Lecture Notes for

Database Management, using
Database Principles, Programming and Performance
Notes by Patrick E. O'Neil and Elizabeth O'Neil

Chapter 4 Object-Relational Databases

Introduction

In the sense that we will use it, an object holds together a bundle of related information. An employee has a name, id, address, etc., so we can have an employee object for each employee. A document has a title, author, length, and date, so we can have a set of document objects in a library object.

All the employee objects are of a defined **object type**. Associated with an object type are the data attributes it has and the functions (methods) defined for it.

Here we use "object" in a loose sense, **without encapsulation**, even though some object-oriented experts would say that encapsulation is so essential to objects that we are mis-using the word object.

Encapsulation is also known as data-hiding, and is a software technology that builds walls around a combination of code and data and only allows interaction with other parts of the code via methods or messages. In Java, a class with only private data members is encapsulated.

The object types we will be using are like Java classes with all public data members, or structs in C; the data inside can be accessed easily. The object can be thought to *organize* the data without *policing* it.

Relational databases have always had the outlook that all data is accessible (except for secret data) so that you don't have to know ahead of time how you might use it. The data is strongly organized into neat tables of rows and columns. The object types give us a new way to organize data items.

But it does break RR1 by allowing combination data items. The collection types break RR1 another way by allowing whole sets of data items as column values. Look at Figure 4.1. Also read Chapter 1.

We will cover:
Section 4.2.1--ORACLE object types.
Section 4.3.1--ORACLE collections

We don't have time to cover how to write object methods (Section 4.4.1) in PL/SQL, but encourage you to read it, and also read the last section on packaged software.

Look at Example 4.2.1. Like a C struct declaration or Java class declaration. Like these, it provides a **new type,** an **object type,** out of a combination of other types, called members in C/Java, **attributes** in Oracle object types.

This new object type can be used wherever ordinary SQL data types can be used, for ex., as a column type.

Ex. 4.2.2 `create table teachers ...` Note that teachers is a normal "relational table" that has a column type that is an object type. After this, have an empty table.

Next, insert some rows. Now finally have "objects", not just object types. Objects are instances of object types, that is, real data held together as prescribed by the type. Here the name_t objects are held as column values of the tname column.

```
select * from teachers;
```

This shows all the column values, including the tname column of object type name_t:

```
tid     tname                              room
----    -------                            ------
1234    name_t('Einstein', 'Albert', 'E')    120
1235   name-t(...)                           144
...
```

Note the display of the name_t objects. Since the name_t objects are column values, they are called **column objects**.

```
select t.tname.fname from teachers t;
```

This prints out just the fname. Note the dotted expression that digs down from the table alias all the way to the fname.

NOTE: The table alias is needed here and anywhere more than one dot is used, in Oracle.

```
select room from teachers
  where t.tname.lname like ’%Neil’;
```

Example 4.2.3--here we have an object type in an object type.

Table teachers was still a relational table, but table people is an **object table**, a new idea. Each row of people is of type person_t, and thus is called a **row object.**

We see that rows (with their columns) and object types (with their attributes) both provide ways of grouping several typed values together.

Table people has row objects of type person_t, and each of these has attributes ssno, pname, and age. These attribute names map naturally to column names. So we can say table people has columns ssno, pname, and age derived from the row object attributes.

Object table
  -- each full row is of an object type
  -- each row is a row object with certain attributes
  -- the attributes of the row object provide effective column names
  -- the table provides a natural home for objects of that object type

Relational table
  -- each full row has one or more columns each of some data type, but no type describes the whole row.
  -- some columns may have object types, providing column objects.

See Figure 4.2 for a picture of this.

There is a "duality" here:
an object table is, at the same time:
  --a table of row objects
  --a table of rows and columns where the column names are the top-level attribute names of the row object.

Look at Examples after Figure 4.2.

The whole row object can be evaluated by the expression "value(p)" where p is the table alias for the object table.

select value(p) vs. select *: select row object vs all-the-columns--see Figures 4.3 and 4.4. Note how the whole object is represented by

```
        name_t('Delaney', 'Patrick', 'X').
```

This is also the form of the object constructor, shown in action in Example 4.2.5 to create a name to match in the query.

Dot notation

Just as in C and Java, we use dots to drill down into structured types. For example p.pname.fname specifies the fname attribute of the pname name_t object within the person_t row object referenced by the table alias p. Look at Example 4.2.6 to see this in use in a query.

Oracle is very picky about the use of the table alias in dotted expressions. The easy rule is simply to always use a table alias with dotted paths. In fact, Oracle does support simple cases without table aliases, to be consistent with the relational case. Look at the examples on p. 183.

To insert into an object table, we can view it as having various columns and providing values for them just as in the relational case, as shown in Example 4.2.7.

If we view the same object table in terms of row objects, it would seem natural to be able to insert a whole person_t object in all at once. However, this doesn't fit the Insert statement syntax, which allows either a values clause over the columns or a subquery, and Oracle has not stretched the syntax out to cover this possibility (as of version 8.1.5, anyway.)

However, Oracle has found a way to do a full-object Update of a row by letting the table alias symbolize the whole row, as shown on p. 184.

```
        update scientists s set s = name_t(…) where …;
```

As you can see from Appendix C, p. 773, the use of the table alias in the Update statement is an extension of the standard form.

Object References

The row objects of object tables are somewhat special in that they have unique identifiers (this is related to the fact that they are full-fledged database rows), whereas the column objects are known only by their values. We can utilize this top-level identification information by defining REFs that, in effect, point directly to a row object.

The REF feature provides new built-in datatypes, of type "ref object_type", so we can have table columns that point to row objects in an appropriate object table.

For example in a new version of the CAP database, we can have a "ref product_t" column in orders that points to a whole row object representing a product.

Look at Example 4.2.9, where the details of this new version of CAP are laid out. Each order object has REFs to its customer, agent, and product objects. We still have the old relational identifiers as well, to help with loading the data. But we will utilize the REFs in queries wherever possible.

Note the scope clauses in the create table for orders. These are not required, but allow shorter (thus faster) REFs when they all (of one type) point into one table.

We will gloss over the issue of loading the table now and jump into doing queries on a fully loaded table. The dotted expressions (or "paths") can now follow a REF across to another table. For example

```
        o.ordcust.city
```

(where o is a table alias for orders) accesses the city attribute of the pointed-to customer object via the REF o.ordcust housed in the order object.

In relational CAP, we needed a join on cid to make this association--here it is just a dot away. Further, we know that it is based on a unique identifier that is a "locator." It has information on how to find the row, not just a unique pattern of bits to search for.

Any relational query that does joins on orders with customers on cid, with agents on aid, and/or with products with pid is simplified by using REFs here.

For example, to list all cname, pname pairs where the customer ordered the product, we can just think of the cname and pname "hanging off" the order object:

```
        select distinct o.ordcust.cname, o.ordprod.pname from orders o;
```

Two joins are saved here over the relational query.

Admittedly, only the easiest joins are saved. Many harder queries are not simplified. In Example 4.2.13, the notorious division query is rewritten with REFs. It does make a little clearer how the x row bridges from c to a, but the hard stuff is still all there.

We have to be aware of the possibility of "dangling" REFs. If an order has a REF pointing to a certain customer object, and that row is deleted from the customers table, then the REF in the order object is left "dangling", pointing nowhere, but not NULL.

Oracle has added a new predicate "is dangling" so that these REFs can be found and weeded out. Look at Example 4.2.14.

REF Dependencies

When you execute a Create Type that defines REF attributes, like order_t, the REF-target types (like customer_t, etc.) need to be already defined. We say that type order_t has a "REF type dependency" on customer_t, agent_t, and product_t. Similarly, we need to drop them in the opposite order, order_t first, then the others (in any order.)

This would seem to preclude circular REF type dependencies, but there is a loophole in the system to allow them, since they are so useful in certain circumstances.

For example, and employee_t type could have a REF department_t attribute for the employee's department, while the department_t type could have a REF employee_t attribute for the department's manager.

See pg. 190 for the trick to set this up--we can half-way define employee_t, then fully define department_t, and then finish defining employee_t. To drop these types, a special FORCE keyword needs to be used.

Loading Tables with REFs

Since we are only using REFs as extra columns in the CAP example, we can do the old relational load and then use an Update statement to set all the REFs, as shown in Example 4.2.17. We see it is using matching on the still-existent relational ids.

In general we need a way to specify the target of a REF uniquely by one or more columns, which of course means specifying a key in the target object table. And this key value must be obtainable from the source object, to do the match-up. Thus the simplest approach is to use REF columns in parallel with ordinary relational keys

Collection Types in Oracle

We have talked about the idea of multi-valued attributes before in Chapter 2, and how they break Relational Rule 1. See pg. 32. Relational techniques get rid of multi-valued attributes by setting up additional tables. Then we are constantly joining them back together to get our work done. Now, instead, we'll allow multiple values inside one column value by using collection types.

Look at Figure 4.11 to see collections of dependents for each employee.

Oracle has "nested tables", meaning a whole table (of dependents, for example) in a column value of another table. To make this work, new datatypes are made available, "table types" such as "table of person_t" or "table of int" that can be used as column datatypes.

As with ordinary tables, each nested table is a set of rows, with no order in the rows. Nested tables can be object tables or tables of a single column of a built-in type such as int.

If order needs to be maintained in a collection inside a column value, Oracle provides VARRAYs. We'll look into them after nested tables.

Oracle (version 8.1.5) allows only one level of nesting for nested tables. Employees can have a nested table of dependent_t objects, but these dependents can't have a nested table of hobbies. The employee can have a nested table of hobbies as well as dependents, since that still counts as single-level nesting.

To create a table that contains a nested table within it, we are first expected to define a table type and then come up with two tablenames. One tablename is for the whole table (the "parent" table) and the other is a helper (or "child") table to hold the rows of all the nested tables of one column. If we have two nested table columns, we have to come up with three tablenames.

See Example 4.3.1 for the employees and dependents example.

The syntax for the Create Table statement with nested tables is like the previous Create Table statements with an additional clause or clauses of the form "nested table colname store as tabname". Here the colname matches the columnname of a column of table type.

Although we can see the child tables as database objects (in say "select table_name from user_tables;"--see pg. 448,) we can't use them in queries directly. We always need to access the data through the parent table.

Each nested table is itself a column value of some table type. It is easy to display whole nested tables from SQL, as in Examples 4.3.2 and 4.3.3:

```
        select dependents from employees where eid = 101;    -- one
nested table
        select dependents from employees;  -- one nested table per row
```

Figure 4.14 shows the output format for these retrieved nested tables. Following the object type display form, we see the type name and then a parenthesized list of the component parts, in this case the row objects of the nested table. As with object types, we can use the same form for constructing nested table objects.

Now the challenging part is retrieving just the desired information from nested tables rather than the whole tables. We can use TABLE(e.dependents) to convert the nested table column value (a single database item value) e.dependents into a query-compatible table of its contents, but note that there *is one of these for each row of the employees table.*

We can use TABLE(e.dependents) in a subquery, such as

```
        select * from table(e.dependents)  -- subquery using TABLE
```

but this cannot stand as a top-level select because the table alias e is not defined. We can give e a definition by embedding this subquery in an outer query on employess:

```
        select eid from employees e
                where exists (select * from table(e.dependents));
```

This query finds all employees with dependents. Many queries on nested tables have this format: one select runs over the parent table and the other over one or more child tables. See Examples 4.3.4 and 4.3.5.

We do have to be careful about using TABLE() in the right spots. Example 4.3.6 shows a query that produces different results based on whether or not TABLE() is applied to e.dependents. With it, we are accessing the rows of the table, whereas without it, we are accessing the whole table object, a single thing with count of 1.

In Example 4.3.7, we query one dependents table:

```
   select d.ssno from table(select e.dependents from employees where
e.eid = 101) d;
```

What happens if we remove the "where e.eid = 101" that specified a single nested table? Then TABLE() refuses to work with the multiple nested tables trying to flow into it. TABLE() requires a single nested table as an argument.

OK, but on a more logical level, how can we remove this specification of eid = 101 and retrieve *all* the ssno's for *all* the dependents? This is the "table of tables" data retrieval case. We are asking for data items from all the rows of all the nested tables in all the rows of the parent table.

It turns out there are two ways to do this in Oracle. The first and often easier method is by using a table product between the parent table and the child tables, for example:

```
        select e.eid, d.ssno from employees e, table(e.dependents) d;
```

This table product generates a result row for each dependent. The result row contains one child table element value (here a dependent_t object) plus all the column values of its row in the parent table, even the column value containing the full nested table object.

Thus an employee with 3 dependents would get 3 rows in this result set, with triplicated employee column values. But an employee with no dependents doesn't show up at all!

To include the dependent-less employees, with null of course where the dependent_t object should show up, we put a (+) in the right spot, as follows:

```
        select e.eid, d.ssno from employees e, table(e.dependents) (+) d;
```

Thus this table product has neatly filled out a big table with all (or nearly all, if you leave out the (+)) of the information of all the tables. This big table can be queried like any other.

For example, to find all the possibilities in groups of dependents of exactly the same age, where we are only interested in groups of 3 or more (for planning play groups, say), we could put:

```
        select d.age, count(*) from employees e, table(e.dependents) d
                group by d.age having count(*)>= 3 order by d.age;
```

But exactly who are these dependents? We have their ages from this query, and that can be used to find out their identities:

```
        select d1.ssno, d1.age from employees e1, table(e1.dependents)
d1
                where d1.age in (select d.age from employees e,
table(e.dependents) d
                        group by d.age having count(*)>= 3)
                order by d1.age;
```

Note that table products turn a "table of tables" into an ordinary table, so it maps us back into the normal relational world. That's great for lots of cases, but sometimes we would like to maintain the hierarchical relationship of parent to child all the way through.

The Oracle CURSOR facility allows hierarchical presentation of query results (even from purely relational tables.) Unfortunately, its printout from SqlPlus is pretty ugly. See Figure 4.17. Perhaps this will improve in later releases.

The idea of the hierarchical output is simple: output the parent information once, and then print a table of all the relevant child rows for that parent, then go on to the next parent, and so on.

For our current example, employee 101 has two dependents and employee 102 has one, so we see the 101, then the table of two dependent rows, then 102, and finally a table of one row.

The first thing to note is that this result set is not a relational table! A relational table has rows that all have the same column types, very rectangular.

We see that the CURSOR provides another way of running a loop over rows, thus competing with SELECT for this privilege. But it runs many smaller loops, one for each row of the outer SELECT, and each loop execution produces a little table for that parent row.

We see that a CURSOR runs inside a SELECT. In fact, it can only be used in a select-list of a top-level SELECT, as in select expr, expr, …, cursor (…). See Examples 4.3.9 and 4.3.10 for more examples.

VARRAYs

VARRAYs are collection objects, like nested tables in many ways. But they differ in three important ways:

- VARRAYs maintain the order of their elements
- VARRAYs have a maximum number of elements, set at type definition time
- VARRAYs are held in the main table (usually), not in a separate storage table

In version 8.1.5 or later, VARRAYs work very much like nested tables in queries. You can use TABLE() to convert a VARRAY column value into a query-compatible table. In earlier versions, VARRAYs had to be cast to nested tables first, a terrible inconvenience.

Study the examples for details--there are no real new ideas here.

Inserts and Updates

Collection constructors can be used in the expected way to specify values for nested columns. See Example 4.3.15. The Update there should be:

```
update phonebook pb set extensions = extensions_t(345,999)
        where pb.phperson.ssno = 123897766;
```

The more interesting capability is inserting and updating the individual elements in nested tables or VARRAYs. We can use TABLE() to turn a single collection-value into a table that can be updated. We use a subquery to select the target collection-value, such as

```
      TABLE(select e.dependents from employees e where e.eperson.ssno
=  123897766)
```

and then embed this in an insert statement, as in Example 4.3.16:

```
insert into
table (select e.dependents from employees e where e.eperson.ssno
= 123897766) values (344559112, name_t('Smith', 'Diane', null),
0);
```