CHAPTER 8

Adding, Changing, and Deleting Data

About this chapter

This chapter describes how to modify the data in a database.

Most of the chapter is devoted to the INSERT, UPDATE, and DELETE statements. Statements for bulk loading and unloading are also described.

Contents

Data modification statements

The statements you use to add, change, or delete data are called **data modification** statements. The most common such statements are as follows:

- **Insert** adds new rows to a table.
- **Update** changes existing rows in a table.
- **Delete** removes specific rows from a table.

Any single INSERT, UPDATE, or DELETE statement changes the data in only one table or view.

In addition to the common statements, the LOAD TABLE and TRUNCATE TABLE statements are designed especially for bulk loading and deleting of data.

Sometimes, the data modification statements are collectively called the **data modificaton language** (DML) part of SQL.

Permissions for data modification

You can only execute data modification statements if you have the proper permissions on the database tables you are modifying. The database administrator and the owners of database objects use the GRANT and REVOKE statements to decide who has access to which data modification functions.

Permissions can be granted to individual users, groups, or the public group. For more information on permissions, see "Managing User IDs and Permissions" on page 575.

Transactions and data modification

When data is modified, a copy of the old and new state of each row affected by each data modification statement is written to the transaction log. This means that if you begin a transaction, realize you have made a mistake, and roll the transaction back, the database can be restored to its previous condition.

For more information about transactions, see "Using Transactions and Locks" on page 367.

Adding data using INSERT

You add rows to the database using the INSERT statement. The INSERT statement has two forms: you can use the VALUES keyword or a SELECT statement:

INSERT using values

The VALUES keyword specifies values for some or all of the columns in a new row. A simplified version of the syntax for the INSERT statement using the VALUES keyword is:

INSERT [**INTO**] *table-name* [(*column-name*, ...)]
VALUES (*expression* , ...)

You can omit the list of column names if you provide a value for each column in the table, in the order in which they appear when you execute a query using SELECT *.

INSERT from SELECT

You can use a SELECT statement in an INSERT statement to pull values from one or more tables. A simplified version of the syntax for the insert statement using a select statement is:

INSERT [**INTO**] *table-name* (*column-name*, ...)
select-statement

Inserting values into all columns of a row

The following INSERT statement adds a new row to the **department** table, giving a value for every column in the row:

```
INSERT INTO department
VALUES ( 702, 'Eastern Sales', 902 )
```

Notes

- The values are entered in the same order as the column names in the original CREATE TABLE statement, that is, first the ID number, then the name, then the department head ID.
- The values are surrounded by parentheses.
- All character data is enclosed in single quotes.
- You need to use a separate insert statement for each row you add.

Inserting values into specific columns

You can add data to some columns in a row by specifying only those columns and their values. All other columns that are not included in the column list must be defined to allow NULL or must have defaults. If you skip a column that has a default value, the default is used.

Adding data in only two columns, for example, **dept_id** and **dept_name**, requires a statement like this:

```
INSERT INTO department (dept_id, dept_name)
VALUES ( 703, 'Western Sales' )
```

The **dept_head_id** column has no default, but can allow NULL. A NULL is assigned to that column.

The order in which you list the column names must match the order in which you list the values. The following example produces the same results as the previous one:

```
INSERT INTO department (dept_name, dept_id )
VALUES ('Western Sales', 703)
```

Values for unspecified columns

When you specify values for only some of the columns in a row, one of four things can happen to the columns with no values specified:

- **NULL is entered** This occurs if the column allows NULL and no default value exists for the column.
- **A default value is entered** This occurs if a default exists for the column.
- **A unique, sequential value is entered** This occurs if the column has the AUTOINCREMENT default or the IDENTITY property.
- **The INSERT is rejected and an error message is displayed** This occurs if the column does not allow NULL and no default exists.

By default, columns allow NULL unless you explicitly state NOT NULL in the column definition when creating tables. You can alter the default using the ALLOW_NULLS_BY_DEFAULT option.

Restricting column data using constraints

You can create constraints for a column or user-defined data type. Constraints govern the kind of data that can or cannot be added.

For information on constraints, see "Using table and column constraints" on page 356.

Explicitly inserting NULL

You can explicitly insert NULL into a column by entering NULL. Do not enclose this in quotes, or it will be taken as a string.

For example, the following statement explicitly inserts NULL into the **dept_head_id** column:

```
INSERT INTO department
VALUES (703, 'Western Sales', NULL )
```

Using defaults to supply values

You can define a column so that, even though no value is inserted into the column, a default value is automatically filled in whenever a row is inserted. You do this by supplying a **default** for the column.

For information about defaults, see "Using column defaults" on page 352.

Adding new rows with SELECT

To pull values into a table from one or more other tables, you can use a SELECT clause in the INSERT statement. The select clause can insert values into some or all of the columns in a row.

Inserting values for only some columns can come in handy when you want to take some values from an existing table. Then, you can use update to add the values for the other columns.

Before inserting values for some, but not all, columns in a table, make sure that a default exists or that NULL has been specified for the columns for which you are not inserting values. Otherwise, an error is generated.

When you insert rows from one table into another, the two tables must have compatible structures—that is, the matching columns must be either the same data types or data types between which Adaptive Server automatically converts.

Example

If the columns are in the same order in their create table statements, you do not need to specify column names in either table. Suppose you have a table named **newproduct** that contains some rows of product information in the same format as in the **product** table. To add to **product** all the rows in **newproduct**:

```
INSERT product
SELECT *
FROM newproduct
```

You can use expressions in a SELECT statement inside an INSERT statement.

Inserting data into some columns

You can use the SELECT statement to add data to some, but not all, columns in a row just as you do with the VALUES clause. Simply specify the columns to which you want to add data in the INSERT clause.

Inserting Data from the Same Table

You can insert data into a table based on other data in the same table. Essentially, this means copying all or part of a row.

For example, you can insert new items into the **product** table that are based on existing products. The following statement adds new Extra Large Tee Shirts (of Tank Top, V-neck, and Crew Neck varieties) into the **product** table. The identification number is ten greater than the existing sized shirt:

```
INSERT INTO product
SELECT id+ 10, name, description,
    'Extra large', color, 50, unit_price
```

```
FROM product
WHERE name = 'Tee Shirt'
```

Changing data using UPDATE

You can use the UPDATE statement to change single rows, groups of rows, or all rows in a table. The UPDATE statement is followed by the name of the table or view. As in all data modification statements, you can change the data in only one table or view at a time.

The UPDATE statement specifies the row or rows you want changed and the new data. The new data can be a constant or an expression that you specify or data pulled from other tables.

If an UPDATE statement violates an integrity constraint, the update does not take place and an error message is generated. The update is canceled, for example, if one of the values being added is the wrong data type, or if it violates a constraint that has been defined for one of the columns or data types involved.

UPDATE syntax

A simplified version of the UPDATE syntax is:

UPDATE *table-name*
SET *column_name* = *expression*
WHERE *search-condition*

If the company Newton Ent. (in the customer table of the sample database) is taken over by Einstein, Inc., you can update the name of the company using a statement such as the following:

```
UPDATE customer
SET company_name = 'Einstein, Inc.'
WHERE company_name = 'Newton Ent.'
```

You can use any expression in the WHERE clause. If you are not sure how the company name was entered, you could try updating any company called Newton, with a statement such as the following:

```
UPDATE customer
SET company_name = 'Einstein, Inc.'
WHERE company_name LIKE 'Newton%'
```

The search condition need not refer to the column being updated. The company ID for Newton Entertainments is 109. As the ID value is the primary key for the table, you could be sure of updating the correct row using the following statement:

```
UPDATE customer
SET company_name = 'Einstein, Inc.'
WHERE id = 109
```

The SET clause

The SET clause specifies the columns to be updated, and their new values. The WHERE clause determines the row or rows are to be updated. If you do not have a WHERE clause, the specified columns of all rows are updated with the values given in the SET clause.

You can provide any expression of the correct data type in the SET clause.

The WHERE clause

The WHERE clause specifies the rows to be updated. For example, the following statement replaces the One Size Fits All Tee Shirt with an Extra Large Tee Shirt

```
UPDATE product
SET size  = 'Extra Large'
WHERE name = 'Tee Shirt'
   AND size = 'One Size Fits All'
```

The FROM clause

You can use a FROM clause to pull data from one or more tables into the table you are updating.

Deleting data using DELETE

Simple DELETE statements have the following form:

DELETE [**FROM**] *table-name*
WHERE *column-name* = *expression*

You can also use a more complex form, as follows

DELETE [**FROM**] *table-name*
FROM *table-list*
WHERE *search-condition*

The WHERE clause

The WHERE clause specifies which rows are to be removed. If no WHERE clause is given in the DELETE statement, all rows in the table are removed.

The FROM clause

The FROM clause in the second position of a DELETE statement is a special feature allowing you to select data from a table or tables and delete corresponding data from the first-named table. The rows you select in the FROM clause specify the conditions for the delete.

Example

This example uses the sample database. To execute the statements in the example, you should set the option WAIT_FOR_COMMIT to OFF. The following statement does this for the current connection only:

```
SET TEMPORARY OPTION WAIT_FOR_COMMIT = 'OFF'
```

This allows rows to be deleted even if they contain primary keys referenced by a foreign key, but does not permit a COMMIT unless the corresponding foreign key is deleted also.

The following view displays products and the value of that product that has been sold:

```
CREATE VIEW ProductPopularity as
SELECT  product.id,
   SUM(product.unit_price * sales_order_items.quantity)
as "Value Sold"
FROM  product JOIN sales_order_items
ON product.id = sales_order_items.prod_id
GROUP BY product.id
```

Using this view, you can delete those products which have sold less than $20,000 from the **product** table.

```
DELETE
FROM product
FROM product NATURAL JOIN ProductPopularity
WHERE "Value Sold" < 20000
```

You should roll back your changes when you have completed the example:

```
ROLLBACK
```

Deleting all rows from a table

You can use the TRUNCATE TABLE statement as a fast method of deleting all the rows in a table. It is faster than a DELETE statement with no conditions, because the delete logs each change, while truncate table operations are not recorded individually in the transaction log.

The table definition for a table emptied with the TRUNCATE TABLE statement remains in the database, along with its indexes and other associated objects, unless you enter a DROP TABLE statement.

You cannot use TRUNCATE TABLE if another table has rows that reference it through a referential integrity constraint. Delete the rows from the foreign table, or truncate the foreign table and then truncate the primary table.

TRUNCATE TABLE syntax

The syntax of truncate table is:

TRUNCATE TABLE *table-name*

For example, to remove all the data in the **sales_order** table, type the following:

```
TRUNCATE TABLE sales_order
```

A TRUNCATE TABLE statement does not fire triggers defined on the table.