CHAPTER 15

Updating the Database

About this chapter

This chapter describes how to make changes in the contents of a database. It includes descriptions of how to add rows, remove rows, and change the contents of rows, as well as how to make changes permanent or back out of changes you have made.

Contents

Adding rows to a table

Suppose that a new eastern sales department is created, with the same manager as the current Sales department. You can add this information to the database using the following INSERT statement:

```
INSERT
INTO department ( dept_id, dept_name, dept_head_id )
VALUES ( 220, 'Eastern Sales', 902 )
```

If you make a mistake and forget to specify one of the columns, Adaptive Server Anywhere reports an error.

The NULL value is a special value used to indicate that something is either not known or not applicable. Some columns are allowed to contain the NULL value, and others are not.

A short form for INSERT

There is a short form that can be used if you are entering values for all the columns in a table in the order they appear when you SELECT * from the table (the same as the order in which they were created). The following is equivalent to the previous INSERT command:

```
INSERT
INTO department
VALUES ( 220, 'Eastern Sales', 902 )
```

> ***Caution***
> *You should use this form of* INSERT *with caution; it will not work as expected if you ever change the order of the columns in the table or if you add or remove a column from the table.*

Modifying rows in a table

In most databases, you need to update data stored in the database. For example, suppose that the employee named James Klobucher (employee ID 467) is transferred from the sales department to the marketing department. In SQL, this is done using the UPDATE statement:

```
UPDATE employee
SET dept_id = 400
WHERE emp_id = 467
```

The WHERE clause identifies which employee to update.

Example: using the WHERE clause

SQL can update more than one column at a time. For example, the manager ID should change when employees are transferred between departments, as well as the department ID. The following statement carries out both updates at the same time for employee Marc Dill (employee ID 195):

```
UPDATE employee
SET dept_id = 400,
manager_id = 1576
WHERE emp_id = 195
```

The UPDATE and INSERT commands

SQL allows more than one row to be updated at one time. For example, if a group of sales employees are transferred into marketing and have their **dept_id** column updated, the following statement sets the **manager_id** for all employees in the marketing department to **1576**.

```
UPDATE employee
SET manager_id = 1576
WHERE dept_id = 400
```

For employees already in the marketing department, no change is made.

It is also possible that an UPDATE statement updates no rows. For example, suppose you had made a mistake typing the employee ID in the first UPDATE statement above:

```
UPDATE employee
SET dept_id = 400
WHERE emp_id = 194
```

No rows would be updated since there is no employee with the employee ID **194.**

Canceling changes

You may be a little concerned about all of the changes you have made to the **employee** table. However, SQL allows you to undo all of these changes with one command:

```
ROLLBACK
```

The ROLLBACK statement

The ROLLBACK statement undoes all changes you have made to the database since the last time you made changes permanent (see COMMIT in the next section).

The default action in Interactive SQL is to do a COMMIT on exit. This can be controlled with the Interactive SQL option COMMIT_ON_EXIT.

For more information on Interactive SQL options, see "Interactive SQL options" on page 138 of the book *Adaptive Server Anywhere Reference Manual*.

Making changes permanent

The SQL statement

```
COMMIT
```

makes all changes permanent.

Use COMMIT with care
When trying the examples in this tutorial, be careful not to COMMIT any changes until you are sure that you want to change the database permanently.

Making changes permanent in Interactive SQL

The default action in Interactive SQL is to do a COMMIT on exit. This can be controlled with the Interactive SQL option COMMIT_ON_EXIT.

For more information on Interactive SQL options, see "Interactive SQL options" on page 138 of the book *Adaptive Server Anywhere Reference Manual*.

Interactive SQL has another option, named AUTO_COMMIT. If this option is on, Interactive SQL does a COMMIT operation after every command. The default for this option is OFF. Usually you will want it OFF, giving you the opportunity to ROLLBACK the changes (if, for example, an update or delete operation doesn't produce the intended results).

Deleting rows

Sometimes you will want to remove rows from a table. Suppose Rodrigo Guevara (employee ID 249) leaves the company. The following statement deletes Rodrigo Guevara from the employee table.

```
DELETE
FROM employee
WHERE emp_id = 249
```

Example

You can delete more than one row with one command. For example, the following statement would delete all employees who had a termination date that is not NULL from the **employee** table.

```
DELETE
FROM employee
WHERE termination_date IS NOT NULL
```

This example would not remove any employees from the database as the **termination_date** column is NULL for all employees.

With UPDATE and DELETE, the search condition can be as complicated as you need. For example, if the **employee** table is being reorganized, the following statement removes from the **employee** table all male employees hired between March 3, 1989 and March 3, 1990.

```
DELETE
FROM employee
WHERE sex = 'm'
    AND start_date between '1988-03-03'
    AND '1989-03-03'
```

Since you have made changes to the database that you do not want to keep, you should undo the changes as follows:

```
ROLLBACK
```

Validity checking

Adaptive Server Anywhere automatically checks for some common errors in your data.

Inserting duplicate data

For example, suppose you attempt to create a department but supply a **dept_id** value that is already in use:

To do this, enter the command:

```
INSERT
INTO department ( dept_id, dept_name, dept_head_id )
VALUES ( 200, 'Eastern Sales', 902 )
```

The INSERT is rejected, as it would make the primary key for the table not unique.

Primary key

A primary key is a set of columns that uniquely identifies each row in a table. For example, the **dept_id** column is the primary key for the **department** table; given a valid department ID number, there is exactly one row in the department table with that number. The primary key for the **sales_order_items** table is composed of the **id** and **line_id** columns, meaning that there should never be two items in the same order with the same line number.

Inserting incorrect values

Another mistake is to type an incorrect value. The following statement inserts a new row in the **sales_order** table, but incorrectly supplies a sales_rep ID that does not exist in the employee table.

```
INSERT
INTO sales_order ( id, cust_id, order_date,
   sales_rep)
VALUES ( 2700, 186, '1995-10-19', 284 )
```

Foreign key

The primary key for the **employee** table is the employee ID number. The sales rep ID number in the **sales_rep** table is a foreign key for the **employee** table, meaning that each sales rep number in the **sales_order** table must match the employee ID number for some employee in the **employee** table.

When you try to add an order for sales rep 284 you get an error message:

> No primary key value for foreign key 'ky_so_employee_id' in table 'sales_order'

There isn't an employee in the **employee** table with that ID number. This prevents you from inserting orders without a valid sales rep ID. This kind of validity checking is called **referential integrity** checking, as it maintains the integrity of references among the tables in the database.

Errors on DELETE or UPDATE

Foreign key errors can also arise when doing update or delete operations. For example, suppose you try to remove the R&D department from the department table.

```
DELETE
FROM department
WHERE dept_id = 100
```

Example: DELETE errors

An error is reported indicating that there are other records in the database that reference the R&D department, and the delete operation is not carried out.

primary key for row in table 'department' is referenced in another table

In order to remove the R&D department, you need to first get rid of all employees in that department:

```
DELETE
FROM employee
WHERE dept_id = 100
```

You can now perform the deletion of the R&D department.

You should cancel these changes to the database (for future use) by entering a ROLLBACK statement:

```
ROLLBACK WORK
```

All changes made since the last successful COMMIT WORK will be undone. If you have not done a COMMIT, then all changes since you started Interactive SQL will be undone.

Example: UPDATE errors

The same error message is generated if you perform an update operation that makes the database inconsistent.

For example, the following UPDATE statement causes an integrity error:

```
UPDATE department
    SET dept_id = 600
    WHERE dept_id = 100
```

In all of the above examples, the integrity of the database was checked as each command was executed. Any operation that would result in an inconsistent database is not performed.

Example: checking the integrity after the COMMIT WORK is complete

It is possible to configure the database so that the integrity is not checked until the COMMIT WORK is done. This is important if you want to change the value of a referenced primary key; for example, changing the R&D department's ID from 100 to 600 in the **department** and **employee** tables. In order to make these changes, the database has to be inconsistent in between the changes. In this case, you must configure the database to check only on commits.

☞ For more information, see "WAIT_FOR_COMMIT option" on page 178 of the book *Adaptive Server Anywhere Reference Manual*.

You can also define foreign keys in such a way that they are automatically fixed. In the above example, if the foreign key from **employee** to **department** were defined with ON UPDATE CASCADE, then updating the department ID would automatically update the employee table.

In the above cases, there is no way to have an inconsistent database committed as permanent. Adaptive Server Anywhere also supports alternative actions if changes would render the database inconsistent.

☞ For more information, see the chapter "Ensuring Data Integrity" on page 347 of the book *Adaptive Server Anywhere User's Guide*.