A COMPARISON OF OBJECT-RELATIONAL AND RELATIONAL DATABASES

A Thesis

Presented to

the Faculty of California Polytechnic State University

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Lara Nichols

December 2007

AUTHORIZATION FOR REPRODUCTION OF MASTER'S THESIS

I reserve the reproduction rights of this thesis for a period of seven years from the date of submission. I waive reproduction rights after the time span has expired.

_____

Signature

_____

Date

APPROVAL PAGE

TITLE: A Comparison of Object-Relational and Relational Databases

AUTHOR: Lara Nichols

DATE SUBMITTED: December 2007

Dr. Laurian Chirica	
Advisor or Committee Chair	Signature
Dr. David Janzen	
Committee Member	Signature
Dr. Alexander Dekhtyar	
Committee Member	Signature

Abstract

A Comparison of Object-Relational and Relational Databases

by

Lara Nichols

Object-oriented programming concepts have been studied and used in academics and industry for some time. These concepts, although highly disputed by some, are not new [19]. However, the ability for engineers to have persistent objects in an object-oriented application has more recently become an area interest for developers and researchers [13] [20]. Application programmers have traditionally used Relational Database Management Systems (RDBMSs) to retain application data [12] . This method of data persistence, although familiar and standard practice for some engineers, is inadequate for object-oriented applications [7]. This work investigates the reasons why traditional relational databases are inadequate for object persistence, an overview of object-relational database systems (ORDBMSs), a comparison of ORDBMs to object-oriented programming language and RDBMS features, and object-relational database performance testing results.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Ever since Postgres released the first object-relational database system (ORDBMS) in 1986, ORDBMSs have been seen as the next generation database system [26] [12]. Currently the three leading database management systems —Oracle, Microsoft, and IBM —have extended their database systems to support the SQL:2003 standard which includes object-relational features [12]. Some analysts predict that because of support for ORDMBSs from all three database management vendors, in the near future ORDBMSs will have a 50% larger share of the market than the RDBMS market [12]. Since an ORDBMS is an extension of RDBMSs with object-oriented programming concepts, using an ORDBMS is appealing to many users because the concepts involved are already known from relational database systems and object-oriented programming languages.

According to Stonebraker the world of database systems and uses for these systems are broken into four quadrants —shown in Figure 1.1 [1]. The first lower-left quadrant represents applications that have simple data and do not require any query capability such as word processing applications. These types of applications do not need a database system. The lower-right quadrant represents applications

that have complex data and do not need query capabilities such as computer-aided design applications (CAD). These applications are best suited using an object-oriented database system (OODBMS). The upper-left quadrant represents applications that have simple data and need query capabilities such as traditional banking applications. Finally, the upper-right quadrant represents applications that have complex data, but also need query capabilities. Stonebraker proposes that applications of this type can benefit most from using ORDMBSs [1].

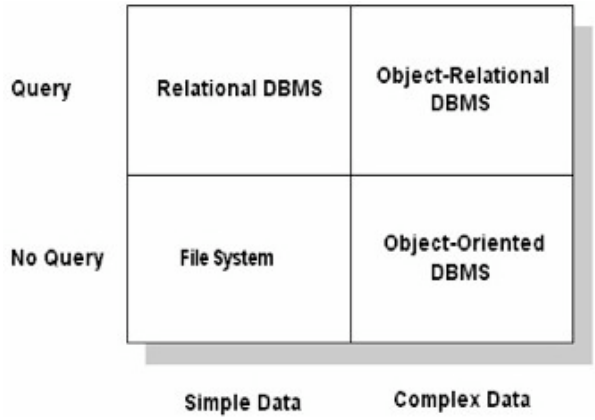


Figure 1.1. Stonebraker's four-quadrants

This thesis presents an overview of ORDBMSs comparing OOPL features to OR features and also gives performance comparisons between using object tables and relational tables for SQL insert, select, update, delete, and table joins operations. The remaining chapters are organized as follows. Chapter two gives background information for OOPLs and relational database systems. Chapter three gives an overview of features available in an ORDBMS. Chapter four provides performance testing for Data Manipulation Language (DML) statements for object-relational tables and relational tables. Chapter five gives an overview of future work. Finally, Chapter six provides a conclusion for this work.

Chapter 2

Background Information

When software engineers design applications they often need a system that provides reliability, durability, concurrency control, recovery, and data integrity for persistent application data. These features are available to engineers using both RDBMSs and ORDBMSs. In addition, both database systems provide a non-procedural query language that allows ad hoc data retrieval and simple data representation in the form of relational tables [24]. However, the object data model allows designers to represent real-world relationships and complex data while also taking advantage of object-oriented programming features such as reusability, robustness, convenience, maintainability, and expressiveness. In addition, features of object-relational database systems also include support for object oriented language concepts such as extensibility, inheritance, encapsulation, polymorphism, dynamic binding, and user defined types (ADT) by extending the RDBMS using the SQL:2003 standard [26] [24] [38] [27] [12] [5] [17].

This chapter gives some background information about relational database systems (RDBMSs) the advantages and limitations of RDBMSs that create the need for ORDBMSs.

2.1 Relational database basics

RDBMSs have been the leading data management software used in industry since the late 1970's [12] [27]. The relational model defined by Codd in 1970 is the basis of a RDBMS [11]. The relational model logically stores data in relations, also referred to as tables. A relation is structured with attributes (columns) and tuples (rows) of data. Humans can easily understand this logical representation since it can be mapped to something that is known in the real world. For example, long before computers, mathematicians and office workers stored information in tables (columns and rows). The relational model's simplicity and mathematical foundation are two reasons why it has remained the leading database management system for the past thirty years [36]. This subsection presents RDBMS terminology and concepts and an overview of RDBMS functionality.

2.1.1 Relational data model

The relational model, despite its simple structure, theoretically bases itself on a mathematical relation [17] [36]. In mathematics, the definition of a relation is "a subset of a Cartesian product of a list of domains" [36]. In a RDBMS, a mathematical relation is a database table. The relational model's mathematical basis defines a set of basic algebraic operations that can operate on tables in a relational database. Basic algebraic operations include insert, delete, update, select, projection, rename, set difference, union, and Cartesian product using the assignment operators used in relational algebra [36]. Additional operators defined in the relational model extend these relational algebra operations. Following are the three components of Coddś relational model [11]:

1. **Data structures used to store data:** relation (table), attributes (columns), tuples (rows), relation instance, and relation schema (table header)

2. **Integrity Constraints:** domain, key, refererential, procedural (application dependant constraints)

3. **Operations:** relational algebra and relational calculus

2.1.1.1 Data structures used to store data in a relational database

The relational model defines an attribute as a <`Name, Domain`> pair where the domain is the set of values and operators that are in the attributes domain. For example, an attribute is <`HireDate, DATE`>where the attribute name is `HireDate` and the domain is the set of valid dates and operations of the `DATE` attribute type.

The relational model defines a relation schema as a < `Name, Set of Attributes` > pair. For example, a table schema used to store business information could have a relation schema `EMPLOYEE(SSN, Lname, Fname, HireDate, Dept)`. Given a relation schema R, a tuple t for R is a mapping from each attribute of R into the domain of that attribute. As shown in Figure 2.1, tuple t for the table `EMPLOYEE, t(Lname) = 'Smith'`. A relation instance is defined as any finite set of tuples for a particular schema. For the schema in Figure 2.1, the relation instance r is as follows: `r(EMPLOYEE) = {t1,t2} where t1 = (234, 'Jones', 'Tom', '1-Mar-1998','IT') and t2=(123,'Smith','Joe','1-Dec-2006','IT')`.

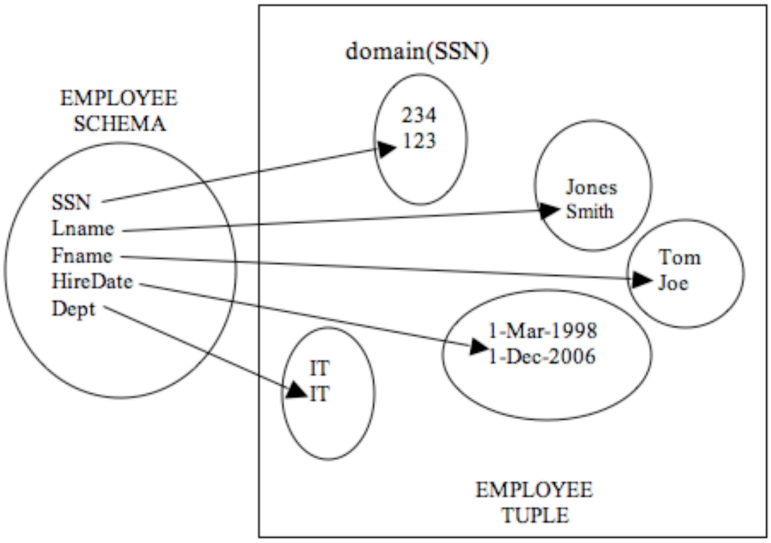


Figure 2.1. `EMPLOYEE` schema and tuple

2.1.1.2 Integrity constraints

The second component of the relational model components is a set of integrity constraints. A database system requires integrity constraints to guarantee its users that data stored in the database is valid. Business rules are implemented as integrity constraints to guarantee that only valid data is stored. For example, in the `EMPLOYEE` schema in Figure 2.1 business rules for the attribute `HireDate` to be less than or equal to today's date can be implemented using the integrity constraint `HireDate <= trunc(sysdate)`. Primary and foreign keys can also be used as integrity constraints to prevent storing duplicate data. More information regarding integrity constraints can be found in Dietrich and Urban's 2005 study and also in Connelly and Begg's 2005 textbook [12] [37].

2.1.1.3 Operations

Relational algebra provides operations on tuples (rows) a set-at-a-time in a single operation that is non-procedural. Codd's model defines operations to manipulate and retrieve data from a database using relational algebra [11] [36]. The fundamental relational algebra operations are separated into two groups, unary and binary operations. The unary operation selection is used to select tuples that satisfy a predicate. For example, using the `EMPLOYEE` schema in Figure 2.1, an application would use selection to select employees that have worked at the company for more than one year. In general, selections use boolean expressions to specify the logical properties of data to be retrieved. A projection operation allows users to specify what relation attributes (columns) they want to see. For example, in Figure 2.1, a projection would be "show the first names of all employees". This would project only the first name relation attribute from the EMPLOYEE schema. The relational model's binary set operations Union, Set difference, Intersection, and Cartesian Product operate on relational pairs. In addition, the relational model defines division and join operations for Theta join, Equijoin, Natural join, Outer join, and Semi joins. Extensions to relational-algebra operations include grouping operations and aggregate functions that take collections of values and return a single value as a result [36].

In 1974, IBM researchers used Codd's relational model as specifications to create the query language Structured English Query Language (SEQUEL) [12] [11]. Designers of SEQUEL were force to change the query language name in 1976 to SQL because of legal reasons. Soon after, researchers at IBM created the first relational database system called System R —used as an engineering prototype —to validate Codd's relational model [36]. Since then, SQL has been

the "only standard database language to gain wide acceptance" by over one hundred database management vendors [12].

SQL allows users to manipulate data by using the DML operations `SELECT`, `INSERT`, `UPDATE`, and `DELETE` with a `WHERE` clause for row selection (equivalent to the relational algebra selection operator). SQL also allows users to use comparison and compound search strings by providing the `WHERE` clause and the `ORDER BY` clause to sort query results. In addition, SQL provides aggregate and grouping functions to operate on data sets.

The importance of SQL is two-fold. First, SQL gives the database community a standard that all database systems can use in order to allow compatibility between different DBMS vendors. Secondly, SQL is non-procedural and because it uses a Standard English language structure, different types of users can easily use SQL whether they are database administrators, applications developers, or business users.

2.1.2 Physical data independence

The relational model supports physical data independence between an application and a database. This is very important because before the relational model, changes made to physical data storage required changing application code. The relational model provides a more flexible data management approach in that physical data storage changes do not affect how applications access the data. The relational model can support physical data independence because it contains a separation between the physical and logical data structure.

2.1.3 Access Control

In a relational database access control is managed by using SQL grant and revoke statements on database objects (tables, views, methods, packages, procedures, database roles, and system privileges). Since every user of the database has a username and password, database administrators control access to data and methods based on usernames. For example, a database user account can exist in the database and administrators can revoke what tables, methods, or columns of data the user can access. Administrators an also grant users of the database access to execute procedures and packages. This provides database users the ability to create procedures and packages once and allow multiple users in the database to access them.

The granularity of access control provided by relational database systems allows database administrators an easy method of managing the need for users to access data and business rules regarding what data users can access.

2.1.4 Transaction Management

A transaction is a "collection of operations that form a single logical unit of work" [36]. For example, adding a class to a student's class list includes updating the class enrollment number (the number of seats let in a class) and the student's class list. It is important that both of these updates complete successfully or both fail. This is referred to as the *atomicity* property of transactions. Additionally, it is important if more then one user is adding classes to their schedule that each transaction is processed separately so inconsistent data is not stored in the database. This is referred to as *isolation*, which guarantees that transactions will not be affected by other transactions running concurrently.

Once a class has been successfully added, it is important that the transaction persist even if there is a power or system failure. This is called the *durability* property [36].

The last transaction property is *consistency*. Consistency is the requirement for data to remain consistent before and after a transaction. For instance, the number of students allowed in a class should remain consistent when added a student to a class. If the student capacity is 30 before the transaction, the seats available before the transaction is 10 and students enrolled is 20, if the add transaction is successful, the available seats should be 9, the capacity should remain at 30, and enrolled students should be 21. The consistency property eliminates the possibility of class seats being created or lost during the student add class transaction.

Transaction management in a relational database with support for the ACID properties gives application developers and database administrators the guarantee of concurrency control and fault tolerance.

2.2 Relational databases weaknesses

Developers most often use a relational database system because of its maturity and simplicity [24]. However, according to many researchers and industry leaders, the relational model in not adequate to model real-world static and dynamic relationships that exist in applications currently being developed [38] [34]. For instance, application developers often need persistence for nontraditional data structures, such as graphics, multimedia, or voice data [37] [14]. The relational model does not support storing and manipulating complex data types such as nested objects, multi-valued attributes, user-defined types, unstructured data

(voice, video), and inheritance relationships [36] [9]. Therefore, developers cannot use a relational database to store data for complex object-oriented applications.

The relational model is not scalable for applications needing access to many related tables, which requires joins [38]. Joining several tables leads to inefficient query processing times [37] [38]. Application developers encountered this weakness when mapping inheritance relationships from object-oriented applications to relational tables. Developers can use different methods to map inheritance to relational tables; however, no matter what method is used, overhead from translation is unavoidable [38]. In object-oriented programming languages (OOPLs), developers use inheritance primarily because of its reuse advantages and elimination of duplicate data. Developers introduce duplicate attributes for each inherited object (storing attributes multiple times for different tables) when converting inheritance to relational tables, since advantages of OOPL inheritance cannot be utilized. According to Rahayu et al.[38], the cost of converting inheritance to relational tables is dependant on the number of objects inherited from a superclass in addition to the number of attributes each object contains.

In addition, converting inheritance to relational tables also decreases overall application performance because it involves joining tables. The cost to join tables in a RDBMS is very significant since joins are the most expensive operation in relational databases [27] [38]. In contrast, an object-relational database can handle access to related tables using object references instead of expensive relational joins to access tables (see Chapter 3.0).

There are many advantages to using the relational data model for data access and storage; however, because of new requirements for object-oriented applications with complex data and relationships, the relational model is not adequate for storing all object-oriented data [12]. The following section presents object-

oriented programming language concepts as background information to object-relational database systems.

2.3 Object-Oriented Language Concepts

According to Coad and Yourdon [10], an object is an abstraction of something in the domain of a problem or its implementation, reflecting the capabilities of a system to keep information about it, interact with it, or both. Software engineers can represent real-world artifacts as objects that applications can manipulate using classes and methods [4]. For example, in an object-oriented banking application, engineers can create objects to represent different types of customers, accounts, and transactions. These objects can be manipulated by users or applications using classes and methods to model real-world phenomena that a banking institutions need to manage.

The most basic critical object-oriented model concepts are abstraction, encapsulation, and inheritance [3]. Object technology goals are to construct software out of standard, reusable parts whenever possible and to reduce software maintenance cost [12]. This development approach allows software engineers to reuse components from one application to another. For example, if developers create a geographic information system (GIS) application that has the objects water, land, location, roads, and methods to manipulate those objects, applications using the same types of objects could later use the objects, classes, and methods developers have already created. The following contains brief definitions of basic object-oriented concepts. These section definitions are important in understanding the concepts presented in Chapter 3.

2.3.1 Objects, Methods, and Classes

An object is an abstract representation of an artifact that engineers can use when developing applications. Each object is an instance of a class and has a state and behavior describing it at any point in time. An object state such as the person object in Figure 2.2, is defined by the attributes `name, address,` and `SSN`. An object's behavior includes the set of methods that are used to manipulate the object, such as `changeAddress()` in Figure 2.2. The concept of an object is simple, but, at the same time, very powerful: each object can be defined and maintained independently of the others [12].

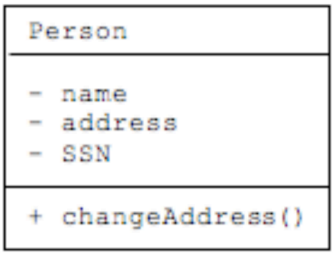


Figure 2.2. Object with attributes and method

2.3.2 Abstraction

Abstraction is the principle of ignoring aspects of a subject that are not relevant to the current purpose in order to concentrated more fully on those that are [10]. By using abstraction, software engineers can break processes into sub processes, making complex problems easier to solve by focusing on a few concepts at a time. Abstraction also allows developers to abstract common procedures out of an algorithm and write the procedure once, but use it multiple times throughout the program as needed.

2.3.3 Encapsulation

Encapsulation involves hiding information from other modules so that users or programs cannot see what methods contain [22]. Encapsulation groups data and methods together creating a well-defined boundary around an object. Classes allow software engineers to use public or private access to control access to object variables and methods. This is helpful for developers who do not want outside modules or applications to either access or manipulate methods while still providing reuse advantages [5] [15].

2.3.4 Inheritance

Inheritance is the means of defining one class in terms of another [3]. Objects with similar attributes are abstracted into generalized and specialized classes. The generalized classes create superclasses and specialized classes become subclasses inheriting from the superclass [16]. For example, a university application could create a superclass `person` and subclasses `student` and `instructor` that inherit from the `person` class. Attributes from a superclass are inherited by a subclass. In the university example, all attributes in the `person` class are available to the `instructor` and `student` classes.

In addition to a class inheriting from one superclass, subclasses can also inherit from multiple superclasses. This is referred to as multiple inheritance. For instance, the subclass `graduate_instructor` is both an `instructor` and a `student` and thus inherits from both superclasses. Using inheritance allows software engineers to develop less applications code since a superclass can define attributes for multiple subclasses.

2.3.5 Polymorphism and Overriding

Using inheritance, generalized methods contained in a superclass can be specialized by overriding them in a subclass. For instance, in a person class the method `getName()` can be declared and then in the subclasses that inherit from it `student, instructor, graduate_instructor` can implement the method to return the correct information when each subclass method is called. This is shown in Figure 2.3. Polymorphism is the capability for the same method to have multiple implementations. The correct implementation is selected that corresponds to the subclass method is called. In Figure 2.3 if the function `getName()` is called for an instance of the employee class, the string `Employee Name:` will be returned and not the string `Person:`. Polymorphism is implemented by overriding superclass methods.

2.4 Object-Oriented Language Features

In addition to abstraction, inheritance, overriding, and polymorphism OOPLs, object-oriented languages also provide features of reusability, maintainability and ease of modeling real-world objects over non-OOPLs. This section gives an overview of reusability, maintainability, and ease of modeling objects in OOPLs.

2.4.1 Reusability

Reusability in OOPLs includes code reuse within a single software project and code reuse between multiple projects. According to Brooks [19], reusability of code in software engineering is the most effective way to increase productivity in software development. An in-depth study done by Lewis et. al [33] found

```
public class Person{
   public String getName(){
   System.out.println(‘‘Person’’);
   return name;
   }
 }
public class Student extends Person{
   public String getName(){
   System.out.println(‘‘Student Name:’’  name);
   return name;
    }
 }

public class Employee extends Person{
   public String getName(){
   System.out.println(‘‘Employee Name:’’  name);
   return name;
   }
 }
```

Figure 2.3. Polymorphism in OOPLs

increased productivity when developers used reusability when developing object-oriented applications. To take full advantage of code reuse, developers must identify redundant code and create methods and procedures that increase code reuse. This process of abstraction can be difficult in large software projects; however, by creating smaller methods that only complete one task and planning for reuse during the design phase, reusability can increase productivity.

2.4.2 Maintainability

Maintenance of a system is the most costly part of the software process [21]. According to Brooks [19], the cost of maintenance can be more than forty per-

cent of the total cost of the entire software system. One of the most important part of the maintenance cycle is being able to understand how changes will affect an application. Fisher et. al. states, "Maintainability and controlled evolution of a system is dependent on the understanding of what is currently present, as changes in design are affected by the prior design" [29]. Using OOPLs to develop applications produces a cleaner, easier to understand system because application implementation is separated into classes and methods instead of having application code in a single function as in non OOPLs [17].

2.4.3 Ease of modeling real-world objects

Ease of use is defined as something that is easy to find, easy to understand, or is sufficient for the task at hand [25]. Using an object-relational database to store application objects gives developers a more expressive way to solve problems. This makes the database design process easier understand, since OOPL objects can be mapped to database objects, and easier to use because developers can create data models that accurately describe application objects and relationships between them[24] .

For example, database designers can use both object and relational tables to store data for an object-oriented application. The developer is not confined to only using object or relational tables but instead has the option of choosing what type of table works best for the data type being stored. Also, object tables can contain one or more object types along with SQL supported data types such as number, varchar2, or date [30]. In addition, there are also many options that developers can choose from when designing the database schema such as object references versus joins, object methods or PL/SQL procedures, ADTs or scalar

types. Using an OR database to store object-oriented data, allows database designers to create database schemas that are easier to understand and that adequately represent application objects and relationships between these objects.

2.4.4 Chapter Summary

The relational data model is important because it is a formal definition for RDBMSs to store, access, and manipulate data with support for data independence, integrity constraints, access control, and transaction management.

RDBMSs provide a high level of abstraction by using a table structure that is easily understood by its‘ users, possessing different data representation levels in column, row, table, tablespace, and schema form, and hiding the physical storage structures from users [11]. Separation of physical data storage from logical data representation allows users and applications to continue to access data even if database administrators move or change physical data structure thus giving a RDBMS data independence [8]. Integrity constraints support data validation and business rules that can be stored and imposed on RDBMS data when applications or users manipulated data [8]. Access control allows businesses to control data access privileges and manipulation for RDBMS data. The relational model also includes transaction management providing concurrency control, failure recovery, and preservation of data [8]. Using the relational model also provides developers ease of use by providing a non-procedural query language and seamless data retrieval for applications..

There are many advantages to using the relational data model for data access and storage; however, because of new requirements for object-oriented applications with complex data and relationships the relational model is not adequate

for storing persistent object-oriented application data [12]. The relational model lacks the ability to adequately map inheritance in object-oriented applications to relational tables without encountering the high cost of duplicate attributes and costly joins or the need to use object-mapping tools (OMTs) such as Hibernate. The relational model inadequacies point to a need for a new data model that can support the different data relationship types used in object-oriented applications. Not all researchers and database designers share this view. For instance, Date and Darwen firmly believe in the relational model and do not see a need to change the model to include object-relational data. Instead, they firmly believe think "those close to the problem should create solutions to address inheritance and user-defined types without changing the relational model" [14].

Object-oriented languages and concepts are not new to the software engineering field [17]. Software engineers use languages such as C++, C#, and Java when creating object-oriented applications. This application development approach promotes maintainability, flexibility, code reusability, software quality, and makes it easier for software engineers to solve complex problems [10].

This chapter presented the most basic relational database and object-oriented concept definitions that are important to understanding this work's remaining chapters. Object-relational database management systems (ORDBMSs), like OOPLs, incorporate the use of objects, classes, methods, inheritance, abstraction, and encapsulation. Chapter 3.0 goes into more detail of how developers can use these concepts when creating object-oriented applications to store persistent objects in ORDBMSs.

Chapter 3

The Object-Relational Approach to Data Management

An object-relational (OR) approach to data management using ORDBMSs includes objects that need persistence, a data model, a query language to manipulate, retrieve, and store data, and a database system [26] [14] [17]. Persistence is the process of storing information that is retained after an application is terminated. There are three basic types of databases that developers can use for persistence: a RDBMS, an object-oriented database system (OODBMS), or an ORDBMS. This work focuses on the use of an ORDBMS for methods of object persistence in contrast to using a traditional RDBMS.

A data model is a "logical organization of the real-world objects (entities), constraints on them, and relationships among them" [26]. Real-world objects and the relationships between them can be represented in entity-relationship (ER) diagrams using modeling techniques such as Coad/Yourdon notation, Shlaer/Mellor notation, Booch notation, or the more widely used Unified Modeling Language

(UML) [22] [10]. No matter what method developers choose, modeling notation is important because it allows developers to represent relationships between entities using a standard representation, which database designers can transform into database schemas used to store database objects and relationships.

An OR database is a collection of objects whose behavior, state, and relationships can be viewed or manipulated using object methods, stored procedures, or a query language [30]. As discussed in Section 2.2.1, in an ORDBMS methods can be stored with objects to manipulate or view the state of an object. Alternatively, stored procedures can be used for the same purpose where procedures are developed using PL/SQL and called by applications to retrieve and manipulate database objects. In Oracle 10g, SQL can also be used to view or manipulate the behavior, state, and relationships for objects stored in its ORDBMS.

The following sections present the need for ORDBMSs, an OR database overview, and a comparison of OOPL and RDBMS features to ORDBMS features.

3.1 The Need for Object-Relational Database Systems

If the relational model has met the data storage needs for the last thirty years, what has changed in order that developers need to store complex data types and relationships when creating applications? To answer this, one must identify why complex data and relationships exist, whether software engineers have created this new phenomenon, or whether this indicates that the relational model has always been inadequate to store the types of data and relationships that are needed

to develop applications. According to Cook and Ibrahim [13], the use of object-oriented programming languages has resulted in a set of new issues that "arise at the boundary between programming languages and databases". Object persistence introduces issues of impedance mismatch between programming languages and databases because of complex relationships and user defined data types [12]. In this paper, complex relationships are defined as many-to-many relationships and inheritance [3]. Likewise, the definition of complex data is nested objects, multi-dimensional arrays, unstructured data (voice, video), data in non-first normal form, and user-defined data types [42]. Complex data and relationships, such as inheritance, nested objects, and user-defined data types are properties of OOPLs. Therefore, the need to store complex relationships and data is in part a result of using OOPLs to develop applications. Before OOPLs, the relational model was adequate for storing application data [14].

The need to store complex data has increased partly because of the increase of unstructured digital media such as photos, voice, and videos due to the reduced cost of storage and the increase of digital recording and transferring technology. As the development of object-oriented applications and digital media storage continue to increase, the issues they create with current RDBMSs will also continue to remain [7]. Therefore, because of the relational model inability to represent concepts of OOPLs and store complex data, developers need a new data model to store persistent application objects.

In order to persuade software engineers to use OR databases for object persistence, this work presents a list of ORDBMSs advantages. Following are the advantages to using an ORDBMS for application object persistence.

1. **Impedance mismatch removal**

2. **Ease of modeling real-world objects and relationships**

3. **Ability to create user-defined types**

4. **Persistent object encapsulation**

5. **Object Referencing**

The following sections present an overview of the advantages to using an OR database for storing persistent objects when creating applications compared to using relational databases..

3.1.1 Removal of impedance mismatch

A main advantage to using an OR database —as opposed to a relational database —for object-oriented applications is the removal of the impedance mismatch between the object-oriented applications and the relational model [12] [4] [41] [7] [37]. Impedance mismatch is the "incompatibilities that occur at each interface between two set of tools due to the different models for importation representation" [18]. Since the application programming language and the database system are based on different data and computation types, application developers are forced to manually map application data to relational tables or use an object mapping tool (OMT) to convert the application object data to the relational model [3] [37] [23]. An impedance mismatch between an application and a database affects overall application development time, performance, and leads to discrepancies between the design and implementation [7] [18].

Added development time for mapping objects to relational tables depends on the applications object types [2]. If developers can easily map application objects

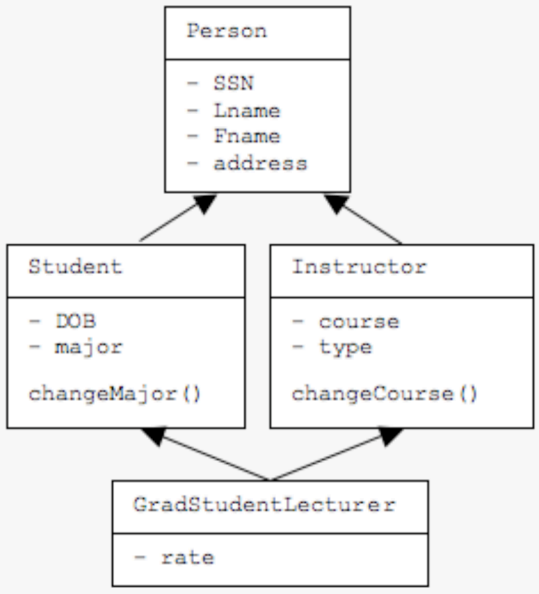
to relational tables, the design does not need substantial changes thus adding negligible development time. However, the higher the impedance mismatch (i.e the more complex the objects and relationships in the application needing persistence), the more developers must develop additional code to represent objects in relational tables, which translates to more application-development time [37] [38]. According to Olofson, as quoted by Leavitt, "Programmers sometimes spend more than 25 percent of their coding time mapping program objects to the database" [27].

When developers map object-oriented applications to relational databases, not only is there added time for development, there is also a decrease in performance because the relational model does not support very well some relationships in OOPLs. One such relationship is inheritance [27] [24] [38] [17] [37]. In order to map object-oriented program inheritance to the relational model —whether developers do it by hand or with a mapping program —the objects that use inheritance must each be created as a table.

This means that if an application needs to retrieve persistent data from all three tables (a superclass query), a relational database must join all three tables. Since joins in a relational database are expensive, this cost can be substantial [27] [38].

As shown in Figure 3.1, four objects (two students, one instructor, and one graduate student lecturer) map to four relational tables. Since the relational model requires each table to have a primary key, each mapped table has an extra attribute ID [38]. Although an added attribute for each relational table may not seem significant, when hundreds of thousands of tuples and numerous tables are involved, an added attribute for each table is a significant overhead. In addition to the added attributes for primary keys, if the rela-

For example, if a superclass `Person` has two subclasses that inherit from it `Student` and `Instructor`, and a class `GradStudentLecturer` that inherits from both `student` and `instructor`, developers must create the following tables to map from objects to relational tables.



Person

ID	SSN	Lname	Fname	address
1234	1111	smith	dave	1 O st
2345	2222	long	jane	2 P st
3456	3333	jones	bill	3 Q st
4567	4444	adams	tom	4 R st

Student

ID	DOB	Major
1234	1-1-1993	CSC
2345	2-1-1993	CSC
4567	10-20-1980	CSC

Instructor

ID	Course	Type
3456	CSC 300	Professor
4567	CSC 101	Graduate Assistant

GradStudentLecturer

ID	Rate
4567	10.75

Figure 3.1. Impedance mismatch between relational model and objects

tional table `graduateStudentLecturer` is queried, the tables `person, student, instructor,` and `graduateStudentLecturer` must all be joined. In comparison, an object-relational database can store all information in one object table where each row is person object. Examples of inserts for a student object type that inherites from the super type person are shown later in Chapter 3 in Figures 3.51 and 3.54.

Using object tables instead of relational tables to store data in an object-oriented application removes the impedance mismatch between OOPLs and the relational model by mapping application objects to database objects. In contrast, if a relational database is used to store application data, application objects must be mapped to relational tables [24]. According to some database researchers

[24] [27], removing the impedance mismatch between object-oriented applications and database systems leads to developing less application code, which reduces development time. Not only is less application code developed, the mapping between application and database code is also less complex since the object-oriented features can be utilized in both the application and the database system.

3.1.2 Ease of modeling real-world objects and the relationships between them

Using ORDBMS for applications that need persistence offers application developers a more natural way to represent real-world entities and relationships. Not only does it improve designers‘ abilities to represent problems in an application, but using an ORDBMS also allows designers to think of problems at a higher abstraction level without having the burden of mapping application data to relational tables [38] [15] [12] [17]. This idea is closely related to Section 3.1.1 in that developers do not have to work around the relational model's limitations. The difference between impedance mismatch and ease of modeling real-world problems is that the former is measured by additional time and code for development, whereas the latter is measured by the developed applications overall quality [5].

The application‘s overall quality could be defined as how closely the overall design models the real world, how resilient it is to change, and how maintainable the application is throughout the software life-cycle [17] [5]. It has been stated by many that an object-oriented development approach can lead to an overall better-designed application that is maintainable and flexible, and can better model artifacts in the natural world [32] [17] [5]. Therefore, using an object-oriented approach to solving application data persistence could allow developers

to design higher-quality applications [22] For example, if developing a computer-aided design (CAD) application, and software engineers are limited to using the relational model, they must not only map objects to tables, they are also limited in application development by what can be represented in the database. Thus, the overall design of the CAD application is limited to data representation of the relational model. This could result in a poorly designed application and make them hard to maintain if new features are needed that are either hard or impossible to represent using the relational model.

3.1.3 Ability to create user defined data types

The ability to create user defined data types (UDTs) in a database is something that experts in database systems agree is needed [14] [35] [1]. The ability to create new data types increases expressiveness and maintainability in object-oriented applications since objects can be stored directly in a database without the need to convert application objects to relational data types. According to Stonebraker and Brown [1], UDTs are necessary in order to solve data storage for applications in quadrant four of Figure 1.1. Without UDTs a table column is limited to data types supported by SQL [1]. Although the SQL data types are sufficient for many applications, quadrant four applications are complex and limiting data types to only those supported by SQL forces developers to create complex application code that increases runtime and could be eliminated if a UDT were created.

For example, the software application ArcGIS (a quadrant four type application) is a geographic information system (GIS) mapping application used by business analysts‘ to determine the number of potential customers in a specific lo-

cation. In order to compute the number of customers from a shopping center, the application needs to be able to compute distance from a customers' home address to the shopping center address. In order to compute the distance between potential customers and a business, a geographic position of each potential customer is needed as a (longitude, latitude) point. Given the person and business table relational schemas in Figure 3.2 queries can be developed to answer the question "how many potential customers are within 5 miles of the business". However, not only are these queries complex they also are inefficient. They are inefficient because since customers' distance from the business is not calculated until the query is ran, indexes will not be used so the distance from the business must be calculated for every person in the database. If the database stored information about every residence address in the United States, it would result in over three hundred million calculations to determine how many potential customers live within five miles of a given business. If instead, a UDT is created that supports (longitude, latitude) point calculations and logic to identify "points that are close by" the application will be less complex and also have much better performance [1].

Overall, UDTs are necessary to support quadrant four applications because they give application developers the ability to solve complex problems without the limitations of only using data types supported by SQL.

```
person( id number, long number, lat number );

business( id number, long number,lat number );
```

Figure 3.2. `person` and `business` table schemas

3.1.4 Objects and methods stored together

In traditional RDBMSs data is stored in relational tables while stored procedures are stored in the database schema. However, in an ORDBMS object data and methods to manipulate that data are stored together [30]. In a RDBMS data is manipulated by application procedures, stored procedures, or SQL commands. In contrast, applications using an ORDMBS can manipulate data using methods stored with the database attributes [24]. This is achieved by having getters and setters for data attributes in addition to basic data manipulation methods as shown in Section 2.2.1 in Figure 2.1. Using database manipulation methods reduces the amount of application methods created by developers and guarantees database administrators correct data manipulation. For example, if the data attribute salary is stored in the database as an annual value the database can provide methods to `getMonthlySal()` and `getAnnualSal()`, to guarantee the correct values are retrieved. The technique of storing methods and data together is a property of OOPL (encapsulation); therefore, it is best practice to provide encapsulation for persistent application objects.

3.1.5 Object references

In a RDBMS, relational joins are considered to be one of the most expensive operations [27] [38]. In an ORDBMS, related tables can be accessed by using object references instead of table joins. Object references provide an easy way to navigate between objects using the object-oriented dot notation. Figure 3.3 shows the commands to create a table with an object reference (REF) as well as the query to access the referenced object.

```
SQL >create type instructor_t as object
(name varchar2(25)
dept_id varchar2(10),
dept_name varchar2(25));

SQL >create table instructor of instructor_t(
primary key(person_id) );

SQL >create table course(
course_id varchar2(25),
course_name varchar2(25),
course_desc varchar2(256),
instructor REF instructor_type scope is
instructor_table);

SQL >select c.* ,c.instructor.name,
c.instructor.dept_name from course c;
```

Figure 3.3. Creating and using object `REF's`

Using an object reference to a related table has the potential of being more efficient then using a relational foreign key. Performance testing for object references versus relational table joins is presented in Chapter 4.

3.2 Overview of Object-Relational Database Systems

As more object-oriented applications are developed, increasingly more complex requirements exist for the types of relationships and data these applications must manipulate [13]. Object-relational databases have the ability to represent complex data and the relationships between complex objects. Object-relational database systems are based on the combination of OOPL and relational database system features [32]. Object types and object tables are the most basic ORDBMS

concepts. An object is defined as a representation of an artifact that is being modeled by the database. Each object represents a class instance and contains the data structures from its class definition, along with access to class methods [5]. Before an object table can exist an object type must be created with the attributes of the object. Once an object type is created, an object table can be created from the type where each row in the object table is an instastance of the object [12]. When an object is created, the database system assigns it a system generated unique object identifier (OID). An OID is never reused even if the object is deleted and will never be modified. These features of OIDs allow each object in an OR database to be unique. Although related objects are referenced using OIDs, database users also have the option of using primary keys that are used in RDBMSs for uniqueness as shown in Figure 3.4. RDBMS primary keys and OIDs are different in that a user can choose and alter a primary key while an OID cannot be chosen or altered (see Section 3.1.5). The stability of an OID is important in an object-relational database when referencing objects because an OID is guaranteed not to change causing an object reference to become invalid [30].

```
SQL >create type PERSON_TYPE as object(
person_id varchar2(10),
first_name varchar2(25),
last_name varchar2(25) );

SQL >create table person of PERSON_TYPE (
person_id primary key)
object identifier primary key
```

Figure 3.4. Creating and using object `REF's`

Once objects are created, the majority of ORDMSs use SQL:2003 standards to define OR integrity constraints, operations, object data structures, relationships,

transaction management, and OOPL features such as encapsulation, abstraction, polymorphism, overriding, and overloading. The following sections present this information in detail.

3.2.1 Data modeling

Data modeling is one of the most important parts of database and application development because it specifies the kinds of real-world properties and operations that must be represented in the database [35] [16]. A data model consists of a set of data structures to store data, relationships, integrity constraints. There are different levels of abstraction involved in database modeling.

The first level of abstraction is mapping real-world problems to a conceptual model. The conceptual model is an in-depth analysis of how the system will be used and the attributes needed to develop the system. For example, engineers developing a conceptual model for a simple banking system might identify that the system will be used to open and close personal and business accounts.

Once the purpose of the system has been identified, developers can determine what attributes are needed to develop the system. This stage of data modeling helps engineers, and users define the attributes and operations needed instead of creating attributes and operations that could relate to the system, but are not needed. For example, many attributes are available to engineers about banking customers that could be used in developing a banking system; however, if the system‘s use is only to open and close accounts only attributes relevant to those operations are represented in the conceptual model. A conceptual model is usually created using design tools and is written in normal language that can be understood by engineers and those the system is being developed for.

Besides determining the use of the system, conceptual modeling also includes identifying objects, methods, relationships and events between objects, and behavior at the system external boundary [16]. During analysis, engineers may determine the classes `person`, `account`, and the operations `getBalance()`, `deposit()`, `withdrawal()`, and `closeAccount()` are needed. Once the use of the system and all the classes, methods, relationships, and events have been identified, the logical model can be developed.

The second level of database modeling is taking the conceptual model and mapping it to a logical model that can be represented by a database system. The purpose of a logical model is to use the capabilities of a database system to implement the conceptual model. From the simple banking system example engineers would map the conceptual model to a logical model by creating a diagram that shows objects, relationships, and methods involved in the data model.

The third level of abstraction involved in database modeling is the physical data model. The physical model is the implementation of the logical model. For the banking example, engineers would develop the SQL commands to create the objects, methods, relationships, events, and interaction at the system boundaries that were identified in the conceptual and logical models.

3.2.2 Standards in object-relational database systems

According to Connolly and Begg [12], there is no single OR data model, instead database management creators use the SQL: 2003 standard as a guideline and implemented the standard to whatever degree they desire. Although there is not one OR model, since database vendors use the SQL:2003 standard, all ORDBMSs do have relational tables, a query language, object types, object ta-

bles, methods, procedures, and the ability to store object data [12]. For example, Oracle's 10g ORDBMS documentation states what standards from SQL: 2003 it has implemented and which it has not implemented [30] .

History

The first standard to support object-relational databases was the ISO/IEC 9075/1999, commonly referred to as SQL:1999 [12]. Included in the SQL: 1999 standard are core and non-core standards. In order for a database vendor to say that they comply with the SQL standard they must at least meet the core SQL standards. The core SQL:1999 standards came from the SQL:1992 standard and the non-core or extended standard was created to manage object-oriented data by adding Binary Large Objects (BLOBs), Character Large Objects (CLOBs), REFs, and user defined data types (UDTs).

Included in the ISO/IEC 9075/2003 standard —usually referred to as SQL: 2003 —is support for user-defined types, procedures, methods, functions, and operators in addition to type constructors for types, and type constructors for collection types such as arrays, sets, lists, and multisets [12]. To date, SQL: 2003 provides the greatest support for managing object-oriented data within a relational framework.

3.2.3 Database modeling

The following sections present an overview of database modeling history and OR database modeling.

3.2.3.1 Database modeling history

Currently the most used database model is the relational model (Section 2.1.1); however, in order to understand object-relational database modeling this section provides information about the hierarchical and network database models. The hierarchical database model, created in the mid 1960's, represents data as collections of records and relationships as trees whose nodes are these records [12]. The model represents the database as a set of trees, where the data (records) are nodes and the relationships (sets) are represented as edges between the nodes (parent/child relationships). The physical storage of data in the hierarchical database model is collections of files linked by physical pointers; a main record is at the top level, and subsequent types of records branch below. The hierarchical data model's best-known product is IBM's IMS DBMS [12]. The model is also used in the lightweight directory access protocol (LDAP) —a protocol to query and modify organization directory information stored in systems such as Oracle Internet Directory [17].

The network database model, created in 1964 by Charles Bachman, is a superset of the hierarchical model where data is stored as collections of records and relationships are stored as sets. The network model stores records and sets the same way the hierarchical model stores them as nodes and edges respectively [12]. The physical storage of data in the network model is also the same as the hierarchical model —collections of files linked by physical pointers [12]. The main difference in the network model from the hierarchical model is that the network model can represent many-to-many relationships between nodes. This creates a network of nodes instead of a tree graph. Although the network model can repre-

sent many-to-many relationships, it is still not adequate to manage data because it still lacks the ability to support data independence.

The disadvantage for both models is that writing queries to retrieve information required a deep understanding of the navigational structure of the data itself. Consequently, users of the system had to know not only what data to retrieve from the system (for example, select business customers), they had to know how the data should be retrieved based on the physical storage of the data. This was a complicated task and was generally left to experienced procedural programmers [36].

In 1970, Codd proposed a solution to data storage in his paper, "A Relational Model of Data for Large Shared Data Banks" [11]. Codd's proposal was to store data independently from hardware and create a nonprocedural language for accessing data. Codd's solution suggested that data should be stored in simple tables with rows and columns instead of being stored in hierarchical or network structures [11]. This method of data storage eliminates the need for a database user or application to know the structure of the data in order to access that data.

Although the relational model has been adequate to this time it does not support objects, UDT, or relationships between multiple objects [1]. To remedy this, database vendors have extended the relational model to include support for UDTs, objects and their relationships. It should be noted that not all researchers and database users agree with extending the relational model to include support for objects. For example, Date and Darwen [14] strongly disagree that the relational model is inadequate and do not believe that the relational model should be changed. The following subsection discusses what the OR approach to data modeling should include and presents its advantages and disadvantages.

3.2.3.2 Object-relational database modeling

Similar to the relational model (see Section 2.1.1), an OR data model must have the following components [11]:

1. **Data Structures used to store data:** object tables
2. **Integrity Constraints:** object identifiers, relationships
3. **Operations:** query language

3.2.4 Data structures used to store data

SQL:2003 provides object types and object tables to store object data [30]. An object type is not the same as an object table. An object type is a logical structure containing the attributes of an object. Before creating an object table, an object type must be created with the attributes that define the object type. Once an object type exists, an object table can be created using the SQL create table statement shown in Figures 3.5 and 3.6.

Each row in an object table is a single object instance with the data types specified in the object type [30]. For example, each row in the object table `name_table` in Figure 3.6 is an object instance with the attributes `first` and `last`.

```
SQL >create type NAME_TYPE as object(
first varchar2(25),
last varchar2(25));
```

Figure 3.5. Object type

Objects are stored in a relational table as either column objects, row objects, or nested tables [30]. Object types used as attributes in an object table are stored as column objects. Object tables created as an object type are stored as row objects [30]. Objects can also be stored in nested tables where each column object is a table. Examples of row, column, and nested table objects are given in Figures 3.6, 3.7, and 3.8.

```
SQL >create table NAME_TABLE of NAME_TYPE;
```

Figure 3.6. Object stored as row object

```
SQL >create table PERSON_TABLE(
id number(10) as primary key,
name_col name_type,
age number);
```

Figure 3.7. Object stored as object column

```
 SQL >create table PERSON_TABLE(
id number(10) as primary key,
name_history name_type,
age number(3) )
NESTED TABLE name_history store as h_name_tb;
```

Figure 3.8. Nested object table

According to Oracle's documentation, their DBMS stores objects as a tree like structure where the branches represent attributes and attributes that are objects are stored as a subbranch of that attribute [30]. Each branch eventually ends with an attribute that is a SQL data type such as `number` or `varchar2` [30]. An example of the storage structure for the `person_table` from Figure 3.7 is shown in Figure 3.9.

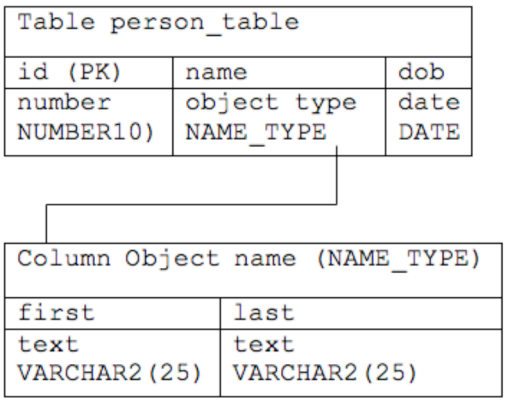


Figure 3.9. Storage of PERSON_TABLE

3.2.5 Integrity Constraints

Constraints exist in both real-world objects and relationships between objects [16]. Dates, times, and physical properties of objects are examples of constraints on physical entities. There are constraints in both the physical world and the abstract world. In the abstract world constraints are placed on objects either by humans or by limits in technology. In databases, constraints are used to implement business rules. For example, in a HR application, constraints on employee hire dates, office numbers, and the department that an employee works in can be defined to implement business rules defined in a company. There may also be constraints placed on attributes from engineers creating the application due to hardware or software limitations. For example, database designers may put constraints on the maximum size of a tablespace because of a limited amount of physical storage.

Constraints define what values are valid for each attribute. Starting with Codd's relational model, the ability to use constraints in database systems became one of the standards for distinguishing relational database systems from non-relational database systems [12]. This section compares constraints in an

OR database to constraints in a relational database system. Comparisons will be made using Oracle's 10g database system for both RDBMS and ORDBMS schemas.

As shown in Figure 3.10, Oracle does not allow developers to create constraints on type attributes [30].

```
SQL>create type ADDRESS_TYPE as object (
street varchar2(50),
city varchar2(25),
state varchar2(2) NOT NULL,
zip varchar2(9));

PLS-00218:  a variable declared NOT NULL must have
an initialization assignment.
```

Figure 3.10. Create constraint on type attribute

Constraints on types can be created in object tables when an object type is used as either a row or column object [30]. Not allowing constraints on type definitions is a feature of usability—enabling the type to be used by many different applications—since constraints are made specifically by each object table using the objec type. For example, `ADDRESS_TYPE` could have a constraint in one application to only except addresses for the state `'CA'` while another application or table may want to include all U.S. states. In both cases, even though the constraints on each type are different, the same type can be used since the constraints are not created on the definition of the type.

Similarly, in OOPLs constraints on types are not allowed. Some researchers have proposed and implemented type constraints in OOPLs to efficiently analyze and optimize object-oriented programs; however, this is an extension to OOPLs [40]. SQL:2003 provides the following five types of constraints for relational table

attributes.

I. Not null

I I. Unique

I I I. Primary key

IV. Foreign key

V. Check constraints

These same five types of constraints should also work with object table and type attributes. In the following subsections the five constraint types are created on object table attributes in the create table statement. In addition, the basic functionality of adding a constraint to an existing object table, and dropping a constraint on an object table are tested.

I. Not Null constraint

Constraints in Oracle's 10g ORDBMS are implemented on columns in object tables. Figure 3.11 shows the SQL statement to implement a not null constraint on the column street in the object table `OBJ_ADDRESS_TABLE`.

The `ORA-01400` error in Figure 3.11 indicates that Oracle used the `NOT NULL` constraint on the `OBJ_ADDRESS_TABLE.STREET` column. For comparison, a similar `NOT NULL` constraint is implemented on the relational table `REL_ADDRESS_TABLE` in Figure 3.12 that has the same attributes as `OBJ_ADDRESS_TABLE` from Figure 3.11. Once again, Oracle uses the `NOT NULL` constraint on the column street and does not allow a NULL value to be inserted into the `REL_ADDRESS_TABLE.`

The next test of Oracle's OR implementation of the `NOT NULL` constraint is testing constraints on an object table created from an object type that has an object type as one of its data types. Figure 3.13 shows implementation for the

```
SQL>create table OBJ_ADDRESS_TABLE of
ADDRESS_TYPE( street not null );

SQL>insert into OBJ_ADDRESS_TABLE values
(null,null,null,99999);

ORA-01400:  cannot insert NULL into (OBJ_USER
.OBJ_ADDRESS_TABLE.STREET)
```

Figure 3.11. Object table `NOT NULL` constraint

```
SQL>create table REL_ADDRESS_TABLE(
street varchar2(50) not null,
city varchar2(25),
state char(2),
zip number );

SQL >insert into REL_ADDRESS_TABLE values
(null,null,null,99999);

ORA-01400:  cannot insert NULL into
(RELATIONAL.REL_ADDRESS_TABLE.STREET)
```

Figure 3.12. Relational table `NOT NULL` constraint

`NAME_TYPE` and `PERSON_TYPE` created for use in the `OBJ_PERSON_TABLE` that has the object types `NAME_TYPE` and `ADDRESS_TYPE`.

However, it may be confusing to application developers or database administrators that the `NOT NULL` constraint on the type address from Figure 3.12 is not checked by Oracle if `NULL` is inserted for all attributes in the `ADDRESS_TYPE` as shown in Figure 3.14. The first insert statement in Figure 3.14 inserts null for all attributes of the `ADDRESS_TYPE` and the values are inserted into the table; however, the second insert statement inserts a null `ADDRESS_TYPE` and the values are not inserted into the table [39].

```
SQL >create type NAME_TYPE as object(
first_name varchar2(15),
middle_name varchar2(10),
last_name varchar2(15)
);

SQL >create type PERSON_TYPE as object(
name NAME_TYPE,
address ADDRESS_TYPE,
dob date );

SQL >create table OBJ_PERSON_TABLE of PERSON_TYPE
( constraint add_street_nn
check(address.street is NOT NULL) );

SQL >insert into OBJ_PERSON_TABLE values (
NAME_TYPE('George', 'D', 'Smith'),
ADDRESS_TYPE(null, 'city', 'CA', 99999),
'12-DEC-1973' );

ORA-02290:  check constraint
(OBJ_USER.OBJ_ADD_STREET_NN) violated
```

Figure 3.13. `OBJ_PERSON_TABLE NOT NULL` constraint

The first `ADDRESS_TYPE` insert is identified as an object with null values and the second `ADDRESS_TYPE` insert is identified as a null object [30]. Although by definition it may be clear why the first insert statement succeeded and the second insert failed this behavior should be understood by application developers and database administrators. To remedy confusion between null attributes of an object type and null object types the syntax in Figure 3.15 should be used for a `NOT NULL` constraint on an object type.

Another feature available in relational tables is the ability to add constraints to existing tables. This feature allows changes to a table without having to take the database offline, export data in the table, drop and recreate the table with the correct not null columns and import the data into the newly created table.

```
SQL >create table OBJ_PERSON_TABLE of PERSON_TYPE(
address not null);

SQL >insert into OBJ_PERSON_TABLE values (
NAME_TYPE('George', 'D', 'Doe'),
ADDRESS_TYPE(null, null, null, null),
'12-DEC-1973' );

1 row inserted

SQL >insert into OBJ_PERSON_TABLE values (
NAME_TYPE('George', 'D', 'Doe'),
null,
'12-DEC-1973' );
ORA-01400:  cannot insert NULL into
(OBJ_USER.OBJ_PERSON_TABLE.ADDRESS)
```

Figure 3.14. All `NULL` values with `NOT NULL` constraint allowed

As shown in Figures 3.16 and 3.17, this feature is easy to implement with both a relational and object tables.

Another feature available to database administrators is to drop `NOT NULL` constraints from a table column. Figure 3.18 illustrates how `NOT NULL` constraints are dropped from columns in a relational table. Likewise, the same syntax is used to drop a `NOT NULL` constraint in an object table (Figure 3.19).

I I. Unique constraint

Another type of integrity constraint available in Oracle‘s 10g relational database are unique constraints. A unique constraint specifies that a column must have a unique value. In Figure 3.20 the object table `OBJ_ADDRESS_TABLE` is created with the street column as unique. This guarantees the street column in the `OBJ_ADDRESS_TABLE` will have unique values.

```
SQL >create table OBJ_PERSON_TABLE of PERSON_TYPE(
        address not null),
        check ((address.street is not null) AND
        (address.city is not null) AND
        (address.state is not null) AND
        (address.zip is not null)) );
```

Figure 3.15. `NOT NULL` constraint on object type

```
SQL>alter table REL_ADDRESS_TABLE modify(
city varchar2(25) not null);

Table altered
```

Figure 3.16. Relational add `NOT NULL` constraint to existing table

However, as shown in Figure 3.21 if a column is not explicitly defined as `NOT NULL`, nulls can be inserted multiple times even if a there is a unique constraint on the attribute. This is the same functionality of nulls inserted in unique attributes in relational tables. This is remedied in an object table by using the unique not null constraint as shown in Figure 3.22.

The next type of unique constraint investigated is a unique constraint imposed on an object table created from an object type that has a nested object type. To show this the object types `ADDRESS_TYPE, NAME_TYPE,` and `PERSON_TYPE` are created (Figure 3.23). The object table `PERSON_TYPE` has the nested objec type `ADDRESS_TYPE, NAME_TYPE`.

As shown in Figure 3.24, a unique constraint is created on the address column in `OBJ_PERSON_TABLE`; however, the SQL statement fails with an ORA-02329 error because unique constraints are not allowed to be created on object type

```
SQL>alter table OBJ_ADDRESS_TABLE modify city not
null;

Table altered

SQL>insert into OBJ_ADDRESS_TABLE values(
'1111 street name',null,null,99999 );

ORA-01400:  cannot insert NULL into
(OBJ_USER.OBJ_ADDRESS_TABLE.CITY)
```

Figure 3.17. Add `NOT NULL` constraint to existing object table

```
SQL>alter table REL_ADDRESS_TABLE modify city null;

Table altered

SQL>commit;

SQL>insert into REL_ADDRESS_TABLE values(
'1111 street name',null,null,99999 );

1 row inserted
```

Figure 3.18. Drop `NOT NULL` constraint on existing relational table

column attributes [30]. Instead, unique constraints on an object type must be implemented by creating a unique constraint on every attribute in the object type as shown in Figure 3.25. The insert statements in Figure 3.26 indicate that the unique constraint is implemented correctly.

Another unique constraint function in Oracle for relational tables is the ability to add unique constraints to existing relational tables. This gives database users the ability to add constraints to a table after it is created [30] [39]. Without this functionality it would be necessary to export the table, recreate the table, and import in to the newly created table anytime a unique constraint was added to

```
SQL>alter table OBJ_ADDRESS_TABLE modify city null;

Table altered

SQL>insert into OBJ_ADDRESS_TABLE values(
'1111 street name',null,null,99999 );

1 row inserted
```

Figure 3.19. Drop `NOT NULL` constraint on existing object table

```
SQL>create table OBJ_ADDRESS_TABLE of ADDRESS_TYPE(
street unique );

SQL>insert into OBJ_ADDRESS_TABLE values(
'111 any street',null,null,'99999' );

1 row inserted
SQL>insert into OBJ_ADDRESS_TABLE values(
'111 any street',null,null,'99999');
ORA-00001:  unique constraint (OBJ_USER.SYS_C006924) violated
```

Figure 3.20. Create unique constraint on object table

a table. As shown in Figure 3.27, an object table can be altered to add a unique constraint in the same way that a unique constraint can be added to a relational table. This does not require recreating the table.

Another functionality available for relational tables is the ability to drop unique constraints. As shown in Figure 3.28, unique constraint `PERSON_UNIQUE` is dropped from the object table `OBJ_PERSON_TABLE`. Once the constraint is dropped, two identical objects are inserted into the `OBJ_PERSON_TABLE`. This is the same result if a unique constraint is dropped on a relational table.

```
SQL>insert into OBJ_ADDRESS_TABLE values( null,null,null,'99999');

1 row inserted

SQL>insert into OBJ_ADDRESS_TABLE values( null,null,null,'99999');


1 row inserted
```

Figure 3.21. Insert multiple NULL valued rows

```
SQL>create table OBJ_ADDRESS_TABLE of ADDRESS_TYPE(
street unique not null);

SQL>insert into OBJ_ADDRESS_TABLE values ( null,null,null,'99999');

ORA-01400:  cannot insert NULL into (OBJ_USER.OBJ_ADDRESS_TABLE.STREET)
```

Figure 3.22. Unique NOT NULL constraint

I I I . Primary key constraint

As discussed in Section 3.2, by default in Oracle REFs are used for object references instead of primary keys. Although REFs are used in an OR database to reference related object tables, primary keys can be created if they are needed. This section gives examples of how to create, add, and drop primary keys on object tables in Oracle.

The SQL syntax to create a primary key for an object table is shown in Figure 3.29. Null values are inserted into the object table OBJ_ADDRESS_TABLE to check if the primary key on the object table OBJ_ADDRESS_TABLE is similar to a primary key in a relational table. The null value inserted in the object table

```
SQL >create type ADDRESS_TYPE as object (
Street varchar2(50),
City varchar2(25),
State varchar2(2),
Zip varchar2(9) );

SQL>create type NAME_TYPE as object(
first_name varchar2(15),
middle_name varchar2(10),
last_name varchar2(15) );

SQL>create type PERSON_TYPE as object(
name NAME_TYPE,
address ADDRESS_TYPE,
dob date);
```

Figure 3.23. Create object types for `OBJ_PERSON_TABLE`

```
SQL>create table OBJ_PERSON_TABLE of PERSON_TYPE(
address unique);

ORA-02329:  column of datatype ADT cannot be unique or a primary
key
```

Figure 3.24. Create `OBJ_PERSON_TABLE.address` as unique

results in an ORA-01400 showing that the primary key constraint in an object table does not allow null values to be inserted for primary key attributes.

As shown in Figures 3.30 and 3.31, similar to primary key constraints in relational tables, primary keys in object tables can be added to existing tables [31]. Like unique constraints on the `OBJ_PERSON_TABLE` attribute address, primary keys can not be created on object types in an object table. Instead, a primary key should be create on the object type attributes [31].

In a relational table a primary key can also be dropped from an existing table (Figure 3.32). This may be necessary when a primary key changes and the old

```
SQL>create table OBJ_PERSON_TABLE of PERSON_TYPE(
constraint person_unique unique(
address.street,address.city,address.state,address.zip) );

Table created
```

Figure 3.25. Create unique constraint on OBJ_PERSON_TABLE

```
SQL>insert into OBJ_PERSON_TABLE values (
NAME_TYPE('george', 'D', 'Doe'),
ADDRESS_TYPE('111 any street',null,null,'99999'),
'12-DEC-1973');

SQL>insert into OBJ_PERSON_TABLE values (
NAME_TYPE('george', 'D', 'Doe'),
ADDRESS_TYPE('111 any street',null,null,'99999'),
'12-DEC-1973');

ORA-00001:  unique constraint (OBJ_USER.PERSON_UNIQUE) violated
```

Figure 3.26. Test PERSON_UNIQUE constraint on address type

primary key is not longer necessary. This same functionality is also available for object tables —dropping a primary keys is supported for an object table (Figure 3.33) [31].

IV. Foreign key constraint

Included in the SQL:2003 standard are reference data types (i.e. REF type) that allow object tables to be referenced from related object tables. This eliminates the need for foreign keys in object tables. However, the SQL standard gives database users the flexibility of adding foreign keys to object tables. This section shows examples of how to implement foreign keys in an ORDBMS.

```
SQL>alter table OBJ_PERSON_TABLE add constraint person_uniq
unique(address.street);
```

Figure 3.27. Add unique constraint to existing object table

```
SQL>alter table OBJ_PERSON_TABLE drop constraint person_unique;

Table altered

SQL>insert into OBJ_PERSON_TABLE values (
NAME_TYPE('george', 'D', 'Doe'),
ADDRESS_TYPE('111 any street',null,null,'99999'),
'12-DEC-1973');

1 row inserted

SQL>insert into OBJ_PERSON_TABLE values (
NAME_TYPE('george', 'D', 'Doe'),
ADDRESS_TYPE('111 any street',null,null,'99999'),
'12-DEC-1973');

1 row inserted
```

Figure 3.28. Test drop `PERSON_UNIQUE` constraint on `address.street`

Since columns cannot be added to object tables, as shown in Figure 3.34, the `PERSON_TYPE` is altered by adding a column person_id. The person_id attribute in the `OBJ_PERSON_TABLE` is used as a foreign key in `OBJ_ADDRESS_TABLE`. If there were any rows in the object table `OBJ_PERSON_TABLE` the column `person_id` would need to be populated with a valid value before it was used as a primary key.

In Figure 3.35, `OBJ_PERSON_TABLE` is altered with a primary key on person_id and the `OBJ_ADDRESS_TABLE` is created for storing secondary addresses. The object table `OBJ_ADDRESS_TABLE` should not have values if the `OBJ_PERSON_TABLE` address is null. This is implemented by a primary key in the `OBJ_PERSON_TABLE` and a foreign key in the `OBJ_ADDRESS_TABLE`. This implementation allows a per-

```
SQL>create table OBJ_ADDRESS_TABLE of ADDRESS_TYPE (
constraint address_pk primary key (street,city,state) );

SQL>insert into OBJ_ADDRESS_TABLE values(
'1111 any street',null,null,'99999') ;

ORA-01400:  cannot insert NULL into (OBJ_USER.OBJ_ADDRESS_TABLE.CITY)
```

Figure 3.29. Create primary key constraint

```
SQL>alter table REL_ADDRESS_TABLE add constraint
address_pk primary key (street,city,state,zip);

Table altered
```

Figure 3.30. Add primary key constraint to relational table

son to have many different addresses, yet not have to store them all in the OBJ_PERSON_TABLE. For example, a student may have an address at school, a home address, a parent's address, and an address while studying abroad. Instead of changing the primary address every time, the business rule for student addresses may be to have the home address as the primary address and add in other addresses as the student moves while at school or studies abroad.

The foreign key created in Figure 3.35 on the person_id column is tested by the insert statement in Figure 3.36. A valid row is inserted into the OBJ_PERSON_TABLE table. A row is then inserted into the OBJ_ADDRESS_TABLE table with a valid foreign key. Both these inserts succeed. In order to test the foreign key in the OBJ_ADDRESS_TABLE an invalid person_id is inserted into the table. Since this person_id does not exist in the OBJ_PERSON_TABLE the row should not be inserted in the the OBJ_ADDRESS_TABLE. Figure 3.36 shows that the invalid values are not inserted into the OBJ_ADDRESS_TABLE.

```
SQL>alter table OBJ_PERSON_TABLE add constraint
person_pk primary key (name.first_name,name.last_name,dob);
Table altered
```

Figure 3.31. Add primary key constraint

```
SQL>alter table REL_ADDRESS_TABLE drop primary key;

Table altered
```

Figure 3.32. Delete relational table primary key

Dropping the foreign key constraint, as shown in Figure 3.38, allows users of the database system to insert a row with a `person_id` that does not exist in the `OBJ_PERSON_TABLE` (since the `person_id` 1111111112 does not exist in the `OBJ_PERSON_TABLE`). This example shows that the foreign key on `person_id` is actually dropped.

V. Check constraints

Check constraints are used to implement business rules and constraints on data stored in a relational database. Since check constraints can be implemented in relational tables, the same functionality should be available when creating object tables. This section compares functionality of check constraints in relational tables to check constraints in object tables.

The types `ADDRESS_TYPE` and `PERSON_TYPE` are used in Figure 3.39 to create a check constraint on the attribute dob in the table `OBJ_PERSON_TABLE`. Testing shown in Figure 3.39 confirms that the check constraint `DOB_CK` is used when inserting into the `OBJ_PERSON_TABLE`.

```
SQL>alter table OBJ_PERSON_TABLE drop constraint person_pk;

Table altered
```

Figure 3.33. Delete object table primary key

```
SQL>alter table OBJ_PERSON_TABLE add person_id number;

ORA-22856:  cannot add columns to object tables

SQL>alter type PERSON_TYPE add attribute person_id number(10)
CASCADE;

Type altered

SQL>desc PERSON_TYPE;


Element_Name Type
------------ ------------
NAME         NAME_TYPE
ADDRESS      ADDRESS_TYPE
DOB          DATE
PERSON_ID    NUMBER(10)
```

Figure 3.34. Alter `PERSON_TYPE`

It should also be possible to create check constraints on a object table created from an object type that has a nested object type. As shown in Figure 3.40, the check constraint `state_ck` is created on the `address.state` attribute in the `OBJ_PERSON_TABLE` . A row is then inserted into the `OBJ_PERSON_TABLE` with an invalid value in the address.state attribute. As shown in Figure 3.40, the invalid row value is not inserted into the `OBJ_PERSON_TABLE`.

As with the previous constraints, it should be possible to add a check constraint to an existing table. In Figure **??**, the address.state is added to the

```
SQL>alter table OBJ_PERSON_TABLE add
constraint person_pk primary key (person_id);
Type altered
SQL>alter type ADDRESS_TYPE add attribute person_id number(10)
cascade;

Type altered

SQL>create table OBJ_ADDRESS_TABLE of ADDRESS_TYPE(
constraint address_fk foreign key(person_id)
referencing OBJ_PERSON_TABLE(person_id) );

Table created
```

Figure 3.35. Create `OBJ_ADDRESS_TABLE` with foreign key

`OBJ_PERSON_TABLE` to check that the state value is `'CA'`. The invalid value `'NV'` in the inserted statement results in an ORA-02290 error "check constraint violated". This shows that it is possible to add a check constraint to an existing object table.

As shown in Figure **??**, it is also possible to drop check constraints on object tables. After the check constraint is dropped, the insert statement that previously failed because of the address.state value `'NV'` is now inserted. The ability to drop check constraints allows database administrators to drop business rules for an object the same way the can be dropped in a relational table.

As shown in the previous five subsections, Oracle provides the same functionality for constraints on object tables as relational tables. Constraints can be created on object tables and attributes of object types in an object table. It is necessary for application developers and database administrators to be aware of using the `NOT NULL` constraint on object types in an object table. Instead of creating the `NOT NULL` constraint on the object type, it should be created on the

```
SQL>insert into OBJ_PERSON_TABLE values (
NAME_TYPE('george', 'D', 'Doe'),
ADDRESS_TYPE('111 L street','Los Osos','CA',93412,111111111),
'12-DEC-1978',
1111111111 );

1 row inserted

SQL>insert into OBJ_ADDRESS_TABLE values (
'111 A street','Los Osos','CA','93412',1111111111 );

1 row inserted

SQL>insert into OBJ_ADDRESS_TABLE values (
'111 A street','Los Osos','CA','93412',1111111112 );

ORA-02291:  integrity constraint (OBJECT_USER.ADDRESS_FK) violated
- parent key not found
```

Figure 3.36. Test foreign key constraint

object type attributes to guarantee that an object type with all null values can not be inserted (Figure 3.14).

It is also possible to create primary and unique constraints on an object type in an object table; however, instead of using the object type name the object type attributes should be used (Figure 3.28 and Figure 3.24).

3.2.6 Operations

In an RDBMS operators for manipulating tables are done using SQL and PL/SQL procedures. An object-relational database system must also provide operators for manipulating objects. These operators are provided through extensions to SQL to include operations on objects, vendor specific database functions, and PL/SQL procedures. These operators include functionality to create, insert,

```
SQL>create type ADDRESS_TYPE as object(
person_id varchar2(10),
street varchar2(25),
city varchar2(25),
state varchar2(2),
zip varchar2(9) );

SQL>create type NAME_TYPE as object(
person_id varchar2(10),
first_name varchar2(15),
middle_name varchar2(10),
last_name varchar2(15) );

SQL>create type PERSON_TYPE as object(
name NAME_TYPE,
address ADDRESS_TYPE);

SQL>create table OBJ_PERSON_TABLE of PERSON_TYPE(
constraint person_pk primary key(person_id));

SQL>alter table OBJ_PERSON_TABLE add
constraint person_pk foreign key(address.person_id)
referencing OBJ_PERSON_TABLE(person_id) ;
```

Figure 3.37. Create nested object foreign key

delete, and update objects, alter and drop object types in addition to functions to compare objects, convert user-defined types to another type, and reference and deference object references.

Operators provided by the SQL standard include select, update, alter, create, and delete statements that contain clauses used to manipulate object types, in addition to functions `CAST, TREAT, DEREF, VALUE, IS OF TYPE`.

The function `CAST` converts one built-in datatype or object type value into another built-in datatype or object type [31]. This is similar to casting in OOPLs. The function `TREAT` allows you to change the declared type of a SQL expression argument, as shown in Figure 3.55. `TREAT` is useful when converting a supertype

```
SQL>alter table OBJ_ADDRESS_TABLE drop constraint address_fk;
SQL>insert into OBJ_ADDRESS_TABLE values (
'111 A street','Los Osos','CA','93412',1111111112 ); 1 row inserted
```

Figure 3.38. Drop foreign key constraint

```
SQL>create table OBJ_PERSON_TABLE of PERSON_TYPE (
constraint dob_ck check(dob >'1-JAN-1900') );

Table created

SQL>insert into OBJ_PERSON_TABLE values (
NAME_TYPE('george', 'D', 'Doe'),
ADDRESS_TYPE('2 street', 'city', 'CA', '99999'),
'12-DEC-1865' );

ORA-02290:  check constraint (OBJ_USER.DOB_CK) violated
```

Figure 3.39. Create and test check constraint DOB_CK

to a more specialized subtype [31]. The DEREF function returns the values in an object instance referenced by a REF [31]. For example, the DEREF SQL statement `select DEREF(c.instructor) from OBJ_COURSE c` using the obj_course table schema in Figure 4.15 to return values in the OBJ_INSTRUCTOR object instances. In comparison, the SQL statement `select c.instructor from OBJ_COURSE c` returns the OID reference to the OBJ_INSTRUCTOR object.

The VALUE function treats a row as an object and returns the attributes for the object within a constructor for the object type. An example of using the VALUE function is shown In Figure 3.56. In addition, Figure 3.55 shows an example of the function IS OF VALUE which has the key words `IS OF TYPE (type_name)` or `IS OF (only type_name)` [31]. This function is useful when specialized subtypes are selected from a supertype table.

```
SQL>drop table OBJ_PERSON_TABLE;

Table dropped

SQL>create table OBJ_PERSON_TABLE of PERSON_TYPE (
constraint state_ck check(address.state in ('CA','WA','OR') );

Table created

SQL>insert into OBJ_PERSON_TABLE values (
NAME_TYPE('Jane', 'D', 'Doe'),
ADDRESS_TYPE('2 B street', 'Los Osos', 'CA', '93412'),
'12-DEC-1969' );

1 row inserted

SQL>insert into OBJ_PERSON_TABLE values (
NAME_TYPE('Jane', 'D', 'Doe'),
ADDRESS_TYPE('2 street', 'Las Vegas', 'NV', '99999'),
'12-DEC-1969' );

ORA-02290:  check constraint (OBJ_USER.STATE_CK) violated
```

Figure 3.40. Create check constraint

The last two SQL object operators are the map and order functions. SQL data types such as varchar2 or number have a predefined order that is used for comparison [30]. In order to perform object comparisons either a map or order method must be implemented [31]. A map method compares objects by mapping an object instance to a SQL scalar data type such as NUMBER shown in Figure **??**. Once a map method is defined, it can be used in less-than and greater-than comparisons in addition to `GROUP BY`, `UNION` and `ORDER BY` clauses.

An order method compares objects instances without mapping them to scalar SQL types [30]. Each order method has one parameter of the object type that is being compared as shown in Figure **??**. The return value from an order method is either -1, 0, or 1 which indicates less than, equal, or greater than respectively.

```
SQL>create table OBJ_PERSON_TABLE of PERSON_TYPE;

Table created

SQL>alter table OBJ_PERSON_TABLE add constraint state_ck
check (address.state in ('CA'));

Table altered

SQL>insert into OBJ_PERSON_TABLE values (
NAME_TYPE('Jane', 'D', 'Doe'),
ADDRESS_TYPE('2 street', 'Las Vegas', 'NV', '99999'),
'12-DEC-1969' );

ORA-02290:  check constraint (OBJ_USER.STATE_CK) violated
```

Figure 3.41. Add check constraint to existing object table

To avoid conflicts, only a map method or an order method can be defined for an object type, not both.

3.2.7 Relationships

One of the main differences between a RDBMS and an ORDBMS is how relationships are managed. A relational database has primary and foreign keys that relate tables to one another; however, in an OR database, relationships between objects are related by default with OIDs. Neither human users nor query languages ever modify or assign OIDs; rather, the ORDBMS automatically assigns the objects an OID, which are only seen internally. An OID is unique to an object for the object's lifespan and operates independently from its attributes' values [12]. Since relationships in an object-relational database are based on values that cannot change, an OID automatically provides entity integrity [24]. Although OID's result in entity integrity, it does not guarantee that object in-

```
SQL>alter table OBJ_PERSON_TABLE drop constraint state_ck;

Table altered

SQL>insert into OBJ_PERSON_TABLE values (
NAME_TYPE('Jane', 'D', 'Doe'),
ADDRESS_TYPE('2 street', 'Las Vegas', 'NV', '99999'),
'12-DEC-1969' );

1 row inserted
```

Figure 3.42. Drop object table check constraint

stance values will be unique [30]. Using an OID to represent relationships, versus using primary and foreign keys, is thought to result in better performance for OR databases since joins are not needed to access data in related tables [15] [22]. Performance testing for object reference versus relational joins is given in Chapter 4. Besides OIDs, other relationships that exist in the OR model are inheritance, association, and aggregation [24].

3.2.7.1 Inheritance

Inheritance allows developers to take objects that have similar attributes and methods and abstract out the similarities —creating subclasses that inherit from a superclass. All instances of subclasses are also instances of the superclass, and all properties of the superclass are properties of the subclass [24] [30]. The relationship between a subclass and an inherited class is called an IS—A relationship. For example, in Figures 3.49 and 3.51, the person and student types have an inheritance relationship.

There are several types of inheritance: single, multiple, union, mutual exclusion, partial, repeated, and selective inheritance [12] [38]. The primary in-

```
SQL> create type employee_type as object(
       start_date date,
       manager NAME TYPE,
       depart\_id varchar2(5),
       salary number,
       MAP MEMBER FUNCTION income RETURN NUMBER);

CREATE TYPE BODY income AS
MAP MEMBER FUNCTION income RETURN NUMBER IS
       BEGIN
         RETURN salary;
    END income;
END;
```

Figure 3.43. MAP method

heritance types software engineers use when developing an application are single inheritance and multiple inheritance. Single inheritance is a subclass that inherits from only one superclass. In contrast, multiple inheritances imply that a subclass inherits from more than one superclass [37]. ORDBMS object inheritance allows direct mapping from application objects to persistent database objects. This is identified as a main shortcoming of the relational model by application developers [13] [14].

Besides direct mapping from application objects to database objects, inheritance allows software engineers to design an overall better application by increasing applications expressiveness, convenience, and maintainability [5] [17]. For instance, changes made in a superclass, will automatically be propagated to all subclasses [26]. This increases maintainability in OR databases compared to a relational database, where database administrators or applications must update each related table [37]. Moreover, inheritance provides convenience and expres-

```
SQL> create type employee_type as object(
       start_date date,
       manager NAME TYPE,
       depart_id varchar2(5),
       salary number,
       ORDER MEMBER FUNCTION income (e employee_type) RETURN INTEGER);

       CREATE TYPE BODY income AS
       MAP MEMBER FUNCTION income (e employee_type) RETURN INTEGER IS
         BEGIN
          IF salary < e.salary then
            RETURN -1;
          IF salary >  e.salary then
            RETURN 1;
          ELSE
           RETURN 0;
         END IF;
       END income;
       END;
```

Figure 3.44. ORDER method

siveness in a database system by creating high coupling between the database and the application accessing it. These are all desirable and well-developed application features that will provide valuable throughout the lifetime of the object-oriented application.

3.2.7.2 Association relationships

Association relates two or more independent objects creating a "membership-of" relationship between them [18]. An association relationship is often referred to as a grouping or partitioning mechanism [16]. For example, an association relationship would be between the employee and corporation objects since an employee has a membership relationship with their employer. Association rela-

tionships are binary and can be one-to-one, one-to-many, or many-to-many [37]. Examples of association relationships include `worksFor, memberOf, worksIn,` and `presidentOf`. SQL: 2003 supports the binary relationships of one-to-one, one-to-many, and many-to-many [35]. It also specifies that an OR database must maintain referential integrity of these relationships [35]. For instance, if an employee object is deleted from the previous example, the relationship paths referencing the object in the corporation object must also be deleted.

```
SQL> create type PERSON_TYPE(
       person_id varchar2(10),
         name NAME_TYPE);

SQL> create type COURSE_TYPE(
       course_id varchar2(10),
       course_name varchar2(30) );

SQL> create table STUDENT_TB of PERSON_TYPE(
       person_id primary key);

SQL> create table COURSE_TB of COURSE_TYPE(
       course_id primary key);

SQL> create table ENROLLS_IN(
       student REF PERSON_TYPE,
       course  REF COURSE_TYPE);
```

Figure 3.45. Association many-to-many

3.2.7.3 Aggregation relationships

In addition to inheritance and association, OR databases also provide aggregation relationships between objects. In an OR database, aggregation is the grouping of objects that have the WHOLE-PART relationship [37]. This rela-

```
SQL> create type PERSON_TYPE(
       person_id varchar2(10),
       name NAME_TYPE);

SQL> create type COURSE_TYPE(
       course_id varchar2(10),
       course_name varchar2(30,
       course_instructor REF PERSON_TYPE);

SQL> create table INSTRUCTOR_TB of PERSON_TYPE(
       person_id primary key);

SQL> create table COURSE_TB of COURSE_TYPE(
       course_id primary key);
```

Figure 3.46. Association one-to-many

tionship is a collection of objects (subparts) that make up a whole collection [18]. For example, in manufacturing an airplane, many objects make up the final artifact being created; an airplane is made of body, wings, rudders, flaps, engines, instruments, wheels, and seats, just to name a few parts. If an engineer uses an OR database with aggregation to model an airplane, the airplane is viewed as a single object and only exists if each object that is part of the whole plane exists [24] [26].

3.2.8 Object-relational encapsulation

Encapsulation, a feature of OOPLs, is the ability to store both data and methods together in an object. Encapsulation in an ORDBM provides logical data independence by allowing modifications to object data methods without having to make changes to applications that access them [7] For example, Fig-

```
SQL> create type OFFICE_TYPE(
       office_id varchar2(10),
       building_num varchar2(10) );

SQL> create type PERSON_TYPE(
       person_id varchar2(10),
       name NAME_TYPE,
       office REF OFFICE_TYPE);

SQL> create table OFFICE_TB of OFFICE_TYPE(
       office_id primary key);

SQL> create table INSTRUCTOR_TB of PERSON_TYPE(
       course_id primary key);
```

Figure 3.47. Association one-to-one

ure ?? has a person object with the data attributes `name, address, id,` and the method `who_am_i()`all stored in the same object. If developers modified the method `who_am_i()` to also return age, the application calling `who_am_i()` would not have to be modified. Encapsulation in ORDBMSs also reduces the amount code applications developers must create (see Section 3.1.4). In most companies Database Administrators (DBAs) that manage data and developers using the data in applications work together closely. In ORDBMSs application developers and DBAs have the flexibility to choose whether methods should be stored with the data and managed by DBA's or created in the application and managed by application developers. One advantage to storing methods in ORDBMSs is that database methods can return objects containing multiple data values instead of applications making multiple selects or joins to get the same data. This reduces the number of calls from the application to the database. A disadvantage to us-

```
SQL> create type QUESTION_TYPE(
       quest_id varchar2(10),
       question varchar2(256) );

SQL> create type QUESTION_TB_TYPE as
       table of QUESTION_TYPE;

SQL> create type HOMEWORK_TYPE(
       HW_id varchar2(10),
       questions QUESTION_TB_TYPE);

SQL> create type HOMEWORK_TB_TYPE

SQL> create table COURSE_MATERIAL(
       course_id primary key,
       homework_assign HOMEWORK_TB_TYPE)
       NESTED TABLE homework_assign STORE AS HW_TB
         (NESTED TABLE questions STORE AS Q_TB);
```

Figure 3.48. Aggregation using nested tables

ing database methods to manipulate database objects is encountered if database methods change names or new methods are added.

3.2.9 Object-relational abstraction

Abstraction, as defined in Section 2.3.2, is a feature of OOPLs that is concerned with having the ability to isolate aspects of a problem or certain data elements that are important while suppressing the information that is unimportant to solving the problem. This section compares OOPLS abstraction to abstraction in an ORDBMS. Abstraction in OOPLs allows developers to break problems into smaller problems by using public data with methods that access public data. For example, developers may need to create software to close bank

accounts. Instead of writing one long algorithm in a file to solve the problem, OOPLs provide functions that allow developers to break the problem into sub problems `getName(), getBalance()` for example.

Abstraction is also available to DBAs through the use of PL/SQL procedures and functions and methods stored with an object as in Figure 3.49. This allows designers to abstract out unimportant information by creating functions that have a single purpose.

3.2.10 Object-relational polymorphism and overriding

One advantage to using OOPLs is polymorphism —the ability to have a subtype to use a supertype. This subsection compares OOPL polymorphism (see Section 2.3.5) to polymorphism in an ORDBMS. SQL:2003 uses `under` in the create type statement in order to implement type polymorphism (see Figure 3.51). As a subtype of `PERSON_TYPE, STUDENT_TYPE` inherits all attributes and methods declared in or inherited by `PERSON_TYPE`.

After a student row is inserted in Figure 3.54, notice that the employee attributes and student attributes are not returned in the SQL statement in Figure 3.55. This is for a very good reason. The attributes salary, major, minor etc are not valid identifiers in the `PERSON_TABLE`. Instead, SQL provides the `TREAT()` function (see Figure 3.56) to retrieve the attributes salary, major, minor etc from the `PERSON_TABLE`. Besides providing object polymorphism through inheritance, SQL also implements OOPL polymorphism with object views.

Another feature of OOPL polymorphism is the ability to override super type member methods with subtype methods [30]. To compare OOPL overriding functionality, the types `PERSON_TYPE, EMPLOYEE_TYPE,` and `STUDENT_TYPE` are

```
SQL>alter type PERSON_TYPE add
member function who_am_i return varchar2
cascade;

SQL>alter type PERSON_TYPE
add attribute id number(10) not final cascade;
Type altered

SQL>create or replace type body PERSON_TYPE as
member function who_am_i return varchar2 is
     begin
     return 'PERSON';
     end;
end;
```

Figure 3.49. Create PERSON_TYPE

```
SQL>create type EMPLOYEE_TYPE under PERSON_TYPE(
start_date date,
manager NAME_TYPE,
depart_id varchar2(5),
salary number
member function who_am_i return varchar2 );
```

Figure 3.50. Create EMPLOYEE_TYPE

altered to override the PERSON_TYPE member methods. Before a method can be overridden, it must be declared as `not final` in the super type, as shown in Figure 3.49. In Figure 3.58, the function `who_am_i()` inherited from the PERSON_TYPE is overridden by the type STUDENT_TYPE.

Next, as shown in Figures 3.59 and 3.60, the type EMPLOYEE_TYPE is altered to also override the `who_am_i()` method inherited from PERSON_TYPE. Now both the EMPLOYEE_TYPE and the STUDENT_TYPE have overridden the `who_am_i()` method. Since the method is overridden, if the method is called for a STUDENT_TYPE or

```
SQL>create type STUDENT_TYPE under PERSON_TYPE(
major varchar2(4),
minor varchar2(4),
gwr_score number(3),
start_date date );
```

Figure 3.51. Create `STUDENT_TYPE`

```
SQL>create table PERSON_TABLE of PERSON_TYPE (
constraint person_pk primary key(id) );

Table created
```

Figure 3.52. Create `PERSON_TABLE`

`EMPLOYEE_TYPE`, the method should return `'STUDENT'` or `'EMPLOYEE'` respectively [30].

As shown in Figure 3.61, when the method `who_am_i()` is used, the correct value is returned depending on what type called the method. This is the same functionality of OOPL when overriding an inherited method [16].

Similar to OOPLs, there are restrictions in an ORDMS for overriding a supertype. The overridden method must have the same default parameter values and static supertype methods can not be overridden.

3.2.11 Access control

Access control is ability for data administrators to have control over what data can be accessed by each user. This is a feature of a RDBMS (see Section 2.3.1). This subsection compares access control in a RDBMS to access control in an ORDBMS.

```
SQL>insert into PERSON_TABLE values (
EMPLOYEE_TYPE(NAME_TYPE('Employee','D','Jones'),
ADDRESS_TYPE('12345 A Street','Los Osos','CA','93412'),
'01-JAN-80',
1234567890,
'01-JAN-07',
NAME_TYPE ('Manager', 'A', 'Jones'),
11111,
37000) );
```

Figure 3.53. Insert EMPLOYEE_TYPE

```
SQL>insert into PERSON_TABLE values (
STUDENT_TYPE(
NAME_TYPE('Student','B','Jones'),
ADDRESS_TYPE('1333 B Street','Los Osos','CA','93412'),
'01-JAN-85',
1234567899,
'CSC',
null,
8,
'01-JAN-07') );
1 row inserted
```

Figure 3.54. Insert STUDENT_TYPE

Database administrators can implement access control in a relational database by granting and revoking privileges to database users [12]. The granted privileges allow users to create or view database objects for example tables, packages, views. Granting or revoking privileges determines what database objects each user has access to and what changes they can make. Likewise in Oracle's 10g ORDBMS, privileges are used to implement access control. Database administrators can grant or deny access to objects, object tables, object types, object view,packages, procedures, views, and tables to allow or deny access to database users.

```
SQL>select * from PERSON_TABLE;
NAME(FIRST_NAME, MIDDLE_NAME, LAST_NAME)
----------------------------------------
NAME_TYPE('Employee', 'D', 'Jones')
NAME_TYPE('Student', 'B', 'Jones')


ADDRESS(STREET, CITY, STATE, ZIP)
---------------------------------
ADDRESS_TYPE('12345 A Street', 'Los Osos', 'CA', '93412')
ADDRESS_TYPE('1333 B Street', 'Los Osos', 'CA', '93412')

 DOB        ID
------     --------
01-JAN-80   1234567890
01-JAN-85   1234567899
```

Figure 3.55. Select from PERSON_TABLE

3.2.12 Transaction Management

In a relational database transaction management provides concurrency control and fault tolerance by using commit, rollback, and locks(Section 2.1) [30]. Before using an ORDBMS to store persistent application data, database administrators and application developers must know that transactions in an ORDBMS also provide concurrency control and fault tolerance. The SQL:2003 standard specifies the use of commit, rollback, and locks for transaction managment. Since the three major ORDBMS vendors —Oracle, IBM and Microsoft —have extended their relational databases to include objects following the SQL:2003 standard, they all provide transaction management.

```
SQL>select
treat(value(p) as PERSON_TYPE).name.first_name ‘‘First’’,
treat(value(p) as PERSON_TYPE).name.last_name ‘‘Last Name’’,
treat(value(p) as EMPLOYEE_TYPE).salary ‘‘Salary’’,
treat(value(p) as EMPLOYEE_TYPE).start_date ‘‘start dt’’,
treat(value(p) as EMPLOYEE_TYPE).depart_id ‘‘Dept’’,
treat(value(p) as EMPLOYEE_TYPE).manager.first_name ‘‘Mgr First’’,
treat(value(p) as EMPLOYEE_TYPE).manager.last_name ‘‘Mgr Last’’
from PERSON_TABLE p
where value(p) is of (only EMPLOYEE_TYPE);

First  Last Name  Salary  Start_dt Deptment Mgr_First Mgr_Last
-------------------------------------------------------------
Worker Jones     37000    1/1/2007 11111    Manager   Jones
```

Figure 3.56. Select EMPLOYEE_TYPE

```
SQL>alter type STUDENT_TYPE add
overriding member function who_am_i return varchar2
cascade;

SQL>create or replace type body STUDENT_TYPE as

overriding member function who_am_i return varchar2 is
    begin
     return ‘STUDENT‘;
    end;
end;
```

Figure 3.57. Add method to STUDENT_TYPE

```
SQL>desc STUDENT_TYPE

Element_Name  Type
------------  ------------
NAME          NAME_TYPE
ADDRESS       ADDRESS_TYPE
DOB           DATE
ID            NUMBER(10)
MAJOR         VARCHAR2(4)
MINOR         VARCHAR2(4)
GWR_SCORE     NUMBER(3)
START_DATE    DATE
who_am_i      FUNCTION
```

Figure 3.58. Add method to STUDENT_TYPE

```
SQL>alter type EMPLOYEE_TYPE add
overriding member function who_am_i return varchar2
cascade;
create or replace type body EMPLOYEE_TYPE as
overriding member function who_am_i return varchar2 is
    begin
    return 'EMPLOYEE';
    end;
end;
```

Figure 3.59. Add method to EMPLOYEE_TYPE

```
SQL>desc EMPLOYEE_TYPE

Element_Name Type
_____________ _____________
NAME          NAME_TYPE
ADDRESS       ADDRESS_TYPE
DOB           DATE
ID            NUMBER(10)
START_DATE    DATE
MANAGER       NAME_TYPE
DEPART_ID     VARCHAR2(5)
SALARY        NUMBER
who_am_i      FUNCTION
```

Figure 3.60. Describe EMPLOYEE_TYPE

```
SQL >select
treat(value(p) as PERSON_TYPE).name.first_name ‘‘First’’,
treat(value(p) as PERSON_TYPE).name.last_name ‘‘Last’’,
p.who_am_i() who
from PERSON_TABLE p;

First Last Who
----------------
Employee Jones EMPLOYEE
Student Jones STUDENT
```

Figure 3.61. Use overridden who_am_i methods

Chapter 4

Performance Comparison Between ORDBMS and RDBMS

In order to determine the cost of using an ORDBMS for persistent object storage, performance testing was done using a database management system that can store object-relational and relational data.

The machine used for testing was an Intel Pentium 3.0GHz with 1G of memory. All tests were done on a cold database by restarting the machine before each test to avoid any data caching problems. The object-relational and relational testing was done using the data schema in Figure 4.1. Similar to BORD and 007 benchmark testing, SQL insert, select, update, delete, and join statements were tested [6] [28]. In order to determine if the number of attributes in an object type increased or decreased performance for inserts, deletes, and updates compared to a relational table, each test was done varying the attributes starting with four and ending with sixteen attributes. Primary keys with indexes were created for each table to follow best practices in the testing scenario.

Results from testing found that overall the cost of using an object table compared to a relational table is minimal for inserts, selects, updates and deletes. However, it is substantially faster to use object reference to access related tables instead of using relational joins.

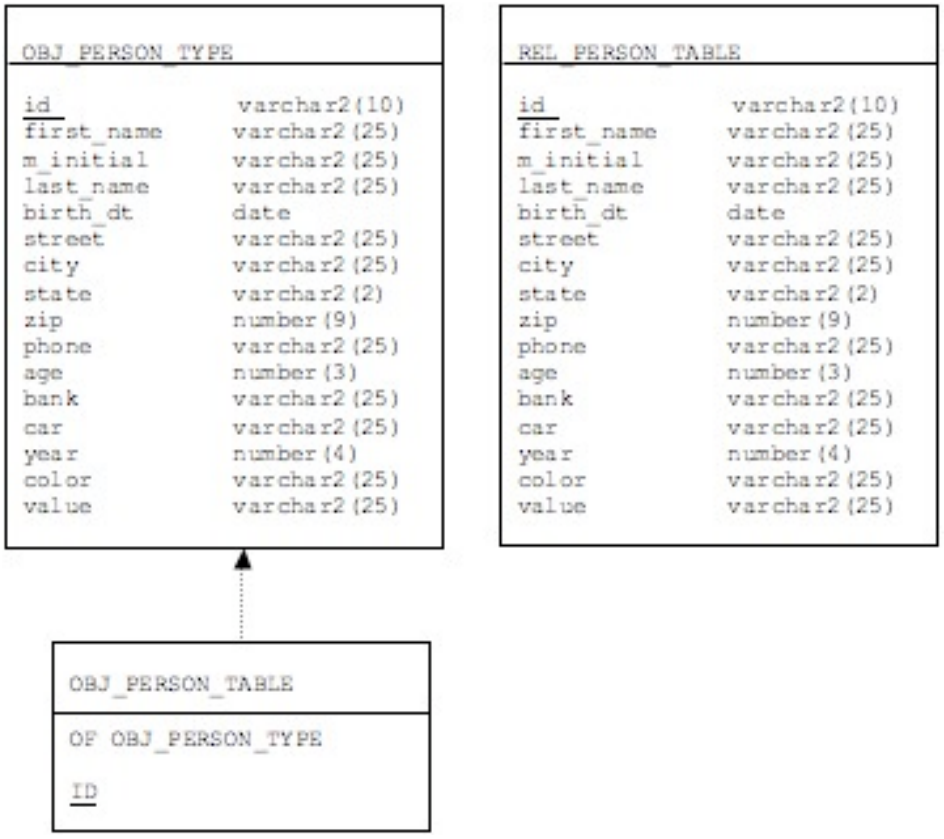


Figure 4.1. Relational and Object table schemes

4.1 Relational table insert vs. Object table insert

For each insert test results where recorded for 10, 100, 1,000, and 10,000 rows inserted. Performance results for inserting 1000 rows are shown in Figure 4.2. Insert testing results for inserting 10, 100, 10,000 rows are in Appendix B. Results

for inserting 10, 100, 1000, 10,000 rows into an object table with one type and four to sixteen attributes show that on average it is ten percent faster to insert into an object table than a relational table.

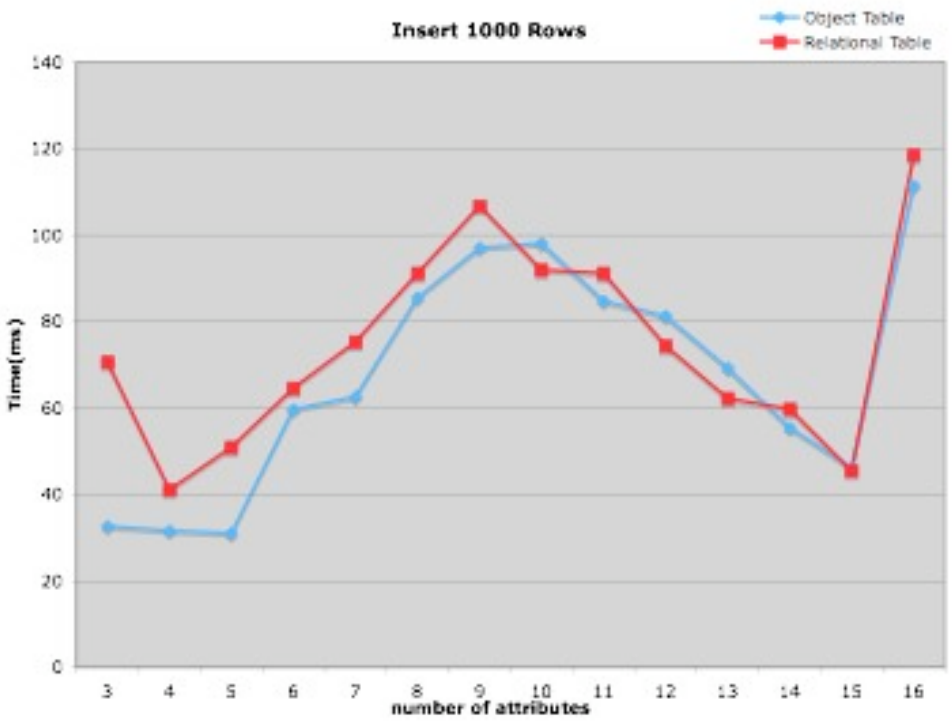


Figure 4.2. Relational vs. Object table insert

4.2 Relational table select vs. Object table select

Selects were tested using the `person_table` schemas in Figure 4.1. For each select test, results where recorded for 10, 100, and 1,000 rows inserted. Performance results for inserting 1000 rows are shown in Figure 4.3. Insert testing results for inserting 10, 100 rows are in Appendix B. The object and relational select statements are shown in Figures 4.4 and 4.5. Results for selecting 10, 100,

and 1000 rows from an object table with one type and four to sixteen attributes show that on average it is five percent slower to select from an object table than a relational table.

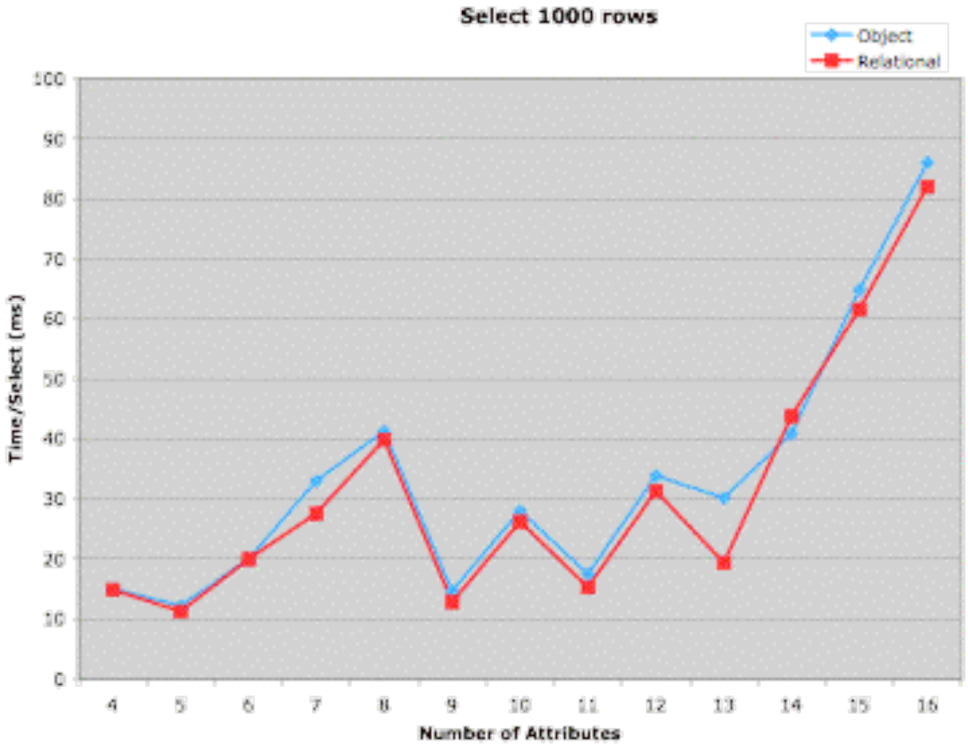


Figure 4.3. Relational vs. Object table select

```
SQL >select * from obj_person_table;
```

Figure 4.4. Object select statement

```
SQL >select * from rel_person_table;
```

Figure 4.5. Relational select statement

4.3 Relational table update vs. Object table update

Performance testing for updates in the `person_table` object and relational table was done using the same number of attributes as inserts and deletes. In the same manner, the number of updates tested were 10, 100, and 1000 for each number of attributes tested. The SQL update statement updated every row in the `OBJ_PERSON_TABLE` and `REL_PERSON_TABLE` tables (Figures 4.6 and 4.7). Each test was run ten times and the average of the results are recorded. Results for updating 1000 rows are shown in Figure 4.8. Results for updating 10, 100 and 1000 rows show that on average it is two percent slower to update an object table then a relational table.

```
SQL >update obj_person_table
set middle_name='S' where first_name='L';
```

Figure 4.6. Object update statement

```
SQL >update rel_person_table
set middle_name='S' where first_name='L';
```

Figure 4.7. Relational update statement

4.4 Relational table delete vs. Object table delete

Performance testing for deletes was done using the `person_table` schema in Figure 4.1 varying the number of attributes in both the object table and the relational table from four to sixteen attributes. In addition, the number of rows

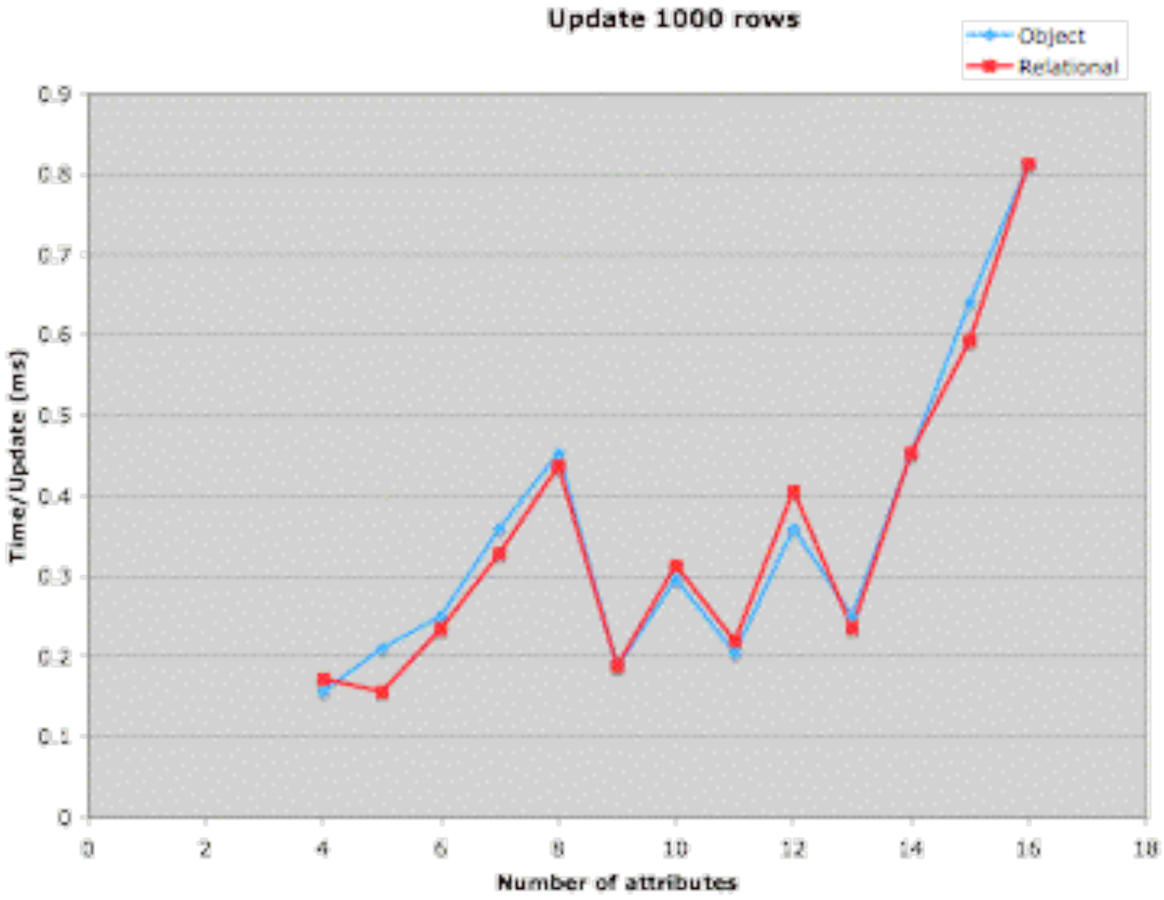


Figure 4.8. Relational vs. Object table update

in the table deleted was tested using 10, 100, and 1000 rows. The SQL statements used when testing deletes are shown in Figures 4.9 and 4.10. Each test was run ten times and the average of the results are recorded. Figure 4.11 shows performance testing results for deleting 100 rows from the object and relational `person_table` varying the number of attributes from four to sixteen.

Comparing the test results for object and relational table deletions shows that on average there is a two percent performance overhead to delete from an object compared to a relational table. These results were expected because the database vendor for the object and relational database is the same and deletes from relational tables are suspected to be the same as deletes from object tables.

```
SQL >delete from obj_person_table;
```

Figure 4.9. Object delete SQL statement

```
SQL >delete from rel_person_table;
```

Figure 4.10. Relational delete SQL statement

4.5 Relational Joins vs. Object References

A more interesting test is the performance difference between using a relational join and object references to access data in related tables. Testing was done using small, medium, and large data sets for the object and relational course and instructor schemas from Figure 4.14. The small data set had 1,200 and 3,500 rows in the instructor and course tables respectively. The medium data set had 17,000 and 50,000 rows in the instructor and course tables respectively. The large data set had 35,000 and 100,000 rows in the instructor and course tables respectively.

First, the relational join and object reference queries shown in Figures 4.12 and 4.13 where tested. Results for the small data set showed that it was one percent slower to use the object reference query in Figure 4.13 than the relational join query in Figure 4.12. In the medium size data set the relational join query in Figure 4.12 was twenty-eight percent slower then the object reference query in Figure 4.13. Finally, for the large data set the object reference query in Figure 4.13 was eight percent slower than the relational join query in Figure 4.12.

After analyzing the queries in Figure 4.12 and Figure 4.13, it was found that by retrieving all attributes from the `COURSE_TABLE` and also using the dot notation to retrieve all rows from the `INSTRUCTOR_TABLE` in the object tables, the object

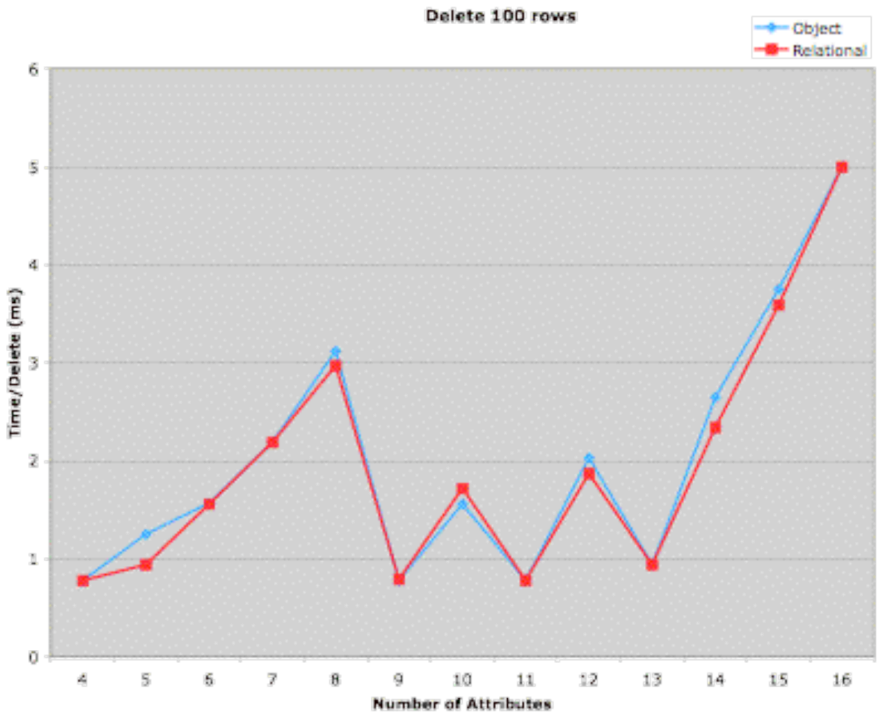


Figure 4.11. Relational vs. Object table deletes

table was accessed multiple times for every row —one access to get the `OID`, and one table access for every attribute for each row. In contrast, using the `DEREF` Oracle operator to retrieve referenced objects in the `INSTRUCTOR_TABLE` resulted in only access the referenced object table once for every `OID` in the `COURSE_TABLE`.

```
SQL>select c.*, i.*
from rel_course_table c, rel_instructor_table i
where c.person_id=i.person_id;
```

Figure 4.12. Relational join query

Performance testing for the object reference query in Figure 4.15 using the `DEREF` operator for the large data set was fifty-two percent faster then using the relational join query in Figure 4.12. The difference in retrieving references using a the `DEFRE` operator compared to retrieving table attributes using the object

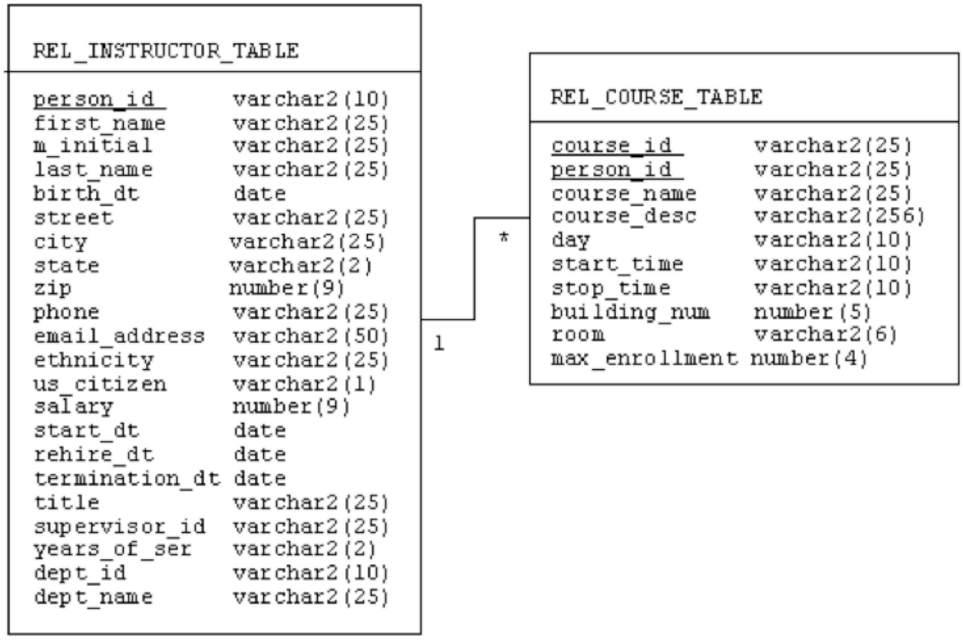


Figure 4.13. `Relational` schema

reference dot notation is very substantial. After realizing this difference, performance tests were also done to determine if the number of objects referenced using the `DEREF` operator in an object table resulted in better or worse performance than joining the same number of relational tables.

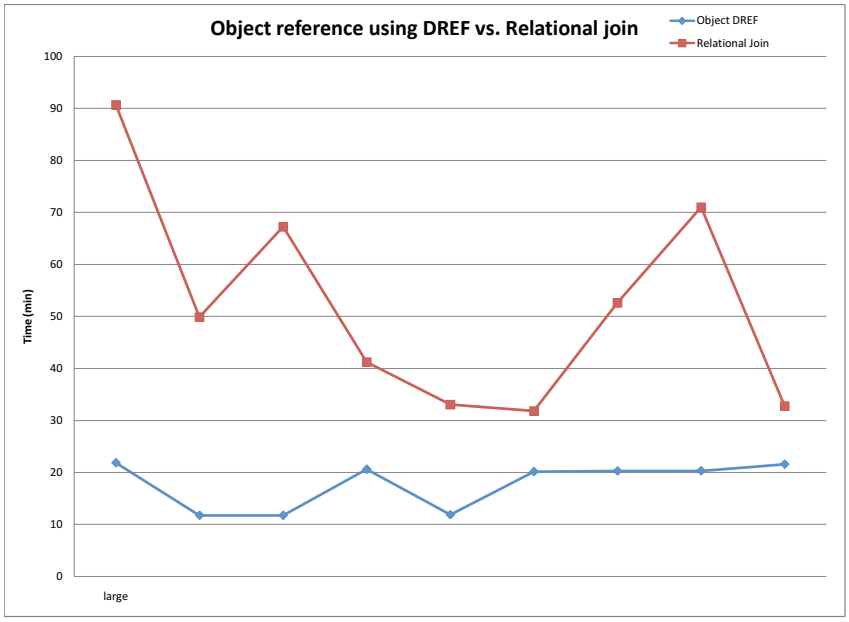
The object and relational schemas from Figure 4.16 were used to test multiple object references versus joining multiple relational tables with small, medium, and large data sets. The small data set had 1,200 in the `DEPT_TABLE` and 3,500 rows in the `PERSON_TABLE, ADDRESS_TABLE`, and `NAME_TABLE` tables The medium data set had 17,000 in the `DEPT_TABLE` and 50,000 rows in the `PERSON_TABLE, ADDRESS_TABLE`, and `NAME_TABLE` tables. The large data set had 35,000 in the `DEPT_TABLE` and 100,000 rows in the `PERSON_TABLE, ADDRESS_TABLE`, and `NAME_TABLE` tables. The queries in Figures 4.17 and 4.18 were used to test relational join of

```
SQL>select c.*,c.instructor.first_name,
c.instructor.m_initial,
c.instructor.last_name,
c.instructor.birth_dt,
c.instructor.street,
c.instructor.city,
c.instructor.state,
c.instructor.zip,
c.instructor.phone,
c.instructor.email_address,
c.instructor.ethnicity,
c.instructor.us_citizen,
c.instructor.salary,
c.instructor.start_dt,
c.instructor.rehire_dt,
c.instructor.termination_dt,
c.instructor.title,
c.instructor.supervisor_id,
c.instructor.years_of_service,
c.instructor.dept_id,
c.instructor.dept_name from course c;
```

Figure 4.14. Object REF query

four tables versus using the `DEREF` operator to retrieve objects from four object tables.

Results from testing Figure 4.17 and 4.18 queries using the small data set showed that on average over ten runs the object deref was eighteen percent slower then the relational join of four tables. Using the same two queries with the medium and large data sets, the object deref for four tables was also eighteen percent slower then relational joins. Considering that with one dref the performance increase was fifty-two percent faster then a relational join and with four object tables the performance decrease was eighteen percent it is clear that as the number of references increase there is a decrease in performance to access the referenced object tables.


Object reference using DREF vs. Relational join
Object DREF
Relational Join
Time (min)
100
90
80
70
60
50
40
30
20
10
0
large

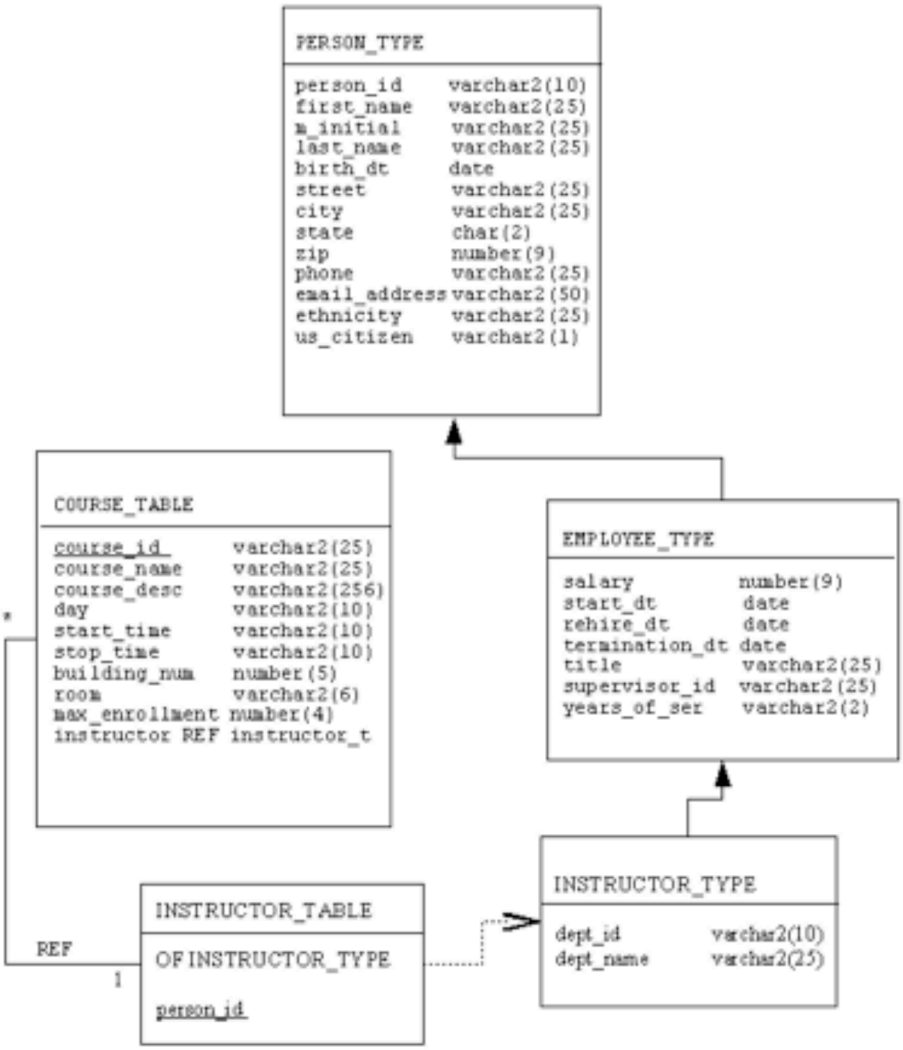
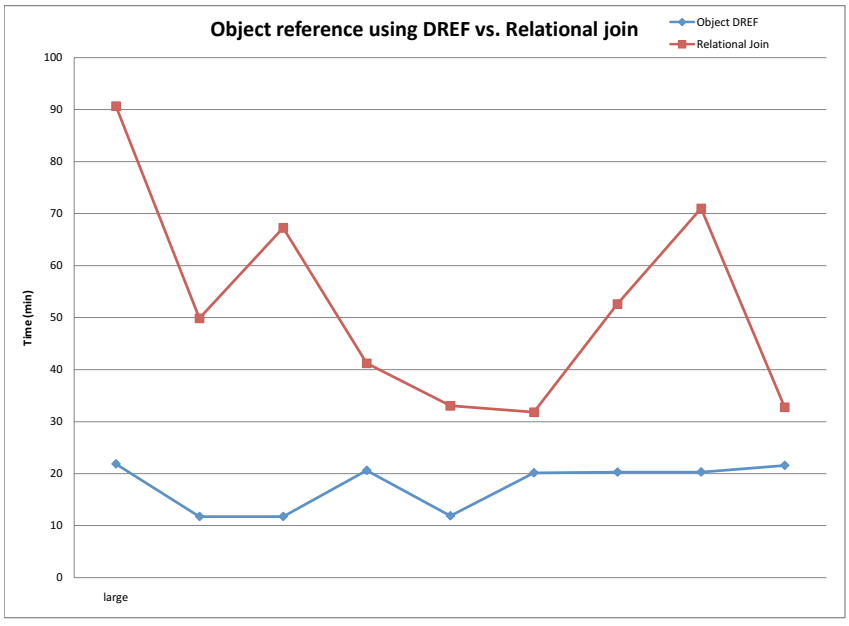


Figure 4.15. `Object` schema

```
SQL>select c.course_id,
c.course_id,
c.course_desc,
c.day,
c.start_time,
c.stop_time,
c.building_num,
c.room,
c.max_enrollment,
deref(c.instructor)
from course c;
```

Figure 4.16. Object DREF query



```
SQL>select p.*, a.*, n.*,d.*
from rel_person p, rel_name n, rel_address a, rel_dept d
where p.person_id = a.person_id
and p.person_id = n.person_id
and p.dept_id =d.dept_id;
```

Figure 4.17. Relational four table join

```
SQL>select p.person_id,
p.dob,
deref(p.name),
deref(p.address),
deref(p.dept)
from obj_person_dref_table;
```

Figure 4.18. Object table DREF with four tables

Chapter 5

Future Work

This thesis has presented background information for relational database systems and OOPLs. In addition, performance testing has been presented for inserts, selects, updates, and deletes for simple object-relational tables. If more time was available, testing done in Chapter 4 could be extended to include testing of more complex object-relational tables including object tables with multiple object references to other object tables. Comparing performance testing of retrieving objects from object-relational tables with one REF with object-relational tables with three REF's, it is clear that performance decreases as the number of object REF increase. By testing inserts, deletes, and updates using more complex object schemas this overhead could be better quantified.

Testing could also be extended to included linear references between multiple tables. For this test, an object-relational table would reference a second object-relational (object table schema in Figure 4.14) and the second table would reference a third object-relational table. In addition to testing linear references in object-relational tables, inserts, updates, deletes, and selects could also be tested for linear reference schemas. These tests would provide helpful information to

application developers as they determine the best design for applications using ORDBMSs.

Information provided in this text has been from a database point of view. No information has been given for how an application would access and store object-oriented data in an ORDBMS. This information is very important to software engineers as they make design decisions of how to best store and retrieve OR data. Further work could investigate the mapping from object-oriented applications to ORDBMS storage and retrieval. This information is vital because at this time it is not available to engineers as they design object-oriented applications.

Chapter 6

Conclusion

This thesis has provided important information by presenting reasons why traditional relational databases are inadequate for object persistence, a comparison of object-relational and relational databases. and performance testing results for select, update, delete, insert, and object references for an OR database. At this time, no other work has investigated the potential increase or decrease of performance from using OR databases.

The advantages presented in this thesis from using an ORDBMS to store object-oriented application data are impedance mismatch removal, ease of modeling real-world objects and relationships, ability to create user-defined types, persistent object encapsulation, and object referencing. In addition, performance testing results presented in Chapter four show that the decrease in performance for OR selects, deletes, inserts, updates is very minimal, and the increase in performance for object tables with one object reference is fifty-two percent faster than using a relational join.

In addition to the advantages presented in this thesis and the performance testing results, the tree major database vendors all support storing and manipulating OR data (object types and tables). Given this information, it can be anticipated that the popularity and number of users of object-relational database systems will increase as more object-oriented application developers become familiar with using ORDBMSs for object persistence.

Acknowledgments

Bibliography

[1] Stonebraker M. (1996). *Object-Relational DRMSs: The Next Great Wave*. Morgan Kaufmann Publishers Inc, San Francisco, CA, 1999.

[2] S. Ambler. *Building Object Applications that Work*. Cambridge University Press and Sigs Books, New York, NY, USA, 1998.

[3] B. K. Barry. *The Object database handbook*. Wiley Computer Publishing, New York, NY, USA, 1996.

[4] E. Bertino and L. Martino. *Object-oriented database systems – concepts and architecutres*. Addison-Westley, Workingham, 1993.

[5] D. K. Burleson. *Inside the Database Object Model*. CRC Press, Boca Raton, 1999.

[6] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. A status report on the 007 00dbms benchmarking effort. In *In Proceedings of the ACM 00PSLA Conference*, pages 414–426, October 1994.

[7] A.B. Chaudhri and R. Zicari. *Succeeding with Object databases: a practical look at today's implementations with Java and XML*. Wiley and Sons, New York, NY, USA, 2001.

[8] L. Chirica. Discussion, May 2007.

[9] D. N. Chorafas and H. Steinmann. *Object-Oriented Databases*. Prentice Hall, Englewood Cliffs, 1993.

[10] P. Coad and E. Yourdon. *Object-oriented analysis*. Yourdon Press, Upper Saddle River.

[11] E. F. Codd. A relational model of data for large shared data banks. *Communications ACM*, 13(6):377–387, 1970.

[12] Thomas M. Connolly and Carolyn Begg. *Database Systems: A Practical Approach to Design, Implementation, and Management*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[13] William R. Cook, Robert Greene, Patrick Linskey, Erik Meijer, Ken Rugg, Craig Russell, Bob Walker, and Christof Wittig. Objects and databases: state of the union in 2006. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 926–928, New York, NY, USA, 2006. ACM.

[14] C. J. Date and H. Darwen. *Foundation for Future Database Systems the Third Manifesto*. Addison-Westley, New York, NY, USA, 2000.

[15] R. S. Devarakonda. Object-relational database systems - the road ahead. *ACM Crossroads*, 7(3):15–18, March 2001.

[16] Tharam S. Dillon and Poh L. Tan. *Object-Oriented Conceptual Modeling*. Prentice Hall PTR, Upper Saddle River, NJ, 1993.

[17] J. Rumbaugh M. Blaha W. Premerlani F. Eddy and W. Lorensen. *Object-oriented modeling and design*. Prentice-Hall, Inc, New York, NY, USA, 1991.

[18] M. J. Egenhofer and A.U. Frank. Object-oriented modeling for gis. *URISA Journal*, 4(2):3–19, 1992.

[19] Jr. Frederick P. Brooks. No silver bullet: essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987.

[20] Jim Grey. The next database revolution. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 1–4, New York, NY, USA, 2004. ACM.

[21] Jonathan Grudin. Evaluating opportunities for design capture. *Design rationale: concepts, techniques, and use*, 1996.

[22] J. L. Harrington. *Object-Oriented Database Design Clearly Explained*. Morgan Kaufmann, San Diego, CA USA, 2000.

[23] T. Dillon J. Raharu, E. Chang and D. Taniar. A methology for transforming inheritance relationships in an object-oriented conceptual model tables. *Information and Software Technology*, 42:571–592, 2000.

[24] Eric Pardede Johanna Wenny Rahayu, David Taniar. *Object-Oriented Oracle*. Hersey: CyberTech Publishing, London, UK, 2005.

[25] J. Lee K. Sau, S. Kim. End-user computing abilities and the use of information systems. *SIGCPR Computing*, 15(1):3–14, 1994.

[26] W. Kim and F.H. Lochovsky. *Object-oriented concepts, databases, and applications*. ACM Press, New York, NY, USA, 1989.

[27] N. Leavitt. Whatever happened to object-oriented databases? *Computer*, 33(8):16–19, August 2000.

[28] Sang Ho Lee, Sung Jin Kim, and Won Kim. The bord benchmark for object-relational databases. In *DEXA '00: Proceedings of the 11th International Conference on Database and Expert Systems Applications*, pages 6–20, London, UK, 2000. Springer-Verlag.

[29] G. Fisher A. C. Lemke R. McCall A. I. Morch. Making argumentation serve design. *Human-Computer Interaction*, 6(3), 1991.

[30] Oracle. *Oracle Database Application Developer's Guide - Object Relational Features*, 2005.

[31] Oracle. *Oracle Database SQL Reference*, 2005.

[32] B. R. Rao. *Object-Oriented Databases: Technology, Applications, and Products*. McGraw-Hill, New York, NY, USA, 1994.

[33] J. A. Lewis S.M. Henry D. G. Kafura R. S. and Schulman. An empirical study of the object-oriented paradigm and software reuse. In *In Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications*, pages 184–196, Phoenix, Arizona, United States, October 1991. ACM Press.

[34] Karen E. Smith and Stanley B. Zdonik. Intermedia: A case study of the differences between relational and object-oriented database systems. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 452–465, New York, NY, USA, 1987. ACM.

[35] R. Cattell B. Barry M. Berler J. Eastman D. Jordan C. Russell O. Schadow T. Stanienda and F. Velez. *The Object data standard: ODMG 3.0*. Morgan Kaufmann Publishers Inc, San Francisco, 2000.

[36] A. Silberschatz H.F. Korth S. Sudarhan. *Database System Concepts*. McGraw-Hill, Inc, New York, NY, USA, 2006.

[37] S. W. S.W. Dietrich and S.D. Urban. *An Advanced Course in Database Systems Deyond Relational Databases*. Pearson Prentice Hall, Upper Saddle River, 2005.

[38] J.W. Rahayu E. Chang T.S. Dillon D. Taniar. Performance evaluation of the object-relational transformation methodology. *Data Knowledge Engineering*, 38(3):265–300, September 2001.

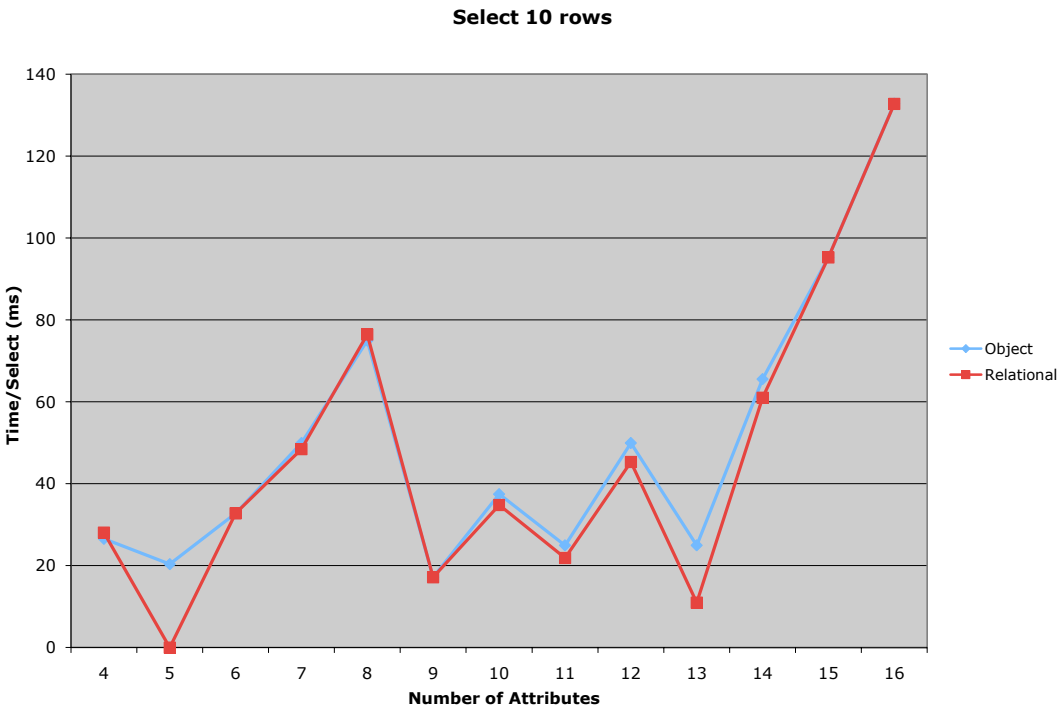
[39] Can Türker and Michael Gertz. Semantic integrity support in sql:1999 and commercial (object-)relational database management systems. *The VLDB Journal*, 10(4):241–269, 2001.

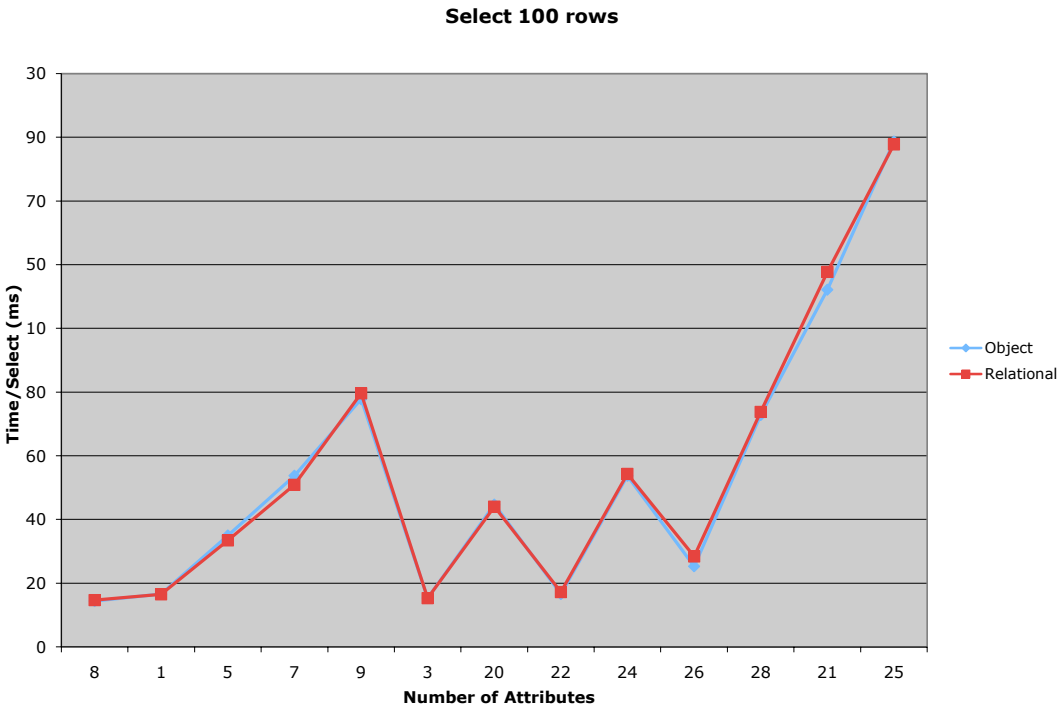
[40] Tiejun Wang and Scott F. Smith. Precise constraint-based type inference for Java. *Leture Notes in Computer Science*, 2072, 2001.

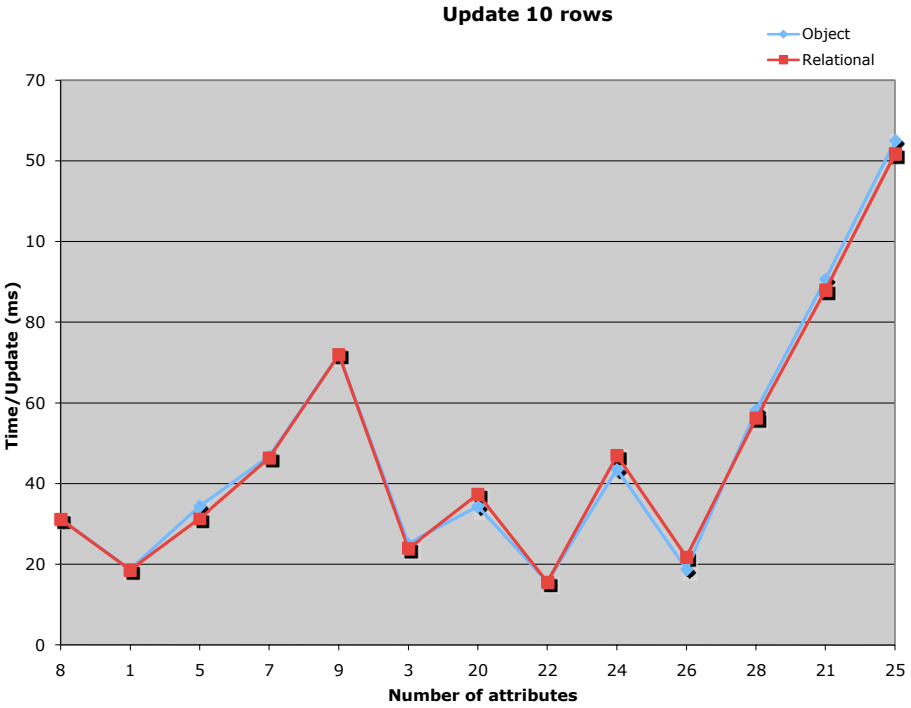
[41] M. P. Atkinson F. Bancilhon D. J. DeWitt K. R. Dittrich D. Maier S. B. Zdonik. The object-oriented database system manifesto. In *International Conference on Deductive and Object-Oriented Databases*, pages 223–240, 1989.

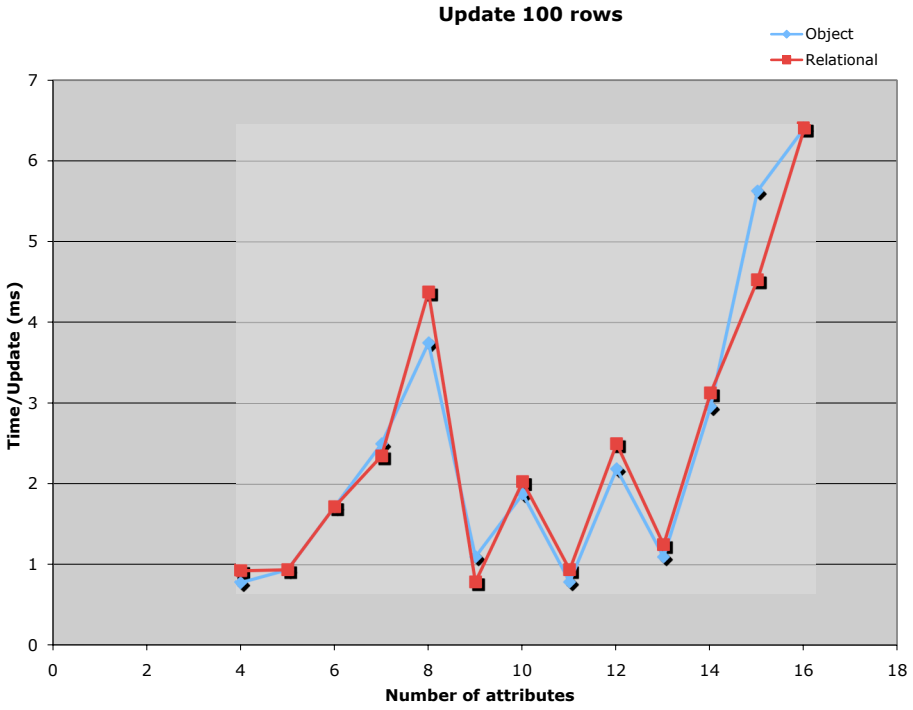
[42] Q. Zhang. Object-oriented database systems in manufacturing: selection and applications. *Industrial Management and Data Systems*, 101(3):97–105, 2001.

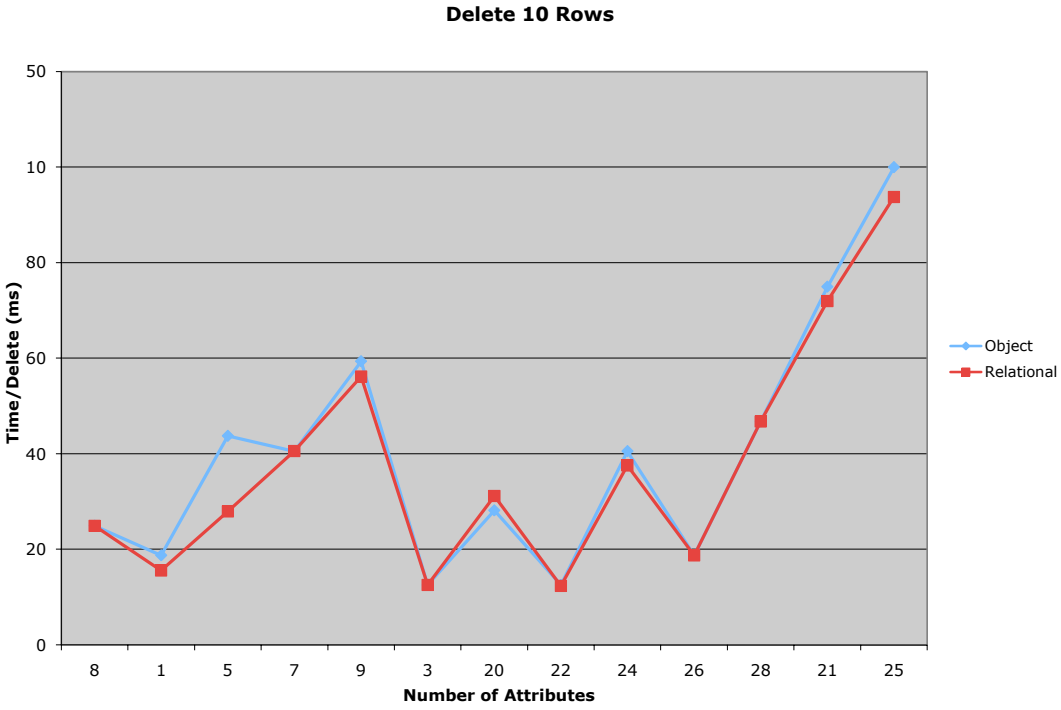
Appendix A

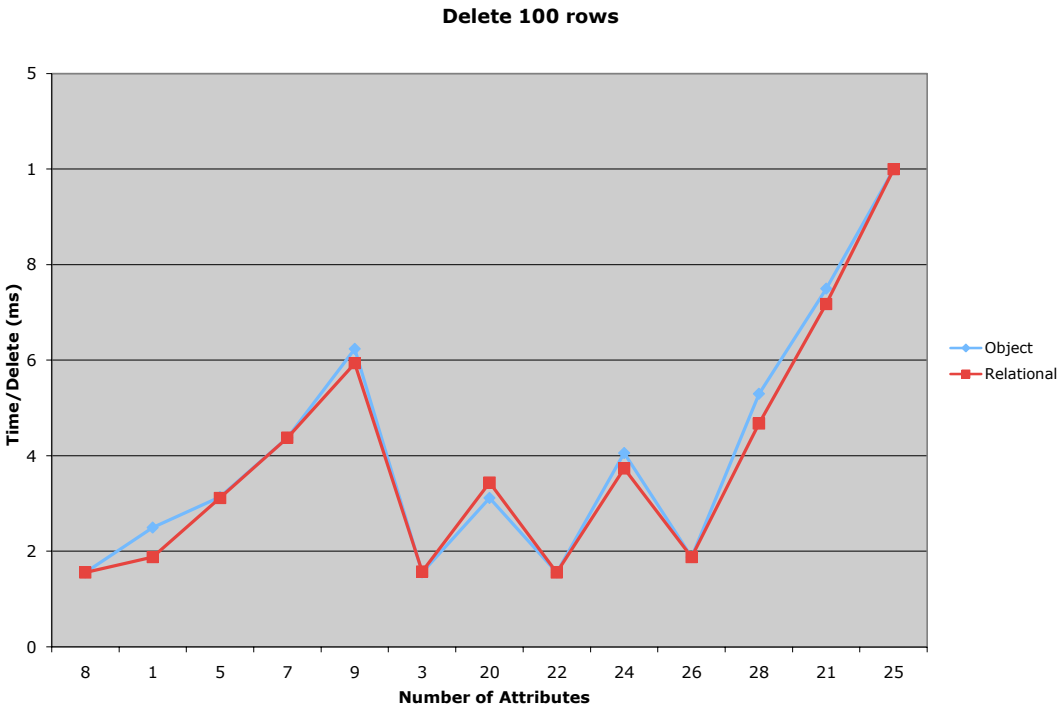

Select 10 rows
Time/Select (ms)
140
120
100
80
60
40
20
0
4
5
6
7
8
9
10
11
12
13
14
15
16
Number of Attributes
Object
Relational


Select 100 rows
Time/Select (ms)
30
90
70
50
10
80
60
40
20
0
8
1
5
7
9
3
20
22
24
26
28
21
25
Number of Attributes
Object
Relational


Update 10 rows
Object
Relational
Time/Update (ms)
70
50
10
80
60
40
20
0
8
1
5
7
9
3
20
22
24
26
28
21
25
Number of attributes


Update 100 rows
Object
Relational
Time/Update (ms)
0
1
2
3
4
5
6
7
0
2
4
6
8
10
12
14
16
18
Number of attributes


Delete 10 Rows
Time/Delete (ms)
50
10
80
60
40
20
0
8
1
5
7
9
3
20
22
24
26
28
21
25
Number of Attributes
Object
Relational


Delete 100 rows
Time/Delete (ms)
0
2
4
6
8
1
5
8 1 5 7 9 3 20 22 24 26 28 21 25
Number of Attributes
Object
Relational