
Working with Oracle8 Datatypes

Elizabeth Boss

Boss Consulting Services, Inc.

Abstract

Oracle8 introduces many new datatypes that provide additional functionality in both PL/SQL programs and database table columns. Oracle8 can incorporate object-relational database concepts including the use of the new object data type. An object-relational database includes standard relational features and object-oriented constructs such as varying arrays, nested tables, and abstract datatypes. The primary advantage of incorporating object-oriented methodology in a database is to allow data modeling to more closely depict the real world than standard relational tables do. Other advantages of using object-oriented design include object reuse, standardization, and pre-defined access methods.

The abstract datatypes include object tables that contain multiple columns and rows within a single object. Nested tables contain a table inside the table. Nested tables are similar to a one-to-many relationship that would be modeled using two tables in a relational design. Varying arrays are similar to nested tables but can contain only a predefined number of rows.

Object Datatypes

An object datatype can be created using the CREATE OR REPLACE TYPE command. The new datatype will be stored in the database and therefore, be accessible to any other developer who is provided the necessary access privileges. The basic syntax for creating a user-defined data type is:

```
CREATE [OR REPLACE] TYPE type_name AS OBJECT
[attribute_name datatype [, attribute_name datatype. . .]]
[MAP|ORDER MEMBER function_spec]
[MEMBER function_spec|procedure_spec . . .]
[PRAGMA RESTRICT_REFERENCES(method_name, constraint_name). . .]);
```

The *type_name* is the name of the abstract data type being created.

The *attribute_name* is the name of the attribute being specified.

The *datatype* is the Oracle datatype being specified for the attribute.

The *function_spec* is the specification statement of the function that will be used as a method for the object.

The *procedure_spec* is the specification statement of the procedure that will be used.

The *method_name* is the name of the method specified for the abstract datatype.

The *constraint_name* is the name of the constraint that limits the access a method has against the object and includes the following options: WNDS – write no database state (no tables modified), WNPS – write no package state (no package variables modified), RNDS – read no database state (no queries performed), and RNPS – read no package state (no package variables referenced).

Object Type Example

The following example uses the CREATE TYPE command to create a new object datatype:

```
SQL> CREATE TYPE address_type as OBJECT
      (street varchar2(120),
       city varchar2(30),
       state char(2),
       zip number(5));
      /
```

This example creates a new abstract data type that is an object type named address_type containing four attributes. When the CREATE TYPE command is issued, a new database object is created, in this case, a new data type. Because the datatype is stored in the data dictionary, any application, database table, or program that has the appropriate privileges can access it.

To determine what user-defined types have been created, query the USER_TYPES or ALL_TYPES data dictionary table.

```
SQL> SELECT type_name,
           typecode
      FROM user_types
      ORDER BY type_name;
```

The following GRANT command grants privileges on the object type named address_type to the user student1. Note: a user cannot obtain the EXECUTE privilege through a role.

```
SQL> GRANT EXECUTE ON address_type TO student1;
```

Using Objects in Database Tables

After the new datatype has been created, it can be referenced in the column list of a CREATE TABLE command. The following example creates the customer table using the address_type datatype for a column datatype.

```
SQL> CREATE TABLE customer
      (customer_id varchar2(5),
       last_name varchar2(20),
       first_name varchar2(14),
       address address_type);
```

When the customer table is described, the column defined using the object datatype is displayed showing only the named datatype. When an object is described that contains an abstract object, the attributes of the abstract object are not displayed. The following example describes the customer table:

Name	Null?	Type
CUSTOMER_ID		VARCHAR2(5)
LAST_NAME		VARCHAR2(20)
FIRST_NAME		VARCHAR2(14)
ADDRESS		ADDRESS_TYPE

The attributes of the address_type are not displayed when the customer table is displayed. The DESCRIBE command simply identifies column names and the datatypes associated with the column names. Because address_type is a user-defined datatype, no further information is displayed other than the datatype. Additional information pertaining to the datatype address_type can be obtained through a query against the data dictionary table USER_TYPE_ATTRS or ALL_TYPE_ATTRS.

The following query provides attribute information for the address_type datatype.

```
SQL> SELECT attr_name,
           attr_type_name,
           length
FROM user_type_attrs
WHERE type_name = 'ADDRESS_TYPE';
```

ATTR_NAME	ATTR_TYPE_NAME	LENGTH
STREET	VARCHAR2	40
CITY	VARCHAR2	30
STATE	CHAR	2
ZIP	NUMBER	5

Inserting into Abstract Datatypes

When an abstract datatype is created, a special method called a constructor method is created for the datatype. The constructor method is a program that contains input arguments that are the attributes of the abstract datatype. The constructor method can be used to insert data into the abstract datatype. The following example inserts into the customer table:

```
SQL> INSERT INTO customer
(customer_id,
 last_name,
 address)
VALUES('11111',
 'Smith',
 address_type('111 S. Main',
 'Moscow',
 'ID',
 83843));
```

Notice that the constructor name is the same as the object name; in this case, address_type. The address datatype in this example is scalar – only a single address can be added for each customer. Later you will see how to create a datatype that allows multiple columns and rows.

Querying Abstract Datatypes

A simple SELECT statement can be used to retrieve the data from the table containing the abstract datatype. When selecting an attribute from an abstract datatype an alias must be used for the table name. The following example retrieves all information from the customer table:

```
SQL> SELECT * FROM customer;

CUSTO LAST_NAME          FIRST_NAME
-----
ADDRESS(STREET, CITY, STATE, ZIP)
-----
11111 Smith
ADDRESS_TYPE('111 S. Main', 'Moscow', 'ID', 83843)
```

The following example retrieves a single attribute from the abstract datatype – notice the use of the alias for the table name.

```
SQL> SELECT c.address.street
      FROM customer c;
```

```
ADDRESS.STREET
-----
111 S. Main
```

Updating Abstract Datatypes

An individual attribute from an abstract datatype may be updated in a table using a simple UPDATE command. The UPDATE command must use an alias for the table name. The following example updates the zip code attribute of the address column:

```
SQL> UPDATE customer c
      SET c.address.zip = 83844
      WHERE customer_id = '11111';
```

Notice in this example, the column name, address, used prefaced with the table alias. The syntax for updating the column is column.attribute_name.

Deleting Using an Attribute

A row may be deleted from a table using an attribute from an abstract datatype as the filter criteria in the WHERE clause of the DELETE statement. A table alias must be used when referencing an attribute from an abstract datatype. The following example deletes using the zip code attribute in the WHERE clause:

```
SQL> DELETE FROM customer c
      WHERE c.address.zip = 83843;
```

Creating an Object View

If a database contains relational tables one method for implementing an object-oriented design is through the use of object views. An object view allows the developer to use the existing relational tables to set up views that contain references to abstract datatypes. Using the standard relational table Customer, object types can be created.

```
SQL> DESC customer
```

Name	Null?	Type
CUSTOMER_ID	NOT NULL	VARCHAR2(5)
LAST_NAME		VARCHAR2(20)
FIRST_NAME		VARCHAR2(14)
ADDRESS		VARCHAR2(120)
CITY		VARCHAR2(30)
STATE		CHAR(2)
ZIP		NUMBER(5)
AIRLINE_PREF		CHAR(2)
SEAT_PREF		CHAR(1)
SALUTATION		VARCHAR2(10)
GENDER		CHAR(1)
DATE_OF_BIRTH		DATE
LAST_PURCH_DATE		DATE

An object view can be created containing the appropriate address attributes found in the customer table. This object datatype was created earlier and named address_type. A second object type can be created that contains the other appropriate attributes from the customer table and the address type.

```
SQL> CREATE TYPE cust_type AS OBJECT
      (customer_id varchar2(5),
       date_of_birth date,
       address address_type);
```

This object, `cust_type` includes two standard attributes – `customer_id` and `date_of_birth`. A nested object, `address`, is also included. A view can then be created using the abstract datatypes defined earlier – `address_type` and `cust_type`.

```
SQL> CREATE VIEW customer_view
      (customer_id,address,date_of_birth)
AS SELECT customer_id,
          address_type(address,
                      city,
                      state,
                      zip),
          date_of_birth
FROM customer;
```

Accessing an Object View

An object view can be described as any other table or view would be, using the `DESCRIBE` command. The following example describes the `customer_view`.

```
SQL> DESC customer_view
```

Name	Null?	Type
CUSTOMER_ID		VARCHAR2(5)
ADDRESS		ADDRESS_TYPE
DATE_OF_BIRTH		DATE

Columns and rows can be selected from the object view as they would from any other table that contains an abstract datatype. It is important to remember that an alias must be used for the view.

```
SQL> SELECT customer_id,
          c.address.city
FROM customer_view c
WHERE c.address.state = 'CO';
```

Creating a Method

A method can be created that defines an access path for an object view. When the object datatype is created the member function must be specified. The following example creates an object datatype named `cust_type`.

```
SQL> CREATE OR REPLACE TYPE cust_type AS OBJECT
      (customer_id varchar2(5),
       date_of_birth date,
       address address_type,
       MEMBER FUNCTION age(date_of_birth date) RETURN number,
       PRAGMA RESTRICT_REFERENCES(age, wnds)
      )
/
```

The `CREATE TYPE BODY` command is used to create a method.

```
SQL> CREATE OR REPLACE TYPE BODY cust_type as
MEMBER FUNCTION age(date_of_birth date) RETURN number
IS
BEGIN
  RETURN(TRUNC(months_between(sysdate,date_of_birth)/12));
END;
END;
/
```

Next, a table could be created that references the new object `cust_type`. The following CREATE TABLE command creates a table named `cust`, which includes the `cust_type` datatype.

```
SQL> CREATE TABLE cust
      (last_name varchar2(40),
       first_name varchar2(30),
       cust_info cust_type);
```

Data would be inserted into the table as described in the section *Inserting into Abstract Datatypes*. To retrieve customer information and the age of the customer, the `age` method can be called from an SQL SELECT statement. The following query displays the customer name and age.

```
SQL> SELECT last_name,
           first_name,
           c.cust_info.age(c.cust_info.date_of_birth)age
      FROM cust c;
```

Collection Datatypes

A collection datatype allows a group of individual variables to be combined into a single variable. Collection datatypes allow the programmer to manipulate a series of values within a single datatype. Oracle8 provides two new collection datatypes: nested tables and varrays. Oracle7 introduced the PL/SQL table collection datatype.

Nested Tables

A nested table is similar in functionality to a PL/SQL table datatype. The advantage that a nested table provides is that the nested table can be stored in a standard Oracle database table. The nested table is similar in structure to a database table, in that it is made up of a key column (similar to the index by column in a PL/SQL table) and the value column. The key values that are loaded into the nested table may be sparse, i.e., the values do not need to be sequential numbers.

Creating a Nested Table

The syntax for creating a nested table is very similar to that of a PL/SQL table. The nested table syntax does not require the `INDEX BY BINARY_INTEGER` clause that the PL/SQL table datatype uses. The basic syntax for a nested table is:

```
TYPE table_type_name is
TABLE of datatype [NOT NULL];
```

The following examples declare several nested tables and create variables based on the table datatypes.

```
declare
    type name_type is table of varchar2(30);
    type person_type is table of agent%rowtype;
    customer_name name_type;
    agent_table person_type;
```

Initializing Nested Tables

When a table datatype is initially created, it is automatically set to NULL. If the program attempts to initialize the table datatype to NULL, the predefined exception `COLLECTION_IS_NULL` is raised. Values may be added to a nested table using the constructor method. The constructor is created automatically when the table is created. The constructor for a table has the same name as the table (Just like an object constructor).

The following example adds values to a nested table:

```
declare
    type name_type is table of varchar2(30);
    type person_type is table of agent%rowtype;
    type num_type is table of number;

    number_table num_type := num_type(1,2,3,4,5);
    customer_name name_type := name_type(1,2);
    agent_table person_type;

begin
    customer_name(1) := 'Andy Jones';
    customer_name(2) := 'Jennifer Johnson';
end;
/
```

Adding Elements to a Table

The following PL/SQL program adds the agent first name and last name to the nested table for all agents in the database.

```
Declare
    type name_type is table of varchar2(30);

    cursor agent_cursor is
        select first_name||' '||last_name name
        from agent;

    customer_name name_type := name_type(1,2,3,4,5);

begin
    for agent_rec in agent_cursor loop
        customer_name(agent_cursor%rowcount) := agent_rec.name;
        dbms_output.put_line(customer_name(agent_cursor%rowcount)|| ' Added. ');
    end loop;
end;
/
```

In this example, the customer_name table is initialized and contains 5 elements. The loop causes an error condition to be raised when the 6th element is added to the nested table.

```
Joyce Davis Added.
Mark Lindquist Added.
Denise Abrahams Added.
Alan Smith Added.
Jennifer Johnson Added.
Declare
*
ERROR at line 1:
ORA-06533: Subscript beyond count
ORA-06512: at line 18
```

The EXTEND method can be used to add additional values to an existing nested table. The EXTEND method is a predefined method that can be used with collection datatypes. The EXTEND method can be called with the following input arguments:

EXTEND – adds a null element to the table.

EXTEND(n) – adds n elements to the table.

EXTEND(n,i) adds n copies of element i to the table.

The following example adds one copy of the first element to the table for each iteration of the loop.

```
Declare
  type name_type is table of varchar2(30);
  cursor agent_cursor is
    select first_name||' '||last_name name
    from agent;
  customer_name name_type := name_type(1,2,3,4,5);
begin
  for agent_rec in agent_cursor loop
    customer_name(agent_cursor%rowcount) := agent_rec.name;
    dbms_output.put_line(customer_name(agent_cursor%rowcount)||
      ' Added. ');
    customer_name.extend(1,1);
  end loop;
end;
/
```

Using Nested Tables in the Database

A nested table can be added to a database table as a column in the table. When a nested table is stored within a column, each row in the database table contains the nested table. Each row can have a different nested table. The CREATE TYPE statement can be used to create a nested table type that will be stored in the data dictionary. The datatype will then be available for use in the CREATE TABLE statement.

The following example creates an object named address and then creates a table datatype named address_table.

```
SQL> create type address as object
  (address_type char(1),
  street varchar2(20),
  city varchar2(20),
  state char(2),
  zip varchar2(9));

SQL> create type address_table
  as table of address;
```

After the new table datatype has been created, it can be referenced as a column datatype in the CREATE TABLE command. The following example uses the recently created address_table datatype as a column datatype.

```
SQL> create table new_agent
  (agent_id varchar2(5),
  first_name varchar2(20),
  last_name varchar2(25),
  agent_address address_table)
  nested table agent_address store as address_tab
  /
```

The NESTED TABLE clause is required when a nested table is included in a database table. The NESTED TABLE clause specifies the name of the system-generated table that will actually be used to store the data. This system generated table is called the “store” table. Using the STORE AS clause allows the specification of a tablespace for the nested table.

Inserting into a Nested Table

To insert values into a table containing a nested table use a standard SQL INSERT command. The basic syntax for the INSERT command is:

```
INSERT INTO table_name
values(value1,value2...,
  table_datatype(object_name(value3,
  value4...))
```

The *table_datatype* is the name of the table datatype that was created earlier.

The *object_name* is the name of the object datatype that was created earlier and used as the datatype for the table type.

The following example inserts one row into the database table for a travel agent and two rows into the *agent_address* nested table.

```
SQL> insert into new_agent
      values('11111',
            'James',
            'Hartman',
            address_table(address('H',
                                  '1190 S. Bend',
                                  'Pullman',
                                  'WA',
                                  '98119'),
                          address('W',
                                  '500 Main Street N.',
                                  'Moscow',
                                  'ID',
                                  '83843'))));
```

Updating a Nested Table

The values in a nested table stored in a database table may be modified using the UPDATE command. The basic syntax for the UPDATE command is:

```
UPDATE TABLE table_name
SET nested_tablecolumn = table_datatype(
    object_name(value1,
                value2...),
    object_name(value1,
                value2...))
```

The following example updates the *new_agent* table and sets the nested table column *agent_address* to two new rows.

```
SQL> update new_agent
      set agent_address = address_table(address('H',
                                                '6042 Highland',
                                                'Englewood',
                                                'CO',
                                                '80112'),
                                       address('W',
                                                '500 E. Arapahoe',
                                                'Englewood',
                                                'CO',
                                                '80111'))
      where agent_id = '22222';
```

Deleting a Row with a Nested Table

The DELETE statement can be used to remove a row containing a nested table. The basic syntax is:

```
DELETE FROM TABLE table_name
WHERE column = value;
```

The following example deletes the travel agent with an *agent_id* of '22222'.

```
SQL> DELETE FROM new_agent
      WHERE agent_id = '22222';
```

Displaying Data from a Nested Table

The following PL/SQL program unit displays the agents and their addresses as stored in the nested table `agent_address`. Because the built-in package `DBMS_OUTPUT` is referenced, the SQL*Plus session variable `SERVEROUTPUT` would need to be set to `ON` prior to executing this procedure.

```
Declare
  v_address new_agent.agent_address%type;
  v_agent   new_agent.agent_id%type;
  cursor agent_cursor is
    select agent_id,
           agent_address
    from new_agent
    where agent_address is not null;
begin
  if not agent_cursor%isopen then
    open agent_cursor;
  end if;
  loop
    fetch agent_cursor
    into v_agent,
         v_address;
  exit when agent_cursor%notfound;
    dbms_output.put('Agent: '||v_agent||' has street addresses: ');
    dbms_output.new_line;

    for inx in 1..v_address.count loop
      dbms_output.put_line(inx||'. '||v_address(inx).street);
    end loop;
  end loop;
  if agent_cursor%isopen then
    close agent_cursor;
  end if;
end;
/
```

The output from this example would be:

```
Agent: 11111 has street addresses:
1. 1190 S. Bend
2. 500 Main Street N.
```

Using Varrays

A variable length array (varray) is similar to an array in the C programming language. The declaration for a varray is similar to that of a nested table or PL/SQL index-by table. A varray is unlike a nested table in that it is not sparsely populated; the varray starts with an index value of 1 and is incremented until it reaches the upper bound for the array.

Declaring a Varray

The basic syntax for creating a varray is:

```
TYPE varray_name IS [VARRAY|VARYING ARRAY]
(maximum_size)
OF datatype [NOT NULL]
```

The *varray_name* is the user-defined name for the new datatype.

The *maximum_size* is the upper bound for the number of elements in the array.

The *datatype* is the datatype for the varying array. Varrays may not be declared with the following types: `BOOLEAN`, `NCHAR`, `NCLOB`, `NVARCHAR2`, `REF CURSOR`, `TABLE` or another `VARRAY` type.

The following DECLARE section shows examples of declaring a varray in a PL/SQL program unit.

```
Declare
  type v_addresses is varray(5) of address;
  type v_agents is varray(12) of name_type;
```

Initializing a Varray

A varray is similar to a table type in that it must be initialized prior to adding elements to the array. The following program initializes and adds elements to a varray.

```
Declare
  type v_agents is varray(13) of varchar2(20);
  agent_name v_agents := v_agents(NULL);

  cursor agent_cursor is
    select first_name
    from agent
    order by first_name;
begin
  for agent_rec in agent_cursor loop
    agent_name(agent_cursor%rowcount) := agent_rec.first_name;
    dbms_output.put_line(agent_name(agent_cursor%rowcount));
    agent_name.extend(1,1);
  end loop;
end;
/
```

The array cannot be extended beyond the size of the array. Only use a varray when you know the number of elements that will be used in the array up front.

Storing Varrays in the Database

Varrays may be stored in the database as a datatype for a column in a database table. Varrays provide a more limited manipulation capability than nested tables. Varrays may only be manipulated in their entirety. The individual elements of a varray may not be manipulated.

The following example creates a varray datatype that is stored as a database object:

```
SQL> create or replace type address_table2
  as varray(13) of address;
/
```

Next, a table is created using the new data type.

```
SQL> create table new_agent2
  (agent_id varchar2(5),
  last_name varchar2(25),
  first_name varchar2(20),
  agent_address address_type);
```

Data would be inserted into the varray column just as the nested table data was loaded.

Conclusion

Object datatypes can be used to more closely model the real world and the way in which we describe a “thing” in real life. The CREATE TYPE command is used to create a user-defined datatype that is stored in the database. A table can be created that uses a user-defined datatype for one or more columns in the table. Methods are used as a means of accessing an object datatype and are written as PL/SQL functions or procedures. Object views can be used as a method of implementing object-oriented design without recreating existing relational tables.

Nested tables can be created and stored as objects in the database. A nested table can be the datatype for a column in a database table. Nested tables allow a single row in a table to have a column that also contains rows – essentially “pre-joining” two tables. Varrays are similar to nested tables in that they allow for storage of multiple rows. The varray datatype can be stored in the database as an object or declared in a PL/SQL program unit. A varray has a predetermined number of elements and cannot be extended beyond that number.

About the Author

Elizabeth Boss is president of Boss Consulting Services, Inc. She has more than 15 years of experience in application development and database administration as a database administrator, consultant, instructor, and curriculum developer. As a senior consultant/instructor, Elizabeth has worked directly with customers in all phases and aspects of the design, development, and administration of Oracle systems. She is a frequent presenter at international and local user groups, and, in 1997, was listed among the top 25 speakers at the IOUG-A Conference.

Elizabeth can be reached at:

Boss Consulting Services, Inc.

710 Kings Deer Point

Monument, CO 80132

Phone: (877) 489-7745 Fax: (719)481-5820

E-mail: eboss@boss-consulting-inc.com

Web: www.boss-consulting-inc.com