Enforcing Referential Integrity in Oracle

Creating an Entity-Relationship Diagram for the PARCELS Database

By now the PARCELS database should be very familiar to you. Feel free to review the PARCELS database schema[*]. We'll use a subset of the PARCELS database for this discussion: **parcels**[*], **owners**[*], **fires**[*], and **tax**[*]. Treat each of these tables as an entity.

You can use Microsoft Access's "Relationships" feature to create an entity-relationship (E-R) diagram for this database. To do so, link in to your Access database the tables listed above. Next, open the "Relationships" window using the **Tools > Relationships** menu item. You'll be asked to add tables; add all the tables listed above. You define relationships by clicking-and-dragging from the primary key to the foreign key, much like adding joins to a query. Notice that a "Relationships" dialog box appears when you draw a relationship. This dialog gives you the opportunity to edit the relationship. Since some of the tables have multiple-column primary keys, you will need to use this interface to add the additional columns. You can bring back this dialog box by clicking on one of black relationship lines in the window. Your diagram should include:

- The attributes of each entity
- The primary key of each entity
- Primary key-foreign key relationships

Since Access will not let you print out the relationships window, the only way to preserve the diagram is to copy-and-paste an image of it into in another application such as Microsoft Word. To copy the image of the current window, press the "Alt" key, then press the "Print Screen" key. Open Word, then use **Edit > Paste** to paste the image into a Word document.

Consider listing of the foreign key relationships among the tables in the following format:

FK_TABLE (FK_COL1, FK_COL2, ...) references PK_TABLE (PK_COL1, PK_COL2) where

- FK_TABLE is the table in which the foreign key appears
- FK_COL1, FK_COL2, ... are the columns that make up the foreign key

[*] Kindly refer back to the Tools section.
[*] Kindly refer back to the Tools section.
[*] Kindly refer back to the Tools section.
[*] Kindly refer back to the Tools section.
[*] Kindly refer back to the Tools section.

- PK_TABLE is the table containing the primary key referenced by the foreign key
- PK_COL1, PK_COL2, ... are the columns that make up the primary key
- FK_COL1 corresponds to PK_COL1, FK_COL2 corresponds to PK_COL2, etc.

For example, for the **fires** and **parcels** tables, you would have the following entry:
FIRES (PARCELID) references PARCELS (PARCELID)

Defining Primary and Foreign Keys in Oracle

Once we have defined the primary and foreign keys in a database schema, it is important that the proper relationships be maintained so that the database does not become inconsistent. For example, once we've determined that the column **fires.parcelid** constitutes a foreign key that refer to the columns of the same name in the **parcels** table, we want to ensure that for every **parcelid** in **fires** has is a corresponding row in **parcels.** To restate this wish in database jargon, we want to ensure that the **referential integrity** of the database is maintained. Fortunately, Oracle has the power to enforce referential integrity, but to do so, we must inform the database of the relationships involved. We do so by adding **constraints** on the tables.

Once you've created tables in Oracle using the **CREATE TABLE** statement, you can use the **ALTER TABLE** **ADD CONSTRAINT** statement to add primary key and foreign key designations to the tables. For example, to define the primary key **parcelid** for the **parcels** table, we would use the following SQL statement:

```
ALTER TABLE parcels
  ADD CONSTRAINT parcels
  PRIMARY KEY (parcelid);
```

Constraints in Oracle have names just as tables and views do. Since a constraint can have the same name as a table, I recommend giving primary key constraints names the same name as that of the table it applies to. If you omit the constraint name, Oracle assigns a default name of the form 'SYS_Cnnnnn', where nnnnn are digits. By defining constraint names in this fashion, it is easy to identify what a constraint is for. When you run a SQL statement that violates a constraint, the Oracle will display the name of the constraint in the error message. An informative constraint name will save you the trouble of querying the data dictionary to figure out the nature of the constraint you violated.

Similarly, we can define the multi-column primary key **(parcelid, fdate)** for the **fires** table**:**

```
ALTER TABLE fires
  ADD CONSTRAINT fires
  PRIMARY KEY (parcelid, fdate);
```

The foreign key **parcel** in the **fires** table, refers to we would use the following SQL statement:

```
ALTER TABLE fires
  ADD CONSTRAINT fires$parcelid
  FOREIGN KEY (parcelid)
  REFERENCES parcels (parcelid);
```

Here, we've used a more elaborate convention for naming the foreign key constraint. Specifically, we use constraint names of the following form:
table$foreign_key_column
table$foreign_key_column_1#foreign_key_column_2
table$foreign_key_column_1#foreign_key_column_2#foreign_key_column_3
Here, 'table' is the name of the table that is being constrained (here, the table **fires**). The 'foreign_key_columns' in the list are column names from the constrained table (here, **fires.parcelid**). This naming convention uses the dollar sign ($) to separate the table name from the column names and the pound sign (#) to separate the column names, if necessary. There are some pitfalls with this approach:

- Oracle constraint names cannot exceed thirty (30) characters, so it may be necessary to abbreviate the table and column names in some instances.
- The convention of using the dollar and pound signs as separators only works if the tables themselves eschew using these characters in table or column names. Generally, avoiding these special characters in identifiers is a good practice to follow regardless.

To view the constraints you've created, first make your SQL*Plus window at least 113 columns wide and run the following SQL*Plus commands:

```
SET LINESIZE 113
COLUMN TABLE_NAME   FORMAT A12
COLUMN COLUMN_NAME  FORMAT A12
COLUMN OWNER        FORMAT A12
COLUMN R_OWNER      FORMAT A12
```

Now, run the following query against the Oracle data dictionary to view your constraints:

```
  SELECT CO.TABLE_NAME, CO.CONSTRAINT_NAME,
CO.CONSTRAINT_TYPE,
         CC.COLUMN_NAME, CC.POSITION,
         CO.R_OWNER, CO.R_CONSTRAINT_NAME
    FROM USER_CONSTRAINTS CO, USER_CONS_COLUMNS CC
   WHERE CO.OWNER = CC.OWNER
     AND CO.CONSTRAINT_NAME = CC.CONSTRAINT_NAME
     AND CO.CONSTRAINT_TYPE IN ('P', 'R')
ORDER BY CO.TABLE_NAME, CO.CONSTRAINT_TYPE,
         CO.CONSTRAINT_NAME, CC.POSITION;
```

Primary key constraints show up with a 'P' in the **constraint_type** column; foreign key constraints show up with an 'R' (for 'references'). Foreign key constraints reference a primary key constraint on another table. This referenced constraint is identified in the columns **r_owner** and **r_constraint_name**. Oracle also supports other kinds of constraints that this query will not show. For example, 'NOT NULL' conditions on the columns of tables are recorded as constraints. Oracle also supports a "uniqueness" constraint, which enforces uniqueness on any column or group of columns, not just the primary key. Referential constraints can also refer to these "unique" constraints.

Constraints are a powerful database management tool, but further discussion of their considerable capabilities is beyond the scope of this class. I recommend you look at the Oracle documentation on constraints and data integrity if you want to learn more about the functionality of constraints. The documentation for the ALL_CONSTRAINTS data dictionary view (similar to the USER_CONSTRAINTS view used above) explains the information Oracle stores about constraints in the data dictionary.

If you need to drop a constraint, use the **ALTER TABLE DROP CONSTRAINT** command. For example, to drop the primary key constraint we created on the table **parcel**s earlier, we could try the following SQL statement:

```
ALTER TABLE parcels
  DROP CONSTRAINT parcels;
```

However, this statement will fail if the foreign key constraint **fires$parcelid** we created earlier still exists, since the foreign key constraint references this primary key constraint. To get around this, we can either drop the referencing constraints individually or we can add the keyword **CASCADE** to the end of the command; this instructs Oracle to drop all other constraints that depend on this constraint:

```
ALTER TABLE parcels
  DROP CONSTRAINT parcels
  CASCADE;
```

Similarly, we will not be able to drop the table **parcels** if any constraints refer to primary (or unique) keys in the table. To circumvent this, we could either drop the referencing constraints individually first or we can add the **CASCADE CONSTRAINTS** phrase to the end of the **DROP TABLE** statement:

```
DROP TABLE parcels
  CASCADE CONSTRAINTS;
```

When creating new tables and defining constraints, I recommend that you write a SQL script that performs these tasks in the following order:

- Create **all** tables first. Precede each **CREATE TABLE** statement with a corresponding **DROP TABLE ... CASCADE CONSTRAINTS** statement to facilitate rebuilding the schema if you make changes.
- Alter the tables to add **all** the **primary key** constraints.
- Alter the tables to add the **foreign key** constraints.

Following this order avoids most chicken-and-egg problems that may occur in this process. Keep in mind, however, that you cannot define a foreign key in the **fires** table that references the primary key in the **parcels** table unless the primary key in **parcels** has been defined first. (Technical note: a foreign key can actually reference a uniqueness constraint instead of primary key constraint, but that it is of little import here.)

This is all you need to know to use primary and foreign keys in Oracle. Note that while many modern relational database systems allow you to define the primary and foreign keys, the SQL syntax for doing so varies from one database system to another. Hence, while the concepts presented here should apply to most relational database systems, your SQL statements to implement them may look different if you are not using Oracle.
