Teaching Case

Solving Relational Database Problems with ORDBMS in an Advanced Database Course

Ming Wang
ming.wang@calstatela.edu
Department of Information Systems
California State University
Los Angeles, CA 90032

Abstract

This paper introduces how to use the object-relational database management system (ORDBMS) to solve relational database (RDB) problems in an advanced database course. The purpose of the paper is to provide a guideline for database instructors who desire to incorporate the ORDB technology in their traditional database courses. The paper presents how to use the specific object-relational database (ORDB) technology to solve three normalization problems: Transitive dependency, Multi-value attributes, and Non-1st Normal Form. The paper also provides the solutions to data complexity problems with three specific ORDBMS techniques: object view, object inheritance, and object integration. The paper summarizes the significance and advantages of teaching ORDBMSs in advanced database courses. Course contents and students' learning outcomes are discussed. To be more helpful to database educators, the paper presents a complete object-relational database development case study from the UML class diagram design to Oracle ORDBMS implementation.

Keywords: Object-relational database, Database Curriculum, Oracle Database, Normalization

1. INTRODUCTION

The success of relational database management systems (RDBMSs) cannot be denied, but they experience difficulty when confronted with the kinds of "complex data" found in advanced application areas such as hardware and software design, science and medicine, and mechanical and electrical engineering. To meet the challenges, Oracle, IBM and Microsoft have moved to incorporate object-oriented database features into their relational DBMSs under the name of object-relational DBMSs. The major database vendors presently support object-relational data model, a data model that combines features of the object-oriented model and relational model (Silberschatz, et al., 2009). The emergence of object-relational technology into the commercial database market has caused the database professional's attention in seeking how to utilize its object-oriented features in the database development and has brought new challenges for IS instructors in teaching ORDBMS in their database courses. In response to this challenge, the author has incorporated the ORDB technology into her advanced database course. ORDBMS enhances object-oriented technology into the relational database management system (RDBMS) and extends traditional RDBMS with object-oriented features. As an evolutionary technology, ORDBMS allows users to take advantages of reuse features in object-oriented technology, to map objects into relations and to maintain a consistent data structure in the existing RDBMS.

The purpose of the paper is to provide a guide for database instructors who desire to incorporate the ORDB technology in their traditional database courses. This paper presents how to use ORDBMS to overcome relational database weaknesses and solve some existing normalization problems. The paper first introduces the background and features of ORDBMS, then presents how to use the specific ORDBMS techniques to solve normalization problems in 1) Transitive dependency, 2) Multi-value attributes, 3) and Non-1st Normal Form, and how to use the specific ORDBMS features: 1) object view 2) object inheritance and 3) object integration to solve data complexity problems. Course content and students' learning outcomes are discussed. Many of the ORDBMS features appear in Oracle. Thus, the author utilizes Oracle as a tool to demonstrate how to overcome some weaknesses of relational DBMS. The ORDBMS script in the case study has been tested in the Oracle 9i, 10g, and 11g SQLPlus environment. The solution to the presented case can be utilized in the classroom demonstration and can also be generalized the homework assignments and projects of advanced database courses.

2. ORDB TECHNOLOGY

The object-relational database technology occurrence can be traced back to the middle of 1990s after emergence of object-oriented database (OODB). In their book "Object-relational DBMSs: the Next Great Wave", Stonebraker and Moore (1996) define their four-quadrant view (two by two matrix) of the data processing world: relational database, object-relational database, data file processing, and object-oriented database. Their purpose is to indicate the kinds of problems each of four-quadrants solves. As will be seen, "one size does not fit all"; i.e. there is no DBMS that solves all the applications. They suggest that there is a natural choice of data manager for each of the four database applications. They conclude why the problems addressed by object-relational DBMSs are expected to become increasingly important over the next decade. As such, it is "the next wave".

Theoretically, as Stonebraker and Moore (1996) predict in their four-quadrant view of the database world, ORDBMS has been the most appropriate DBMS that processes complex data and complex queries. The object-oriented database management systems have made limited inroads during the 1990's, but have since been dying off. Instead of a migration from relational to object-oriented systems, as was widely predicted around 1990, the vendors of relational systems have incorporated many object-oriented database features into their DBMS products. As a result, many DBMS products that used to be called "relational" are now called "object-relational." (Garcia-Molina, *et al.* 2003).

Practically, ORDBMS bridges the gap between OODBMS and RDBMS by allowing users to take advantage of OODB'MSs great productivity and complex data type without losing their existing investment in relational data (Connolly & Begg, 2006). In fact, an ORDBMS engine supports both relational and object-relational features in an integrated fashion (Frank, 1995). The underlying ORDB data model is relational because object data is stored in tables or columns. ORDB designers can work with familiar tabular structures and data definition languages (DDLs) while assimilating new object-oriented features (Krishnamurthy et al., 1999).It is essentially a relational data model with object-oriented extensions. In response to the evolutional change of ORDB technology, SQL:1999 started supporting object-relational data modeling features in database management standardization and SQL:2003 continues this evolution. Currently, all the major database vendors have upgraded their relational database products to object-relational database management systems to reflect the new SQL standards (Hoffer et al., 2009) and use by industrial practitioners.

Although each of the object-relational DBMS vendors has implemented OO principles: encapsulation and inheritance in their own way, all of them share the combination of the OO principles and follow SQL standardization, incorporate object-oriented paradigms. All the ORDBMSs have the ability to store object data and methods in databases. Many of the SQL:2003 standard ORDBMS features appear in Oracle. These features are listed as follows.

Object Types**:** User-defined data types (UDT) or abstract types (ADT) are referred to as object types.

Functions/Methods: For each object type, the user can define the methods for data access. Methods define the behavior of data.

***Varray*:** The varray is a collection type that allows the user to embed homogenous data into an array to form an object in a pre-defined array data type.

Nested table: A nested table is a collection type that can be stored within another table. With a nested table, a collection of multiple columns from one table can be placed into a single column in another table.

***Inheritance*:** With Object type inheritance, users can build subtypes in hierarchies of database types in ORDBs.

***Object View*:** Object view allows users to develop object structures in existing relational tables. It allows data to be accessed or viewed in an object-oriented way even if the data are really stored in a traditional relational format.

There is some research that has been done in ORDBMS technology as ORDBMSs have become commonplace in recent years. He and Darmont (2005) propose the Dynamic Evaluation Framework (DEF) that simulates access pattern changes using configurable styles of change. Pardede, Rahayu, & Taniar (2006) propose an innovative methodology to store XML data into new ORDB data structures, such as user-defined type, row type and collection type. The methodology has preserved the conceptual relationship structure in the XML data, including aggregation, composition and association. Wok (2007) and Cho, et. al. (2007) present a methodology for designing proper nesting structures of user-defined types in object-relational database. The proposed schema trees schema are transformed to Oracle 10g. Their purpose is to develop an automatic ORDB design tool.

But very little research has been done in using ORDBMS to overcome relational database weaknesses and solve some existing normalization problems. The significance of the paper is to promote teaching ORDBMS features for problem solving and object reuse and integration among IS educators. The use of ORDBMSs to develop database applications can enforce the reuse of varying user-defined object types, provide developers' an integrated view of data and allow multiple database applications to operate cooperatively. Ultimately, this can result in improved operational efficiency for the IT department, increase programmers' productivity, lower development effort, decrease maintenance cost, reduce the defect rate, and raise the applications' reliability. If multiple database applications use the same set of database objects in ORDBMS, a de facto standard for the database objects is created, and these objects can be extended, reused and integrated in the ORDB.

3. CASE STUDY

3.1 Case Scenario

Pacific Bike Traders assembles and sells bikes to customers. The company currently accepts customer orders online and wants to be able to track orders and bike inventory. The existing database system cannot handle the current transaction volume generated by employees processing incoming sales orders. When a customer orders a bike, the system must confirm that the ordered item is in stock. The system must update the available quantity on hand to reflect that the bike has been sold. When Pacific Bike Traders receives new shipments, a receiving clerk must update the inventory to show the new quantity on hand. The system must produce invoices and reports showing inventory levels.

3.2. Business Rules

The following business rules are developed for the new database system:

One customer may originate many orders.
One order must be originated from a customer.

One order must contain one or more bikes.
One bike may be in many orders.

One employee may place many orders.
One order must be placed by an employee.

One bike is composed with a front wheel, rear wheel, crank, and stem.
One front wheel, rear wheel, crank, and stem compose one bike.

One employee must be either a full-time or part-time.
One full-time or part-time employee must be an employee.

3.2. ORDB Design

The Pacific Trader Object-Relational Database design is illustrated with the UML class diagram in Appendix 1. Each of the classes is displayed as a rectangle that includes three sections: the top section gives the class name; the middle section displays the attributes of the class; and the last section displays methods that operate on the data in the object. Associations between classes are indicated with multiplicity ("min..max." notation). Inheritance is indicated with an empty triangle. Aggregation is marked with an empty diamond, whereas composition is marked with a solid diamond. Aggregation models a whole-part relationship where individual items become elements in a new class. In Appendix 1, a sales order is made of line items (bikes). Aggregation is indicated by a small empty diamond next to the SalesOrder class. The dotted line links to the associative class generated from the many-to-many relationship.

Based on the Pacific Trader's Object-Relational Database Design in Appendix 1, ORDB features are implemented with Oracle for the case in the following sections. The implementation shows how the UML class diagram maps and supports major ORDB features. For the sake of simplicity, it is assumed that referential integrity constraints will be added later.

4. ORDBMS FOR NORMALIZATION

Normalization is a logical data modeling technique for the development of a well structured relational database. The process is decomposing tables with anomalies to produce smaller tables. Traditional normalization processes are normalizing tables in non-1NF form and multi-value attributes to at least 3NF; and removing transitive dependency. Such processes can be eliminated if ORDB technology is used.

4.1. Object Type & Transitive Dependency

The address attribute is usually split into four columns such as street, city, state and zip code in order to store address dada in a customer table since it is a composite attribute in a traditional database.

Customer table

Cu_id	First	Last	Street	City	State	Zip
1	John	Smih	12 Pine	Bell	CA	90201
2	Mary	Fox	6 Circle	Brea	CA	92821

The above Customer table is in Second Normalization Form (2NF) and violates the Third Normalization Form (3NF) rule because there is the transitive dependency in the customer table. Zip is a determinant of street, city and state. Functional dependency analysis shows transitive dependency:

Zip -> Street, City, State (transitive dependency)

There are three solutions to this transitive dependency problem. Solution 1 keeps the customer table in the Second Normalization Form (2NF) though it is not an ideal normal form for a relational database.

Solution 2 is to create a new customer address table by splitting the address from the original customer table (3NF). This solution implies more joins of records in the Customer table and Zip table.

Customer Table

Cu_id	First	Last	Zip
1	John	Smith	90201
2	Mary	Fox	92821

Zip Table

Zip	Street	City	State
96123	12 Pine	Bell	CA
25678	6 Circle	Brea	VA

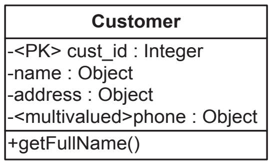
Solution 3 is to store all the customer address information in one column. This solution creates difficulty in data retrieval. For example, it is impossible to retrieve or sort customer records by city, state or zip code.

Customer table

Cu_id	First	Last	Address
1	John	Smith	12 Pine, Bell, CA 90201
2	Mary	Fox	6 Circle, Brea, CA 92821

None of the above three solutions is considered ideal in terms of efficient database design and operations. The first solution is not satisfactory since 2NF is not ideal for relational database design. The second solution implies that more joins might occur in the query process, since the zip table has been added to the database. The third solution creates difficulty in data retrieval. For example, it is impossible to retrieve or sort customer records by city, state, or zip code.

With ORDBMS technology, the attribute address can be defined as a user-defined abstract data type with a number of attributes using s the same internal format. User-defined types (UDT) or abstract data types (ADT) are referred to as object types. Object types are used to define either object columns or object tables. The following UML Customer class illustrates the address object column.



Object types need to be defined before the customer table. The following SQL statements define the object types: address_ty and name_ty.

```
CREATE OR REPLACE TYPE address_ty AS
OBJECT
(street          NVARCHAR2(30),
 city            VARCHAR2(25),
 state           CHAR(2),
 zip             NUMBER(10));
```

```
CREATE OR REPLACE TYPE name_ty AS OBJECT
(
f_name    VARCHAR2(25),
l_name    VARCHAR2(25));
```

Mapping the above customer class, the following statement is used to create the Customer table with the CustName and CustAddress object columns using name_ty and address_ty. The column phone is to be added to the table later.

```
CREATE TABLE Customer2(
Cust_ID          Number(5),
CustName         name_ty,
CustAddress      address_ty);
```

Object type constructors are used to insert object data into the table. The following INSERT statement uses constructors name_ty() and address_ty() to add data into the two object columns.

```
INSERT INTO Customer VALUES (1,
name_ty ('John', 'Smith',),
address_ty ('12 Road', 'Bell', 'CA', 90201));
```

The following statements retrieve the data from the Customer2 table.

```
SELECT c.custName.l_name, c.custAddress.City,
c.custAddress.state
          FROM Customer2 c;
```

CUSTNAME.L_NAME	CUSTADDRESS.CITY	CU
John Smith	Bell	CA

SELECT * from Customer2;

CUST_ID	CUSTNAME(F_NAME, L_NAME, INITIALS)	CUSTADDRESS(STREET, CITY, STATE, ZIP)
1	NAME_TY('John', 'Smith')	ADDRESS_TY('12 Pine', 'Bell', 'CA', 90201)

4.2 Varray and Multi-value Attributes

In a relational model, multi-valued attributes are not allowed in the first normalization form. The traditional solution to the problem is that each multiple-valued attribute is handled by forming a new table in a relational database. If a table has five multi-valued attributes, that table would have to be split into six tables. The Oracle ORDBMS allows users to create the varying length array (VARRAY) data type as a new data storage method for multi-valued attributes. The following statement defines a varray type of three VARCHAR2 string named varray_phone_ty to represent a list of phone numbers.

VARRAY is a collection type in ORDBMSs. A VARRAY consists of a set of objects that have the same predefined data type in an array. In a relational model, multi-valued attributes are not allowed in the first normalization form. The solution to the problem is that each multiple-valued attribute is handled by forming a new table. If a table has five multi-valued attributes, that table would have to be split into six tables after the First Form of normalization. To retrieve the data back from that original table, the student would have to do five joins across these six tables. ORDBMs allow multi-valued attributes to be represented in a database. ORDBMSs allow users to create the varying length array (VARRAY) data type can be used as a new data storage method for multi-valued attributes. The following statement defines a VARRAY type of three VARCHAR2 strings named varray_phone_ty to represent a list of three phone numbers in the Customer2 table.

```
CREATE TYPE varray_phone_ty AS VARRAY(3)
OF VARCHAR2(14);
```

```
ALTER TABLE Customer ADD (phones
varray_phone_ty);
```

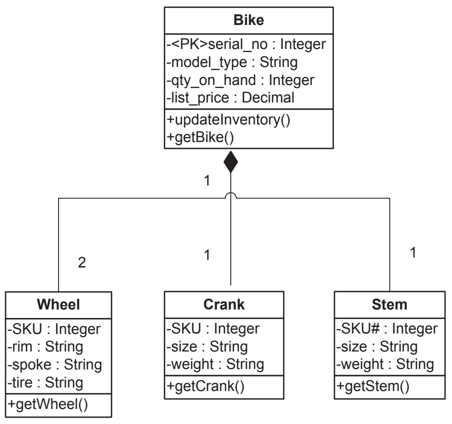
```
UPDATE customer
            SET phones =
(varray_phone_ty('(800)555-1211',
'(800)555-1212','(800)555-1213'))
           WHERE cust_id = 1;

INSERT INTO customer(phones) values
(varray_phone_ty('(800)555-
1211','(800)555-1212','(800)555-1213'));
```

The above example shows that using the varying length array (VARRAY) data type not only can solve multi-value attribute problem for the customer table, but also can speed up the query process on customer data.

4.3 Nested Table and Non-1NF

A nested table is a table that can be stored within another table. With a nested table, a collection of multiple columns from one table can be placed into a single column in another table. Nested tables allow user to embed multi-valued attributes into a table, thus forming an object.



```
CREATE TYPE wheel_type AS OBJECT(
          SKU       VARCHAR2(15),
          rim       VARCHAR2(30),
          spoke     VARCHAR2(30),
          tire      VARCHAR2(30));

CREATE TYPE crank_type AS OBJECT
          (SKU           VARCHAR2(15),
          crank_size          VARCHAR2(15),
          crank_weight      VARCHAR2(15) );

CREATE TYPE stem_type AS OBJECT(
          SKU       VARCHAR2(15),
          stem_size              VARCHAR2(15),
          stem_weight    VARCHAR2(15));
```

The following statement creates nested table types: wheel_type, crank_type and stem_type:

```
CREATE TYPE nested_table_wheel_type AS
TABLE OF wheel_type;

CREATE TYPE nested_table_crank_type AS
TABLE OF crank_type;

CREATE TYPE nested_table_stem_type AS
TABLE OF stem_type;
```

The following example creates the table named Bike with that contains four nested tables:

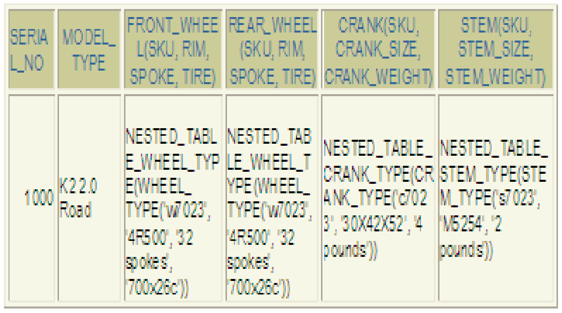
```
CREATE TABLE bike   (
serial_no          INTEGER PRIMARY KEY,
          model_type                  VARCHAR2(20),
          front_wheel
          nested_table_wheel_type,
    rear_wheel
          nested_table_wheel_type,
    crank
          nested_table_crank_type,
    stem
          nested_table_stem_type
          )
          NESTED TABLE
                    front_wheel
          STORE AS
                    front_wheel,
    NESTED TABLE
                    rear_wheel
          STORE AS
                    rear_wheel,
          NESTED TABLE
                    crank
          STORE AS
                    nested_crank,
          NESTED TABLE
                    stem
          STORE AS
                    nested_stem;

INSERT INTO bike VALUES (1000, 'K2 2.0 Road',
nested_table_wheel_type(   wheel_type('w7023',
'4R500', '32 spokes', '700x26c' )),
nested_table_wheel_type(
wheel_type('w7023',    '4R500',    '32    spokes',
'700x26c' )),
nested_table_crank_type(
crank_type('c7023', '30X42X52', '4 pounds')),
nested_table_stem_type(
stem_type('s7023', 'M5254', '2 pounds')));
```

Finally the previous statement inserts a row into the Bike table with nested tables using the three defined constructors: wheel_type, crank_type and stem_type.

The above example shows that using the NESTED TABLE can implement the composition association, store multiple parts and also speed up the data retrieval speed for the Bike table. The following statement shows the nested tables in the table Bike.

```
SELECT * from bike;
```



SERIAL_NO	MODEL_TYPE	FRONT_WHEEL(SKU, RIM, SPOKE, TIRE)	REAR_WHEEL (SKU, RIM, SPOKE, TIRE)	CRANK(SKU, CRANK_SIZE, CRANK_WEIGHT)	STEM(SKU, STEM_SIZE, STEM_WEIGHT)
1000	K2 2.0 Road	NESTED_TABLE_WHEEL_TYPE(WHEEL_TYPE('w7023', '4R500', '32 spokes', '700x26c'))	NESTED_TABLE_WHEEL_TYPE(WHEEL_TYPE('w7023', '4R500', '32 spokes', '700x26c'))	NESTED_TABLE_CRANK_TYPE(CRANK_TYPE('c7023', '30X42X52', '4 pounds'))	NESTED_TABLE_STEM_TYPE(STEM_TYPE('s7023', 'M5254', '2 pounds'))

5. ORDBMS FOR OBJECT INTEGRATION

The beauty of ORDBMSs is reusability and sharing. Reusability mainly comes from storing data and methods together in object types and performing their functionality on the ORDBMS server, rather than have the methods coded separately in each application. Sharing comes from using user-defined standard data types to make the database structure more standardized (Breg & Connolly. 2010)

5.1. Object Views on a Relational Table

Object views are virtual object tables, which allow database developers to add OOP structures on top of their existing relational tables and enable them to develop OOP features with existing relational data. The object view is a bridge between the relational database and OOP. Object view creates a layer on top of the relational database so that the database can be viewed in terms of objects (Loney & Koch, 2002). This enables you to develop OOP features with existing relational data. The following statements show how to create the SalesOrder table:

```
CREATE TABLE SalesOrder (
ord_id   NUMBER(10),
ord_date DATE,
cust_id NUMBER(10),
emp_id NUMBER(10));
```

```
INSERT INTO SalesOrder VALUES
      (100,'5-Sep-05', 1, '1000');
INSERT INTO salesOrder VALUES
      (101, '1-Sep-05', 1, '1000');
```

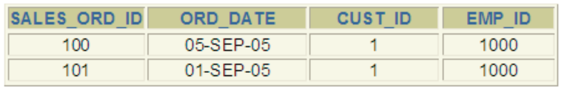
The following statements show how to create an object view on the top of the SalesOrder relational table:

```
CREATE TYPE SalesOrder_type AS OBJECT(
sales_ord_id     NUMBER(10),
         ord_date DATE,
         cust_id NUMBER(10),
         emp_id NUMBER(10));
```

```
CREATE VIEW customer_order_view OF
SalesOrder_type WITH OBJECT IDENTIFIER
(sales_ord_id)
AS SELECT  o.ord_id, o.ord_date, o.cust_id,
o.emp_id
         FROM salesOrder o
           WHERE o.cust_id = 1;
```

The following SQL statement generates the view output:

```
SELECT * FROM customer_order_view;
```



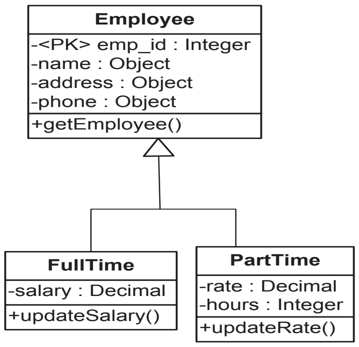
SALES_ORD_ID	ORD_DATE	CUST_ID	EMP_ID
100	05-SEP-05	1	1000
101	01-SEP-05	1	1000

The object view is a bridge that can be used to create object-oriented applications without modifying existing relational database schemas. By calling object views, relational data can be retrieved, updated, inserted, and deleted as if such data were stored as objects. The following statement can retrieve Analysts as object data from the relational SalesOrder table. Using object views to group logically-related data can lead to better database performance.

5.2 Inheritance for Object Reuse

The main advantages of extending the relational data model come from reuse and sharing. If multiple applications use the same set of database objects, then you have created a de facto standard for the database objects, and these objects can be extended (Price, 2002). ORDBMSs allow users to define hierarchies of data types. With this feature, users can build subtypes in hierarchies of database types. If users create standard data types to use for all employees, then all of the employees in the database will use the same internal format.

Users might want to define a full time employee object type and have that type inherit existing attributes from employee_ty. The full_time_ty type can extend employee_ty with attributes to store the full time employee's salary. The part_time_ty type can extend employee_ty with attributes to store the part-time employee's hourly rates and wages. Inheritance allows for the reuse of the employee_ty object data type. The details are illustrated in the following class diagram:



Object type inheritance is one of new features of Oracle 9i. For employee_ty to be inherited from, it must be defined using the NOT FINAL clause because the default is FINAL, meaning that object type cannot be inherited. Oracle 9i can also mark an object type as NOT INSTANTIABLE; this prevents objects of that type derived. Users can mark an object type as NOT INSTANTIABLE when they use the type only as part of another type or as a super_type with NOT FINAL. The following example marks address type as NOT INSTANTIABLE:

```
CREATE TYPE employee_ty AS OBJECT (
  emp_id        NUMBER,
  SSN           NUMBER,
  name          name_ty,
    dob              DATE,
  phone              varray_phone_ty,
  address            address_ty
) NOT FINAL NOT INSTANTIABLE;
```

To define a new subtype full_time_ty inheriting attributes and methods from existing types, users need to use the UNDER clause. Users can then use full_time_ty to define column objects or table objects. For example, the following statement creates an object table named FullTimeEmp.

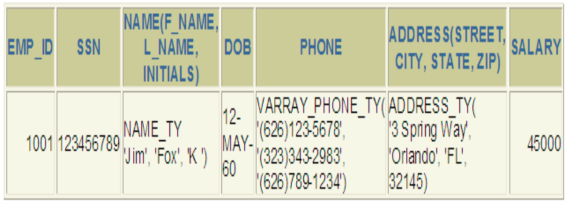
```
CREATE TYPE full_time_ty UNDER employee_ty (
Salary  NUMBER(8,2));

CREATE TABLE FullTimeEmp of full_time_ty;
```

The preceding statement creates full_time_typ as a subtype of employee_typ. As a subtype of employee_ty, full_time_ty inherits all the attributes declared in employee_ty and any methods declared in employee_ty. The statement that defines full_time_ty specializes employee_ty by adding a new attribute "salary". New attributes declared in a subtype must have names that are different from the names of any attributes or methods declared in any of its supertypes, higher up in its type hierarchy. The following example inserts row into the FullTimeEmp table. Notice that the additional salary attribute is supplied

```
INSERT INTO FullTimeEmp VALUES
(1000, 123456789, name_ty('Jim',   'Fox', 'K'),
'12-MAY-1960',
varray_phone_ty('(626)123-5678',    '(323)343-
2983', '(626)789-1234'),
Address_ty ('3 Lost Spring Way', 'Orlando', 'FL',
32145), 45000.00);

SELECT * FROM FullTimeEmp;
```



EMP_ID	SSN	NAME(F_NAME, L_NAME, INITIALS)	DOB	PHONE	ADDRESS(STREET, CITY, STATE, ZIP)	SALARY
1001	123456789	NAME_TY 'Jim', 'Fox', 'K ')	12-MAY-60	VARRAY_PHONE_TY('(626)123-5678', '(323)343-2983', '(626)789-1234')	ADDRESS_TY('3 Spring Way', 'Orlando', 'FL', 32145)	45000

A supertype can have multiple child subtypes called siblings, and these can also have subtypes. The following statement creates another subtype part_time_ty under Employee_ty.

```
CREATE OR REPLACE TYPE part_time_ty UNDER
employee_ty (
rate Number(7,2),
hours Number(3))NOT FINAL;

CREATE TABLE PartTimeEmp of part_time_ty;
```

A subtype can be defined under another subtype. Again, the new subtype inherits all the 87attributes and methods that its parent type has, both declared and inherited. For example, the following statement defines a new subtype student_part_time _ty under part_time_ty. The new subtype inherits all the attributes and methods of student_part_time _ty and adds two attributes.

CREATE TYPE student_part_time_ty UNDER part_time_ty
(school VARCHAR2(20),
year VARCHAR2(10));

5.3 Object Integration with Interface

ORDBMS combines attributes and methods together in the structure of object type. The object type interface includes both attributes and its methods. The public interface declares the data structure and the method header shows how to access the data. This public interface serves as an interface to applications. The private implementation fully defines the specified methods.

Public Interface

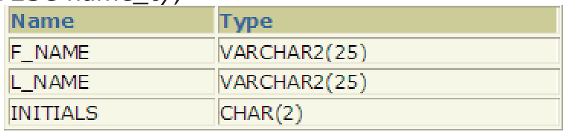
Specification:
Attribute declarations
Method specifications

Private Implementation

Body:
Method implementations

The following statement displays the public interface of the object type name_type. The output of the name_type public interface shows attributes and method headers as follows:

DESC name_ty;



Name	Type
F_NAME	VARCHAR2(25)
L_NAME	VARCHAR2(25)
INITIALS	CHAR(2)

METHOD
MEMBER FUNCTION ***FULL_NAME***
RETURNS ***VARCHAR2***

Although the user-defined methods are defined with object data within the object type, they can be shared and reused in multiple database application programs. This can result in improved operational efficiency for the IT department, as well, by improving communication and cooperation between applications. An object-relational database schema consists of a number of related tables that forms connected user-defined object-types. Object-types possess all the properties of a class, data abstraction, encapsulation, inheritance and polymorphism. These traits of object-types are embedded in the relational nature of the database; data model, security, concurrency, normalization. In more precise words, the underlying ORDB data model is relational because object data is stored in tables or columns.

6. LEARNING OUTCOMES

The provided ORDB script guides students with hands-on learning experience in the classroom. Once they have understood they can use the script as templates to do their homework assignments and projects. ORDB, implement it with Oracle 9i/10g, and create ORDB applications using various tools. As a result, the following learning outcomes are demonstrated at the end of the class. Students are able to:

1. Map UML class diagrams to ORDB databases
2. Use Object Types to remove transitive dependency
3. Use VARRAY types for multi-value attribute
4. Use NESTED TABLE types to Solve non-1NF problems
5. Implement inheritance with sub-object types
6. Create object views in the existing relational databases

ORDB technology helps students to better understand object-oriented principles such as encapsulation, inheritance, and reusability. During the learning process, they have reviewed the object-oriented paradigm they learned from their previous programming courses and are able to tie it to ORDBMS and object-oriented system design.

With a grasp of ORDB technology, students are able to make their database design more structured and consistent. With object reuse and standard adherence, students are able to create a de facto standard for database objects and multiple database applications. The motivation to learn in class is high because students have realized that object-relational technology is incorporated in most commercial DBMS. Learning it will help their career development in the future competitive job market.

7. REFERRNCES

Begg, C., & Connolly, T. (2010). Database systems: A practical approach to design, implementation, and management, 5th Ed. Addison Wesley.

Cho, W., Hong, K. & Loh, W. (2007). Estimating nested selectivity in object-oriented and object-relational databases *Information and Software Technology*, (49)7, 806-816

Connolly, T. and Begg, C. (2006). Database systems: A practical approach to design, implementation, and management, 4th Ed. Addison Wesley.

Elmasri, R. & Navathe, S. (2011). Fundamentals of Database Systems, 6th Edition, Addison Wesley.

Fortier, P. (1999). SQL3: Implementing the Object-Relational Database, Osborne McGraw-Hill,

Frank, M. (1995). Object-relational Hybrids, *DBMS*, 8/8, 46-56.

Garcia-Molina, H., Ullman, J. & Widom, J. 2003. Database Systems: The Complete Book, Prentice Hall, Upper Saddle River.

He, Z., & Jérôme, D. (2005). Evaluating the Dynamic Behavior of Database Applications, *Journal of Database Management*; 16:2, 21-45.

Hoffer, J., Prescott, M., & Topi, H., 2009 Modern Database Management, 9th Edition, Pearson Prentice Hall.

Krishnamurthy, Banerjee and Nori, 1999. Bringing object-relational technology to the mainstream, Proceedings of the ACM SIGMOD International Conference on Management of Data and Symposium on Principles of Database Systems, Philadelphia, PA

Loney, K. & Koch, G. (2002) Oracle 9i: The complete reference, Oracle Press/McGraw-Hill/Osborne.

Mok, W. Y. (2007) Designing nesting structures of user-defined types in object-relational databases, *Information and Software Technology,* 49, 1017–1029.

Pardede, E., Rahayu, J. Wenny, T. & Taniar, D., 2006, Object-relational complex structures for XML, *Information & Software Technology*, 48(6), 370-384.

Philippi, S. 2005, Model driven generation and testing of object-relational mappings, *Journal of Systems and Software*, 77:2, 193-207.

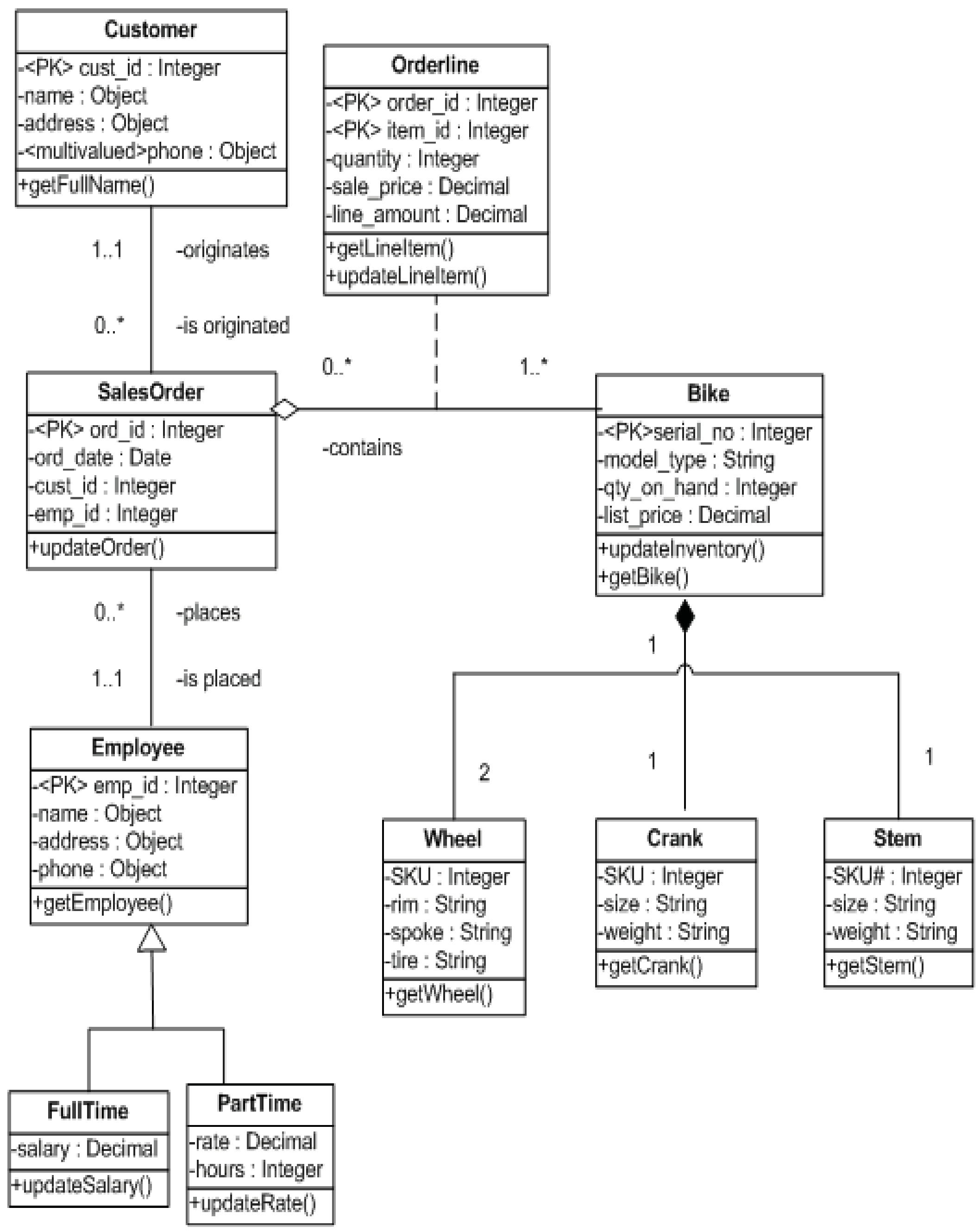
Price, J. 2002. Oracle9i, JDBC Programming, Oracle Press/McGraw-Hill/Osborne

Rahayu, J. W., Taniar, D. And Pardede, E. (2005) Object-Oriented Oracle, IRM Press

Silberschatz, A., Korth, H. and Sudarshan, S. 2009, Database System Concepts, Six Edition, McGraw-Hill

Stonebraker M. and Moore, D. 1996. Object-relational DBMSs: the Next Great Wave. San Francisco, CA: Morgan Kaufmann Publishers, Inc.

Appendix 1 Pacific Trader's Object-Relational Database Design


Customer
-<PK> cust_id : Integer
-name : Object
-address : Object
-<multivalued>phone : Object
+getFullName()
Orderline
-<PK> order_id : Integer
-<PK> item_id : Integer
-quantity : Integer
-sale_price : Decimal
-line_amount : Decimal
+getLineItem()
+updateLineItem()
1..1
-originates
0..*
-is originated
0..*
1..*
SalesOrder
-<PK> ord_id : Integer
-ord_date : Date
-cust_id : Integer
-emp_id : Integer
+updateOrder()
-contains
Bike
-<PK>serial_no : Integer
-model_type : String
-qty_on_hand : Integer
-list_price : Decimal
+updateInventory()
+getBike()
0..*
-places
1..1
-is placed
1
2
1
1
Employee
-<PK> emp_id : Integer
-name : Object
-address : Object
-phone : Object
+getEmployee()
Wheel
-SKU : Integer
-rim : String
-spoke : String
-tire : String
+getWheel()
Crank
-SKU : Integer
-size : String
-weight : String
+getCrank()
Stem
-SKU# : Integer
-size : String
-weight : String
+getStem()
FullTime
-salary : Decimal
+updateSalary()
PartTime
-rate : Decimal
-hours : Integer
+updateRate()