Nested Relations

In relational databases, all relations are at least in First-Normal-Form (1NF) which requires all attributes to have *atomic domains*. That is, elements of the domains are considered to be indivisible units.

In the *nested relational model* – as an extension of the relational model – domains may be either atomic or relation valued. That is, an attribute value of a tuple can be a relation.

Example: Representation of a document relation in Non-1NF:

title	author-list	date	keyword-list
DB Theory	{Smith, Jones}	1 April 79	{algebra, logic}
Programming	{Jones, Frick}	17 June 85	{Pascal, C}

A nested relation can be decomposed ("flattened") into a relation having the 1NF property. It then can be decomposed into a set of relations that satisfy the Third Normal Form (3NF).

☞ Nested relations are based on a type constructor for collection types.
☞ Requires operators to "flatten" nested relation.

Nested Tables in Oracle 10g

Another object-relational feature provided in Oracle 10g is the ability to have a nested relation.

- The keyword **table** allows you to treat a nested relation as a normal relation, e.g., in a **from** clause.
- The keywords **cast**(**multiset**( )) allow you to turn the result of a query into a nested relation.

Defining a Nested Table

If you have an object type, you can create a new type that is a bag of that type by using **as table of**

Suppose we have a more complex beer type:

```
create type BeerType as object (
    name char(20),
    kind char(10),
    color char(10) );
```

We may create a type that is a nested table of objects of this type by

```
create type BeerTableType as
      table of BeerType;
```

Define a relation of manufacturers that will nest their beers inside.

```
create table  manfs (
      name char(30),
      addr AddrType,
      beers BeerTableType)
nested table beers store as BeerTable;
```

The last line in the **create table** statement indicates that the nested table is not stored "in-line" with the rest of the table (Oracle maintains pointers between tables); you cannot refer to BeerTable in any query!

Inserting into Nested Tables

```
insert into manfs values
('Anheuser',
       AddrType('LoopRoad','Boga','CA',56789),
       BeerTableType(
                BeerType('sweet','ale','yellow'),
                BeerType('sour','lager','pale')
       )
);
```


Querying With Nested Tables

An attribute that is a nested table can be printed like any other attribute. The value of the attribute has two type constructors, one for the table, and one for the type of its tuples.

Example: List the beers made by Anheuser:

```
select beers from manfs
where name = 'Anheuser';
```

This query gives you a single value that looks like this:

```
BeerTableType(
          BeerType('sweet','ale','yellow'),
          BeerType('sour','lager','pale'))
```

✗ **select** kind **from** manfs or **select** kind **from** beers

Use **table** to get to the nested table itself, then treat it like any other (normal) relation; **table** "flattens" the nested table.

Example: Find the ales made by Anheuser:

```
select bb. name
from table(
       select beers from manfs
       where name = 'Anheuser'
) bb
where  bb.kind = 'ale';
```

Find the name and address (city) of manufacturers that produce green beer:

```
select m.name, m.addr.city
from manfs m
where exists (select b.* from
                table(select beers from manfs
                    where name = m.name) b
                    where b.color = 'green');
```

List all manufacturers (names) together with all information about the beers they produce:

```
select m.name, b.*
from manfs m, table(m.beers) b;
```

Meaning: This statement "unnests" the nested table that is associated with each tuple, i.e., for a row in that table, it joins the "normal" attributes with each tuple of the associated nested table. The above query is equivalent to:

```
select m.name , nt.*
from manfs m, table (select beers
                     from manfs where name = m.name) nt;
```


Output:

NAME	NAME	KIND	COLOR
Anheuser	A1	ale	yellow
Anheuser	A2	lager	pale
Budwiser	B1	ale	yellow
Budwiser	B2	lager	pale
Budwiser	B3	lager	pale
Citrus	C1	ale	yellow
Citrus	C2	lager	pale
Citrus	C3	lager	yellow
Citrus	C4	ale	green

.....

More on Database Modifications

Updating tuples in a nested table:

```
update table (select beers from manfs m
                where m.name = 'Anheuser') b
set b.color = 'blue';
```

That is, first you have to flatten the nested table using **table** before you can do an update on the nested table. Not quite obvious because **update** essentially refers only to nested table (beers) and not the table manfs.

Delete operations on a nested table occur in an analogous way.

Insertion of a tuple into manfs for which no address and beer is known:

```
insert into manfs values
    ('Dodger', null, BeerTableType() );
```

This is different from

```
insert into manfs values
    ('Dodger', AddrType(null,null,null,null), BeerTableType() );
```

where the object addr exists, but its values are not known.

"Fodors produces the same beers as Anheuser, i.e., we have to insert a tuple into manfs where the table valued attribute beers has the same values as for Anheuser:

```
insert into manfs values
    ('Fodors', AddrType('CircleWay','Roga','VA',72441),
      cast( multiset (
            select * from
            table (select m.beers from manfs m
                  where m.name = 'Anheuser'))
                              as BeerTableType));
```

cast keyword allows you to "cast" the result of a query as a nested table. **multiset** keyword allows the cast query to contain multiple records. In the example above, the result of the query is cast as BeerTableType.

This **insert** statement won't work!

```
insert into manfs values
    ('Fodors', AddrType('CircleWay','Roga','VA',72441),
      cast( multiset (select b.* from
            table (select m.beers from manfs m
                  where m.name = 'Anheuser') b
                      ) as BeerType)
);
```

Varying Arrays

Ordered set of data elements that are all of the same data type. Each element has an index which is a number corresponding to the element's position in the array.

```
create type StandBeersType as varray(5) of BeerType;
```

```
create type PhoneType as varray(3) of number(9);
```

If the size of an array is smaller than 4,000 bytes, Oracle stores it in-line. Otherwise, Oracle stores it in a BLOB.

Using Varying Arrays

```
create table BeerSellers (
    name char(30),
    beers StandBeersType);

insert into BeerSellers values (
    'Miller', StandBeersType(
            BeerType('sweet', 'ale', 'yellow'),
            BeerType('sour', 'lager', 'pale') )
);

create table Departments (
    deptno integer,
    deptname char(40),
    phones PhoneType);

insert into departments values
    (10, 'Sales', PhoneType(5307821234, 5307829876));
```

Varying arrays are can be queried and manipulated in the same way as nested tables. Piece-wise updates of a varray value are not supported. Thus, when a varray is updated, the entire old collection is replaced by the new collection.

Other Features

Since Oracle 7, one can use the data type **long** to store character data up to 2GB in length per row. Since Oracle 8, additional data types are available to store types of *long objects* (LOBs).

- **BLOB**: Binary large object; up to 4GB in length, stored in the database (i.e., tablespaces).
- **CLOB**: Character LOB; character data, up to 4GB in length, stored in the database.
- **BFILE**: Binary file; read-only binary data stored outside the database, length is limited by the operating system.

Example:

```
create table Proposal (
    proposal_id number(8) primary key,
    recipient_name varchar2(25),
    proposal_name varchar2(50),
    short_description varchar2(1000),
    proposal_text CLOB,
    budget BLOB,
    cover_letter BFILE );
```

budget contains the spreadsheet showing the calculations of the cost and profit of the proposed project; values of the attribute cover_letter are stored outside the database.

Oracle uses normal database capabilities to support data integrity and and concurrency for proposal_text and budget entries, but not for cover_letter.

A **create table** statement may contain **storage** clause for a LOB.

PL/SQL packages (set of PL/SQL functions and procedures) provide useful operators on LOBs:

– **getlength**(<LOB>)
– **compare**(<LOB1>, <LOB2>)
– **write, append, erase, trim, copy, . . .**

External Procedures

Oracle 10g (9/8) supports *external procedures* $\hat{=}$ third-generation language routine stored in a *dynamic link library* (DLL), registered with PL/SQL, and can be called by users to do special purpose processing. At run-time, PL/SQL loads the library dynamically, then calls the routine as if it were a PL/SQL subprogram.

External procedures promote reusability, efficiency, and modularity. DLLs are loaded only when needed, so memory is conserved. Moreover, DLLs can be modified without affecting the calling program.

Typically, external procedures are used to interface with embedded systems, solve scientific and engineering problems, analyze data, or control real-time devices and processors.

Usage (examples): send instructions to a robot, solve partial differential equations, process signals, analyze time series, or create animation on a video display.

Choosing a Language for Method Functions

A method implemented in C executes in a separate process from the server using external routines. In contrast, a method implemented in Java or PL/SQL executes in the same process as the server.

```
CREATE TYPE ImageType AS OBJECT (
   id   NUMBER,
   img  BLOB,
   MEMBER FUNCTION get_name() return VARCHAR2,
   MEMBER FUNCTION rotate() return BLOB,
   STATIC FUNCTION clear(color NUMBER) return BLOB
   );

CREATE TYPE BODY ImageType AS
   MEMBER FUNCTION get_name() RETURN VARCHAR2
   AS
   imgname VARCHAR2(100);
   BEGIN
      SELECT name INTO imgname FROM imgtab WHERE imgid = self;
      RETURN imgname;
   END;

   MEMBER FUNCTION rotate() RETURN BLOB
   AS LANGUAGE C
   NAME "Crotate"
   LIBRARY myCfuncs;

   STATIC FUNCTION clear(color NUMBER) RETURN BLOB
   AS LANGUAGE JAVA
   NAME 'myJavaClass.clear(color oracle.sql.NUMBER)
   RETURN oracle.sql.BLOB';
END;
```