Oracle Nested Tables

Another structuring tool provided in Oracle is the ability to have a relation with an attribute whose value is not just an object, but a (multi)set of objects, i.e., a relation.

- Keyword `THE` allows us to treat a nested relation as a regular relation, e.g., in `FROM` clauses.
- Keywords `CAST(MULTISET(...))` let us turn the result of a query into a nested relation.

Defining Table Types

If we have an object type, we can create a new type that is a bag of that type by `AS TABLE OF`.

Example

Suppose we have a more complicated 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;
/
```

Now, we can define a relation of manufacturers that will nest their beers inside.

- In a sense, we normalize an unnormalized relation, since other data about the manufacturer appears only once no matter how many beers they produce.

```
CREATE TABLE Manfs (
    name CHAR(30),
    addr CHAR(50),
    beers BeerTableType
)
```

- However, to tell the system how to store the little `beers` tables, we must follow this statement, prior to the semicolon, by a statement

```
NESTED TABLE beers STORE AS
    BeerTable;
```

- The name of the table that stores the tuples for the nested `beers` relations is arbitrary; here we used `BeerTable`.

Querying With Nested Tables

An attribute that is a nested table can be printed like any other attribute.

- The value has two type constructors, one for the table, one for the type of its tuples.

Example

List the beers made by Anheuser-Busch.

```
SELECT beers
FROM Manfs
WHERE name = 'Anheuser-Busch';
```

- A single value will be printed, looking something like:

```
BeerTableType(
    BeerType('Bud', 'lager', 'yellow'),
    BeerType('Lite', 'malt', 'pale'),...
)
```

Operating on Nested Tables

Use `THE` to get the nested table itself, then treat it like any other relation.

Example

Find the ales made by Anheuser-Busch.

```
SELECT bb.name
FROM THE(
    SELECT beers
    FROM Manfs
    WHERE name = 'Anheuser-Busch'
) bb
WHERE bb.kind = 'ale';
```

Casting to Create Nested Tables

Create a value for a nested table by using a select-from-where query and "casting" it to the table type.

Example

- Suppose we have a relation `Beers(beer, manf)`, where `beer` is a `BeerType` object and `manf` its manufacturer.

- We want to insert into `Manfs` a tuple for Pete's Brewing Co., with all the beers brewed by Pete's (according to `Beers`) in one nested table.

```
INSERT INTO Manfs VALUES(
    'Pete''s', 'Palo Alto',
    CAST(
        MULTISET(
            SELECT bb.beer
            FROM Beers bb
            WHERE bb.manf = 'Pete''s'
        ) AS BeerType
    )
);
```

Transactions

= units of work that must be:

1. *Isolated* = appear to have been executed when no other DB operations were being performed.
   - Often called *serializable* behavior.
   - In modern DBMS's, serializability is often one of several options for how behavior is restricted.
2. *Atomic* = either all work is done, or none of it.

Commit/Abort Decision

Each transaction ends with either:

1. *Commit* = the work of the transaction is installed in the database; previously its changes may be invisible to other transactions.

2. *Abort* = no changes by the transaction appear in the database; it is as if the transaction never occurred.

   - `ROLLBACK` is the term used in SQL and the Oracle system.

- In the ad-hoc query interface (e.g., Oracle's SQLplus), transactions are single queries or modification statements.

  - Oracle allows `SET TRANSACTION READ ONLY` to begin a multistatement transaction that doesn't change any data, but needs to see a consistent "snapshot" of the data.

- In program interfaces (e.g., Pro*C or PL/SQL), transactions begin whenever the database is accessed, and end when either a `COMMIT` or `ROLLBACK` statement is executed.

Example

Sells(<u>bar</u>, <u>beer</u>, price)

- Joe's Bar sells Bud for \$2.50 and Miller for \$3.00.
- Sally is querying the database for the highest and lowest price Joe charges:

(1)
```
SELECT MAX(price) FROM Sells
WHERE bar = 'Joe''s Bar';
```
(2)
```
SELECT MIN(price) FROM Sells
WHERE bar = 'Joe''s Bar';
```

- At the same time, Joe has decided to replace Miller and Bud by Heineken at \$3.50:

(3)
```
DELETE FROM Sells
WHERE bar = 'Joe''s Bar' AND
    (beer = 'Miller' OR beer = 'Bud');
```
(4)
```
INSERT INTO Sells
VALUES('Joe''s bar', 'Heineken',
              3.50);
```

- If the order of statements is 1, 3, 4, 2, then it appears to Sally that Joe's minimum price is greater than his maximum price.
- Fix the problem by grouping Sally's two statements into one transaction, e.g. with one PL/SQL statement.

Example: Problem With Rollback

Suppose Joe executes statement 4 (insert Heineken), but then, during the transaction thinks better of it and issues a `ROLLBACK` statement.

- If Sally is allowed to execute her statement 1 (find max) just before the rollback, she gets the answer $3.50, even though Joe doesn't sell any beer for $3.50.
- Fix by making statement 4 a transaction, or part of a transaction, so its effects cannot be seen by Sally unless there is a `COMMIT` action.

SQL Isolation Levels

Isolation levels determine what a transaction is allowed to see. The declaration, valid for one transaction, is:

`SET TRANSACTION ISOLATION LEVEL` X`;`

where:

- X = `SERIALIZABLE`: this transaction must execute as if at a point in time, where all other transactions occurred either completely before or completely after.

Example

Suppose Sally's statements 1 and 2 are one transaction and Joe's statements 3 and 4 are another transaction. If Sally's transaction runs at isolation level `SERIALIZABLE`, she would see the `Sells` relation either before or after statements 3 and 4 ran, but not in the middle.

- X = `READ COMMITTED`: this transaction can only read committed data.

Example

If transactions are as above, Sally could see the original `Sells` for statement 1 and the completely changed `Sells` for statement 2.

- X = `REPEATABLE READ`: if a transaction reads data twice, then what it saw the first time, it will see the second time (it may see more the second time).
  - Moreover, all data read at any time must be committed; i.e., `REPEATABLE READ` is a strictly stronger condition than `READ COMMITTED`.

Example

If 1 is executed before 3, then 2 must see the Bud and Miller tuples when it computes the min, even if it executes after 3. But 2 may see the Heineken tuple, even if 1 didn't.

- X = `READ UNCOMMITTED`: essentially no constraint, even on reading data written and then removed by a rollback.

Example

1 and 2 could see Heineken, even if Joe rolled back his transaction.

Independence of Isolation Levels

Isolation levels describe what a transaction T with that isolation level sees.

- They *do not* constrain what other transactions, perhaps at different isolation levels, can see of the work done by T.

Example

If transaction 3-4 (Joe) runs serializable, but transaction 1-2 (Sally) does not, then Sally might see `NULL` as the value for both min and max, since it could appear to Sally that her transaction ran between steps 3 and 4.

Authorization in SQL

- File systems identify certain access privileges on files, e.g., read, write, execute.
- In partial analogy, SQL identifies nine access privileges on relations, of which the most important are:

  1. `SELECT` = the right to query the relation.
  2. `INSERT` = the right to insert tuples into the relation — may refer to one attribute, in which case the privilege is to specify only one column of the inserted tuple.
  3. `DELETE` = the right to delete tuples from the relation.
  4. `UPDATE` = the right to update tuples of the relation — may refer to one attribute.

Granting Privileges

- You have all possible privileges to the relations you create.
- You may grant privileges to any user if you have those privileges "with grant option."
  - ❖ You have this option to your own relations.

Example

1. Here, Sally can query `Sells` and can change prices, but cannot pass on this power:

```
GRANT SELECT ON Sells,
    UPDATE(price) ON Sells
TO sally;
```

2. Here, Sally can also pass these privileges to whom she chooses:

```
GRANT SELECT ON Sells,
    UPDATE(price) ON Sells
TO sally
WITH GRANT OPTION;
```

Revoking Privileges

- Your privileges can be revoked.
- Syntax is like granting, but `REVOKE ... FROM` instead of `GRANT ... TO`.
- Determining whether or not you have a privilege is tricky, involving "grant diagrams" as in text. However, the basic principles are:

  a) If you have been given a privilege by several different people, then all of them have to revoke in order for you to lose the privilege.

  b) Revocation is transitive. If A granted P to B, who granted P to C, and then A revokes P from B, it is as if B also revoked P from C.