supported (for example, using class libraries)
Integrity constraints	Strong support	No support

Recovery	Strong support	Supported but degree of support differs between products
Advanced transaction models	No support	Supported but degree of support differs between products
Security, integrity, and views	Strong support	Limited support

3.5 Summary

The concept of the ORDBMS is as a hybrid of the RDBMS and OODBMS. The object-oriented features proposed in SQL3 support type constructors for row types and reference types, user-defined types, user-defined procedures, functions and operators, and support for large objects Binary Large Objects (BLOBs) and Character Large Objects (CLOBs).

CHAPTER FOUR

OBJECT RELATIONAL ALGEBRA

4.1 Overview

Object relational algebra is a set of operators that take relations as their operands and return a relation as their result. There are eight operators restrict, project, product, union, intersection, difference, join, and divide. In this chapter these operators will be explain in detail.

4.2 Introduction to the Original Algebra

The traditional set operators union, intersection, difference, and Cartesian product (all of them modified somewhat to take account of the fact that their operands are, specifically, relations instead of arbitrary sets). The special relational operators restrict (also know as select), project, join, and divide.

Here are simplified definitions of these eight operators (refer to fig 4.1):

Restrict :	Returns a relation containing all tuples from a specified relation that satisfy a specified condition.
Project :	Returns a relation containing all (sub) tuples that remain in a specified relation after specified attributes have been removed.
Product :	Returns a relation containing all possible tuples that are a combination of two tuples, one from each of tows specified relations.
Union :	Returns a relation containing all tuples that appear in either or both of tow specified relations.
Intersect :	Returns a relation containing all tuples that appear in both of two specified relations.
Difference:	Returns a relation containing all tuples that appear in the first and not the second of two specified relations.
Join :	Returns a relation containing all possible tuples that are

	combinations of two tuples, one from each of two specified relations, such that the two tuples contributing to any given combination have a common value for the common attributes of the two relations (and that common value appears just once, not twice, in the result tuple).
Divide :	takes two unary relations and one binary relation and returns a relation containing all tuples from one unary relation that appear in the binary relation matched with all tuples in the other unary relation [5].

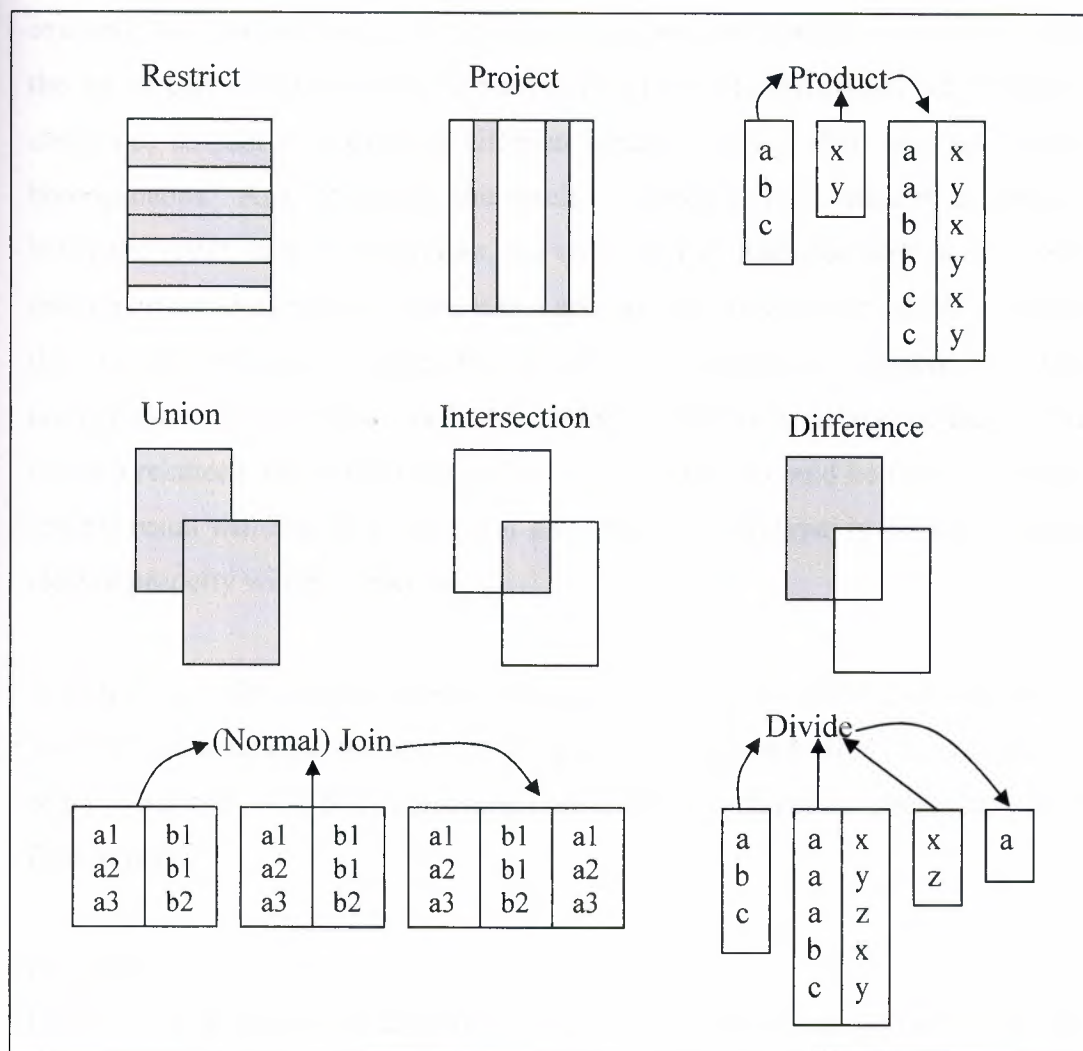


Figure 4.1 Original Eight Operators (Overview)

4.3 Semantics

In this section there is an explanation of the previous eight operators, with fully explained examples. Where it considers the operators in the sequence union, intersection, difference, product, restrict, project, join, and divide.

4.3.1 Union

In mathematics, the union of two sets in the set of all elements belonging to either or both of the original sets. Since a relation is a set, namely a set of tuples, it is obviously possible to construct the union of two such sets; the result will be a set consisting of tuples appearing in either or both of the original relation. For example, the union of the set of supplier tuples currently appearing in relvar S and the set of part tuples currently appearing in relvar P is certainly a set. Where it could not contain a mixture of different kinds of tuples, they must be "tuple-homogeneous." And, of course, the result is wanted to be a relation, in spite of being the result as a set. Therefore, the union in the relational algebra is not the usual mathematical union; rather, it is a special kind of union, in which required the two input relations to be of the same type- meaning, for example, that they both contain supplier tuples, or both part tuples, but not a mixture of the two. If the two relations are of the same (relation) type, then it could be taken as union, and the result will also be a relation of the same (relation) type; in other words, the closure property will be preserved.

Here then is a definition of the relational union operator: Given two relations A and B of the same type, the union of those tow relations, $A \text{ UNION } B$, is a relation of the same type, with body consisting of all tuples t such that t appears in A or in B or in both .

Example:

Let relation A and B be as shown in Figure 4.2 (both are derived from the suppliers relvar S; A is the suppliers in London, and B is the suppliers who supply

part P1, intuitively speaking). Then $A \cup B$ - see part a. of the Figure- is the suppliers who either are located in London or supply part P1 (or both).

$$\sigma_{S\#, sname, status, city}(A) \cup \sigma_{S\#, sname, status, city}(B)$$

Notice that the result has three tuples, not four; duplicate tuples are eliminated, by definition. It can be remarked in passing that the only other operation for which this question of duplicate elimination arises is in projection [5].

4.3.2 Intersect

Like union and for essentially the same reason, the relational intersection operator requires its operands to be of the same type. Given two relations A and B of the same type, then, the intersection of those two relations, $A \cap B$, is a relation of the same type, with body consisting of all tuples t such that t appears in both A and B .

Example: Again, let A and B be as shown in figure 4.2. Then $A \cap B$ – see part b of the figure- is the suppliers who are located in London and supply part P1 [5].

$$\sigma_{S\#, sname, status, city}(A) \cap \sigma_{S\#, sname, status, city}(B)$$

Notice that the result has one tuple; this tuple is repeated in both sets A and B .

4.3.3 Difference

Like union and intersection, the relational difference operator also requires its operands to be of the same type. Given two relation A and B of the same type, then, the difference between those two relations, $A - B$ (in that order), is a relation of the same type, with body consisting of all tuples t such that t appears in A and not in B .

Example: Let A and B again be as shown in figure 4.2. Then A MINUS B- see part c. of the figure – is the suppliers who are located in London and do not supply part P1.

$\sigma.S\#, sname, status, city(A) - \sigma.S\#, sname, status, city(B)$

Also B MINUS A -see part d . of the figure- is the suppliers who supply part P1 and are not located in London. Observe that MINUS has a directionality to it, just as subtraction does in ordinary arithmetic [5].

$\sigma.S\#, sname, status, city(B) - \sigma.S\#, sname, status, city(A)$

A			
S#	SNAME	STATUS	CITY
S1	Smith	20	London
S4	Clark	20	London

a. Union (A UNION B)

b. Intersection (A INTERSECT B)

c. Difference (A MINUS B)			
S#	SNAME	STATUS	CITY
S4	Clark	20	London

B			
S#	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S4	Clark	20	London
S2	Jones	10	Paris

S#	SNAME	STATUS	CITY
S1	Smith	20	London

d. Difference (B MINUS A)			
S#	SNAME	STATUS	CITY
S3	Jones	10	Paris

Figure 4.2 Union, Intersection and Difference Examples

4.3.4 Product

In mathematics, the Cartesian product (product for short) of two sets in the set of all ordered pairs such that, in each pair, the first element comes from the first set and the second element comes from the second set. Thus, the Cartesian product of two relations would be a set of ordered pairs of tuples, loosely speaking. But again

it can be want to preserve the closure property; in other words, the result wanted to contain tuples per se, not ordered pairs of tuples. Therefore, the relational version of Cartesian product is an extended form of the operation, in which each ordered pair of tuples is replaced by the single tuple that is the union of the two tuples in question (using "union" in its normal set theory sense, not its special relational sense). That is, given the tuples $\{A1:a1, A2:a2, \dots, Am:am\}$ and $\{B1:b1, B2:b2, \dots, Bn:bn\}$ this union of the two is the single tuples $\{A1:a1, A2:a2, \dots, Am:am, B1:b1, B2:b2, \dots, Bn:bn\}$. Another problem that occurs in connection with Cartesian product is that (of course) the result relation required to have a well-formed heading (i.e., to be of a proper relation type). Now, clearly the heading of the result consists of all of the attributes from both of the two input relations. A problem will therefore arise if those two headings have any attribute names in common; if the operation were permitted, the result heading would have two attributes with the same name and would thus not be "well-formed." If construct the Cartesian product needed for two relation that do have any such common attribute names, therefore it must use the RENAME operator first to rename attributes appropriately. It can be defined that the (relational) Cartesian product of tow relations A and B, $A \text{ TIMES } B$, where A and B have no common attribute names, to be a relation with a heading that is the (set theory) union of the headings of A and B and with a body consisting of the set of all tuples t such that t is the (set theory) union of a tuple appearing in A and a tuple appearing in B. Note that the cardinality of the result is the product of the cardinalities of A and B, and the degree of the result is the sum of their degrees.

Example: Let relations A and B be as shown in figure 4.3 (A is all current supplier numbers and B is all current part numbers, intuitively speaking). Then $A \text{ TIMES } B$ -see the lower part of the figure- is all current supplier-number/part-number pairs [5].

For example $R = S \# \times p \#$

A	S#	B	P#
	S1		P1
	S2		P2
	S3		P3
	S4		P4
	S5		P5

Cartesian product (A TIMES B)

S#	P#
S1	P1	S2	P1	S3	P1	S4	P1	S5	P1
S1	P2	S2	P2	S3	P2	S4	P2	S5	P2
S1	P3	S2	P3	S3	P3	S4	P3	S5	P3
S1	P4	S2	P4	S3	P4	S4	P4	S5	P4
S1	P5	S2	P5	S3	P5	S4	P5	S5	P5
...

Figure 4.3 Cartesian Product Examples**4.3.5 Restrict**

Let relation A have attributes X and Y (and possibly others), and let Θ be an operator typically "=", ">", etc. Such that the condition $X \Theta Y$ is well defined and, given particular values for X and Y, evaluates to a truth value (true or false). Then the Θ -restriction of relation A on attributes X and Y (in that order).

S WHERE CITY='London'

Is a relation with the same heading as A and with body consisting of all tuples t of A such that the condition $X \Theta Y$ evaluates to true for that tuple t. Restriction permits only a single condition in the WHERE clause. By virtue of the closure property, however, it is possible to extend it unambiguously to a form in which the expression in the WHERE clause consist of an arbitrary boolean combination of such conditions, thanks to the following equivalences :

$A \text{ WHERE } C1 \text{ AND } C2 \equiv (A \text{ WHERE } c1) \text{ INTERSECT } (A \text{ WHERE } C2)$

$A \text{ WHERE } C1 \text{ OR } C2 \equiv (A \text{ WHERE } c1) \text{ UNION } (A \text{ WHERE } C2)$

$A \text{ WHERE NOT } C \equiv A \text{ MINUS } (A \text{ WHERE } C)$

Henceforth, therefore, it will be assumed that the <boolean expression> in the WHERE clause of restriction consists of such an arbitrary combination of conditions (with parentheses if necessary in order to indicate a desired order of evaluation). Where each condition in turn involves only attributes of the pertinent relation or selector invocations or both. Note that such a <boolean expression> can be established as true or false for a given tuple by examining just that tuple in isolation. Such a <boolean expression> is said to be a restriction condition. The restriction operator effectively yields a "horizontal" subset of a given relation that is, that subset of the tuples of the given relation for which some specified restriction condition is satisfied. Some examples are given in figure 4.4 [5].

$\sigma.S\#, sname, status, city(A) \text{ where } city='London'$

S WHERE CITY='London'	S#	SNAME	STATUS	CITY
	S1	Smith	20	London
	S4	Clark	20	London

P WHERE WEIGHT < WEIGHT(14.0)	P#	PNAME	COLOR	WEIGHT	CITY
	P1	Nut	Red	12.0	London
	P5	Cam	Blue	12.0	Paris

SP WHERE S#=S#('S6') OR P#=P#('P7')	S#	P#	QTY

Figure 4.4 Restriction Examples

4.3.6 Project

Let relation A have attributes X,Y,.....Z (and possibly others). Then the projection of relation A on X,Y,.....Z.

$A\{X,Y,\dots,Z\}$ is a relation with:

- A heading derived from the heading of A by removing all attributes not mentioned in the set $\{X,Y,\dots,Z\}$ and
- A body consisting of all tuples $\{X:x,Y:y,\dots,Z:z\}$ such that a tuple appears in A with X value x , Y value y , ..., and Z value z

The projection operator thus effectively yields a "vertical" subset of a given relation- that is, that subset obtained by removing all attributes not mentioned in the specified commalist of attribute names and then eliminating duplicate (sub)tuples from what is left. A projection of the form $A\{\}$ -i.e., one in which the attributes name commalist is empty- is legal. it represents a nullary projection. Some examples of projection are given in figure 4.5. Notice in the first examples (the projection of suppliers over CITY) that, although relvar S currently contains five tuples and hence five cities, there are only three cities in the result- duplicates (duplicate tuples, that is) are eliminated. Analogous remarks apply to the other examples also, of course. For example: $\pi_{\text{color,city}}(P)$

S{ CITY }	CITY	
	London	
	Paris	
	Athens	

P{ COLOR,CITY }	COLOR	CITY
	Red	London
	Green	Paris
	Blue	Rome
	Blue	Paris

(S WHERE CITY='Paris') { S# }	S#
	S2
	S3

Figure 4.5 Projection Examples

In practice, it is often convenient to be able to specify, not the attributes over which the projection is to be taken, but rather the ones that are to be "projected away" (i.e., removed). Instead of saying "project relation P over the P#, PNAME, COLOR, and CITY attributes, "for example, it might be say" project the WEIGHT attribute away from relation P," as here:

$P\{\text{ALL BUT WEIGHT}\}$

4.3.7 Join

Join comes in several different varieties. Easily the most important, however, is the so-called natural join specifically. here then is the definition (it is a little abstract, but you should already be familiar with natural join at an intuitive level. Let relations A and B have headings $\{X_1, X_2, \dots, X_m, Y_1, Y_2, \dots, Y_n\}$ and $\{Y_1, Y_2, \dots, Y_n, Z_1, Z_2, \dots, Z_p\}$ respectively; i.e., the Y attributes Y_1, Y_2, \dots, Y_n (only) are common to the tow relations. The X attributes X_1, X_2, \dots, X_m are the other attribute of A, and the Z attributes Z_1, Z_2, \dots, Z_p are the other attributes of B. Now consider $\{X_1, X_2, \dots, X_m\}$, $\{Y_1, Y_2, \dots, Y_n\}$, and $\{Z_1, Z_2, \dots, Z_p\}$ as three composite attributes X, Y, and Z, respectively. Then the natural join of A and B is the following.

$A \text{ JOIN } B$

Is a relation with heading $\{X, Y, Z\}$ and body consisting of the set of all tuples $\{X:x, Y:y, Z:z\}$ such that a tuple appears in A with X value x and Y value y and a tuple appears in B with Y value y and Z value z. An example of a natural join (the natural join $S \text{ JOIN } P$, over the common attribute CITY) is given in figure 4.6.

It is still worth stating explicitly that joins are not always between a foreign key and a matching primary key, even though such joins are a very common and important special case.



S#	SNAME	STATUS	CITY	P#	PNAME	COLOR	WEIGHT
S1	Smith	20	London	P1	Nut	Red	12.0
S1	Smith	20	London	P4	Screw	Red	14.0
S1	Smith	20	London	P6	Cog	Red	19.0
S2	Jones	10	Paris	P2	Bolt	Green	17.0
S2	Jones	10	Paris	P5	Cam	Blue	12.0
S3	Blake	30	Paris	P2	Bolt	Green	17.0
S3	Blake	30	Paris	P5	Cam	Blue	12.0
S4	Clark	20	London	P1	Nut	Red	12.0
S4	Clark	20	London	P4	Screw	Red	14.0
S4	Clark	20	London	P6	Cog	Red	19.0

Figure 4.6 The Natural Join S JOIN P

Now, in the view point of the Θ -join operation. This operation is intended for those occasions (comparatively rare, but by no means unknown) where it need to join two relations to-together on the basis of some comparison operator other than equality. Let relations A and B satisfy the requirements for Cartesian product(i.e., they have no attribute names in common);let A have an attribute X and let B have an attribute Y,and let X,Y, and Θ satisfy the requirements for restriction. Then the Θ of relation A on attribute X with relation B on attribute Y is defined to be the result of evaluating the expression $(A \text{ TIMES } B) \text{ WHERE } X \Theta Y$. In other words, it is a relation with the same heading as the Cartesian product of A and B, and with a body consisting of the set of all tuples t such that t appears in that Cartesian product and the condition " $X \Theta Y$ " evaluates to true for that tuple t. By way of example, suppose we wish to compute the greater-than join of relation S on CITY with relation p on CITY (so Θ here is ">"; it can be assume that ">" makes sense for cities, and interpret it to mean simply "greater in alphabetic ordering"). An appropriate relational expression is as follows:

```
(( S RENAME CITY AS SCITY ) TIMES ( P RENAME CITY AS PCITY ) )
WHERE SCITY > PCITY
```

Note the attribute renaming in this example. (Of course, it would be sufficient to rename just one of the two CITY attributes; the only reason for renaming both is symmetry.) The result of the overall expression is shown in figure 4.7.

$$\sigma_{S\#, sname, status, scity(S) \bowtie \sigma_{S\#, pname, status, pcity(B)} \\ SCITY > PCITY$$

If Θ is "=", the Θ -join is called an equijoin. It follows from the definition that the result of an equijoin must include two attributes with the property that the values of those two attributes are equal in every tuple in the relation. If one of those two attributes is projected away and the other renamed appropriately (if necessary), the result is the natural join! For example, the expression representing the natural join of suppliers and parts (over cities) is as the following:

S JOIN P

Is equivalent to the following more complex expression:

((S TIMES(P RENAME CITY AS PCITY))
WHERE CITY=PCITY)
{ALL BUT PCITY}

S#	SNAME	STATUS	SCITY	P#	PNAME	COLOR	WEIGHT	PCITY
S2	Jones	10	Paris	P1	Nut	Red	12.0	London
S2	Jones	10	Paris	P4	Screw	Red	14.0	London
S2	Jones	10	Paris	P6	Cog	Red	19.0	London
S3	Blake	30	Paris	P1	Nut	Red	12.0	London
S3	Blake	30	Paris	P4	Screw	Red	14.0	London
S3	Blake	30	Paris	P6	Cog	Red	19.0	London

Figure 4.7 Greater-than Join of suppliers and parts on cities

4.3.8 Divide

Reference [7] define two distinct "divide" operators that small divide and the great Divide, respectively. $a \langle \text{divide} \rangle$ in which the $\langle \text{per} \rangle$ consists of just one $\langle \text{relational expression} \rangle$ s is a Small Divide, $a \langle \text{divide} \rangle$ in which it consists of a parenthesized commalist of two $\langle \text{relational expression} \rangle$ s is a Great Divide. The description that follows applied to the small Divide only, and only to a particular limited form of the Small Divide at that. See reference [7] for a discussion of the Great Divide and for further details regarding the Small Divide as well.

It should be said that the version of the Small Divide as discussed here is not the same as original operator-in fact, it is an improved version that overcomes certain difficulties that arose with that original operator in connection with empty relations.

Here then is the definition. Let relations A and B have headings

$\{X_1, X_2, \dots, X_m\}$ and $\{Y_1, Y_2, \dots, Y_n\}$

respectively (i.e., A and B have disjoint headings), and let relation C have heading

$\{X_1, X_2, \dots, X_m, Y_1, Y_2, \dots, Y_n\}$

(i.e., C has a heading that is the union of the headings of A and B). Let us now regard $\{X_1, X_2, \dots, X_m\}$ and $\{Y_1, Y_2, \dots, Y_n\}$ as composite attributes X and Y, respectively. Then the division of A by B per C (where A is the dividend, B is the divisor, and C is the "mediator") as the following.

A DIVIDE BY B PER C

Is a relation with heading $\{X\}$ and body consisting of the tuples $\{X:x\}$ such that a tuple $\{X:x, Y:y\}$ appears in C for all tuples $\{Y:y\}$ appearing in B. In other words, the result consists of those X values from A whose corresponding Y values in C include all Y values from B, loosely speaking. Figure 4.8 shows some simple examples of division. The dividend (DEND) in each case is the projection of the current value of relvar S over S#; the mediator (MED) in each case is the projection of the current value of relvar SP over S# and P#; the three divisors

(DOR) are as indicated in the figure. Notice the last example in particular, in which the divisor is a relation containing part numbers for all currently known parts; the result (obviously) shows supplier numbers for suppliers who supply all of those parts. As this example suggests, the DIVIDEBY operator is intended for queries of this same general nature; in fact, whenever the natural language version of the query contains the word "all" ("Get suppliers who supply all parts"), it is a strong possibility that division will be involved. However, it is worth pointing out that such queries are often more readily expressed in terms of the relational comparisons anyway [5].

DEND

S#
S1
S2
S3
S4
S5

MED

S#	P#
S1	P1
S1	P2
S1	P3
S1	P4
S1	P5
S1	P6
...	...

...	...
S2	P1
S2	P2
S3	P2
S4	P2
S4	P4
S4	P5
...	...

DOR

P#
P1

P#
P2
P4

DOR

P#
P1
P2
P3
P4
P5
P6

DEND DIVIDEBY DOR PER MED

S#
S1
S2

S#
S1
S2

S#
S1

Figure 4.8 Division Examples

4.4 Associativity and Commutativity

It is easy to verify that UNION is associative, that is, if A, B, and C are arbitrary relational expressions yielding relations of the same type, then the expressions

$(A \text{ UNION } B) \text{ UNION } C$ and $A \text{ UNION } (B \text{ UNION } C)$

Are logically equivalent. For convenience, therefore, it could be a sequence of UNIONS to be written without any embedded parentheses; i.e., each of the foregoing expressions can be unambiguously simplified to just

$A \text{ UNION } B \text{ UNION } C$

Analogous remarks apply to INTERSECT, TIMES, and JOIN (but not to MINUS).

It should be mentioned that UNION, INTERSECT, TIMES and JOIN (but not MINUS) are commutative – that is, the expressions as well.

$A \text{ UNION } B$ and $B \text{ UNION } A$

Are also logically equivalent, and similarly for INTERSECT, TIMES and JOIN. Finally, We remark that if A and B have no attribute names in common, then $A \text{ JOIN } B$ is equivalent to $A \text{ TIMES } B$, i.e., natural join degenerates to Cartesian product in this case [5, 7].

4.5 Summary

There are eight operators that take relations as their operands. These operators are: Restrict, Project, Product, Union, Intersection, Difference, Join, and Divide.

CHAPTER FIVE

OBJECT RELATIONAL SQL

5.1 Overview

Structured Query Language (SQL) was introduced by IBM as the language to interface with its prototype object relational database management system. The first commercially available SQL object relational database management system was introduced in 1979 by Oracle Corporation. Today, SQL has become an industry standard, and Oracle Corporation clearly leads the world in relational database management system technology.

Because SQL is a non-procedural language, sets of records can be manipulated instead of one record at a time. The syntax is free-flowing, enabling you to concentrate on the data presentation. Oracle has two optimizers (cost- and rule-based) that will parse the syntax and format it into an efficient statement before the database engine receives it for processing. The database administrator (DBA) determines which optimizer is in effect for each database instance.

5.2 The Object Definition Language (ODL)

The Object Definition Language (ODL) is a specification language for defining the specifications of object types for ODMG-compliant systems. Its main objective is to facilitate portability of schemas between compliant systems while helping to provide interoperability between OODBMSs. ODL is equivalent to the data definition language(DDL) of traditional DBMSs. It defines the attributes and relationships of types and specifies the signature of the operations. It does not address the implementation of signatures. The syntax of ODL extends the interface Definition Language(IDL) of the common Object Request Broker Architecture(CORBA). The ODMG hope that the ODL will be the basis for integrating schemas from multiple sources and applications[8].

DDL (Data Definition Language). These SQL statements define the structure of a database, including rows, columns, tables, and database specifics such as file locations. DDL SQL statements are more part of the DBMS and have large differences between the SQL variations. CREATE; ALTER, DROP are called DDL COMMANDS [16].

- CREATE - to create objects in the database.
- ALTER - alters the structure of the database.
- DROP - delete objects from the database.

5.2.1 Tables

1) Create table statement

Create tables to store data by executing the SQL CREATE TABLE statement. This statement is one of the data definition language (DDL) statements, that are covered in subsequent lessons. DDL statements are a subset of SQL statements used to create, modify, or remove Oracle9i database structures. These statements have an immediate effect on the database, and they also record information in the data dictionary.

To create a table, a user must have the CREATE TABLE privilege and a storage area in which to create objects. The database administrator uses data control language (DCL) statements, which are covered in a later lesson, to grant privileges to users [16].

Syntax:

```
Create Table [Schema.] table  
            (Column Datatype [Default Expr][, .....]);
```

In the syntax:

Schema:	is the same as the owner's name.
table:	is the name of the table.
Default Expr:	specifies a default value if a value is omitted.
Column:	is the name of the column.
Datatype:	is the column's data type and length.

2) Alter table statement

It is possible to add, modify, and drop columns in a table by using the ALTER TABLE statement.

Syntax:

Alter Table table

```
ADD (    Column Datatype [Default Expr] [,
        Column Datatype].....);
```

Alter Table table

```
MODIFY( Column Datatype [Default Expr] [,
        Column Datatype].....);
```

Drop Table table

```
DROP    (Column);
```

In the syntax:

ADD MODIFY DROP:	is the type of modification
------------------	-----------------------------

3) Drop table statement

The DROP TABLE statement removes the definition of an Oracle table. When the dropping of table occurs the database loses all the data in the table, and any views remain but are invalid. Only the creator of the table or a user with the DROP ANY TABLE privilege can remove a table.

Syntax:

Drop Table table

4) Truncate table statement

Another DDL statement is the TRUNCATE TABLE statement, which is used to remove all rows from a table and to release the storage space used by that table. When using the TRUNCATE TABLE statement, it is not possible to roll back row removal.

Syntax:

Truncate Table table;

It must be the owner of the table or have DELETE TABLE system privileges to truncate a table. The DELETE statement can also remove all rows from a table, but it does not release storage space. The TRUNCATE command is faster. Removing rows with the TRUNCATE statement is faster than removing them with the DELETE statement for the following reasons:

- The TRUNCATE statement is a data definition language (DDL) statement and generates no rollback information.

- Truncating a table does not fire the delete triggers of the table.
- If the table is the parent of a referential integrity constraint, you cannot truncate the table.

5.2.2 Views

It is possible to present logical subsets or combinations of data by creating views of tables. A view is a logical table based on a table or another view. A view contains no data of its own but is like a window through which data from tables can be viewed or changed. The tables on which a view is based are called base tables. The view is stored as a SELECT statement in the data dictionary, also it used to restrict data access, and to make complex queries easy [9].

Syntax:

```
Create      [Or Replace] View view
            [(Alias[, Alias]...)]
            As Subquery ;
```

In the syntax:

Or Replace:	re-creates the view if it already exists
view:	is the name of the view
Alias:	specifies names for the expressions selected by the view's query(The number of aliases must match the number of expressions selected by the view.)
Subquery:	is a complete SELECT statement (You can use aliases for the columns in the SELECT list.)

Alter view and drop view are similar to alter table and drop table

5.2.3 Types

Create types is executing by using the SQL CREATE TYPE statements. These statements are object definition language (ODL) statements. To create a type, a user must have the privileges and a storage area in which to create objects. The database administrator uses data control language (DCL) statements, to grant privileges to users. An object type has attributes of various types, analogous to columns of a table. After making the SQL statement "create type" it can be used these new defined type as an attributes type.

Syntax:

```
Create Type      [Schema.] type As Object
                  (Column Datatype [Default Expr][, .....]);
```

In the syntax:

Schema:	is the same as the owner's name.
type:	is the name of the type.
Default Expr:	specifies a default value if a value is omitted in the INSERT statement.
Column:	is the name of the column.
Datatype:	is the column's data type and length

Alter type and drop type are similar to alter table and drop table.

5.2.4 Procedures

CREATE PROCEDURE statement can be used to create a new procedure, which may declare a list of parameters and must define the action to be performed by the standard PL/SQL block. The CREATE clause enables you to create stand-alone procedures, which are stored in an oracle database.

- PL/SQL blocks start with either BEGIN or the declaration of local variables and end with either END or END PROCEDURE_NAME. You cannot reference host or bind variables in the PL/SQL block of a stored procedure.
- The REPLACE option indicates that if the procedure exists, it will be dropped and replaced with the new version created by the statement.
- You can not restrict the size of the data type in the parameter .

Syntax:

```
Create      [Or Replace] Procedure Procedure_Name [(
            Parameter1[Mode1] Datatype1,
            Parameter2[Mode2] Datatype2 ,.....)]
Is|As
Pl/Sql Block;
```

In the syntax:

Procedure_Name:	Name of the procedure
Parameter:	Name of a PL/SQL variable whose value is passed to or populated by the calling environment, or both, depending on the mode begin used.
Mode:	Type of argument: IN(default), OUT, and IN OUT.
Datatype:	Data type of the argument can be any SQL/PLSQL data type. Can be of %TYPE, %ROWTYPE, or any scalar or composite data type.
Pl/Sql Block:	procedural body that defines the action performed by the procedure.

5.2.5 Functions

A function returns a value. You create new functions with the CREATE FUNCTION statement, which may declare a list of parameters, must return one value, and must define the actions to be performed by the standard PL/SQL block.

- The REPLACE option indicates that if the function exist, it will be dropped and replaced with the new version created by the statement.
- The RETURN data type must not include a size specification of local variables and end with either END or END function_name. There must be at least one RETURN (expression) statement. You cannot reference host or bind variables in the PL/SQL block of a stored function.

Syntax:

```
Create      [Or Replace] Function Function_Name [(
            Parameter1[Mode1] Datatype1,
            Parameter2[Mode2] Datatype2 ,.....)]
Return Datatype
Is|As
Pl/Sql Block;
```

In the syntax:

Function_Name:	Name of the function
Parameter:	Name of a PL/SQL variable whose value is passed into the function.
Mode:	The type of the parameter; only IN parameter should be declared
Return Datatype:	Data type of the RETURN value that must be output by the function

5.2.6 Triggers

The trigger body represents a complete PL/SQL block. It can create triggers for these events on DATABASE or SCHEMA. Also it can be specified BEFORE or AFTER the timing of the trigger.

Syntax:

Create	[Or Replace] Trigger Trigger_Name
Timing	
	[Ddl_Event1 [Or Ddl_Event2 Or]]
On {Database Schema}	
Trigger_Body	

In the syntax:

Ddl_Event:	Possible Values
Create:	Causes the oracle server to fire the trigger whenever a CREATE statement adds a new database object to the dictionary.
Alter:	Causes the oracle server to fire the trigger whenever a ALTER statement modifies a database object in the data dictionary.
Drop:	Causes the oracle server to fire the trigger whenever a DROP statement removes a database object in the data dictionary.

DDL triggers fire only if the object begins created is a cluster, function, index, package, procedure, role, sequence, table, tablespace, trigger, type, view, or user [9].

5.3 The Object Query Language (OQL)

The Object Query Language (OQL) provide declarative access to the object database using an SQL-like syntax. It does not provide explicit update operators, but leaves this to the operations defined on object types. As with SQL, OQL can be used as a standalone language and as a language embedded in another language, for which an ODMG binding is defined. The currently supported languages are smalltalk C++, and Java, OQL can also invoke operations programmed in these languages. An OQL query is a function that delivers an object whose type may be inferred from the operator contributing to the query expression. Before defining an OQL query, it can be understand the composition of expressions[8].

DML (Data Manipulation Language). These SQL statements are used to retrieve and manipulate data. This category encompasses the most fundamental commands including DELETE, INSERT, SELECT, and UPDATE. DML SQL statements have only minor differences between SQL variations. DML SQL commands include the following:

- INSERT to add a row.
- UPDATE to change data in specified columns.
- DELETE to remove rows.
- SELECT to retrieve row.

DML commands can't be rollback when a DDL command is executed immediately after a DML. DDL after DML means "auto commit". The changes will return on disk not on the buffer. If the changes return on the buffer it is possible to rollback not from the disk [9].

5.3.1 Select statement

A SELECT statement retrieves information from the database. Using a SELECT statement, you can do the following:

- Projection: It could be used as the projection capability in SQL to choose the columns in a table that returns by the query. Also it can be chosen as few or as many columns of the table as you require.
- Selection: It could be used as the selection capability in SQL to choose the rows in a table that returns by the query. Also it can be used as various criteria to restrict the rows that have been seen.
- Joining: it could be used as the join capability in SQL to bring together data that is stored in different tables by creating a link between them.

In its simplest form, a SELECT statement must include the following:

- A SELECT clause, which specifies the columns to be displayed
- A FROM clause, which specifies the table containing the columns listed in the SELECT clause

Syntax:

Select	* {[Distinct] Column Expression [Alias],...}
From	table;

In the syntax:

Select:	is a list of one or more columns
*:	selects all columns
Distinct:	suppresses duplicates
Column Expression:	selects the named column or the expression
Alias:	gives selected columns different headings
From:	table specifies the table containing the columns

5.3.2 Insert statement

New rows can be added to a table by issuing the INSERT statement.

Syntax:

```
Insert Into table [(Column [, Column...])]
Values (Value [, Value...]);
```

In the syntax:

table:	is the name of the table
Column:	is the name of the column in the table to populate
Value:	is the corresponding value for the column

5.3.3 Update statement

It could be modified as the existing rows by using the UPDATE statement.

Syntax:

```
Update table
Set Column = Value [, Column = Value, ...]
[Where Condition];
```

In the syntax:

table:	is the name of the table
Column:	is the name of the column in the table to populate
Value:	is the corresponding value or subquery for the column
Condition:	identifies the rows to be updated and is composed of column names expressions, constants, subqueries, and comparison operators

5.3.4 Delete statement

It can be remove existing rows from a table by using the DELETE statement [9].

Syntax:

Delete [From] table [Where Condition];

5.4 Summary

The Object Definition Language (ODL) is equivalent to the data definition language (DDL) of traditional DBMSs. It defines the attributes and relationships of types and specifies the signature of the operations. DDL (Data Definition Language). These SQL statements define the structure of a database, including rows, columns, tables, and database specifics such as file locations. DDL SQL statements are more part of the DBMS and have large differences between the SQL variations.

The Object Query Language (OQL) can be used as a standalone language and as a language embedded in another language, for which an ODMG binding is defined. The currently supported languages are smalltalk C++, and Java, OQL can also invoke operations programmed in these languages. DML (Data Manipulation Language). These SQL statements are used to retrieve and manipulate data. This category encompasses the most fundamental commands including DELETE, INSERT, SELECT, and UPDATE.

CHAPTER SIX

DATABASE DESIGN WITH OBJECT DATA MODELING

6.1 Overview

This chapter applies the syntaxes shown in chapter (5) in the trading company application, which contains employees' and customers' data. At the same time there is a comparison to show the main differences between using object modeling approaches, and without using it.

6.2 Object Relational DB Application

The applications have focused on accessing and modifying corporate data that is stored in tables composed of native SQL data types such as INTEGER, NUMBER, DATE, and CHAR. Oracle is not supported only for these native types, but also for new 'object' data types [5].

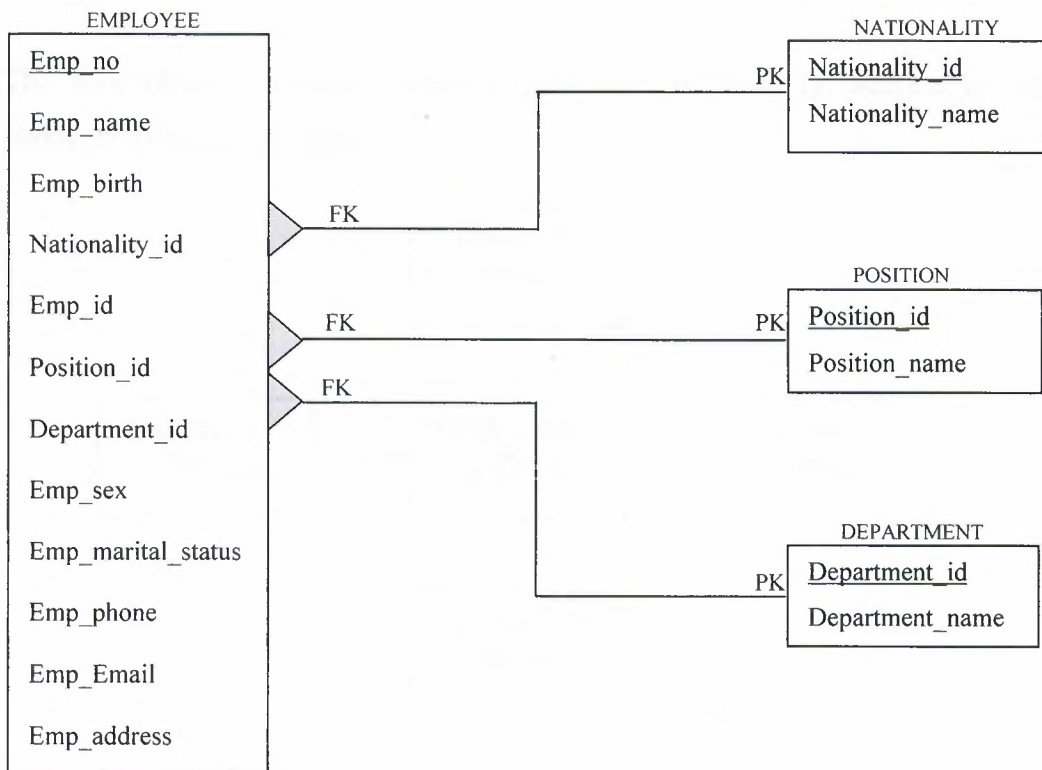


Figure 6.1 The Employee Entity Relationship Diagram

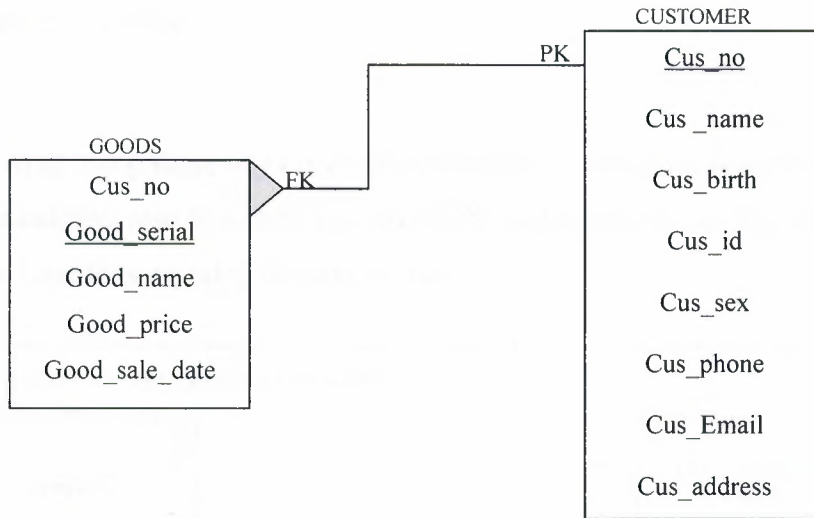


Figure 6.2 The Customer Entity Relationship Diagram

Figure 6.1 and figure 6.2 explain the entity's relationship diagram (ER diagram) of trading company application.

The new objects in this application are name_ty, birth_ty, address_ty, and phone_ty as shown in figure 6.3 .

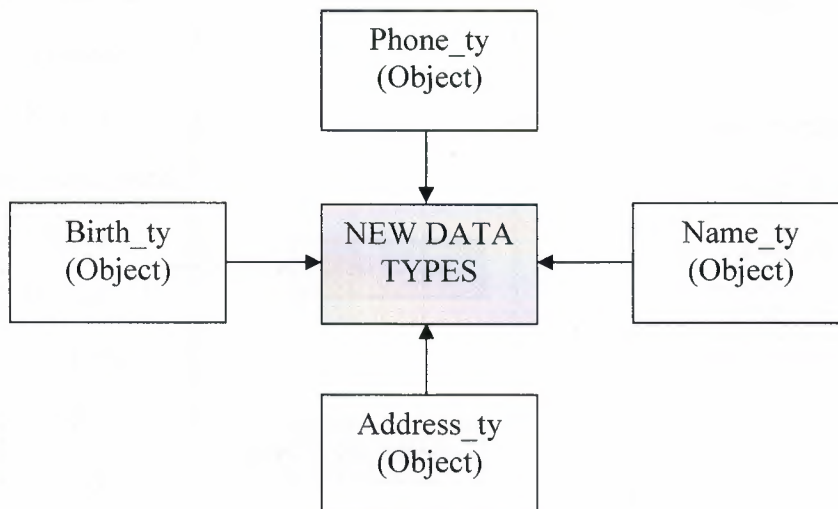


Figure 6.3 New DataTypes(Objects) that used in this Application

As explained in chapter 5 an object type has attributes of various types, analogous to columns of a table.

The goal of using these objects is structure tables, to be more readable, usable, and understandable, also to reduce the size of the code that reduces the time for design and the time for executing the application.

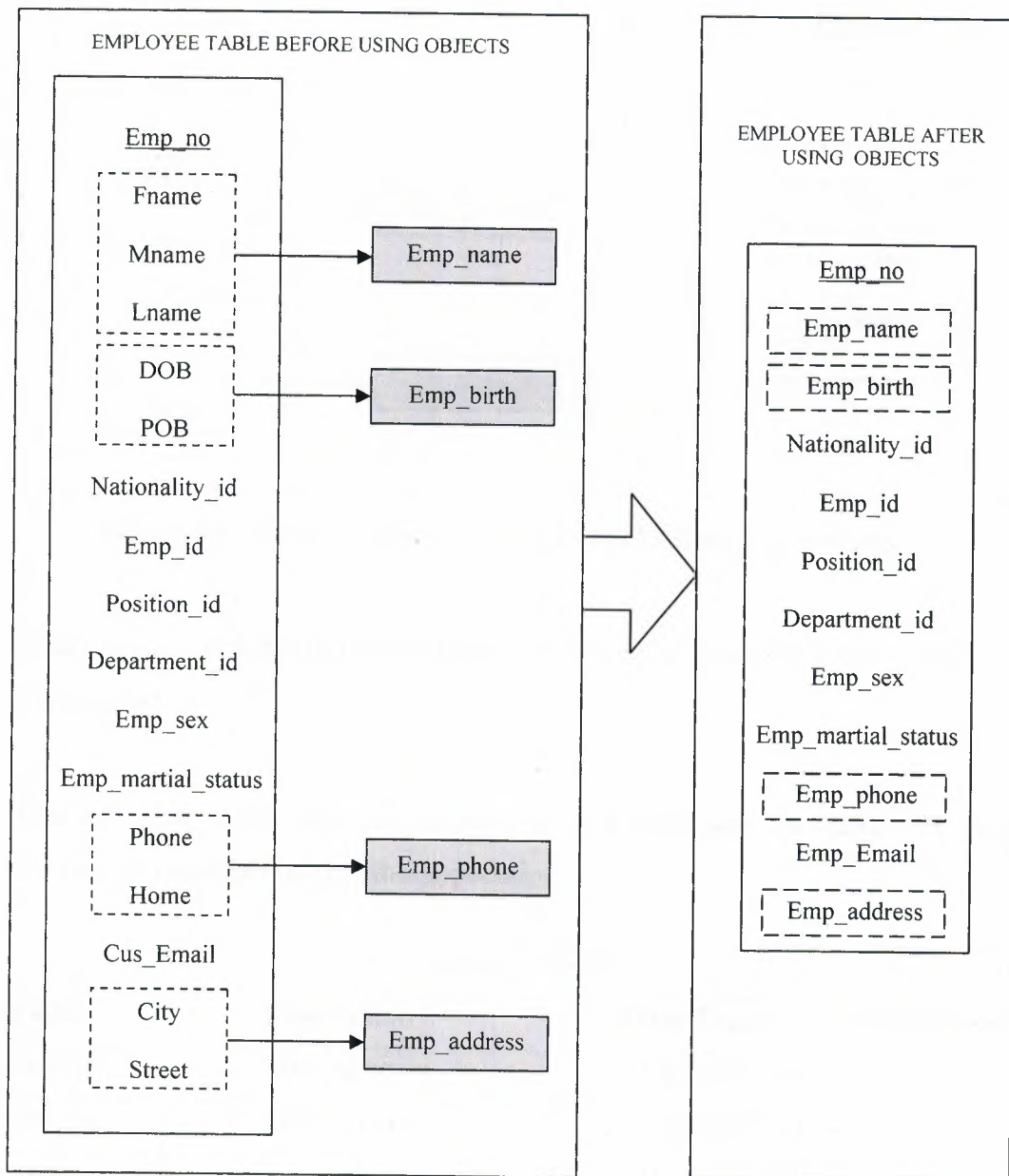


Figure 6.4 The Employee Table Before And After Using Objects

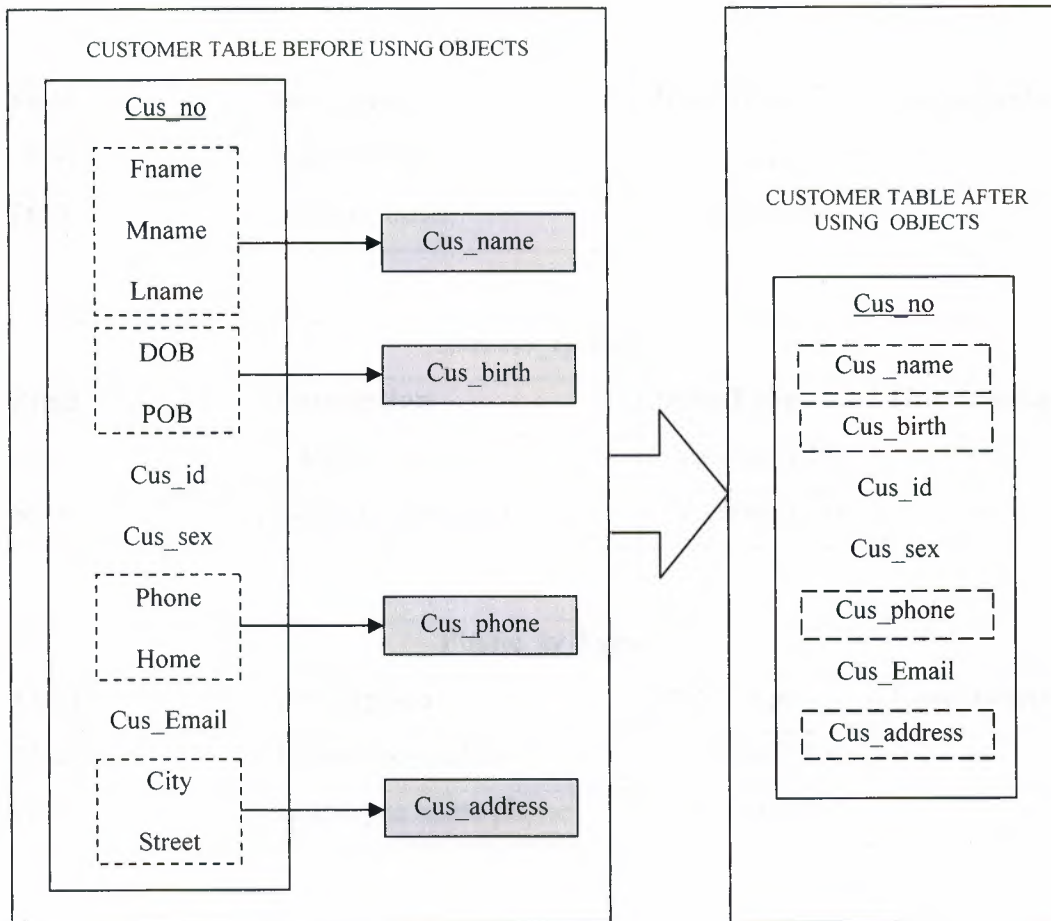


Figure 6.5 The Customer Table Before And After Using Objects

Figure 6.4 and 6.5 shows the differences between the same table before and after using objects.

The following tables explain description, data type, and constraints for each attribute in types that used in this application.

Name_ty Type			
Field	Description	Data Type	Constraints
Fname	First name	Varchar2(25)	--
Mname	Middle name	Varchar2(25)	--
Lname	Last name	Varchar2(25)	--

Birth_ty Type			
Field	Description	Data Type	Constraints
DOB	Data of birth	date	--
POB	Place of birth	Varchar2(25)	--

Address_ty Type			
Field	Description	Data Type	Constraints
City	Name of city	Varchar2(25)	--
Street	Name of the street	Varchar2(25)	--

Phone_ty Type			
Field	Description	Data Type	Constraints
Mobile	Employee mobile	Varchar2(13)	--
Home	Employee home phone	Varchar2(13)	--

This application consists of the tables: nationality, position, department, goods, employee, and customer. Also the following tables explain description, data type, and constraints for each attribute in these tables:

Nationality Table			
Field	Description	Data Type	Constraints
Nationality_id	Nationality id	Number(2)	PK
Nationality name	Nationality name	Varchar2(25)	

Position Table			
Field	Description	Data Type	Constraints
Position_id	Position id	Number(2)	PK
Position_name	Position name	Varchar2(25)	

Department Table			
Field	Description	Data Type	Constraints
Department_id	Department id	Number(2)	PK
Department_name	Department name	Varchar2(25)	

Goods Table			
Field	Description	Data Type	Constraints
Cus_no	Customer number	Number(9)	FK
Good_serial	Serial number	Number(15)	PK
Good_name	Good name	Varchar2(25)	
Good_price	Good price	Number(8)	
Good_sale_date	Date of sale	Date	

Employee Table			
Field	Description	Data Type	Constraints
Emp_no	Employee number	Number(9)	PK
Emp_name	Employee name	Name_ty	
Emp_birth	Employee DOB & POB	Birth_ty	
Nationality_id	Nationality id	Number(2)	FK
Emp_id	Employee id	Number(9)	
Position_id	Position id	Number(2)	FK
Department_id	Employee Department	Number(2)	FK
Emp_sex	Employee sex	Number(1)	
Emp_marital_status	Employee marital status	Number(1)	
Emp_phone	Employee phones	Phone_ty	
Emp_Email	Employee Email address	Varchar2(25)	
Emp_address	Employee address	Address_ty	

Customer Table			
Field	Description	Data Type	Constraints
Cus_no	Customer number	Number(9)	PK
Cus_name	Customer name	Name_ty	
Cus_birth	Customer DOB & POB	Birth_ty	
Nationality_id	Nationality id	Number(2)	FK
Cus_id	Customer id	Number(9)	
Cus_sex	Customer sex	Number(1)	
Cus_phone	Customer phones	Phone_ty	
Cus_Email	Customer Email address	Varchar2(25)	
Cus_address	Customer address	Address_ty	

The following figure shows the new data type that is used in employee and customer table.

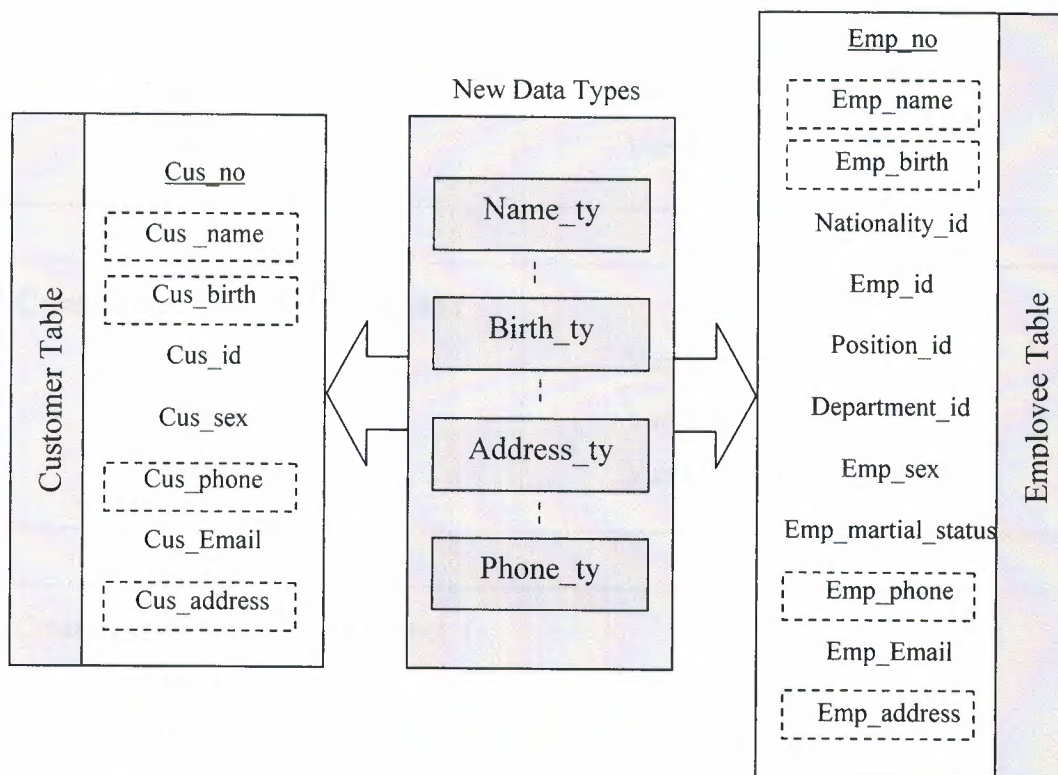


Figure 6.6 New DataType used in Employee & Customer Tables

6.3 The Object Definition Language

Now it is possible to apply the Object Definition Language (ODL) in this Application.

6.3.1 Create objects in the database.

The following SQL statements create Address_ty, Birth_ty, Name_ty, and Phone_ty as objects or as new defined type. After that, it is possible to use these new defined types as an attribute datatype.

```
Create Type Address_Ty As Object (
    City                Varchar2(25),
    Street              Varchar2(25));
```

```
Create Type Birth_Ty As Object (
    DOB                Date,
    POB                Varchar2(25));
```

```
Create Type Name_Ty As Object (
    Fname              Varchar2(25),
    Mname              Varchar2(25),
    Lname              Varchar2(25));
```

```
Create Type Phone_Ty As Object (
    Mobile              Varchar2(13),
    Home                Varchar2(13));
```


6.3.2 Create tables in the database.

The application consists of nationality, position, department, and goods tables that created by the normal way (without new defined type).

```
Create Table Nationality (
    Nationality_Id          Number(2)Primary Key,
    Nationality_Name        Varchar2(25));
```

```
Create Table Position      (
    Position_Id             Number(2)Primary Key,
    Position_Name           Varchar2(25));
```

```
Create Table Department (
    Department_Id           Number(2)Primary Key,
    Department_Name         Varchar2(25));
```

```
Create Table Goods (
    Cus_No                  Number(9),
    Good_Serial             Number(15) Primary Key,
    Good_Name               Varchar2(25),
    Good_Price              Number(8),
    Good_Sale_Date          Date,
    Constraint Good_Cus_Fk Foreign Key(Cus_No)
    References Customer(Cus_No));
```

Also, the application consists of employee and customer tables that involve new defined types as an attributes datatypes.

As shown in creation of employee table, using of objects address_ty, birth_ty, name_ty, and phone_ty reduce the number of the SQL statements, structure tables, and make it more readable and understandable.

Create Table Employee (

Emp_No	Number(9)Primary Key,
Emp_Name	Name_Ty,
Emp_Birth	Birth_Ty,
Emp_Id	Number(9),
Emp_Sex	Number(1),
Emp_Marital_Status	Number(1),
Emp_Phone	Phone_Ty,
Emp_Email	Varchar2(25),
Emp_Address	Address_Ty,
Nationality_Id	Number(2),
Position_Id	Number(2),
Department_Id	Number(2),
Constraint Emp_Nat_Fk Foreign Key(Nationality_Id)	
References Nationality(Nationality_Id),	
Constraint Emp_Pos_Fk Foreign Key(Position_Id)	
References Position(Position_Id),	
Constraint Emp_Dep_Fk Foreign Key(Department_Id)	
References Department(Department_Id));	

Now if employee table does not contain new defined types as attributes datatypes (objects), the SQL Statements will be unreadable, ununderstandable, and the number of it will be more, as the following:

Create Table Employee \

Emp_No	Number(9)Primary Key,
Fname	Varchar2(25),
Mname	Varchar2(25),
Lname	Varchar2(25),
DOB	Date,
POB	Varchar2(25),
Emp_Id	Number(9),
Emp_Sex	Number(1),
Emp_Marital_Status	Number(1),
Mobile	Varchar2(13),
Home	Varchar2(13),
Emp_Email	Varchar2(25),
City	Varchar2(25),
Street	Varchar2(25),
Nationality_Id	Number(2),
Position_Id	Number(2),
Department_Id	Number(2),
Constraint Emp_Nat_Fk Foreign Key(Nationality_Id)	
References Nationality(Nationality_Id),	
Constraint Emp_Pos_Fk Foreign Key(Position_Id)	
References Position(Position_Id),	
Constraint Emp_Dep_Fk Foreign Key(Department_Id)	
References Department(Department_Id));	

Also as seen in creation of customer table, use of objects address_ty, birth_ty, name_ty, and phone_ty reduce the number of SQL statements, structure tables, and make it more readable and understadable.

Create Table Customer (

Cus_No	Number(9)Primary Key,
Cus_Name	Name_Ty,
Cus_Birth	Birth_Ty,
Cus_Id	Number(9),
Cus_Sex	Number(1),
Cus_Phone	Phone_Ty,
Cus_Email	Varchar2(25),
Cus_Address	Address_Ty);

If customer table does not contain new defined types as attributes datatypes (objects), the SQL statements will be unreadable, ununderstandable, the number of it will be more, and the run time will be more as the following:

Create Table Customer (

Cus_No	Number(9)Primary Key,
Fname	Varchar2(25),
Mname	Varchar2(25),
Lname	Varchar2(25),
DOB	Date,
POB	Varchar2(25),
Cus_Id	Number(9),
Cus_Sex	Number(1),
Mobile	Varchar2(13),
Home	Varchar2(13),
Cus_Email	Varchar2(25),
City	Varchar2(25),
Street	Varchar2(25));

6.3.3 Create views in the database.

It is possible to present logical subsets or combinations of data by creating views from tables. This application contains some views, the following view is one of it. As seen in creation of employee_vw view, using of objects name_ty, and birth_ty reduce the number of the SQL statements, structure view, and make it more readable and understandable.

Create Or Replace View Employee_Vw As

```

Select      E.Emp_No,
            E.Emp_Name,
            E.Emp_Birth,
            N.Nationality_Name,
            P.Position_Name,
            D.Department_Name
From        Employee E , Nationality N , Position P , Department D
Where       N.Nationality_Id = E.Nationality_Id      And
            P.Position_Id = E.Position_Id            And
            D.Department_Id = E.Department_Id;
```

When using the normal way (the employee table without new defined type) the view will be more unreadable, ununderstandable, and the number for its SQL statements will be more, as in the following example:

Create Or Replace View Employee_Vw As

```

Select      E.Emp_No,
            E.Fname,
            E.Mname,
            E.Lname,
```

E.DOB

E.POB

N.Nationality_Name,

P.Position_Name,

D.Department_Name

From Employee E , Nationality N , Position P , Department D

Where N.Nationality_Id = E.Nationality_Id And

P.Position_Id = E.Position_Id And

D.Department_Id = E.Department_Id;

6.3.4 Create procedures in the database.

Procedures are defined by a routine name and the parameters to be passed in and out of the routine. The parameters may be native SQL data types like employee_no or new defined type like employee_name and employee_birth. As seen in reation of new_employee procedure, using of objects name_ty, and birth_ty reduce the size of the code, structure procedure, and make it more readable and understandable.

Create Or Replace Procedure New_Employee

```
( Employee_No      In   Number,
  Employee_Name     In   Name_Ty,
  Employee_Birth    In   Birth_Ty )
```

As

Begin

Insert Into Employee(Emp_No,Emp_Name,Emp_Birth)

Values(Employee_No,Employee_Name,Employee_Birth);

End;

To execute the last procedure it must be written in a SQL statement like the following.

```
Execute New_Employee( 20060009,  
                      Name_Ty('Ahmed','R','Dawoud'),  
                      Birth_Ty('11-Nov-1997','Palestine'));
```

When using the normal way (the employee table without new defined type) the procedure will be more unreadable, ununderstandable, and the size for its code will be more, as in the following example:

```
Create Or Replace Procedure New_Employee  
( Employee_No      In   Number,  
  Employee_Fname    In   Varchar2,  
  Employee_Mname     In   Varchar2,  
  Employee_Lname     In   Varchar2,  
  Employee_Dob       In   Date,  
  Employee_Pob       In   Varchar2  )  
As  
Begin  
  Insert Into Employee(Emp_No,Fname,Mname, Lname, Dob, Pob)  
  Values( Employee_No,Employee_Fname,Employee_Mname,  
         Employee_Lname, Employee_Dob ,Employee_Pob);  
End;
```

To execute the last procedure it must be written in a SQL statement like the following.

```
Execute New_Employee( 20060009, 'Ahmed','R','Dawoud', '11-Nov-1997',  
                      'Palestine');
```

6.3.5 Create functions in the database.

A function may declare a list of parameters, and it must return one value. The following function uses the birth date as input, and it returns the age of employee as output. As seen in creation of age_employee_no function, use of object birth_ty structure function, and makes the SQL statements more readable and understandable.

```
Create Or Replace Function Age_Employee_No (X Number) Return Number
Is
    I      Date;
Begin
    Select  E.Emp_Birth.Dob Into I
    From    Employee E
    Where    Emp_No=X;
    Return  (Sysdate-I)/365;
End;
```

When using the normal way (the employee table without new defined type) the function will be more unreadable, and ununderstandable, as in the following example:

```
Create Or Replace Function Age_Employee_No (X Number) Return Number
Is
    I      Date;
Begin
    Select  Dob Into I
    From    Employee
    Where    Emp_No=X;
    Return  (Sysdate-I)/365;
End;
```

6.3.6 Create triggers in the database.

A trigger is a message or something that accomplishes the user's a tension.

```
Create Or Replace Trigger Emp_Trigger
Before Insert Or Update Or Delete On Employee
Begin
    If Deleting Then
        Raise_Application_Error(-20502,'Again');
    End If;
End;
```

As seen when the deletion process occurs this message will be appear.

6.4 Object Query Language

After finishing of ODL, it is possible to apply the Object Query Language (OQL) in application.

6.4.1 Insert a row in the table

To insert a row into the employee table there exists two cases. The first case is when there are some attributes with new defined data type, this makes the insert process more readable and understandable as in the following example.

```
Insert Into Employee( Emp_No, Emp_Name, Emp_Birth )
Values ( 20043315,
        Name_Ty('Anwar','M','Dawoud'),
        Birth_Ty('12-Nov-82','Palistine') );
```


The second case is when using the normal way (the employee table without new defined type) the insert statement will be more ambiguous, unreadable, and ununderstandable, as in the following example:

```
Insert Into Employee ( Emp_No, Fname, Mname, Lname, Dob, Pob )
Values ( 20043315, 'Anwar', 'M', 'Dawoud', '12-Nov-82', 'Palistine' );
```

6.4.2 Update a row in the table

To update a row into the employee table there are two cases. The first case is when there are some attribute with new defined data type, this makes the update process more readable and understandable as in the following example.

```
Update      Employee
Set         Emp_Name=Name_Ty('Ahmed','A','Dalo')
Where      Emp_No=20043815;
```

The second case is when using the normal way (the employee table without new defined type) the update statement will be more ambiguous, unreadable, and ununderstandable, as in the following example:

```
Update      Employee
Set         Fname='Ahmed', Mname='A', Lname='Dalo'
Where      Emp_No=20043815;
```

6.4.3 Delete a row in the table

To delete a row from the employee table there exist two cases. The first case is when there are some attribute with new defined data type, this makes the delete process more readable and understandable as in the following example.

Delete	From Employee
Where	Emp_Name=Name_Ty('Ahmed', 'R', 'Dawoud');

The second case is when using the normal way (the employee table without new defined type) the delete statement will be more ambiguous, unreadable, and ununderstandable, as in the following example:

Delete	From Employee
Where	Fname='Ahmed' And
	Mname='R' And
	Lname='Dawoud';

6.4.4 Select rows from the table

To select some rows from the employee table there exist two cases. The first case is when there are some attribute with new defined data type. This makes the select process more readable and understandable as in the following example.

Select	Emp_No, Emp_Name
From	Employee;

The second case is when using the normal way (the employee table without new defined type) the select statement will be more ambiguous, unreadable, and ununderstandable, as in the following example:

Select	E.Emp_No, E.Fname, E.Mname, E.Lname
From	Employee E;

In this application there are some graphical user interfaces. Figures 6.7 and 6.8 show graphical user interfaces for employee and customer tables.

Figure 6.7 Graphical user Interface for Employee Table

Good Serial	Good Name	Good Price	Date Off Sale
1200023511444	Keyboard	20	21/08/2004
2211215554141	Mouse	15	27/09/2005

Figure 6.8 Graphical user Interface for Customer Table

Throughout the last graphical user interfaces it could be generated as a new number of every employee and customer. After that it could be added some information about them such as name, birth of date, and place of birth.

The following report show employee number, name, phone, and email.

EMPLOYEE PHONES+EMAIL ADDRESS

Staff No.	Name	Mobile	Tel	Email Address
20060001	Anwar Dawoud	05330477111	2655779	anw_daw@hotmail.com
20060002	Ahmed Dale	05330498514	2675514	ahmed2000@yahoo.com

Figure 6.9 Report Showing Employee Name, Phone, and Email

Comparison of SQL statements for trading company database design has been listed for with objects and without objects respectively as following :

Number of SQL Statements	DDL Statements			DML Statements			
	Tables	Procedures	Functions	Select	Insert	Update	delete
With Objects	24	8	8	4	8	5	4
Without Objects	34	11	10	6	14	8	6
Improvement	30%	27%	20%	33%	43%	38%	34%

6.5 Summary

The main differences between using Object approaches and without have been declared using the application of trading company.

As seen in this chapter using the ORDBMS concepts make the application more usable, readable, and reduce the size of the code.

CONCLUSION

In the thesis the Object-oriented database technology is discussed as a symbiosis of object-oriented modeling and database technologies. The object-oriented data modeling provides the technique to encapsulate the attributes and the methods for processing the attributes.

It is shown how database storage mechanism incorporates object-oriented data and establishes the Object-Oriented Database Systems. Object-Oriented Database Systems are based on Object-Oriented Programming Language and Database storage mechanisms that depend on the data model established.

The Relational Database System (RDBS) as most popular one is discussed but with disadvantage not to use an object data modeling. The implementation of the object data modeling to RDBS significantly improves and simplifies the database design. This combination is named as Object Relational Database System (ORDBS) and allows a designer to use convenient structured query language (SQL) with object features.

The thesis analyzes declared the features of Object Structured Query Language (OSQL) that includes Object Data Definition Language and Object Query Language. This analysis with discussion of the advantages of OSQL was made by structuring the problem with object data modeling in Trading Company Database Design. The comparisons of the SQL statements made with and without object features showed that the implementing object data modeling approach in database design gives the following advantages:

- simplify the design by reduction of the steps of abstractions for mapping the real life problem to the database structure design;
- simplify the structuring of problem by using object data modeling
- reduce the number of SQL statements in query descriptions;
- speed up the query processing;

REFERENCES

- [1] Object-Oriented Database Management Systems Revisited, An Updated DACS State-of-the-Art Report. Prepared by: Gregory McFarland, Andres Rudmik, and David Lange Modus Operandi, Inc, 1999.
- [2] Cattell, R.G.G., Object Data Management: Object-Oriented and Extended Relational Database Systems. Massachusetts: Addison-Wesley, 1997.
- [3] Atkinson, Malcolm, et al., "The Object-Oriented Database System Manifesto." Building An Object-Oriented Database System, California: Morgan Kaufmann, 1999.
- [4] Lecluse, C., et al., "O2, an Object-Oriented Data Model." Building An Object-Oriented Database System, California: Morgan Kaufmann, 1995.
- [5] C. J DATE An Introduction to Database System – 7th ed, Addison Wesley, 2005.
- [6] The Object-Oriented Database System Manifesto "Malcolm Atkinson"
<http://www.cs.cmu.edu/People/clamen/OODBMS/Manifesto/htManifesto/Manifesto.html>
- [7] Hugh Darwen and C.J.Date: " Into the Great Divide," in C.J.Date and hugh Drawen, Relational Database Writings 1989-1991. Reading,, mass.: Addison-Wesley 1992.
- [8] Thomas Connolly and Carolyn Begg, Database systems of particular approach to Design, Implementation and management. Addison Welley 2001.

- [9] Introduction to Oracle9i_SQL-Instructor Guide Volume 2.
- [10] Elisa Bertino and Lorenzo: Object Database Systems: Concepts and Architectures. Reading, Mass.: Addison-wesley (1993).
- [11] Versant Object Technology. How to Evaluate Object Database Management Systems, 1992.
- [12] Wirfs-Brock, Rebecca, et al., Designing Object-Oriented Software. New Jersey: Prentice-Hall, 1998.
- [13] Zdonik, Stanley B. and Peter Wegner, "Language and Methodology for Object-Oriented Database Environments." Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences, (1996).
- [14] Arkinson M. ed. (1995). Proc. Of Workshop on Persistent Object System. Springer-Verlag.
- [15] Embley D. (2003). Object Database Development: concepts and principles. Harlow. Addison Wesley Longman.
- [16] Stonebraker M. (2005). Object-Relational DBMS: The Next Great Wave. San Francisco, CA: Morgan Kaufmann Publishers Inc.
- [17] E. F. Codd: "Relational Completeness of Data Base Sublanguages," in Randall J. Rustin (ed.), Data Base System, Courant Computer Science Symposia Series 6. Englewood Cliffs, N.J.: Prentice-Hall (1988).

- [18] Elmasri R. and Navathe S. Fundamentals of Database Systems 3rd ed.
New York, NY : Benjamin/Cummings, 2001.
- [19] S. Abiteboul and P. Kanellakis, "Object identity as a query language primitive", Proceedings of the 1989 ACM SIGMOD, Portland, Oregon, June 89.
- [20] Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches 2002
- [21] Optimization of object-oriented programs using static class hierarchy analysis. In Proceedings of the European Conference on Object-Oriented Programming (Aarhus, Denmark, Aug.). Lecture Notes in Computer Science, vol. 952. Springer-Verlag. 2004
- [22] Profile-Guided receiver class prediction. In Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (Austin, Texas, Oct.) 2005
- [23] Atkinson, Malcolm et al. The Object-Oriented Database Manifesto. In Proceedings of the First International Conference on Deductive and Object-Oriented Databases, pages 223-40, Kyoto, Japan, December 1999
<http://www.cs.cmu.edu/People/clamen/OODBMS/Manifesto/htManifesto/Manifesto.html>
- [24] R. J. Peters and M. T. " Ozsu. An Axiomatic Model of Dynamic Schema Evolution in Objectbase Systems. ACM Transactions on Database Systems, 22(1):75{114, March 1997.

6.3.6 Create triggers in the database.

A trigger is a message or something that accomplishes the user's a tension.

```
Create Or Replace Trigger Emp_Trigger
Before Insert Or Update Or Delete On Employee
Begin
    If Deleting Then
        Raise_Application_Error(-20502,'Again');
    End If;
End;
```

As seen when the deletion process occurs this message will be appear.

6.4 Object Query Language

After finishing of ODL, it is possible to apply the Object Query Language (OQL) in application.

6.4.1 Insert a row in the table

To insert a row into the employee table there exists two cases. The first case is when there are some attributes with new defined data type, this makes the insert process more readable and understandable as in the following example.

```
Insert Into Employee( Emp_No, Emp_Name, Emp_Birth )
Values ( 20043315,
        Name_Ty('Anwar','M','Dawoud'),
        Birth_Ty('12-Nov-82','Palistine') );
```

The second case is when using the normal way (the employee table without new defined type) the insert statement will be more ambiguous, unreadable, and ununderstandable, as in the following example:

```
Insert Into Employee ( Emp_No, Fname, Mname, Lname, Dob, Pob )
Values ( 20043315, 'Anwar', 'M', 'Dawoud', '12-Nov-82', 'Palistine' );
```

6.4.2 Update a row in the table

To update a row into the employee table there are two cases. The first case is when there are some attribute with new defined data type, this makes the update process more readable and understandable as in the following example.

```
Update      Employee
Set         Emp_Name=Name_Ty('Ahmed','A','Dalo')
Where      Emp_No=20043815;
```

The second case is when using the normal way (the employee table without new defined type) the update statement will be more ambiguous, unreadable, and ununderstandable, as in the following example:

```
Update      Employee
Set         Fname='Ahmed', Mname='A', Lname='Dalo'
Where      Emp_No=20043815;
```

6.4.3 Delete a row in the table

To delete a row from the employee table there exist two cases. The first case is when there are some attribute with new defined data type, this makes the delete process more readable and understandable as in the following example.

Delete	From Employee
Where	Emp_Name=Name_Ty('Ahmed', 'R', 'Dawoud');

The second case is when using the normal way (the employee table without new defined type) the delete statement will be more ambiguous, unreadable, and ununderstandable, as in the following example:

Delete	From Employee
Where	Fname='Ahmed' And
	Mname='R' And
	Lname='Dawoud';

6.4.4 Select rows from the table

To select some rows from the employee table there exist two cases. The first case is when there are some attribute with new defined data type. This makes the select process more readable and understandable as in the following example.

Select	Emp_No, Emp_Name
From	Employee;

The second case is when using the normal way (the employee table without new defined type) the select statement will be more ambiguous, unreadable, and ununderstandable, as in the following example:

Select	E.Emp_No, E.Fname, E.Mname, E.Lname
From	Employee E;

In this application there are some graphical user interfaces. Figures 6.7 and 6.8 show graphical user interfaces for employee and customer tables.

Figure 6.7 Graphical user Interface for Employee Table

Good Serial	Good Name	Good Price	Date Off Sale
1200023511444	Keyboard	20	21/08/2004
2211215554141	Mouse	15	27/09/2005

Figure 6.8 Graphical user Interface for Customer Table

Throughout the last graphical user interfaces it could be generated as a new number of every employee and customer. After that it could be added some information about them such as name, birth of date, and place of birth.

The following report show employee number, name, phone, and email.

EMPLOYEE PHONES+EMAIL ADDRESS

Staff No.	Name	Mobile	Tel	Email Address
20060001	Anwar Dawoud	05330477111	2655779	anw_daw@hotmail.com
20060002	Ahmed Dale	05330498514	2675514	ahmed2000@yahoo.com

Figure 6.9 Report Showing Employee Name, Phone, and Email

Comparison of SQL statements for trading company database design has been listed for with objects and without objects respectively as following :

Number of SQL Statements	DDL Statements			DML Statements			
	Tables	Procedures	Functions	Select	Insert	Update	delete
With Objects	24	8	8	4	8	5	4
Without Objects	34	11	10	6	14	8	6
Improvement	30%	27%	20%	33%	43%	38%	34%

6.5 Summary

The main differences between using Object approaches and without have been declared using the application of trading company.

As seen in this chapter using the ORDBMS concepts make the application more usable, readable, and reduce the size of the code.

CONCLUSION

In the thesis the Object-oriented database technology is discussed as a symbiosis of object-oriented modeling and database technologies. The object-oriented data modeling provides the technique to encapsulate the attributes and the methods for processing the attributes.

It is shown how database storage mechanism incorporates object-oriented data and establishes the Object-Oriented Database Systems. Object-Oriented Database Systems are based on Object-Oriented Programming Language and Database storage mechanisms that depend on the data model established.

The Relational Database System (RDBS) as most popular one is discussed but with disadvantage not to use an object data modeling. The implementation of the object data modeling to RDBS significantly improves and simplifies the database design. This combination is named as Object Relational Database System (ORDBS) and allows a designer to use convenient structured query language (SQL) with object features.

The thesis analyzes declared the features of Object Structured Query Language (OSQL) that includes Object Data Definition Language and Object Query Language. This analysis with discussion of the advantages of OSQL was made by structuring the problem with object data modeling in Trading Company Database Design. The comparisons of the SQL statements made with and without object features showed that the implementing object data modeling approach in database design gives the following advantages:

- simplify the design by reduction of the steps of abstractions for mapping the real life problem to the database structure design;
- simplify the structuring of problem by using object data modeling
- reduce the number of SQL statements in query descriptions;
- speed up the query processing;

REFERENCES

- [1] Object-Oriented Database Management Systems Revisited, An Updated DACS State-of-the-Art Report. Prepared by: Gregory McFarland, Andres Rudmik, and David Lange Modus Operandi, Inc, 1999.
- [2] Cattell, R.G.G., Object Data Management: Object-Oriented and Extended Relational Database Systems. Massachusetts: Addison-Wesley, 1997.
- [3] Atkinson, Malcolm, et al., "The Object-Oriented Database System Manifesto." Building An Object-Oriented Database System, California: Morgan Kaufmann, 1999.
- [4] Lecluse, C., et al., "O2, an Object-Oriented Data Model." Building An Object-Oriented Database System, California: Morgan Kaufmann, 1995.
- [5] C. J DATE An Introduction to Database System – 7th ed, Addison Wesley, 2005.
- [6] The Object-Oriented Database System Manifesto "Malcolm Atkinson"
<http://www.cs.cmu.edu/People/clamen/OODBMS/Manifesto/htManifesto/Manifesto.html>
- [7] Hugh Darwen and C.J.Date: " Into the Great Divide," in C.J.Date and hugh Drawen, Relational Database Writings 1989-1991. Reading,, mass.: Addison-Wesley 1992.
- [8] Thomas Connolly and Carolyn Begg, Database systems of particular approach to Design, Implementation and management. Addison Welley 2001.

- [9] Introduction to Oracle9i_SQL-Instructor Guide Volume 2.
- [10] Elisa Bertino and Lorenzo: Object Database Systems: Concepts and Architectures. Reading, Mass.: Addison-wesley (1993).
- [11] Versant Object Technology. How to Evaluate Object Database Management Systems, 1992.
- [12] Wirfs-Brock, Rebecca, et al., Designing Object-Oriented Software. New Jersey: Prentice-Hall, 1998.
- [13] Zdonik, Stanley B. and Peter Wegner, "Language and Methodology for Object-Oriented Database Environments." Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences, (1996).
- [14] Arkinson M. ed. (1995). Proc. Of Workshop on Persistent Object System. Springer-Verlag.
- [15] Embley D. (2003). Object Database Development: concepts and principles. Harlow. Addison Wesley Longman.
- [16] Stonebraker M. (2005). Object-Relational DBMS: The Next Great Wave. San Francisco, CA: Morgan Kaufmann Publishers Inc.
- [17] E. F. Codd: "Relational Completeness of Data Base Sublanguages," in Randall J. Rustin (ed.), Data Base System, Courant Computer Science Symposia Series 6. Englewood Cliffs, N.J.: Prentice-Hall (1988).

- [18] Elmasri R. and Navathe S. Fundamentals of Database Systems 3rd ed.
New York, NY : Benjamin/Cummings, 2001.
- [19] S. Abiteboul and P. Kanellakis, "Object identity as a query language primitive", Proceedings of the 1989 ACM SIGMOD, Portland, Oregon, June 89.
- [20] Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches 2002
- [21] Optimization of object-oriented programs using static class hierarchy analysis. In Proceedings of the European Conference on Object-Oriented Programming (Aarhus, Denmark, Aug.). Lecture Notes in Computer Science, vol. 952. Springer-Verlag. 2004
- [22] Profile-Guided receiver class prediction. In Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (Austin, Texas, Oct.) 2005
- [23] Atkinson, Malcolm et al. The Object-Oriented Database Manifesto. In Proceedings of the First International Conference on Deductive and Object-Oriented Databases, pages 223-40, Kyoto, Japan, December 1999
<http://www.cs.cmu.edu/People/clamen/OODBMS/Manifesto/htManifesto/Manifesto.html>
- [24] R. J. Peters and M. T. " Ozsu. An Axiomatic Model of Dynamic Schema Evolution in Objectbase Systems. ACM Transactions on Database Systems, 22(1):75{114, March 1997.

- [25] Rob, Peter and C. Coronel, Database Systems Design Implementation and Management, Thompson Pub., 2001.
- [26] Mapping Objects to Tables: A Pattern Language, in "Proceedings of the 2000 European Pattern Languages of Programming Conference," Irsee, Germany, Siemens Technical Report 120/SW1/FB 2000.
- [27] Object Oriented Software Engineering, Jacobson, I., christerson, M., Josson, P. & Overgaard, G., Addison-Wesley, Wokingham, England, 2005.
- [28] Object Oriented Modelling and Design, Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. & Lorensen, W., Prentice Hall, 1991.
- [29] The Common Object Request Broker: Architecture and Specification, Revision 2.0, OMG, July 2004.

APPENDIX A

SQL STATEMENTS

The SQL and PL/SQL statements for the trading company program are shown below.

- SQL statements that create the new data types (objects)

Create Type Address_Ty As Object (

City	Varchar2(25),
Street	Varchar2(25));

Create Type Birth_Ty As Object (

DOB	Date,
POB	Varchar2(25));

Create Type Name_Ty As Object (

Fname	Varchar2(25),
Mname	Varchar2(25),
Lname	Varchar2(25));

Create Type Phone_Ty As Object (

Mobile	Varchar2(13),
Home	Varchar2(13));

- SQL statements that create the main tables of the program

Create Table Nationality (

Nationality_Id	Number(2)Primary Key,
Nationality_Name	Varchar2(25));

Create Table Position (

Position_Id	Number(2)Primary Key,
Position_Name	Varchar2(25));

Create Table Department (

Department_Id	Number(2)Primary Key,
Department_Name	Varchar2(25));

Create Table Employee (

Emp_No	Number(9)Primary Key,
Emp_Name	Name_Ty,
Emp_Birth	Birth_Ty,
Emp_Id	Number(9),
Emp_Sex	Number(1),
Emp_Marital_Status	Number(1),
Phone	Phone_Ty,
Emp_Email	Varchar2(25),
Emp_Address	Address_Ty,
Nationality_Id	Number(2),
Position_Id	Number(2),

```

Department_Id                Number(2),
Constraint Emp_Nat_Fk Foreign Key(Nationality_Id)
References Nationality(Nationality_Id),
Constraint Emp_Pos_Fk Foreign Key(Position_Id)
References Position(Position_Id),
Constraint Emp_Dep_Fk Foreign Key(Department_Id)
References Department(Department_Id));

```

Create Table Customer (

Cus_No	Number(9)Primary Key,
Cus_Name	Name_Ty,
Cus_Birth	Birth_Ty,
Cus_Id	Number(9),
Cus_Sex	Number(1),
Cus_Phone	Phone_Ty,
Cus_Email	Varchar2(25),
Cus_Address	Address_Ty);

Create Table Goods (

Cus_No	Number(9),
Good_Serial	Number(15) Primary Key,
Good_Name	Varchar2(25),
Good_Price	Number(8),
Good_Sale_Date	Date,
Constraint Good_Cus_Fk Foreign Key(Cus_No)	
References Customer(Cus_No));	

- SQL statements that create the main views of the program

Create Or Replace View Employee_Vw As

```
Select      E.Emp_No,
            E.Emp_Name,
            E.Emp_Birth,
            N.Nationality_Name,
            P.Position_Name,
            D.Department_Name
From        Employee E , Nationality N , Position P , Department D
Where       N.Nationality_Id = E.Nationality_Id      And
            P.Position_Id = E.Position_Id            And
            D.Department_Id = E.Department_Id;
```

Create Or Replace View Emp As

```
Select      Employee.Emp_No,
            Employee.Emp_Name,
            Employee.Emp_Birth,
            Employee.Emp_Id,
            Employee.Emp_Sex,
            Employee.Emp_Marital_Status,
            Employee.Emp_Email,
            Employee.Emp_Address,
            N.Nationality_Name,
            P.Position_Name,
            D.Department_Name
From        Employee, Department D, Nationality N , Position P
Where       ((Employee.Department_Id = D.Department_Id) And
            (Employee.Nationality_Id = N.Nationality_Id) And
```


(Employee.Position_Id = P.Position_Id))

- SQL and PL/SQL statements that create the main functions of the program.

Create Or Replace Function Empid

Return Number

Is

```

Max_Id_For_This_Year          Number;
This_Year                    Number;
Max_Year_In                  Number;
Begin
    Select To_Char(Sysdate,'Yyyy')Into This_Year From Dual;
    Select Max(Employee.Emp_No) Into Max_Id_For_This_Year
    From Employee
    Where Substr(Employee.Emp_No,1,4)=This_Year;
    Select Max(Substr(Employee.Emp_No,1,4)) Into Max_Year_In
    From Employee;
    If Max_Year_In Is Null Then
        Max_Id_For_This_Year:=To_Number(This_Year||'0000');
        Max_Id_For_This_Year:=Max_Id_For_This_Year+1;
        Return(To_Number(Max_Id_For_This_Year));
    End If;
    If Max_Id_For_This_Year Is Null Then
        Max_Id_For_This_Year:=To_Number(This_Year||'0000');
        If This_Year=Max_Year_In Then
            Max_Id_For_This_Year:=Max_Id_For_This_Year+1;
            Return(To_Number(Max_Id_For_This_Year));
        End If;
    End If;
    If(This_Year>=Max_Year_In And
    To_Number(Substr(Max_Id_For_This_Year,5,4))<('9999'))Then

```

```

Max_Id_For_This_Year:=Max_Id_For_This_Year+1;
Return(To_Number(Max_Id_For_This_Year));
Else
    Return Null;
End If;
End;
```

Create Or Replace Function Cusid

Return Number

Is

```

Max_Id_For_This_Year          Number;
This_Year                     Number;
Max_Year_In                   Number;
Begin
    Select To_Char(Sysdate,'Yyyy')Into This_Year From Dual;
    Select Max(Customer.Cus_No)Into Max_Id_For_This_Year
    From Customer
    Where Substr(Customer.Cus_No,2,4)=This_Year;
    Select Max(Substr(Customer.Cus_No,2,4))Into Max_Year_In From
    Customer;
    If Max_Year_In Is Null Then
        Max_Id_For_This_Year:=To_Number('1'||This_Year||'0000');
        Max_Id_For_This_Year:=Max_Id_For_This_Year+1;
        Return(To_Number(Max_Id_For_This_Year));
    End If;
    If Max_Id_For_This_Year Is Null Then
        Max_Id_For_This_Year:=To_Number('1'||This_Year||'0000');
        If This_Year=Max_Year_In Then
            Max_Id_For_This_Year:=Max_Id_For_This_Year+1;
```

```

        Return(To_Number(Max_Id_For_This_Year));
    End If;
End If;
If(This_Year>=Max_Year_In And
To_Number(Substr(Max_Id_For_This_Year,6,4))<('9999'))Then
    Max_Id_For_This_Year:=Max_Id_For_This_Year+1;
    Return(To_Number(Max_Id_For_This_Year));
Else
    Return Null;
End If;
End;
```

Create Or Replace Function Age_Employee_No (X Number)

Return Number

Is

I Date;

Begin

Select Dob Into I From Employee

Where Emp_No=X;

Return (Sysdate-I)/365;

End;

- SQL and PL/SQL statements that create the main Procedures of the program.

Create Or Replace Procedure New_Employee (

Employee_No	In Number,
Employee_Name	In Name_Ty,
Employee_Birth	In Birth_Ty)

As

Begin

```
Insert Into Employee(Emp_No,Emp_Name,Emp_Birth)
Values(Employee_No,Employee_Name,Employee_Birth);
```

End;

```
Execute New_Employee( 20060009,
                      Name_Ty('Ahmed','R','Dawoud'),
                      Birth_Ty('11-Nov-1997','Palestine'));
```

The PL/SQL statements that used in the trading company program are shown below.

- When-new-form-instance

```
Set_Window_Property(Forms_Mdi_Window, Window_State, Maximize);
Set_Window_Property('Window1', Window_State, Maximize);
Set_Window_Property('Window1', Title, 'Anwar M Dawoud');
Set_Window_Property(Forms_Mdi_Window, Title, 'Personal Departement');
Go_Block('Main');
Go_Item('Main.T');
Hide_View('Nationality');
Hide_View('Position');
Hide_View('Department');
Declare
```

```
Group_Id      Recordgroup;
List_Id1      Item:=Find_Item('Employee.Nationality_Id') ;
List_Id2      Item:=Find_Item('Employee.Position_Id') ;
List_Id3      Item:=Find_Item('Employee.Department_Id') ;
```



```

X          Number;
Timer_Id   Timer;

Begin
  Timer_Id:=Create_Timer('Clock',1000,Repeat);
  Group_Id:=Find_Group('Rg_Nationality');
  X:=Populate_Group(Group_Id);
  X:=Populate_Group(Group_Id);
  Clear_List (List_Id1);
  Populate_List(List_Id1,Group_Id);
  Group_Id:=Find_Group('Rg_Position');
  X:=Populate_Group(Group_Id);
  X:=Populate_Group(Group_Id);
  Clear_List (List_Id2);
  Populate_List(List_Id2,Group_Id);
  Group_Id:=Find_Group('Rg_Department');
  X:=Populate_Group(Group_Id);
  X:=Populate_Group(Group_Id);
  Clear_List (List_Id3);
  Populate_List(List_Id3,Group_Id);
End;

```

- On-error

```

Declare
  Errnum      Number      := Error_Code;
  Errtxt      Varchar2(80) := Error_Text;
  Errtyp      Varchar2(3)  := Error_Type;
  Yy          Number

Begin

```

```
If Errnum = 41051 Then
```

```
    Null;
```

```
End If;
```

```
End;
```

- When-timer-expired

```
Select To_Char(Sysdate,'Day','Nls_Date_Language=English')
```

```
Into :Time_Date_Day.Day1 From Dual;
```

```
Select To_Char(Sysdate,'Yyy/Mm/Dd') Into :Time_Date_Day.Date1 From Dual;
```

```
Select To_Char(Sysdate,'Hh:Mi:Ss') Into :Time_Date_Day.Time1 From Dual;
```

- Employee form (Insert Button)

```
Clear_Form(No_Commit);
```

```
Show_View('Employee');
```

```
Set_Item_Property('Time_Date_Day.Ins_Pro',Visible, Property_True);
```

```
Set_Item_Property('Time_Date_Day.Que_Pro',Visible, Property_False);
```

```
Set_Item_Property('Time_Date_Day.Del_Pro',Visible, Property_False);
```

```
Set_Item_Property('Time_Date_Day.Upd_Pro',Visible, Property_False);
```

```
Set_Item_Property('Employee.Emp_No' ,Visual_Attribute,'V1');
```

```
Set_Item_Property('Employee.Emp_No' ,Enabled, Property_False);
```

```
Set_Item_Property('Employee.Delete' ,Enabled, Property_False);
```

```
Set_Item_Property('Employee.Insert' ,Enabled, Property_True);
```

```
Set_Item_Property('Employee.Save' ,Enabled, Property_True);
```

```
Set_Block_Property('Employee',Delete_Allowed, Property_False);
```

```
Set_Block_Property('Employee',Update_Allowed, Property_False);
```

```
Set_Block_Property('Employee',Insert_Allowed, Property_True);
```

```
Last_Record;
```

Next_Record;
Go_Item('Employee.Emp_Name_Fname');

- Customer form (Insert Button)

Clear_Form(No_Commit);
Show_View('Customer');
Set_Item_Property('Time_Date_Day.Ins_Pro1',Visible,Property_True);
Set_Item_Property('Time_Date_Day.Que_Pro1',Visible,Property_False);
Set_Item_Property('Time_Date_Day.Del_Pro1',Visible,Property_False);
Set_Item_Property('Time_Date_Day.Upd_Pro1',Visible,Property_False);
Set_Item_Property('Customer.Cus_No' ,Visual_Attribute,'V1');
Set_Item_Property('Customer.Cus_No' ,Enabled,Property_False);
Set_Item_Property('Customer.Delete' ,Enabled,Property_False);
Set_Item_Property('Customer.Insert' ,Enabled,Property_True);
Set_Item_Property('Customer.Save' ,Enabled,Property_True);
Set_Block_Property('Customer',Delete_Allowed,Property_False);
Set_Block_Property('Customer',Update_Allowed,Property_False);
Set_Block_Property('Customer',Insert_Allowed,Property_True);
Last_Record;
Next_Record;
Go_Item('Customer.Cus_Name_Fname');

- Employee form (Update Button)

Clear_Block(No_Commit);
Show_View('Employee');

```

Set_Item_Property('Time_Date_Day.Ins_Pro',Visible, Property_False);
Set_Item_Property('Time_Date_Day.Que_Pro',Visible, Property_False);
Set_Item_Property('Time_Date_Day.Del_Pro',Visible, Property_False);
Set_Item_Property('Time_Date_Day.Upd_Pro',Visible, Property_True);
Set_Item_Property('Employee.Emp_No' ,Visual_Attribute, 'V2');
Set_Item_Property('Employee.Emp_No' ,Enabled, Property_True);
Set_Item_Property('Employee.Delete' ,Enabled, Property_False);
Set_Item_Property('Employee.Insert' ,Enabled, Property_False);
Set_Item_Property('Employee.Save' ,Enabled, Property_True);
Set_Block_Property('Employee',Delete_Allowed,Property_False);
Set_Block_Property('Employee',Update_Allowed,Property_True);
Set_Block_Property('Employee',Insert_Allowed,Property_False);
Last_Record;
Next_Record;
Go_Block('Employee');
Go_Item('Employee.Emp_No');

```

- Customer form (Update Button)

```

Clear_Block(No_Commit);
Show_View('Customer');
Set_Item_Property('Time_Date_Day.Ins_Pro1',Visible,Property_False);
Set_Item_Property('Time_Date_Day.Que_Pro1',Visible,Property_False);
Set_Item_Property('Time_Date_Day.Del_Pro1',Visible,Property_False);
Set_Item_Property('Time_Date_Day.Upd_Pro1',Visible,Property_True);
Set_Item_Property('Customer.Cus_No' ,Visual_Attribute,'V2');
Set_Item_Property('Customer.Cus_No' ,Enabled,Property_True);
Set_Item_Property('Customer.Delete' ,Enabled,Property_False);
Set_Item_Property('Customer.Insert' ,Enabled,Property_False);

```



```
Set_Item_Property('Customer.Save',Enabled,Property_True);
Set_Block_Property('Customer',Delete_Allowed,Property_False);
Set_Block_Property('Customer',Update_Allowed,Property_True);
Set_Block_Property('Customer',Insert_Allowed,Property_False);
Last_Record;
Next_Record;
Go_Block('Customer');
Go_Item('Customer.Cus_No');
```

- Employee form (Delete Button)

```
Clear_Block(No_Commit);
Show_View('Employee');
Set_Item_Property('Time_Date_Day.Ins_Pro',Visible,Property_False);
Set_Item_Property('Time_Date_Day.Que_Pro',Visible,Property_False);
Set_Item_Property('Time_Date_Day.Del_Pro',Visible,Property_True);
Set_Item_Property('Time_Date_Day.Upd_Pro',Visible,Property_False);
Set_Item_Property('Employee.Emp_No',Visual_Attribute,'V2');
Set_Item_Property('Employee.Emp_No',Enabled,Property_True);
Set_Item_Property('Employee.Delete',Enabled,Property_True);
Set_Item_Property('Employee.Insert',Enabled,Property_False);
Set_Item_Property('Employee.Save',Enabled,Property_True);
Set_Block_Property('Employee',Delete_Allowed,Property_True);
Set_Block_Property('Employee',Update_Allowed,Property_False);
Set_Block_Property('Employee',Insert_Allowed,Property_False);
Last_Record;
Next_Record;
```

- Customer form (Delete Button)

```

clear_block(no_commit);
show_view('CUSTOMER');
Set_ITEM_Property('TIME_DATE_DAY.INS_PRO1',VISIBLE,property_false);
Set_ITEM_Property('TIME_DATE_DAY.que_PRO1',VISIBLE,property_false);
Set_ITEM_Property('TIME_DATE_DAY.DEL_PRO1',VISIBLE,property_true);
Set_ITEM_Property('TIME_DATE_DAY.upd_PRO1',VISIBLE,property_false);
Set_Item_Property('CUSTOMER.CUS_no',visual_attribute,'v2');
Set_Item_Property('CUSTOMER.CUS_no',Enabled,property_true);
Set_Item_Property('CUSTOMER.delete',Enabled,property_true);
Set_Item_Property('CUSTOMER.insert',Enabled,property_false);
Set_Item_Property('CUSTOMER.save',Enabled,property_true);
Set_Block_Property('CUSTOMER',DELETE_ALLOWED,PROPERTY_TRUE);
Set_Block_Property('CUSTOMER',UPDATE_ALLOWED,PROPERTY_FALSE);
Set_Block_Property('CUSTOMER',INSERT_ALLOWED,PROPERTY_FALSE);
LAST_RECORD;
NEXT_RECORD;
go_item('CUSTOMER.CUS_no');

```

- Call Report

Declare

X Paramlist

Begin

```

X:=Create_Parameter_List('Order_List');
Add_Parameter(X,'Q_1',Text_Parameter,'P_1');
Run_Product(Reports,'M_Thesis\Application\Report\Report1',
Asynchronous,Runtime,Filesystem,X);

```

```
Destroy_Parameter_List(X);  
End;
```

- The main form (General Constant Button)

```
Show_View('Constant');  
Show_View('Constant');  
Go_Block('Constant');  
Go_Item('Constant.W');  
Show_View('Department');  
Go_Item('Department.Department_Id');
```

APPENDIX B

THE ABBREVIATIONS AND ACRONYMS

The Abbreviations and Acronyms that are used in this thesis are shown below.

ADT	Abstract Data Type
API	Application Programming Interface
BLOB	Binary Large Object
CLOB	Character Large Object
CORBA	Common Object Request Broker Architecture(
DB	Database
DBA	Database Administrator
DBMS	Database Management System
DCL	Data Control Language
DDBMS	Distributed Database
DDL	Data Definition Language
DML	Data Manipulation Language
ERDBMS	Extended Relational Database Management System
IDL	Interface Definision Language
ODBMS	Object Database Management System
ODL	Object Definition Language
OID	Object Identifier
OMG	Object Management Group
OO	Object-Oriented
OODB	Object-Oriented Database
OOP	Object-Oriented Programming
OQL	Object Query Language
ORDBMS	Object Relational Database Management System
OSQL	Object Structured Query Language

RDBMS	Relational Database Management System
SQL	Structured Query Language
UDR	User Defined Routine
UDT	User Defined Type