OBJECT ORIENTED EXTENSIONS TO SQL

Thomas B. Gendreau
Computer Science Department
University Wisconsin – La Crosse
La Crosse, WI 54601
gendreau@cs.uwlax.edu

Abstract

Object oriented technology is influencing many areas of software development including database systems. Extensions to SQL in the 1999 and 2003 standards include support for some object-oriented concepts. The data model used in these standards is called the object-relational data model. This model includes all the traditional ideas found in the relational data model and includes some ideas from the object data model. The object-oriented additions to SQL include features that enable the creation of classes, class hierarchies and objects. The object-relational data model is not a pure object data model because the top-level items in the database are restricted to be tables. Tables can include object values as attributes and typed tables include only objects. Oracle servers implement some of the object-relational features and can be used for object-oriented design projects that include a database component.

Introduction

Object-oriented analysis and design is a common methodology for analyzing and designing software systems. Frequently these systems include a database component. Standard rules exist to translate an object-oriented design of a database system into a relational database implementation but it would be better if the database software directly supported the object-oriented design.

SQL is the standard relational database definition and manipulation language. Extensions to SQL in the 1999 and 2003 standards include object-oriented features [1][2]. These features support an extension of the relation data model called the object-relational data model. This model includes all the traditional ideas found in the relational data model and some features of the object-oriented data model.

This paper presents an overview of the object-oriented data model, the features of that model that are included in the object-relational extensions to SQL and a brief example illustrating the use of these features in the context of an Oracle 9i database server.

Object Oriented Data Model

The object-oriented data model is an extension of object-oriented ideas into the area of database systems. The fundamental concept is that a database consists of persistent objects that can be defined and manipulated by object-oriented languages such as C++ or Java. A persistent object is an object that continues to exist after the program that created the object is no longer running. The object can store any type of data. A database could include table objects but the database would not be restricted to only table objects.

One advantage of the object-oriented data model is increased flexibility in the kinds of data that can be stored in a database. The relational data model is good at manipulating values like names and prices but is limited in its ability to store more complex objects such as images or movies. In an object-oriented database the contents can be queried through methods associated with objects in the database. This creates the possibility of creating search strategies more closely tied to the data types being search.

Other potential advantages of object-oriented databases are opportunities to reduce redundancy of data by keeping references to objects instead of copies of values, the ability to use the same (object-oriented) analysis and design methodology for both software development and database design (a database is frequently a large component of software systems) and the possibility of using the same language for software development and database definition and manipulation.

Two sources of information on object-oriented databases include work done by the Object Database Management Group[3] and work done on Java Data Objects standard[4].

SQL Object-Relational Features

The object-relational extensions to SQL include row types, collection types, user defined types (UDT), typed tables and reference types. The object-relational data model does not support all the features of the object-oriented data model. Instead it represents an evolutionary step from the relational data model to the object-oriented data model. In this section examples are given based on the standard syntax as described in [1] and [2].

A row type allows a table to have attributes that are not atomic. This violates the first normal form constraint (1NF) and in general the object-relational data model does not require tables to be in 1NF. The following table includes a row attribute called name and a row attribute called address. Name contains fields for the first name and last name. Address contains fields for street, city, state and zip.

```
create table Faculty   (fid varchar(10) primary key,
                        name  row(first varchar(10), last varchar(30)),
                        address row(street varchar(30), city varchar(30), state
                                                          char(2), zip char(10))
```

The fields of a row type can be accessed by path expressions. For example if f is an alias for the Faculty table in a query, the path expression f.name.first could be used to refer to the first name of a Faculty member.

Collections in SQL:1999 include arrays and SQL:2003 added multisets. Both types allow collecting related data under one attribute name. Both collection types can be used to represent one-to-many relations by keeping references (see below) to objects that are participating in the relationship in a collection attribute of the subject table.

The most important object-oriented feature added to SQL is a user defined type (UDT). The definition of a UDT is similar to the definition of a class. It can include attributes and methods and supports inheritance. The following expression defines a new UDT called Person.

```
create type Person as (pid varchar(10), pname varchar(20)) not final
```

Given this definition a table could have an attribute of type person and the parts of an instance of a person can be accessed through a path expression. For instance a table that keeps track of a call list could be created by a statement like “create table CallList (callee Person, …)”. With this table structure, the name of a person in a row in the table CallList could be referenced by the path expression c.callee.pname (where c is an alias for the CallList table).

The UDT Person could also be the root of an inheritance hierarchy (limited to single inheritance). The key words “not final” indicate that subtypes of Person can be defined. The following expressions create two new UDTs called customer and employee both of

which inherit from Person. The key word "under" is similar to the key word "extend" in Java.

create type Customer **under** Person **as** (…)

create type Employee **under** Person **as** (…)

Other modifiers associated with create type statements include "not instantiable", "instantiable" and "final". When a type is labeled not instantiable it means objects of that type cannot be create. This is similar to abstract classes in Java. Objects can be created from instantiable types. When final is attached to a type it means the type cannot have subtypes (or subclasses).

New types can be used as the type of an attribute in a table. When types are used this way separate object values of the type are created for each row in the table. Another way to use the new types is to create a typed table. The expression below creates a typed table of Customer objects.

create table Clients **of** Customer (primary key (pid)
ref is clientId **system generated)**

The Clients table is now a table of objects. The objects in the table include the attributes pid and pname from Person and all the additional attributes defined in Customer (since it inherits from Person). The pid attribute will be used as the primary key (Note the type definition does not include a primary key designation). The name clientId will be an object reference. Each object of a typed table has a unique reference. This reference is considered one of the attributes in the table. So in addition to all the attributes the Client table gets from Customer it also has an attribute called clientID that has type ref. In this case the value of that reference will be system generated. A reference value can be generated by the system or by a user. Reference (ref) types can be attributes of a table. Reference types can be used to represent relationships and can help reduced data redundancy by keeping references to an object instead of copies of an object's value. For example information about the customers an agent sells to, could be saved in a collection attribute, such as an array or multiset, of references.

Methods can also be associated with a UDT. For instance an Agent UDT could include an addCustomer method that would be used to add new clients to the agent's collection of clients. Clients could be added directly with a SQL expression but by creating a method the Agent UDT can hide the representation of the collection of customers. The SQL standard says that all types should have a default constructor method and observer and mutator methods for all attributes.

The SQL UDT does not support all the features that are commonly considered part of a class definition. The concept of private or public attributes is not supported. All attributes and methods are considered public. (The database can still restrict rights though access rights and views but those actions are not part of the UDT definition).

Examples

Oracle 9i implements some of the object-relational features of SQL. The following give some examples of using object-relational features in Oracle 9i. Some of the syntax is Oracle specific since all these examples were executed on an Oracle 9i server.

The first example shows the creation of a simple UDT hierarchy for a business. The hierarchy implements a Person UDT and two subtypes: Agent and Customer.

```
create type Person as Object (pid varchar(10), pname varchar(30),
        address varchar(50)) not final

create type Agent under Person (commission int) not final

create type Customer under Person (discount int) not final
```

Objects can also be attributes of other objects. To illustrate this a new Person type is shown below. The Person type has been created to include an address UDT. (Assume the Agent and Customer types were recreated as above.) After the types are created a typed table of Agent objects is created and a new row is inserted into the table. Note the constructor for Address that is used to create a instance of an Address object that is part of the row inserted into Agents. The constructor is automatically created for a UDT. It is possible to create programmer-defined constructors.

```
create type Address as Object (street varchar(30), city varchar(30),
                                state char(2), zip char(10)) not final

create type Person as Object ( pid varchar(10), pname varchar(30),
                                pAddr Address) not final

create table Agents of Agent (pid primary key) object identifier is primary key

insert into Agents values ('0000000001', 'John Smith', Address('123 Main',
                                'La Crosse', 'WI', '12345-6789'), 4)
```

An alternative to the above insert statement is shown below. Notice the explicit use of the Agent constructor.

```
insert into Agents values (Agent('0000000011', 'John Smith',
            Address('123 Main', 'La Crosse', 'WI', '12345-6789'), 4))
```

The example can be extended to implement a one-to-many relationship between agents and customers. Assume that an agent can sell to many customers and a customer buys from exactly one agent. The standard way to do this in a relational database is to put the primary key of the agent table into the customer table as a foreign key. The object-

relational features of Oracle create two additional ways to implement this relationship. The first way includes adding a varray of customer references as an attribute in the Agent UDT. The second way is to use a nested table in the Agent UDT.

The first statement shown below creates a new type for a varray of customer references. A varray is a variable length array supported by Oracle. It is similar to the array type specified by SQL:1999. The second statement shows the Agent type modified to include a varray of customer references.

```
create type customerList as varray of ref Customer

create type Agent under Person (commission int, clist customerList)
```

The following statements illustrate the use of nested tables. The first statement creates a table type that can be used as an attribute within another UDT. The second statement redefines Agent to include a nested table. The nested table is a table of customer references. Remember customer is defined above as a type. In order to make use of the nested table in the Agents table a table of customer objects would have to be created and references to the objects in that table (the table of customer objects) would be inserted into the Agents table. The third statement creates the Agents table. In SQL select statements the attribute “clients” would be used to access the nested table. The name clientList id used by Oracle to access additional storage structures created by the nested table. The alter statement indicates that the customer references in Agents should reference objects in the Customers table (as opposed to any Customer object).

```
create type nestedCustomerTable as table of ref Customer

create type Agent under Person (commision int,
                                clients nestedCustomerTable)

create table Agents of Agent (pid primary key)
                    object identifier is primary key
                    nested table clients store as clientList

alter table clientList add (scope for (column_value) is Customers)
```

After the above students are executed rows can be inserted into the Agents tables by the following statements.

```
insert into Agents values ('0000000021', 'John Smith', Address('123 Main',
      'La Crosse', 'WI', '12345-6789'), 4, nestedCustomerTable())

insert into table (select a.clients from Agents a where a.pid = '0000000021')
      select ref(c) from Customers c where c.pid = '0000000011'
```

The first insert statement puts a new row in Agents. The expression "nestedCustomerTable()" indicates an empty nested table is the initial value for the clients attribute. The second insert statement puts the reference value for the customer with id 0000000011 into the nested table called clients in the Agents table. The first select in the insert statement selects the table (that is the client nested table of Agent 0000000021) into which the reference should be stored.

The last feature we will show in this example is a method. UDTs can define their own methods (functions and procedures in PL/SQL terminology). Methods can be either static methods or member methods. In Oracle methods are most frequently written in PL/SQL although Java is also supported as a stored procedure language. Other languages can be used to write UDT methods but only PL/SQL and Java methods will be stored in the database. The statement below includes a member function as part of the Agent UDT. In PL/SQL the implementation would be create in separate "create type body" expression.

```
create type Agent under Person (commision int,
        clients nestedCustomerTable,
        member function customerCount return int)
```

These examples give a brief look and some of the object-relational features that Oracles implementation of SQL support. Top-level objects must be tables but otherwise there are many ways to practice with object-oriented database design. For more details see [1][2][5]

Conclusion

The object-relational features of SQL provide a way to illustrate object-oriented concepts in a database environment. I have taught this material as part of an advanced database class. I think it would accessible to students in my introductory database class but I have not taught it in that class. Another place this material would be useful is in a software engineering class that emphasizes object-oriented design.

References

1. Kifer, Michael, Bernstein, Arthur and Lewis, Philip. (2004) *Database Systems: An Applications-Oriented Approach* 2nd Edition. Addison Wesley.

2. Dietrich, Suzanne and Urban, Susan. *An Advanced Course in Database Systems: Beyond Relational Databases*. Pearson Prentice Hall.

3. www.odmg.org

4. www.jdocentral.com

5. *Oracle9i Application Developer's Guide – Object-Relational Features (2002).* Oracle.