codecademy

SQL: Creating, Updating, and Deleting Data

`CREATE TABLE` Statement

The `CREATE TABLE` statement creates a new table in a database. It allows one to specify the name of the table and the name of each column in the table.

```
CREATE TABLE table_name (
  column1 datatype,
  column2 datatype,
  column3 datatype
);
```

`INSERT` Statement

The `INSERT INTO` statement is used to add a new record (row) to a table.
It has two forms as shown:

Insert into columns in order.

Insert into columns by name.

```
-- Insert into columns in order:
INSERT INTO table_name
VALUES (value1, value2);

-- Insert into columns by name:
INSERT INTO table_name (column1, column2)
VALUES (value1, value2);
```

`UPDATE` Statement

The `UPDATE` statement is used to edit records (rows) in a table. It includes a `SET` clause that indicates the column to edit and a `WHERE` clause for specifying the record(s).

```
UPDATE table_name
SET column1 = value1, column2 = value2
WHERE some_column = some_value;
```

`ALTER TABLE` Statement

The `ALTER TABLE` statement is used to modify the columns of an existing table. When combined with the `ADD COLUMN` clause, it is used to add a new column.

```
ALTER TABLE table_name
ADD column_name datatype;
```

`DELETE` Statement

The `DELETE` statement is used to delete records (rows) in a table. The `WHERE` clause specifies which record or records that should be deleted. If the `WHERE` clause is omitted, all records will be deleted.

```
DELETE FROM table_name
WHERE some_column = some_value;
```

Column Constraints

Column constraints are the rules applied to the values of individual columns:

- `PRIMARY KEY` constraint can be used to uniquely identify the row.
- `UNIQUE` columns have a different value for every row.
- `NOT NULL` columns must have a value.
- `DEFAULT` assigns a default value for the column when no value is specified.

There can be only one `PRIMARY KEY` column per table and multiple `UNIQUE` columns.

```
CREATE TABLE student (
  id INTEGER PRIMARY KEY,
  name TEXT UNIQUE,
  grade INTEGER NOT NULL,
  age INTEGER DEFAULT 10
);
```

Data Types As Constraints

Columns of a PostgreSQL database table must have a data type, which constrains the type of information that can be entered into that column. This is important in order to ensure data integrity and consistency over time. Some common PostgreSQL types are `integer`, `decimal`, `varchar`, and `boolean`. Data types are defined in a `CREATE TABLE` statement by indicating the data type after each column name.

```
CREATE TABLE tablename (
    myNum integer,
    myString varchar(50)
);
```

Check Constraints

When using PostgreSQL, it can be important to enforce check constraints on columns of a database table in order to ensure data integrity and consistency over time. Check constraints can be enforced on a single column, multiple columns, or on all columns. They are implemented within a `CREATE TABLE` statement using the `CHECK` keyword.

```
CREATE TABLE table_name (
    column_1 integer,
    column_2 text,
    column_3 numeric CHECK (column_3 > 0),
    column_4 numeric CHECK (column_4 > 0),
    CHECK (column_3 > column_4)
);
```

Multiple Constraints

Columns in a database table can have multiple constraints. Multiple constraints can be implemented by listing them in a row following the relevant column name and data type in a `CREATE TABLE` statement. The order of the constraints does not matter.

```
CREATE TABLE table_name (
    column_1 integer NOT NULL CHECK
(column_3 > 0),
    column_2 text UNIQUE NOT NULL,
    column_3 numeric
);
```

NOT NULL

In PostgreSQL, `NOT NULL` constraints can be used to ensure that particular columns of a database table do not contain missing data. This is important for ensuring database integrity and consistency over time. `NOT NULL` constraints can be enforced within a `CREATE TABLE` statement using `NOT NULL` .

```
CREATE TABLE table_name (
    column_1 integer NOT NULL,
    column_2 text NOT NULL,
    column_3 numeric
);
```

UNIQUE

In PostgreSQL, `UNIQUE` constraints can be used to ensure that elements of a particular column (or group of columns) are unique (i.e., no two rows have the same value or combination of values). This is important for ensuring database integrity and consistency over time. `UNIQUE` constraints can be enforced within a `CREATE TABLE` statement using the `UNIQUE` keyword.

```
CREATE TABLE table_name (
    column_1 integer UNIQUE,
    column_2 text UNIQUE,
    column_3 numeric,
    column_4 text,
    UNIQUE(column_3, column_4)
);
```

Primary Key Constraint

In PostgreSQL, a primary key constraint indicates that a particular column (or group of columns) in a database table can be used to identify a unique row in that table. In terms of restrictions, this is equivalent to a `UNIQUE NOT NULL` constraint; however, a table may only have one primary key, whereas multiple columns can be constrained as `UNIQUE NOT NULL` . A primary key constraint can be enforced within a `CREATE TABLE` statement using `PRIMARY KEY`

```
-- A primary key on one column
CREATE TABLE table_name (
    column_1 integer PRIMARY KEY,
    column_2 text,
);

-- A composite primary key
CREATE TABLE table_name (
    column_1 integer,
    column_2 text,
    column_2 integer,
    PRIMARY KEY (column_1, column_2),
);
```

Foreign Key Constraint

In PostgreSQL, a foreign key constraint ensures that the values in a particular column of a database table exactly match values in another database table. This is important to ensure the “referential integrity” of the database. A foreign key constraint can be enforced within a `CREATE TABLE` statement either by adding `REFERENCES other_table_name (other_table_primary_key)` after the relevant column name and type or using `FOREIGN KEY (column_1, column_2) REFERENCES other_table_name (other_key1, other_key2)` to indicate a link between groups of columns.

```
CREATE TABLE table_1 (
    column_1 integer PRIMARY KEY,
    column_2 text,
    column_3 numeric
);

-- Option 1
CREATE TABLE table_2 (
    column_a integer PRIMARY KEY,
    column_b integer REFERENCES table_1
(column_2),
    column_c integer
);

-- Option 2 - Creating columns b and c are
a composite foreign key referencing
columns c1 and c2 from other_table
CREATE TABLE t1 (
  a integer PRIMARY KEY,
  b integer,
  c integer,
  FOREIGN KEY (b, c) REFERENCES
other_table (c1, c2)
);
```

Cascade and Restrict

In PostgreSQL, when implementing foreign key constraints in a database table, it is possible to preemptively specify database behavior when values in a referenced column are updated or deleted. Specifying `ON DELETE RESTRICT` or `ON UPDATE RESTRICT` after a foreign key constraint ensures that referenced values/rows cannot be deleted/updated. Specifying `ON DELETE CASCADE` or `ON UPDATE CASCADE` ensures that updated/deleted values/rows in the referenced table are automatically updated/deleted in the referencing table.

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

CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    shipping_address text,
);

CREATE TABLE order_items (
    product_no integer REFERENCES products
ON DELETE RESTRICT,
    order_id integer REFERENCES orders ON
DELETE CASCADE,
    quantity integer,
    PRIMARY KEY (product_no, order_id)
);
```

Updating A Table With Constraints

In PostgreSQL, when implementing a constraint on an existing table, the table must already be consistent with the constraint or PostgreSQL will reject the new constraint. A DB user may backfill the table using `UPDATE` or `ALTER TABLE` statements to make the table consistent with the constraint.

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


--Assume some values in `sale_price` are
missing, and we'd like to apply a `NOT
NULL` constraint on `sale_price`. We must
`UPDATE` the table before applying our
constraint.

UPDATE TABLE products
SET sale_price = 0 WHERE sale_price IS
NULL
```