Data Manipulation

- Inserting Data
- Updating Data
- Deleting Data
- Returning Data from Modified Rows

The previous chapter discussed how to create tables and other structures to hold your data. Now it is time to fill the tables with data. This chapter covers how to insert, update, and delete table data. The chapter after this will finally explain how to extract your long-lost data from the database.

Inserting Data

inserting

INSERT

When a table is created, it contains no data. The first thing to do before a database can be of much use is to insert data. Data is inserted one row at a time. You can also insert more than one row in a single command, but it is not possible to insert something that is not a complete row. Even if you know only some column values, a complete row must be created.

To create a new row, use the ??? command. The command requires the table name and column values. For example, consider the products table from ???:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric
);
```

An example command to insert a row would be:

```
INSERT INTO products VALUES (1, 'Cheese', 9.99);
```

The data values are listed in the order in which the columns appear in the table, separated by commas. Usually, the data values will be literals (constants), but scalar expressions are also allowed.

The above syntax has the drawback that you need to know the order of the columns in the table. To avoid this you can also list the columns explicitly. For example, both of the following commands have the same effect as the one above:

```
INSERT INTO products (product_no, name, price) VALUES (1, 'Cheese', 9.99);
INSERT INTO products (name, price, product_no) VALUES ('Cheese', 9.99, 1);
```

Many users consider it good practice to always list the column names.

If you don't have values for all the columns, you can omit some of them. In that case, the columns will be filled with their default values. For example:

```
INSERT INTO products (product_no, name) VALUES (1, 'Cheese');
INSERT INTO products VALUES (1, 'Cheese');
```

The second form is a PostgreSQL extension. It fills the columns from the left with as many values as are given, and the rest will be defaulted.

For clarity, you can also request default values explicitly, for individual columns or for the entire row:

```
INSERT INTO products (product_no, name, price) VALUES (1, 'Cheese', DEFAULT);
INSERT INTO products DEFAULT VALUES;
```

You can insert multiple rows in a single command:

```
INSERT INTO products (product_no, name, price) VALUES
    (1, 'Cheese', 9.99),
    (2, 'Bread', 1.99),
    (3, 'Milk', 2.99);
```

It is also possible to insert the result of a query (which might be no rows, one row, or many rows):

```
INSERT INTO products (product_no, name, price)
  SELECT product_no, name, price FROM new_products
    WHERE release_date = 'today';
```

This provides the full power of the SQL query mechanism (???) for computing the rows to be inserted.

> When inserting a lot of data at the same time, consider using the ??? command. It is not as flexible as the ??? command, but is more efficient. Refer to ??? for more information on improving bulk loading performance.

Updating Data

updating

UPDATE

The modification of data that is already in the database is referred to as updating. You can update individual rows, all the rows in a table, or a subset of all rows. Each column can be updated separately; the other columns are not affected.

To update existing rows, use the ??? command. This requires three pieces of information:

1. The name of the table and column to update
2. The new value of the column
3. Which row(s) to update

Recall from ??? that SQL does not, in general, provide a unique identifier for rows. Therefore it is not always possible to directly specify which row to update. Instead, you specify which conditions a row must meet in order to be updated. Only if you have a primary key in the table (independent of whether you declared it or not) can you reliably address individual rows by choosing a condition that matches the primary key. Graphical database access tools rely on this fact to allow you to update rows individually.

For example, this command updates all products that have a price of 5 to have a price of 10:

```
UPDATE products SET price = 10 WHERE price = 5;
```

This might cause zero, one, or many rows to be updated. It is not an error to attempt an update that does not match any rows.

Let's look at that command in detail. First is the key word `UPDATE` followed by the table name. As usual, the table name can be schema-qualified, otherwise it is looked up in the path. Next is the key word `SET` followed by the column name, an equal sign, and the new column value. The new column value can be any scalar expression, not just a constant. For example, if you want to raise the price of all products by 10% you could use:

```
UPDATE products SET price = price * 1.10;
```

As you see, the expression for the new value can refer to the existing value(s) in the row. We also left out the `WHERE` clause. If it is omitted, it means that all rows in the table are updated. If it is present, only those rows that match the `WHERE` condition are updated. Note that the equals sign in the `SET` clause is an assignment while the one in the `WHERE` clause is a comparison, but this does not create any ambiguity. Of course, the `WHERE` condition does not have to be an equality test. Many other operators are available (see ???). But the expression needs to evaluate to a Boolean result.

You can update more than one column in an `UPDATE` command by listing more than one assignment in the `SET` clause. For example:

```
UPDATE mytable SET a = 5, b = 3, c = 1 WHERE a > 0;
```

Deleting Data

deleting

DELETE

So far we have explained how to add data to tables and how to change data. What remains is to discuss how to remove data that is no longer needed. Just as adding data is only possible in whole rows, you can only remove entire rows from a table. In the previous section we explained that SQL does not provide a way to directly address individual rows. Therefore, removing rows can only be done by specifying conditions that the rows to be removed have to match. If you have a primary key in the table then you can specify the exact row. But you can also remove groups of rows matching a condition, or you can remove all rows in the table at once.

You use the ??? command to remove rows; the syntax is very similar to the ??? command. For instance, to remove all rows from the products table that have a price of 10, use:

```
DELETE FROM products WHERE price = 10;
```

If you simply write:

```
DELETE FROM products;
```

then all rows in the table will be deleted! Caveat programmer.

Returning Data from Modified Rows

RETURNING

INSERT

RETURNING

UPDATE

RETURNING

DELETE

RETURNING

MERGE

RETURNING

Sometimes it is useful to obtain data from modified rows while they are being manipulated. The `INSERT`, `UPDATE`, `DELETE`, and `MERGE` commands all have an optional `RETURNING` clause that supports this. Use of `RETURNING` avoids performing an extra database query to collect the data, and is especially valuable when it would otherwise be difficult to identify the modified rows reliably.

The allowed contents of a `RETURNING` clause are the same as a `SELECT` command's output list (see ???). It can contain column names of the command's target table, or value expressions using those columns. A common shorthand is `RETURNING *`, which selects all columns of the target table in order.

In an `INSERT`, the default data available to `RETURNING` is the row as it was inserted. This is not so useful in trivial inserts, since it would just repeat the data provided by the client. But it can be very handy when relying on computed default values. For example, when using a `serial` column to provide unique identifiers, `RETURNING` can return the ID assigned to a new row:

```
CREATE TABLE users (firstname text, lastname text, id serial PRIMARY KEY);


INSERT INTO users (firstname, lastname) VALUES ('Joe', 'Cool') RETURNING id;
```

The `RETURNING` clause is also very useful with `INSERT ... SELECT`.

In an `UPDATE`, the default data available to `RETURNING` is the new content of the modified row. For example:

```
UPDATE products SET price = price * 1.10
  WHERE price <= 99.99
  RETURNING name, price AS new_price;
```

In a `DELETE`, the default data available to `RETURNING` is the content of the deleted row. For example:

```
DELETE FROM products
  WHERE obsoletion_date = 'today'
  RETURNING *;
```

In a `MERGE`, the default data available to `RETURNING` is the content of the source row plus the content of the inserted, updated, or deleted target row. Since it is quite common for the source and target to have many of the same columns, specifying `RETURNING *` can lead to a lot of duplicated columns, so it is often more useful to qualify it so as to return just the source or target row. For example:

```
MERGE INTO products p USING new_products n ON p.product_no = n.product_no
  WHEN NOT MATCHED THEN INSERT VALUES (n.product_no, n.name, n.price)
  WHEN MATCHED THEN UPDATE SET name = n.name, price = n.price
  RETURNING p.*;
```

In each of these commands, it is also possible to explicitly return the old and new content of the modified row. For example:

```
UPDATE products SET price = price * 1.10
  WHERE price <= 99.99
  RETURNING name, old.price AS old_price, new.price AS new_price,
            new.price - old.price AS price_change;
```

In this example, writing `new.price` is the same as just writing `price`, but it makes the meaning clearer.

This syntax for returning old and new values is available in `INSERT`, `UPDATE`, `DELETE`, and `MERGE` commands, but typically old values will be `NULL` for an `INSERT`, and new values will be `NULL` for a `DELETE`. However, there are situations where it can still be useful for those commands. For example, in an `INSERT` with an `ON CONFLICT DO SELECT/UPDATE` clause, the old values will be non-`NULL` for conflicting rows. Similarly, if a `DELETE` is turned into an `UPDATE` by a rewrite rule, the new values may be non-`NULL`.

If there are triggers (???) on the target table, the data available to `RETURNING` is the row as modified by the triggers. Thus, inspecting columns computed by triggers is another common use-case for

`RETURNING`.

□□□ #2
□ □□□ □□□ 2026-10-31 14:06:39 UTC
□ □□□ □□□ 2026-10-31 14:06:40 UTC