CHAPTER 20

Data Modification

Abstract

This chapter covers SQL statements for inserting, modifying, and deleting data. It includes a discussion of the effect of deleting data on referential integrity, as well as merging data (updating on a condition).

Keywords

SQL
SQL INSERT
SQL UPDATE
SQL DELETE
SQL MERGE
SQL data modification

SQL includes commands for modifying the data in tables: INSERT, UPDATE, and DELETE. In many business environments, application programs provide forms-driven data modification, removing the need for end users to issue SQL data modification statements directly to the DBMS. (As you will see, this is a good thing because using SQL data modification statements are rather clumsy.) Nonetheless, if you are developing and testing database elements and need to populate tables and modify data, you will probably be working at the command line with the SQL syntax.

Note: This chapter is where it will make sense that we covered retrieval before data modification.

Inserting Rows

The SQL INSERT statement has two variations: one that inserts a single row into a table and a second that copies one or more rows from another table.

Inserting One Row

To add one row to a table, you use the general syntax

```
INSERT INTO table_name VALUES (value_list)
```

In the preceding form, the value list contains one value for every column in the table, in the order in which the columns were created. For example, to insert a new row into the *customer* table someone at the rare book store might use

```
INSERT INTO customer VALUES (8,'Helen','Jerry',
     '16 Main Street','Newtown','NJ','18886','209-555-8888');
```

There are two things to keep in mind when inserting data in this way:

- The format of the values in the value list must match the data types of the columns into which the data will be placed. In the current example, the first column requires an integer. The remaining columns all require characters and therefore the values have been surrounded by single quotes.
- When you insert a row of data, the DBMS checks any integrity constraints that you have placed on the table. For the preceding example, it will verify that the customer number is unique and not null. If the constraints are not met, you will receive an error message and the row will not be added to the table.

If you do not want to insert data into every column of a table, you can specify the columns into which data should be placed:

```
INSERT INTO table_name (column_list) VALUES (value_list)
```

There must be a one-to-one correspondence between the columns in the column list and the values in the value list because the DBMS matches them by their relative positions in the lists.

As an example, assume that someone at the rare book store wants to insert a row into the *book* table, but doesn't know the binding type. The SQL would then be written

```
INSERT INTO book (isbn, work_numb, publisher_id,
    edition, copyright_year)
    VALUES ('978-11111-100-1',16,2,12,1960);
```

There are five columns in the column list and therefore five values in the value list. The first value in the list will be inserted into the *isbn* column, the second value into the *work_numb* column, and so on. The column omitted from the lists—*binding*—will remain null. You therefore must be sure to place values at least in primary key columns. Otherwise, the DBMS will not permit the insertion to occur.

Although it is not necessary to list column names when inserting values for every column in a table, there is one good reason to do so, especially when embedding the INSERT statement in an application program. If the structure of the table changes—if columns are added, deleted, or rearranged—then an INSERT without column names will no longer work properly. By always specifying column names, you can avoid unnecessary program modifications as your database changes to meet your changing needs.

Placement of New Rows

Where do new rows go when you add them? That depends on your DBMS. Typically, a DBMS maintains a unique internal identifier for each row that is not accessible to users (something akin to the combination of a row number and a table identifier) to provide information about the row's physical storage location. These identifiers continue to increase in value.

If you were to use the SELECT * syntax on a table, you would see the rows in internal identifier order. At the beginning of a table's life, this order corresponds to the order in which rows were added to the table. New rows appear to go at the "bottom" of the table, after all existing rows. As rows are deleted from the table, there will be gaps in the sequence of row identifiers. However, the DBMS does not reuse them (to "fill in the holes") until it has used up all available identifiers. If a database is very old, very large, and/or very active, the DBMS will run out of new identifiers and will then start to reuse those made available by deleted rows. In that case, new rows may appear anywhere in the table. Given that you can view rows in any order by using the ORDER BY clause, it should make absolutely no difference to an end user or an application program where a new row is added.

Copying Existing Rows

The SQL INSERT statement can also be used to copy one or more rows from one table to another. The rows that will be copied are specified with a SELECT, giving the statement the following general syntax:

```
INSERT INTO table_name SELECT complete_SELECT_statement
```

The columns in the SELECT must match the columns of the table. For the purposes of this example, we will add a simple table to the rare book store database:

```
summary (isbn, how_many)
```

This table will contain summary information gathered from the *volume* table. To add rows to the new table, the INSERT statement can be written:

```
INSERT INTO summary
SELECT isbn, COUNT (*)
FROM volume
```

```
GROUP BY isbn;
```

The result is 27 rows copied into the *summary* table, one for each unique ISBN in the *volume* table.

Note: Should you store summary data like that placed in the table created in the preceding example? The answer is "it depends." If it takes a long time to generate the summary data and you use the data frequently, then storing it probably makes sense. But if you can generate the summary data easily and quickly, then it is just as easy not to store it and to create the data whenever it is needed for output.

Updating Data

Although most of today's end users modify existing data using an on-screen form, the SQL statements to modify the data must nonetheless be issued by the program providing the form. For example, as someone at the rare book store adds volumes to a sale, the *volume* table is updated with the selling price and the sale ID. The selling price is also added to the total amount of the sale in the *sale* table.

The SQL UPDATE statement affects one or more rows in a table, based on row selection criteria in a WHERE predicate. UPDATE has the following general syntax:

```
UPDATE table_name
SET column1 = new_value, column2 = new_value, …
WHERE row_selection_predicate
```

If the WHERE predicate contains a primary key expression, then the UPDATE will affect only one row. For example, to change a customer's address, the rare book store could use

```
UPDATE customer
SET street = '195 Main Street'
      city = 'New Town'
       zip = '11111'
WHERE customer_numb = 5;
```

However, if the WHERE predicate identifies multiple rows, each row that meets the criteria in the predicate will be modified. To raise all $50 prices to $55, someone at the rare book store might write a query as

```
UPDATE books
SET asking_price = 55
WHERE asking_price = 50;
```

Notice that it is possible to modify the value in a column being used to identify rows. The DBMS will select the rows to be modified before making any changes to them.

If you leave the WHERE clause off an UPDATE, the same modification will be applied to every row in the table. For example, assume that we add a column for sales tax to the *sale* table. Someone at the rare book store could use the following statement to compute the tax for every sale:

```
UPDATE sale
SET sales_tax = sale_total_amt * 0.075;
```

The expression in the SET clause takes the current value in the *sale_total_amt* column, multiplies it by the tax rate, and puts it in the *sales_tax* column.

Deleting Rows

Like the UPDATE statement, the DELETE statement affects one or more rows in a table based on row selection criteria in a WHERE predicate. The general syntax for DELETE is

```
DELETE FROM table_name
WHERE row_selection_predicate
```

For example, if a customer decided to cancel an entire purchase, then someone at the rare book store would use something like

```
DELETE FROM sale
WHERE customer_numb = 12 AND sale_date = '05-Jul-2021';
```

Assuming that all purchases on the same date are considered a single sale, the WHERE predicate identifies only one row. Therefore, only one row is deleted.

When the criteria in a WHERE predicate identify multiple rows, all those matching rows are removed. If someone at the rare book store wanted to delete all sales for a specific customer, then the SQL would be written

```
DELETE FROM sale
WHERE customer_numb = 6;
```

In this case, there are multiple rows for customer number 6, all of which will be deleted.

DELETE is a potentially dangerous operation. If you leave off the WHERE clause—

```
DELETE FROM sale
```

—you will delete every row in the table! (The table remains in the database without any rows.)

Deletes and Referential Integrity

The preceding examples of DELETE involve a table that has a foreign key in another table (*sale_id* in *volume*) referencing it. It also has a foreign key of its own (*customer_numb* referencing the primary key of *customer*). You can delete rows containing foreign keys without any effect on the rest of the database, but what happens when you attempt to delete rows that *do* have foreign keys referencing them?

Note: The statement in the preceding paragraph refers to database integrity issues and clearly misses the logical issue of the need to decrement the total sale amount in the sale table whenever a volume is removed from the sale.

Assume, for example, that a customer cancels a purchase. Your first thought might be to delete the row for that sale from the *sale* table. There are, however, rows in the *volume* table that reference that sale and if the row for the sale is removed from *sale*, there will be no primary key for the rows in *volume* to reference and referential integrity will be violated.

As you will remember from our discussion of creating tables with foreign keys, what actually happens in such a situation depends on what was specified when the table containing the primary key being referenced was created (SET NULL, SET DEFAULT, CASCADE, NO ACTION).

specified when the table containing the primary key being referenced was created (SET NULL, SET DEFAULT, CASCADE, NO ACTION).

Deleting All Rows: TRUNCATE TABLE

The 2008 SQL standard introduced a new command—TRUNCATE TABLE—that removes all rows from a table more quickly than a DELETE, without a WHERE clause. This can be particularly useful when you are developing a database and need to remove test data but leave the table structure untouched. The command's general syntax is

```
TRUNCATE TABLE table_name
```

Like the DELETE without a WHERE clause, the table structure remains intact, and in the data dictionary.

There are some limits to using the command:

- It cannot be used on a table that has foreign keys referencing it.
- It cannot be used on a table on which indexed views are based.
- It cannot activate a trigger (a program module stored in the database that is initiated when a specific event, such as a delete, occurs).

Although DELETE and TRUNCATE TABLE seem to have the same effect, they do work differently. DELETE removes the rows one at a time and writes an entry into the database log file for each row. In contrast, TRUNCATE TABLE deallocates space in the database files, making the space formerly occupied by the truncated table's rows available for other use.

Inserting, Updating, or Deleting on a Condition: MERGE

The SQL:2003 standard introduced a very powerful and flexible way to insert, update, or delete data using the MERGE statement. MERGE[1] includes a condition to be tested and alternative sets of actions that are performed when the condition is or is not met. The model behind this statement is the merging of a table of transactions into a master table.

MERGE has the following general syntax:

```
MERGE INTO target_table_name
      USING source_table_name ON merge_condition
WHEN MATCHED THEN
      update/delete_specification
WHEN NOT MATCHED THEN
      insert specification
```

Notice that when the merge condition is matched (in other words, evaluates as true for a given row), an update and/or delete is performed. When the condition is not matched, an insert is performed. Either the MATCHED or NOT MATCHED clause is optional.

The target table is the table that will be affected by the changes made by the statement. The source table—which can be a base table or a virtual table generated by a SELECT—provides the source of the table. To help you understand how MERGE works, let's use the classic model of applying transactions to a master table. First, we need a transaction table:

```
transaction (sale_id, inventory_id, selling_price,
      sale_date, customer_numb)
```

The *transaction* table contains information about the sale of a single volume. (It really doesn't contain all the necessary columns for the *sale* table but it will do for this example.) If a row for the sale exists in the *sale* table, then the selling price of the volume should be added to existing sale total. However, if the sale is not in the *sale* table, then a new row should be created and the sale total set to the selling price of the volume. A MERGE statement that will do the trick might be written as

```
MERGE INTO sale S USING transaction T ON (S.sale_id = T.sale_id)
WHEN MATCHED THEN
      UPDATE SET sale_total_amt = sale_total_amt + selling_price
WHEN NOT MATCHED
      INSERT (sale_id, customer_numb, sale_date, sale_total_amt)
```

```
INSERT (sale_id, customer_numb, sale_date, sale_total_amt)
      VALUES (T.sale_id, T.customer_numb, T.sale_date,
      T.selling_price);
```

The target table is *sale*; the source table is *transaction*. The merge condition looks for a match between sale IDs. If a match is found, then the UPDATE portion of the command performs the modification of the *sale_total_amt* column. If no match is found, then the insert occurs. Notice that the INSERT portion of the command does not need a table name because the table affected by the INSERT has already been specified as the target table.

As we said earlier, the source table for a merge operation doesn't need to be a base table; it can be a virtual table created on the fly using a SELECT. For example, assume that someone at the rare book store needs to keep a table of total purchases made by each customer. The following table can be used to hold that data:

```
summary_stats (customer_numb, year, total_purchases)
```

You can find the MERGE statement below. The statement assembles the summary data using a SELECT that extracts the year from the sale date and sums the sale amounts. Then, if a summary row for a year already exists in *summary_stats*, the MERGE adds the amount from the source table to what is stored already in the target table. Otherwise, it adds a row to the target table.

```
MERGE INTO summary_stats AS S USING
      (SELECT customer_numb,
      EXTRACT (YEAR FROM sale_date) AS Y, sum
(sale_total_amt AS M) AS T
      FROM sale
      GROUP BY customer_numb, Y)
ON (CAST(S.customer_numb AS CHAR (4)
      || CAST (S.year AS CHAR(4) =
CAST(T.customer_numb AS CHAR (4) || CAST (T.year AS CHAR(4)
WHEN MATCHED
      UPDATE SET total_purchases = T.M
WHEN NOT MATCHED
      INSERT VALUES (customer_numb, Y, M);
```

As powerful as MERGE seems to be, the restriction of UPDATE/DELETE to the matched condition and INSERT to the unmatched prevent it from being able to handle some situations. For example, if someone at the rare book store wanted to archive all orders more than two years old, the process would involve creating a row for each sale that didn't exist in the archive table and then deleting the row from the *sale* table. (We're assuming that the delete cascades, removing all rows from *volume* as well.) The problem is that the delete needs to occur on the unmatched condition, which isn't allowed with the MERGE syntax.

[1] Some DBMSs call this functionality UPSERT.

CHAPTER 21

Creating Additional Structural Elements

Abstract

This chapter discusses the creation and use of SQL to manage database elements not covered in previous chapters. Topics include views, temporary tables, common table expressions, and indexes. The discussion of temporary tables covers creating the tables, loading them with data, and disposing of them when they are no longer needed. The chapter not only looks at how to create views, but also includes a discussion of deciding which views are needed.

Keywords

SQL
SQL views
SQL temporary tables
SQL common table expressions
CTEs
SQL CTEs
SQL indexes
CREATE INDEX

Views

As you first read in Chapter 5, views provide a way to give users a specific portion of a larger schema with which they can work. Before you actually can create views, there are two things you should consider: which views you really need and whether the views can be used for updating data.

Deciding Which Views to Create

Views take up very little space in a database, occupying only a few rows in a data dictionary table. That being the case, you can feel free to create views as needed.

A typical database might include the following views:

- One view for every base table that is exactly the same as the base table, but with a different name. Then, you prevent end users from seeing the base tables and do not tell the end users the table names; you give end users access only to the views. This makes it harder for end users to attempt to gain access to the stored tables because they do not know their names. However, as you will see in the next section, it is essential for updating that there be views that do match the base tables.
- One view for each primary key–foreign key relationship over which you join frequently. If the tables are large, the actual syntax of the statement may include structures for avoiding the join operation, but still combining the tables.
- One view for each complex query that you issue frequently.
- Views as needed to restrict user access to specific columns and rows. For example, you might recreate a view for a receptionist that shows employee office numbers and telephone extensions, but leaves off home address, telephone number, and salary.

View Updatability Issues

A database query can apply any operations supported by its DBMS's query language to a view, just as it can to base tables. However, using views for updates is a much more complicated issue. Given that views exist only in main memory, any updates made to a view must be stored in the underlying base tables if the updates are to have any effect on the database.

Not every view is updatable, however. Although the rules for view updatability vary from one DBMS to another, you will find that many DBMSs share the following restrictions:[1]

- A view must be created from one base table or view, or if the view uses joined tables, only one of the underlying base tables can be updated.
- If the source of the view is another view, then the source view must also adhere to the rules for updatability.
- A view must be created from only one query. Two or more queries cannot be assembled into a single view table using operations such as union.
- The view must include the primary key columns of the base table.
- The view must include all columns specified as not null (columns requiring mandatory values).
- The view must not include any groups of data. It must include the original rows of data from the base table, rather than rows based on values common to groups of data.
- The view must not remove duplicate rows.

Creating Views

Creating Views

To create a view whose columns have the same name as the columns in the base tables from which it is derived, you give the view a name and include the SQL query that defines its contents: