Constraints and Triggers

Deferring Constraint Checking

Sometimes it is necessary to defer the checking of certain constraints, most commonly in the "chicken-and-egg" problem. Suppose we want to say:

```sql
CREATE TABLE chicken (cID INT PRIMARY KEY,
                      eID INT REFERENCES egg(eID));
CREATE TABLE egg(eID INT PRIMARY KEY,
                 cID INT REFERENCES chicken(cID));
```

But if we simply type the above statements into Oracle, we'll get an error. The reason is that the `CREATE TABLE` statement for `chicken` refers to table `egg`, which hasn't been created yet! Creating `egg` won't help either, because `egg` refers to `chicken`.

To work around this problem, we need SQL schema modification commands. First, create `chicken` and `egg` without foreign key declarations:

```sql
CREATE TABLE chicken(cID INT PRIMARY KEY, eID INT);
CREATE TABLE egg(eID INT PRIMARY KEY, cID INT);
```

Then, we add foreign key constraints:

```sql
ALTER TABLE chicken ADD CONSTRAINT chickenREFegg
    FOREIGN KEY (eID) REFERENCES egg(eID)
    INITIALLY DEFERRED DEFERRABLE;

ALTER TABLE egg ADD CONSTRAINT eggREFchicken
    FOREIGN KEY (cID) REFERENCES chicken(cID)
    INITIALLY DEFERRED DEFERRABLE;
```

`INITIALLY DEFERRED DEFERRABLE` tells Oracle to do deferred constraint checking. For example, to insert (1, 2) into `chicken` and (2, 1) into `egg`, we use:

```sql
INSERT INTO chicken VALUES(1, 2);
INSERT INTO egg VALUES(2, 1);
COMMIT;
```

Because we've declared the foreign key constraints as "deferred", they are only checked at the commit point. (Without deferred constraint checking, we cannot insert anything into `chicken` and `egg`, because the first `INSERT` would always be a constraint violation.)

Finally, to get rid of the tables, we have to drop the constraints first, because Oracle won't allow us to drop a table that's referenced by another table.

```sql
ALTER TABLE egg DROP CONSTRAINT eggREFchicken;
ALTER TABLE chicken DROP CONSTRAINT chickenREFegg;
DROP TABLE egg;
DROP TABLE chicken;
```
Constraint Violations

In general, Oracle returns an error message when a constraint is violated. Specifically for users of JDBC, this means an SQLException gets thrown, whereas for Pro*C users the SQLCA struct gets updated to reflect the error. Programmers must use the WHENEVER statement and/or check the SQLCA contents (Pro*C users) or catch the exception SQLException (JDBC users) in order to get the error code returned by Oracle.

Some vendor specific error code numbers are 1 for primary key constraint violations, ORA-02291 for foreign key violations, ORA-02290 for attribute and tuple CHECK constraint violations.

Basic Trigger Syntax

Below is the syntax for creating a trigger in Oracle (which differs slightly from standard SQL syntax):

```sql
CREATE [OR REPLACE] TRIGGER <trigger_name>
  {BEFORE | AFTER} {INSERT | DELETE | UPDATE} ON <table_name>
  [REFERENCING [NEW AS <new_row_name>] [OLD AS <old_row_name>]]
  [FOR EACH ROW [WHEN (<trigger_condition>)]]
  <trigger_body>
```

Some important points to note:

- You can create only BEFORE and AFTER triggers for tables. (INSTEAD OF triggers are only available for views; typically they are used to implement view updates.)

- You may specify up to three triggering events using the keyword OR. Furthermore, UPDATE can be optionally followed by the keyword OF and a list of attribute(s) in <table_name>. If present, the OF clause defines the event to be only an update of the attribute(s) listed after OF. Here are some examples:
  - ... INSERT ON R ...
  - ...
  - ... INSERT OR DELETE OR UPDATE ON R ...
  - ...
  - ... UPDATE OF A, B OR INSERT ON R ...

- If FOR EACH ROW option is specified, the trigger is row-level; otherwise, the trigger is statement-level.
Only for row-level triggers:
  o The special variables NEW and OLD are available to refer to new and old tuples respectively. **Note:** In the trigger body, NEW and OLD must be preceded by a colon (":") , but in the WHEN clause, they do not have a preceding colon! See example below.
  o The REFERENCING clause can be used to assign aliases to the variables NEW and OLD.
  o A trigger restriction can be specified in the WHEN clause, enclosed by parentheses. The trigger restriction is a SQL condition that must be satisfied in order for Oracle to fire the trigger. This condition cannot contain subqueries. Without the WHEN clause, the trigger is fired for each row.

<trigger_body> is a PL/SQL block, rather than sequence of SQL statements. Oracle has placed certain restrictions on what you can do in <trigger_body>, in order to avoid situations where one trigger performs an action that triggers a second trigger, which then triggers a third, and so on, which could potentially create an infinite loop. The restrictions on <trigger_body> include:
  o You cannot modify the same relation whose modification is the event triggering the trigger.
  o You cannot modify a relation connected to the triggering relation by another constraint such as a foreign-key constraint.

---

**Trigger Example**

We illustrate Oracle's syntax for creating a trigger through an example based on the following two tables:

```sql
CREATE TABLE T4 (a INTEGER, b CHAR(10));
CREATE TABLE T5 (c CHAR(10), d INTEGER);
```

We create a trigger that may insert a tuple into T5 when a tuple is inserted into T4. Specifically, the trigger checks whether the new tuple has a first component 10 or less, and if so inserts the reverse tuple into T5:

```sql
CREATE TRIGGER trig1
  AFTER INSERT ON T4
  REFERENCING NEW AS newRow
  FOR EACH ROW
  WHEN (newRow.a <= 10)
  BEGIN
    INSERT INTO T5 VALUES(:newRow.b, :newRow.a);
  END trig1;
```

run;
Notice that we end the `CREATE TRIGGER` statement with a dot and `run`, as for all PL/SQL statements in general. Running the `CREATE TRIGGER` statement only creates the trigger; it does not execute the trigger. Only a triggering event, such as an insertion into `T4` in this example, causes the trigger to execute.

---

**Displaying Trigger Definition Errors**

As for PL/SQL procedures, if you get a message

```
Warning: Trigger created with compilation errors.
```

you can see the error messages by typing

```
SHOW ERR <trigger_name>;
```

Alternatively, you can type, `SHO ERR` (short for `SHOW ERRORS`) to see the most recent compilation error. Note that the reported line numbers where the errors occur are not accurate.

---

**Viewing Defined Triggers**

To view a list of all defined triggers, use:

```
SELECT trigger_name FROM user_triggers;
```

For more details on a particular trigger:

```
SELECT trigger_type, triggering_event, table_name, referencing_names, trigger_body
FROM user_triggers
WHERE trigger_name = '<trigger_name>';
```

---

**Dropping Triggers**

To drop a trigger:

```
DROP TRIGGER <trigger_name>;
```

---

**Disabling Triggers**

To disable or enable a trigger:

```
ALTER TRIGGER <trigger_name> {DISABLE|ENABLE};
```
Aborting Triggers with Error

Triggers can often be used to enforce constraints. The WHEN clause or body of the trigger can check for the violation of certain conditions and signal an error accordingly using the Oracle built-in function RAISE_APPLICATION_ERROR. The action that activated the trigger (insert, update, or delete) would be aborted. For example, the following trigger enforces the constraint Person.age >= 0:

```sql
CREATE table Person (age int);

CREATE TRIGGER PersonCheckAge
AFTER INSERT OR UPDATE OF age ON Person
FOR EACH ROW
BEGIN
    IF (:new.age < 0) THEN
        RAISE_APPLICATION_ERROR(-20000, 'no negative age allowed');
    END IF;
END;
.
RUN;
```

If we attempted to execute the insertion:

```sql
insert into Person values (-3);
```

we would get the error message:

```
ERROR at line 1:
ORA-20000: no negative age allowed
ORA-06512: at "MYNAME.PERSONCHECKAGE", line 3
ORA-04088: error during execution of trigger 'MYNAME.PERSONCHECKAGE'
```

and nothing would be inserted. In general, the effects of both the trigger and the triggering statement are rolled back.

Mutating Table Errors

Sometimes you may find that Oracle reports a "mutating table error" when your trigger executes. This happens when the trigger is querying or modifying a "mutating table", which is either the table whose modification activated the trigger, or a table that might need to be updated because of a foreign key constraint with a CASCADE policy. To avoid mutating table errors:

- A row-level trigger must not query or modify a mutating table. (Of course, NEW and OLD still can be accessed by the trigger.)
• A statement-level trigger must not query or modify a mutating table if the trigger is fired as the result of a CASCADE delete.