Lecture 4: Data manipulation and definition

Contents

Lecture outline

- Inserting, updating, and deleting data
- Creating, altering and dropping tables
- Integrity constraints

So far in the course, we concentrated on retrieving data from a database and its tables using `SELECT` statements. In this lecture, you will learn how to make modifications to rows and tables, delete existing ones, and make new ones. You'll also learn about how to enforce constraints on your tables such that your database always stays in good shape.

```
%load_ext sql
%config SqlMagic.displaylimit = 30
```

Skip to main content

```
import json
import urllib.parse

with open('data/credentials.json') as f:
    login = json.load(f)

username = login['user']
password = urllib.parse.quote(login['password'])
host = login['host']
port = login['port']
```

```
%sql postgresql://{username}:{password}@{host}:{port}/mds
```

```
'Connected: postgres@mds'
```

```
%%sql

DROP TABLE IF EXISTS
    instructor,
    instructor_course,
    course_cohort
;

CREATE TABLE instructor (
    id INTEGER PRIMARY KEY,
    name TEXT,
    email TEXT,
    phone VARCHAR(12),
    department VARCHAR(50)
    )
;

INSERT INTO
    instructor (id, name, email, phone, department)
VALUES
    (1, 'Mike', 'mike@mds.ubc.ca', '605-332-2343', 'Computer Science'),
    (2, 'Tiffany', 'tiff@mds.ubc.ca', '445-794-2233', 'Neuroscience'),
    (3, 'Arman', 'arman@mds.ubc.ca', '935-738-5796', 'Physics'),
    (4, 'Varada', 'varada@mds.ubc.ca', '243-924-4446', 'Computer Science'),
    (5, 'Quan', 'quan@mds.ubc.ca', '644-818-0254', 'Economics'),
    (6, 'Joel', 'joel@mds.ubc.ca', '773-432-7669', 'Biomedical Engineering'),
    (7, 'Florencia', 'flor@mds.ubc.ca', '773-926-2837', 'Biology'),
    (8, 'Alexi', 'alexiu@mds.ubc.ca', '421-888-4550', 'Statistics'),
    (15, 'Vincenzo', 'vincenzo@mds.ubc.ca', '776-543-1212', 'Statistics'),
    (19, 'Gittu', 'gittu@mds.ubc.ca', '776-334-1132', 'Biomedical Engineering'
    (16, 'Jessica', 'jessica@mds.ubc.ca', '211-990-1762', 'Computer Science')
;
```

Skip to main content

```
CREATE TABLE instructor_course (
    id SERIAL PRIMARY KEY,
    instructor_id INTEGER,
    course TEXT,
    enrollment INTEGER,
    begins DATE
    )
;

INSERT INTO
    instructor_course (instructor_id, course, enrollment, begins)
VALUES
    (8, 'Statistical Inference and Computation I', 125, '2021-10-01'),
    (8, 'Regression II', 102, '2022-02-05'),
    (1, 'Descriptive Statistics and Probability', 79, '2021-09-10'),
    (1, 'Algorithms and Data Structures', 25, '2021-10-01'),
    (3, 'Algorithms and Data Structures', 25, '2021-10-01'),
    (3, 'Python Programming', 133, '2021-09-07'),
    (3, 'Databases & Data Retrieval', 118, '2021-11-16'),
    (6, 'Visualization I', 155, '2021-10-01'),
    (6, 'Privacy, Ethics & Security', 148, '2022-03-01'),
    (2, 'Programming for Data Manipulation', 160, '2021-09-08'),
    (7, 'Data Science Workflows', 98, '2021-09-15'),
    (2, 'Data Science Workflows', 98, '2021-09-15'),
    (12, 'Web & Cloud Computing', 78, '2022-02-10'),
    (10, 'Introduction to Optimization', NULL, '2022-09-01'),
    (9, 'Parallel Computing', NULL, '2023-01-10'),
    (13, 'Natural Language Processing', NULL, '2023-09-10')
;

CREATE TABLE course_cohort (
    id INTEGER,
    cohort VARCHAR(7)
    )
;

INSERT INTO
    course_cohort (id, cohort)
VALUES
    (13, 'MDS-CL'),
    (8, 'MDS-CL'),
    (1, 'MDS-CL'),
    (3, 'MDS-CL'),
    (1, 'MDS-V'),
    (9, 'MDS-V'),
    (3, 'MDS-V')
;
```

```
 * postgresql://postgres:***@localhost:5432/mds
Done.
Done.
11 rows affected.
```

Skip to main content

```
16 rows affected.
Done.
7 rows affected.
```

```
[]
```

Let's take a look at the tables of the `mds` database that we created in lecture 3 to demonstrate various types of joins:

```
%sql SELECT * FROM instructor;
```

```
 * postgresql://postgres:***@localhost:5432/mds
11 rows affected.
```

id	name	email	phone	department
1	Mike	mike@mds.ubc.ca	605-332-2343	Computer Science
2	Tiffany	tiff@mds.ubc.ca	445-794-2233	Neuroscience
3	Arman	arman@mds.ubc.ca	935-738-5796	Physics
4	Varada	varada@mds.ubc.ca	243-924-4446	Computer Science
5	Quan	quan@mds.ubc.ca	644-818-0254	Economics
6	Joel	joel@mds.ubc.ca	773-432-7669	Biomedical Engineering
7	Florencia	flor@mds.ubc.ca	773-926-2837	Biology
8	Alexi	alexiu@mds.ubc.ca	421-888-4550	Statistics
15	Vincenzo	vincenzo@mds.ubc.ca	776-543-1212	Statistics
19	Gittu	gittu@mds.ubc.ca	776-334-1132	Biomedical Engineering
16	Jessica	jessica@mds.ubc.ca	211-990-1762	Computer Science

```
%sql SELECT * FROM instructor_course;
```

```
 * postgresql://postgres:***@localhost:5432/mds
16 rows affected.
```

Skip to main content

id	instructor_id	course	enrollment	begins
1	8	Statistical Inference and Computation I	125	2021-10-01
2	8	Regression II	102	2022-02-05
3	1	Descriptive Statistics and Probability	79	2021-09-10
4	1	Algorithms and Data Structures	25	2021-10-01
5	3	Algorithms and Data Structures	25	2021-10-01
6	3	Python Programming	133	2021-09-07
7	3	Databases & Data Retrieval	118	2021-11-16
8	6	Visualization I	155	2021-10-01
9	6	Privacy, Ethics & Security	148	2022-03-01
10	2	Programming for Data Manipulation	160	2021-09-08
11	7	Data Science Workflows	98	2021-09-15
12	2	Data Science Workflows	98	2021-09-15
13	12	Web & Cloud Computing	78	2022-02-10
14	10	Introduction to Optimization	None	2022-09-01
15	9	Parallel Computing	None	2023-01-10
16	13	Natural Language Processing	None	2023-09-10

Inserting, modifying, and deleting rows

A database is rarely only used for retrieving data. We often want to insert new data, update existing data, or delete obsolete data. You might remember from lecture 1 that this relates to the data manipulation language (DML) that a DBMS also provides along with its data query language (DQL). For relational DBMSs, SQL provides standard statements for data manipulation using the `INSERT`, `UPDATE`, and `DELETE` keywords.

With row insertion, updating and deletion statements, we typically need to know in advance

Skip to main content

easily inspect the columns of a table using `psql`'s meta-commands that we've learned before. For example, we can find out about the columns and datatypes in the `instructor` table by running `\d instructor` in psql:

```
mds=# \d instructor
                    Table "public.instructor"
   Column   |         Type          | Collation | Nullable | Default
------------+-----------------------+-----------+----------+---------
 id         | integer               |           | not null |
 name       | text                  |           |          |
 email      | text                  |           |          |
 phone      | character varying(12) |           |          |
 department | character varying(50) |           |          |
Indexes:
    "instructor_pkey" PRIMARY KEY, btree (id)
```

INSERT

The `INSERT` statement is used to add new rows to a table, and can be used in three different ways:

- by column position
- by column name
- from a table

By column position

```
INSERT INTO
    table_name
VALUES
    (value1, value2, ...)
;
```

Skip to main content

- The order of values should be the same as the order of columns in the table.
- It's not mandatory to provided a value for every column in the table, unless they are explicitly set as **non-nullable**. This is a Postgres extension; in other RDBMSs you might need to provide a value for every column using this syntax.

For example, let's add two new instructor to our `instructor` table in the `mds` database:

```
%%sql

INSERT INTO
    instructor
VALUES
    (78, 'Rachel', 'rachel@cs.ubc.ca', '766-442-9059', 'Computer Science')
;
```

```
 * postgresql://postgres:***@localhost:5432/mds
1 rows affected.
```

```
[]
```

```
%%sql

INSERT INTO
    instructor
VALUES
    (79, 'Anthony', 'anthony@gmail')
;
```

```
 * postgresql://postgres:***@localhost:5432/mds
1 rows affected.
```

```
[]
```

```
%sql SELECT * FROM instructor;
```

```
 * postgresql://postgres:***@localhost:5432/mds
13 rows affected.
```

Skip to main content

id	name	email	phone	department
1	Mike	mike@mds.ubc.ca	605-332-2343	Computer Science
2	Tiffany	tiff@mds.ubc.ca	445-794-2233	Neuroscience
3	Arman	arman@mds.ubc.ca	935-738-5796	Physics
4	Varada	varada@mds.ubc.ca	243-924-4446	Computer Science
5	Quan	quan@mds.ubc.ca	644-818-0254	Economics
6	Joel	joel@mds.ubc.ca	773-432-7669	Biomedical Engineering
7	Florencia	flor@mds.ubc.ca	773-926-2837	Biology
8	Alexi	alexiu@mds.ubc.ca	421-888-4550	Statistics
15	Vincenzo	vincenzo@mds.ubc.ca	776-543-1212	Statistics
19	Gittu	gittu@mds.ubc.ca	776-334-1132	Biomedical Engineering
16	Jessica	jessica@mds.ubc.ca	211-990-1762	Computer Science
78	Rachel	rachel@cs.ubc.ca	766-442-9059	Computer Science
79	Anthony	anthony@gmail	None	None

> Remember that there is no guarantee that the new row shows up as the last row in our table.

> Columns that are not given a value in the insert statement will either be set to their default value (if they have one) or null.

Skip to main content

By column name

```
INSERT INTO
    table_name(col1, col2, ...)
VALUES
    (value1, value2, ...)
;
```

- A value should be provided for every listed column, but it's not mandatory to provide values for all columns in the table (unless they are primary keys for the table with no default value—more about this later in this lecture)
- The column names and their values can appear in any order with this syntax

For example, here I'll add another row to the `instructor` table using the syntax above, with a shuffled column order:

```
%%sql

INSERT INTO
    instructor(department, name, email, id)
VALUES
    ('Mathematics', 'Carl', 'carl@math.ubc.ca', 65)
;
```

```
 * postgresql://postgres:***@localhost:5432/mds
1 rows affected.
```

```
[]
```

```
%sql SELECT * FROM instructor;
```

```
 * postgresql://postgres:***@localhost:5432/mds
14 rows affected.
```

Skip to main content

id	name	email	phone	department
1	Mike	mike@mds.ubc.ca	605-332-2343	Computer Science
2	Tiffany	tiff@mds.ubc.ca	445-794-2233	Neuroscience
3	Arman	arman@mds.ubc.ca	935-738-5796	Physics
4	Varada	varada@mds.ubc.ca	243-924-4446	Computer Science
5	Quan	quan@mds.ubc.ca	644-818-0254	Economics
6	Joel	joel@mds.ubc.ca	773-432-7669	Biomedical Engineering
7	Florencia	flor@mds.ubc.ca	773-926-2837	Biology
8	Alexi	alexiu@mds.ubc.ca	421-888-4550	Statistics
15	Vincenzo	vincenzo@mds.ubc.ca	776-543-1212	Statistics
19	Gittu	gittu@mds.ubc.ca	776-334-1132	Biomedical Engineering
16	Jessica	jessica@mds.ubc.ca	211-990-1762	Computer Science
78	Rachel	rachel@cs.ubc.ca	766-442-9059	Computer Science
79	Anthony	anthony@gmail	None	None
65	Carl	carl@math.ubc.ca	None	Mathematics

Let's also try to insert a row with only one value for the column `id` (which is required since it's a primary key):

```
%%sql

INSERT INTO
    instructor(id)
VALUES
    (999)
;
```

```
 * postgresql://postgres:***@localhost:5432/mds
1 rows affected.
```

Skip to main content

```
[]
```

```
%sql SELECT * FROM instructor;
```

```
 * postgresql://postgres:***@localhost:5432/mds
15 rows affected.
```

id	name	email	phone	department
1	Mike	mike@mds.ubc.ca	605-332-2343	Computer Science
2	Tiffany	tiff@mds.ubc.ca	445-794-2233	Neuroscience
3	Arman	arman@mds.ubc.ca	935-738-5796	Physics
4	Varada	varada@mds.ubc.ca	243-924-4446	Computer Science
5	Quan	quan@mds.ubc.ca	644-818-0254	Economics
6	Joel	joel@mds.ubc.ca	773-432-7669	Biomedical Engineering
7	Florencia	flor@mds.ubc.ca	773-926-2837	Biology
8	Alexi	alexiu@mds.ubc.ca	421-888-4550	Statistics
15	Vincenzo	vincenzo@mds.ubc.ca	776-543-1212	Statistics
19	Gittu	gittu@mds.ubc.ca	776-334-1132	Biomedical Engineering
16	Jessica	jessica@mds.ubc.ca	211-990-1762	Computer Science
78	Rachel	rachel@cs.ubc.ca	766-442-9059	Computer Science
79	Anthony	anthony@gmail	None	None
65	Carl	carl@math.ubc.ca	None	Mathematics
999	None	None	None	None

Skip to main content

Multiple rows at once

When inserting rows, we don't need to write several `INSERT INTO` statements to insert several rows. It's possible to insert multiple rows with a single `INSERT INTO` statement by separating the rows to be inserted with commas:

```
INSERT INTO
    table_name(col1, col2, ...)
VALUES
    (row1_value1, row1_value2, ...),
    (row2_value1, row2_value2, ...),
    (row3_value1, row3_value2, ...),
    (row4_value1, row4_value2, ...)
;
```

From a table

```
INSERT INTO
    table_name
    [(col1, col2, ...)]
SELECT ...
```

This `INSERT` syntax allows for reading rows from another table and inserting them into the table we want. In terms of column order and default values, it works the same as the previous two methods.

Let's assume that we had another table called `visiting_instructor` in our `mds` database, with the following rows:

```
%%sql

CREATE TABLE visiting_instructor (
    id INTEGER PRIMARY KEY,
    name TEXT,
    email TEXT
    )
.
```

Skip to main content

```
    visiting_instructor (id, name, email)
VALUES
    (501, 'Oliver', 'oliver@gmail.com'),
    (502, 'Adriana', 'adriana@gmail.com')
;
```

```
 * postgresql://postgres:***@localhost:5432/mds
Done.
2 rows affected.
```

```
[]
```

```
%sql SELECT * FROM visiting_instructor;
```

```
 * postgresql://postgres:***@localhost:5432/mds
2 rows affected.
```

id	name	email
501	Oliver	oliver@gmail.com
502	Adriana	adriana@gmail.com

Recently, these two visiting instructors have accepted a permanent position in the MDS program, so we want to add them to our `instructor` table. We can do so using the following `INSERT` statement to bring rows from the `visiting_instructor` to `instructor`:

```
%%sql

INSERT INTO
    instructor(id, name, email)
SELECT
    *
FROM
    visiting_instructor
;
```

```
 * postgresql://postgres:***@localhost:5432/mds
2 rows affected.
```

Skip to main content

```
[]
```

```
%sql SELECT * FROM instructor;
```

```
 * postgresql://postgres:***@localhost:5432/mds
17 rows affected.
```

id	name	email	phone	department
1	Mike	mike@mds.ubc.ca	605-332-2343	Computer Science
2	Tiffany	tiff@mds.ubc.ca	445-794-2233	Neuroscience
3	Arman	arman@mds.ubc.ca	935-738-5796	Physics
4	Varada	varada@mds.ubc.ca	243-924-4446	Computer Science
5	Quan	quan@mds.ubc.ca	644-818-0254	Economics
6	Joel	joel@mds.ubc.ca	773-432-7669	Biomedical Engineering
7	Florencia	flor@mds.ubc.ca	773-926-2837	Biology
8	Alexi	alexiu@mds.ubc.ca	421-888-4550	Statistics
15	Vincenzo	vincenzo@mds.ubc.ca	776-543-1212	Statistics
19	Gittu	gittu@mds.ubc.ca	776-334-1132	Biomedical Engineering
16	Jessica	jessica@mds.ubc.ca	211-990-1762	Computer Science
78	Rachel	rachel@cs.ubc.ca	766-442-9059	Computer Science
79	Anthony	anthony@gmail	None	None
65	Carl	carl@math.ubc.ca	None	Mathematics
999	None	None	None	None
501	Oliver	oliver@gmail.com	None	None
502	Adriana	adriana@gmail.com	None	None

Note that:

- We can retrieve any subset of columns and rows in another table as long as they are consistent with the columns of the destination table.

Skip to main content

UPDATE

In addition to adding rows to our tables, we can also update existing ones. The standard SQL syntax for updating rows is:

```
UPDATE
    table_name
SET
    col1 = expr1,
    col2 = expr2,
    ...
WHERE
     condition
;
```

For example, let's assign our new instructors to the business school:

```
%%sql

UPDATE
    instructor
SET
    department = 'Business'
WHERE
    department IS NULL
    AND
    name IS NOT NULL
;
```

```
 * postgresql://postgres:***@localhost:5432/mds
3 rows affected.
```

```
[]
```

```
%sql SELECT * FROM instructor;
```

Skip to main content

```
 * postgresql://postgres:***@localhost:5432/mds
17 rows affected.
```

id	name	email	phone	department
1	Mike	mike@mds.ubc.ca	605-332-2343	Computer Science
2	Tiffany	tiff@mds.ubc.ca	445-794-2233	Neuroscience
3	Arman	arman@mds.ubc.ca	935-738-5796	Physics
4	Varada	varada@mds.ubc.ca	243-924-4446	Computer Science
5	Quan	quan@mds.ubc.ca	644-818-0254	Economics
6	Joel	joel@mds.ubc.ca	773-432-7669	Biomedical Engineering
7	Florencia	flor@mds.ubc.ca	773-926-2837	Biology
8	Alexi	alexiu@mds.ubc.ca	421-888-4550	Statistics
15	Vincenzo	vincenzo@mds.ubc.ca	776-543-1212	Statistics
19	Gittu	gittu@mds.ubc.ca	776-334-1132	Biomedical Engineering
16	Jessica	jessica@mds.ubc.ca	211-990-1762	Computer Science
78	Rachel	rachel@cs.ubc.ca	766-442-9059	Computer Science
65	Carl	carl@math.ubc.ca	None	Mathematics
999	None	None	None	None
79	Anthony	anthony@gmail	None	Business
501	Oliver	oliver@gmail.com	None	Business
502	Adriana	adriana@gmail.com	None	Business

Remember:

`UPDATE` is a **dangerous** statement; you might accidentally modify all or several rows in a table with a wrong search condition. It is always a good idea to use a `SELECT` statement to make sure the returned rows are actually the ones you want to update, and then modify them

Skip to main content

One other method is to create a temporary table, and use it to test your `UPDATE` statement. You'll learn how to do that later in this lecture.

DELETE

Well, there finally comes a time when some rows need to be deleted (or all rows, who knows...). The `DELETE` statement in SQL removes rows from a table. Most of the time we don't want to delete all rows but only those that meet specific conditions. Similar to `UPDATE`, `DELETE` also accepts a `WHERE` clause:

```
DELETE FROM
    table_name
WHERE
    condition
;
```

For example, I want to remove the row in the `instructor` table that has an `id` of 999:

```
%%sql

DELETE FROM
    instructor
WHERE
    id = 999
;
```

```
 * postgresql://postgres:***@localhost:5432/mds
1 rows affected.
```

```
[]
```

```
%sql SELECT * FROM instructor;
```

Skip to main content

```
 * postgresql://postgres:***@localhost:5432/mds
16 rows affected.
```

id	name	email	phone	department
1	Mike	mike@mds.ubc.ca	605-332-2343	Computer Science
2	Tiffany	tiff@mds.ubc.ca	445-794-2233	Neuroscience
3	Arman	arman@mds.ubc.ca	935-738-5796	Physics
4	Varada	varada@mds.ubc.ca	243-924-4446	Computer Science
5	Quan	quan@mds.ubc.ca	644-818-0254	Economics
6	Joel	joel@mds.ubc.ca	773-432-7669	Biomedical Engineering
7	Florencia	flor@mds.ubc.ca	773-926-2837	Biology
8	Alexi	alexiu@mds.ubc.ca	421-888-4550	Statistics
15	Vincenzo	vincenzo@mds.ubc.ca	776-543-1212	Statistics
19	Gittu	gittu@mds.ubc.ca	776-334-1132	Biomedical Engineering
16	Jessica	jessica@mds.ubc.ca	211-990-1762	Computer Science
78	Rachel	rachel@cs.ubc.ca	766-442-9059	Computer Science
65	Carl	carl@math.ubc.ca	None	Mathematics
79	Anthony	anthony@gmail	None	Business
501	Oliver	oliver@gmail.com	None	Business
502	Adriana	adriana@gmail.com	None	Business

Also, I want to remove the newly hired instructors (which we copied into the `instructor` table) from the `visiting_instructor` table. Since I want to delete all rows in that table, I can write the `DELETE` statement without a `WHERE` condition:

```
%sql DELETE FROM visiting_instructor;
```

```
 * postgresql://postgres:***@localhost:5432/mds
2 rows affected.
```

```
[]
```

Skip to main content

```
%sql SELECT * FROM visiting_instructor;
```

```
 * postgresql://postgres:***@localhost:5432/mds
0 rows affected.
```

id	name	email

Note that although the `DELETE` statement has removed all rows, **the table structure is intact**. In other words, the table is still there but stores no rows at the moment.

`TRUNCATE`

If the goal is to remove all rows from a table, popular RDBMSs including Postgres also support the `TRUNCATE` statement with the following syntax:

```
TRUNCATE TABLE table_name;
```

- `TRUNCATE` is **faster and more efficient** in terms of the RDBMS resources, because it does not scan every row as opposed to `DELETE`.
- During the process, `DELETE` logs the changes made to each and every row, whereas `TRUNCATE` treats the deleting of all rows as a single operation. Even though `TRUNCATE` locks the entire table during the deleting operation (as opposed to `DELETE` which locks individual rows), it is still a better choice to delete all rows because it will release the lock much sooner.

Remember:

Skip to main content

`DELETE` and `TRUNCATE` are both very dangerous statements, and you should use them with extreme caution in real-life databases. With `DELETE`, you should always try a `SELECT` query first to see if your `WHERE` condition returns the rows that you want, and then use `DELETE` to remove them.

One other method is to create a temporary table to test your `DELETE` statement. You'll learn how to do that later in this lecture.

`RETURNING` (OPTIONAL)

The table modifying commands `INSERT`, `UPDATE`, and `DELETE` all accept a `RETURNING` clause which returns the rows that have been modified by these commands. The returning clause can be helpful in reliably identifying the rows that have been modified, without having to run a separate `SELECT` statement after table modification. Here is the syntax for `RETURNING` used along with `INSERT`:

```
INSERT INTO
    table1(col1, col2, ...)
VALUES
    (val1, val2, ...)
RETURNING
    *
;
```

We can select any one of the columns of the modified rows just as in a `SELECT` statement.

Here is an example of returning rows from an `UPDATE` statement:

```
%%sql

UPDATE
    instructor
SET
    department = 'STATS'
WHERE
```

Skip to main content

```
    *
;
```

```
 * postgresql://postgres:***@localhost:5432/mds
2 rows affected.
```

id	name	email	phone	department
8	Alexi	alexiu@mds.ubc.ca	421-888-4550	STATS
15	Vincenzo	vincenzo@mds.ubc.ca	776-543-1212	STATS

We can check that the above two rows in the `instructor` table are actually the ones that are updated:

```
%sql SELECT * FROM instructor;
```

```
 * postgresql://postgres:***@localhost:5432/mds
16 rows affected.
```

Skip to main content

id	name	email	phone	department
1	Mike	mike@mds.ubc.ca	605-332-2343	Computer Science
2	Tiffany	tiff@mds.ubc.ca	445-794-2233	Neuroscience
3	Arman	arman@mds.ubc.ca	935-738-5796	Physics
4	Varada	varada@mds.ubc.ca	243-924-4446	Computer Science
5	Quan	quan@mds.ubc.ca	644-818-0254	Economics
6	Joel	joel@mds.ubc.ca	773-432-7669	Biomedical Engineering
7	Florencia	flor@mds.ubc.ca	773-926-2837	Biology
19	Gittu	gittu@mds.ubc.ca	776-334-1132	Biomedical Engineering
16	Jessica	jessica@mds.ubc.ca	211-990-1762	Computer Science
78	Rachel	rachel@cs.ubc.ca	766-442-9059	Computer Science
65	Carl	carl@math.ubc.ca	None	Mathematics
79	Anthony	anthony@gmail	None	Business
501	Oliver	oliver@gmail.com	None	Business
502	Adriana	adriana@gmail.com	None	Business
8	Alexi	alexiu@mds.ubc.ca	421-888-4550	STATS
15	Vincenzo	vincenzo@mds.ubc.ca	776-543-1212	STATS

Question: Did you notice what happened to the location of the two updated rows? Do you recall what's significant about this?

Creating, modifying, and dropping tables

In this section, we'll learn about the basics of creating tables from scratch, modifying and also dropping them.

- Before moving on to discussing the ways to do this, I need to remind you that designing a

Skip to main content

knowledge and experience, and sometimes many design iterations.

- The goal here is for you to get a basic knowledge of how tables are defined, altered and dropped and how constraints are enforced in relatively simple cases.
- Even if you're not going to design a database or create tables yourself, an understanding of what the process looks is beneficial in how you think about a database in general and how you use it.

Creating tables

The general syntax for creating a table is as follows:

```
CREATE TABLE table_name (
   column1    datatype [column_contraint],
   column2    datatype [column_contraint],
   column3    datatype [column_contraint],
   [table_constraints]
);
```

In order to define a table, we need:

- column names
- column data types

You may optionally also define

- default values
- constraints

Suppose that in a database for a sample online store, we have a table to keep information about our customers. We can create a table called `customer` with the following statement:

Skip to main content

```
CREATE TABLE customer
(
    customer_id    INTEGER,
    title          CHAR(4),
    fname          VARCHAR(32),
    lname          VARCHAR(32),
    addressline    VARCHAR(64),
    town           VARCHAR(32),
    zipcode        CHAR(10),
    phone          VARCHAR(16)
);
```

Integrity constraints

We have seen so far that the particular data type of a column places a constraint on the kind of values that column stores (known as **domain constraints**).

Constraints of this kind, collectively known as **integrity constraints**, ensure that the right data is stored in the right format in the database at all times, such that all design requirements are met. For example, we might want to not accept `NULL` values for a particular column, or explicitly specify the range of values for another column.

Constraints can be imposed either on the **column** level (which apply only to a single column), or on the **table** level (which may apply to one or multiple columns).

- A column constraint immediately follows the column data type in a `CREATE TABLE` statement
- A table constraint is added after the definition of the last column of a table, and optionally starts with the keyword `CONSTRAINT` followed by a constraint name, and the constraint definition.

> The existence of integrity constraints is one of the main differences between storing data in flat data files (e.g. CSV files) and using DBMSs to store and manage data.

Skip to main content

`PRIMARY KEY`

A **key** is one, or a set of attributes that **uniquely** identifies rows in a table. A key that is comprised of one attribute is called a **simple** key, whereas a key comprised of multiple attributes is called a **composite** key.

In general, a **key constraint** is a statement to specify that one or a set of attributes of a table qualify as being keys for that table, i.e. they **uniquely** identify rows in the table.

Candidate key: Any attribute, or set of attributes that is unique and doesn't contain any redundant attributes is a candidate key.

Primary key: A primary key is one of the candidate keys selected by the database designer to be the identifying key for the rows in a table. The primary key needs to be **minimal**, meaning that out of all candidate keys, the primary key must be one that involves the least number of attributes.

Remember that:

- The primary key constraint automatically enforces **uniqueness** and **non-nullability**.
- A table can only have one primary key.

In order to set a simple primary key constraint, we can append `PRIMARY KEY` the definition of a column:

```
CREATE TABLE table1
```

Skip to main content

```
    col1    datatype [CONSTRAINT constraint_name] PRIMARY KEY,
    col2    datatype,
    .
    .
    .
);
```

However, to set a composite primary key constraint, we need to define it as a table-level constraint:

```
CREATE TABLE table1
(
    col1    datatype,
    col2    datatype,
    .
    .
    .
    [CONSTRAINT constraint_name] PRIMARY KEY (col1, col2, ...)
);
```

For example, we can specify the primary key constraint for the customer table in the following ways:

```
CREATE TABLE customer
(
    customer_id    INTEGER PRIMARY KEY,
    .
    .
    .
);
```

or

```
CREATE TABLE customer
(
    customer_id    INTEGER,
    .
    .
    .
    PRIMARY KEY (customer id)
```

Skip to main content

The `SERIAL` pseudo data type

We have seen `SERIAL` as a data type before, but haven't talked about it yet. `SERIAL` is the Postgres equivalent of a generator in Python: it generates an auto-incrementing sequence of integer values starting from 1. Since it is used in place of a data type when creating tables, it's also called a pseudo data type.

This data type is usually used for a primary key column to automatically create incrementing values each time a new row is inserted. It is also possible to create custom sequences in Postgres that start from integers other than 1, and have custom step sizes other than 1.

Here is an example of how the `SERIAL` type works in a sample table:

```
%%sql

CREATE TABLE demo_serial(
    id SERIAL,
    name VARCHAR(20)
);
```

```
 * postgresql://postgres:***@localhost:5432/mds
(psycopg2.errors.DuplicateTable) relation "demo_serial" already exists

[SQL: CREATE TABLE demo_serial(
    id SERIAL,
    name VARCHAR(20)
);]
(Background on this error at: https://sqlalche.me/e/14/f405)
```

```
%sql INSERT INTO demo_serial (name) VALUES ('Arman'), ('Mike'), ('Tiffany');
```

```
 * postgresql://postgres:***@localhost:5432/mds
3 rows affected.
```

Skip to main content

```
%sql SELECT * FROM demo_serial;
```

```
 * postgresql://postgres:***@localhost:5432/mds
6 rows affected.
```

id	name
1	Arman
2	Mike
3	Tiffany
4	Arman
5	Mike
6	Tiffany

```
%sql DROP TABLE demo_serial;
```

```
 * postgresql://postgres:***@localhost:5432/mds
Done.
```

```
[]
```

`FOREIGN KEY`

We've mentioned before when we talked about joins in the previous lecture that the main idea of the relational model is that data is split across multiple tables. We made use of this relatedness to answer more complex and more informative queries by connecting and associating rows from different tables together.

In a properly designed database, there must be ways to ensure that related rows from

Skip to main content

no use if the values in joining columns do not correctly correspond to each other. Foreign key constraints provide a mechanism through which we can enforce **referential integrity**.

A foreign key is a column or multiple columns that reference values of candidate column(s) in another table.

Child table: The table containing the foreign key

Parent table: The table which a foreign key in another table references

Just like primary keys, there are two ways to define foreign key constraint. On the column level:

```
CREATE TABLE table2
(
    col1    datatype
        [CONSTRAINT constraint_name] PRIMARY KEY,
    col2    datatype
        [CONSTRAINT constraint_name] REFERENCES table1(col1),
    .
    .
    .
);
```

or

```
CREATE TABLE table2
(
    col1    datatype [CONSTRAINT constraint_name] PRIMARY KEY,
    col2    datatype,
    .
    .
    .
    [CONSTRAINT constraint_name] FOREIGN KEY (col1, col2)
        REFERENCES table1(col1, col2)
);
```

For example, we can create a new table called `order_info`

Skip to main content

```
CREATE TABLE orderinfo
(
    orderinfo_id    INTEGER,
    customer_id     INTEGER,
    .
    .
    .
    CONSTRAINT orderinfo_pk
        PRIMARY KEY(orderinfo_id),

    CONSTRAINT orderinfo_customer_id_fk
        FOREIGN KEY(customer_id) REFERENCES customer(customer_id)
);
```

or alternatively:

```
CREATE TABLE orderinfo
(
    orderinfo_id    INTEGER,
    customer_id     INTEGER
        CONSTRAINT orderinfo_customer_id_fk
            REFERENCES customer(customer_id),
    .
    .
    .
    CONSTRAINT orderinfo_pk
        PRIMARY KEY(orderinfo_id),
);
```

A foreign key constraint basically ensures that there are no orphan rows in a child table. In other words, every foreign key value in the child table MUST also exist in the referenced candidate key of the parent table. Any change in either of the tables that violates this principle would result in an error by the RDBMS.

For example, these changes would trigger a foreign key constraint check:

- Inserting/updating a new row in the child table (possibility of creating a new orphan)
- Deleting/updating a row in the parent table (possibility of making existing children orphan)

Skip to main content

- Inserting a new row in the parent table
- Deleting a row in the child table

Remember that foreign key values can be **null** or **duplicate**. A common example of nulls occurring in foreign keys is when the parent row is deleted, and we don't necessarily want to lose the child row.

> According to the SQL standard, foreign keys must reference either the primary or unique key of another table. While this requirement should always be respected, it is not strictly enforced in most DBMSs.

Referential actions

As we've seen so far, deleting or updating a row in a parent table might be troublesome, because it can leave rows in child tables bewildered with broken references.

Referential integrity constraints prevent users from accidentally making inconsistent changes. But SQL also provides us with more options in regards to what to do when inconsistencies occur, beyond just showing an error message. These options are called **referential actions**.

There are two SQL clauses that allow us to implement referential actions: `ON UPDATE` and `ON DELETE` followed any one of these actions:

- `NO ACTION`: If a user tries to update or delete a row in the parent table that is referenced by one or more rows in the child table, the database system will reject the operation and generate an error. (default behaviour)
- `CASCADE`: Applies the same change to the child row
  - `ON UPDATE`: Updates the child foreign key with the new parent key value
  - `ON DELETE`: Deletes the child row as well
- `SET NULL`: Sets the foreign key value to `NULL`
- `SET DEFAULT`: Sets the foreign key value to its default value

Skip to main content

> Remember that referential actions are added in the definition of the **child** table.

Example:

```
CREATE TABLE orderinfo
(
    orderinfo_id    INTEGER PRIMARY KEY,
    customer_id     INTEGER,
    .
    .
    .
    CONSTRAINT orderinfo_customer_id_fk
        FOREIGN KEY(customer_id) REFERENCES customer(customer_id)
        ON DELETE CASCADE
);
```

`UNIQUE`

A `UNIQUE` constraint checks whether the value of a column, or a combination of columns are unique, and prevents insertion or update if not.

```
CREATE TABLE table2
(
    col1    datatype
        [CONSTRAINT constraint_name] PRIMARY KEY,
    col2    datatype
        [CONSTRAINT constraint_name] REFERENCES table1(col1),
    col3    datatype
        [CONSTRAINT constraint_name] UNIQUE
    col4    datatype
    col5    datatype
    .
    .
    .
    [CONSTRAINT constraint_name] UNIQUE (col4, col5)
);
```

Skip to main content

`NOT NULL`

This constraint forbids a column value to hold a value of null. `NOT NULL` can only be applied on the column level:

```
CREATE TABLE table1
(
    col1    VARCHAR(5) NOT NULL,
    col2    INTEGER
);
```

`DEFAULT`

You might remember from the previous sections that if we insert new rows without specifying the value of columns, Postgres automatically assigns them a null value (unless the have a `NOT NULL` constraint that we just learned about). This happens because the default value for all columns is null, unless specified otherwise. We can set our own default values for columns of a table when we define a table using the `DEFAULT` keyword:

```
CREATE TABLE table1
(
    col1    VARCHAR(5) NOT NULL,
    col2    REAL DEFAULT 0.0,
    col3    TEXT DEFAULT 'N/A',
    col4    TIMESTAMP DEFAULT NOW()
);
```

Note that:

- The expression in front of `DEFAULT` should always evaluate to a constant
- The default value applies only on the column level, just like `NOT NULL`

Skip to main content

`CHECK`

While specifying data types for columns imposes an overall constraint on the kind of allowable values, there are situations in which we want to have more control on column domains. For example, we may want to not accept negative integers as values for a particular column. There is no standard data type that provides us this level of control. Fortunately, we can use the `CHECK` constraint to limit the values acceptable for a column or a group of columns:

```
CREATE TABLE table2
(
    col1    datatype
        [CONSTRAINT constraint_name] PRIMARY KEY,
    col2    datatype
        [CONSTRAINT constraint_name] CHECK (condition),
    .
    .
    .
    [CONSTRAINT constraint_name] CHECK (condition)
);
```

The condition in a `CHECK` constraint can be any condition typically used in a `WHERE` clause.

In the following example, we'd like to check if the value entered for the `phone` column is a valid phone number with the right format, e.g. 123-456-7891

```
CREATE TABLE customer
(
    customer_id    INTEGER,
    fname          VARCHAR(32),
    lname          VARCHAR(32),
    .
    .
    .
    phone          CHAR(12) CHECK (phone LIKE '___-___-____')
    .
    .
    .
);
```

Skip to main content

Temporary tables

In the previous sections of this lecture, we learned about how we can update or delete existing rows, and that these could be potentially dangerous operations. One useful way to avoid any unintentional changes in your existing tables is use **temporary tables** option provided by SQL, defined using the following syntax:

```
CREATE TEMPORARY TABLE table_name (
    column1    datatype [column_contraint],
    .
    .
    .
   [table_constraints]
);
```

The most important thing to remember is that **temporary tables only persist for the duration of a *session***. Temporary tables may also be used for storing the results of a complex query to save time and resources.

With the above syntax, the temporary table is created and then we need to populate it with appropriate `INSERT` statements.

There is also a way to create a temporary or actually even regular tables from other tables with the following syntax:

```
CREATE [TEMPORARY] TABLE table_name AS
    query
);
```

The newly created table can be treated just like any other regular table.

Skip to main content

> Remember that table creation and modification operations require special privileges that the database admin should have granted to you so that you can perform these operations.

Let's put our new temporary table concept to practice:

```
%%sql

CREATE TEMPORARY TABLE
    temp_instructor
AS
    SELECT name, department FROM instructor
;
```

```
 * postgresql://postgres:***@localhost:5432/mds
16 rows affected.
```

```
[]
```

```
%sql SELECT * FROM temp_instructor;
```

```
 * postgresql://postgres:***@localhost:5432/mds
16 rows affected.
```

Skip to main content

name	department
Mike	Computer Science
Tiffany	Neuroscience
Arman	Physics
Varada	Computer Science
Quan	Economics
Joel	Biomedical Engineering
Florencia	Biology
Gittu	Biomedical Engineering
Jessica	Computer Science
Rachel	Computer Science
Carl	Mathematics
Anthony	Business
Oliver	Business
Adriana	Business
Alexi	STATS
Vincenzo	STATS

We can see that the temporary table exists in this current database session that we have in Jupyter lab:

```
%%sql

SELECT
    table_name
FROM
    information_schema.tables
WHERE
    table_schema ~ 'public|temp'
;
```

```
 * postgresql://postgres:***@localhost:5432/mds
5 rows affected.
```

Skip to main content

table_name
instructor
instructor_course
course_cohort
visiting_instructor
temp_instructor

But if you check the `mds` database at this moment through pgAdmin or `psql`, you will not find the table `temp_instructor`.

This is because pgAdmin has its own independent connection to the database, and is unaware of a temporary table created in another database session.

If you restart the kernel in this notebook, the current Postgres session will be ended and the `temp_instructor` table will be gone!

With `CREATE TABLE table_name AS` syntax, any valid `SELECT` statement can be used to construct a new table. For example, you can use `WHERE`, `GROUP BY`, and any type of join between multiple tables.

(OPTIONAL) Modifying tables

Ideally, it is best if our tables stay the way they are once they are created. In other words, we don't want to change anything about structure of the tables that we've created in our database, because this would probably cause a host of problems.

It's not hard to imagine that many queries would likely need to be restructured, expressed differently or rewritten entirely. But nothing in life is perfect, neither are our decisions about the original design of a database.

Fortunately, standard SQL provides the `ALTER TABLE` command to

Skip to main content

- change column data types
- add, drop, or change constraints
- rename tables

Here is syntax for `ALTER TABLE`:

```
ALTER TABLE table_name
    action
;
```

`action` in the above syntax could be any table- or column-altering action, including:

```
ADD COLUMN datatype [constraints]

ALTER COLUMN column SET DEFAULT value

DROP COLUMN column [CASCADE]

ADD table_constraint

DROP CONSTRAINT constraint_name
```

For a comprehensive list of altering actions, see Postgres documentation here.

Let's take a look at an example: suppose that we no longer need the `phone` column in the `instructor` table:

```
%sql SELECT * FROM instructor LIMIT 5;
```

```
 * postgresql://postgres:***@localhost:5432/mds
5 rows affected.
```

Skip to main content

id	name	email	phone	department
1	Mike	mike@mds.ubc.ca	605-332-2343	Computer Science
2	Tiffany	tiff@mds.ubc.ca	445-794-2233	Neuroscience
3	Arman	arman@mds.ubc.ca	935-738-5796	Physics
4	Varada	varada@mds.ubc.ca	243-924-4446	Computer Science
5	Quan	quan@mds.ubc.ca	644-818-0254	Economics

```
%%sql

ALTER TABLE
    instructor
DROP COLUMN
    phone
;
```

```
 * postgresql://postgres:***@localhost:5432/mds
Done.
```

```
[]
```

```
%sql SELECT * FROM instructor LIMIT 5;
```

```
 * postgresql://postgres:***@localhost:5432/mds
5 rows affected.
```

id	name	email	department
1	Mike	mike@mds.ubc.ca	Computer Science
2	Tiffany	tiff@mds.ubc.ca	Neuroscience
3	Arman	arman@mds.ubc.ca	Physics
4	Varada	varada@mds.ubc.ca	Computer Science
5	Quan	quan@mds.ubc.ca	Economics

It worked! Now let's add a constraint to the `email` column to require that an email address

Skip to main content

```
%%sql

ALTER TABLE
    instructor
ADD CONSTRAINT
    constraint_email CHECK (email LIKE '%@%')
```

```
 * postgresql://postgres:***@localhost:5432/mds
Done.
```

```
[]
```

The constraint name (along with the keyword `CONSTRAINT`) are optional. You can check that this constraint has been added to the table by running the following command in `psql`:

```
\d instructor
```

Also note that if you add a constraint with which some of the rows are in violation of, Postgres will throw an error message and the constraint will not be added. In order to find which rows violate adding a `CHECK` constraint, we can use the following query:

```
SELECT
    *
FROM
    table
WHERE
    NOT constraint_to_be_added
```

Here's another example: Right now there are some courses in the `instructor_course`, the `instructor_id`s of which do not exist in the `instructor` table. We've learned that this violates **referential integrity** of our database.

In order to avoid this problem in the future, we want to require that every instructor referenced in the `instructor course` table also be present in the `instructor` table, or

Skip to main content

have the value of `NULL` for the `instructor_id` field. We can achieve this be defining a foreign key constraint on the `instructor_course` table using the following commands:

```
ALTER TABLE
    instructor_course
ADD CONSTRAINT fk_instructor
    FOREIGN KEY (instructor_id)
    REFERENCES instructor(id)
    ON DELETE SET NULL
    ON UPDATE CASCADE
;
```

Note that I have also set referential integrity actions such that if an instructor is deleted from the `instructor` table, `instructor_id` becomes `NULL`, and if the `id` field of the `instructor` table is updated for some reason, the change cascades down to the `instructor_course` table to avoid inconsistency.

There is one little problem with the above altering commands though: running the above commands would result in an error, because there are some *orphan* courses in the `instructor_course` table, and Postgres doesn't know what to do with them. We can circumvent this problem by setting the `instructor_id` field to `NULL` for courses without instructors that are currently present in the `instructor_course` table:

```
%%sql

UPDATE
    instructor_course
SET
    instructor_id = NULL
WHERE
    instructor_id IN (9, 10, 12, 13)
;
```

```
 * postgresql://postgres:***@localhost:5432/mds
4 rows affected.
```

```
[]
```

```
%%sql
```

Skip to main content

```
    instructor_course
ADD CONSTRAINT fk_instructor
    FOREIGN KEY (instructor_id)
    REFERENCES instructor(id)
    ON DELETE SET NULL
    ON UPDATE CASCADE
;
```

```
 * postgresql://postgres:***@localhost:5432/mds
Done.
```

```
[]
```

We can confirm the addition of the foreign key constraint by running the following command in `psql`:

```
\d instructor_course
```

As an example for referential actions, take a look at courses taught by instructor with `id = 3` in the `instructor_course` table:

```
%sql SELECT * FROM instructor_course
```

```
 * postgresql://postgres:***@localhost:5432/mds
16 rows affected.
```

Skip to main content

id	instructor_id	course	enrollment	begins
1	8	Statistical Inference and Computation I	125	2021-10-01
2	8	Regression II	102	2022-02-05
3	1	Descriptive Statistics and Probability	79	2021-09-10
4	1	Algorithms and Data Structures	25	2021-10-01
5	3	Algorithms and Data Structures	25	2021-10-01
6	3	Python Programming	133	2021-09-07
7	3	Databases & Data Retrieval	118	2021-11-16
8	6	Visualization I	155	2021-10-01
9	6	Privacy, Ethics & Security	148	2022-03-01
10	2	Programming for Data Manipulation	160	2021-09-08
11	7	Data Science Workflows	98	2021-09-15
12	2	Data Science Workflows	98	2021-09-15
13	None	Web & Cloud Computing	78	2022-02-10
14	None	Introduction to Optimization	None	2022-09-01
15	None	Parallel Computing	None	2023-01-10
16	None	Natural Language Processing	None	2023-09-10

Now let's see who that person is:

```
%sql SELECT * FROM instructor WHERE id = 3;
```

```
 * postgresql://postgres:***@localhost:5432/mds
1 rows affected.
```

id	name	email	department
3	Arman	arman@mds.ubc.ca	Physics

Suppose that the id of this person changes to `100`:

```
%%sql
```

Skip to main content

```
    instructor
SET
    id = 100
WHERE
    name = 'Arman'
;
```

```
 * postgresql://postgres:***@localhost:5432/mds
1 rows affected.
```

```
[]
```

We expect to see that this change is cascaded down to the `instructor_course` table according to the foreign key constraint that we defined earlier:

```
%sql SELECT * FROM instructor_course
```

```
 * postgresql://postgres:***@localhost:5432/mds
16 rows affected.
```

Skip to main content

id	instructor_id	course	enrollment	begins
1	8	Statistical Inference and Computation I	125	2021-10-01
2	8	Regression II	102	2022-02-05
3	1	Descriptive Statistics and Probability	79	2021-09-10
4	1	Algorithms and Data Structures	25	2021-10-01
8	6	Visualization I	155	2021-10-01
9	6	Privacy, Ethics & Security	148	2022-03-01
10	2	Programming for Data Manipulation	160	2021-09-08
11	7	Data Science Workflows	98	2021-09-15
12	2	Data Science Workflows	98	2021-09-15
13	None	Web & Cloud Computing	78	2022-02-10
14	None	Introduction to Optimization	None	2022-09-01
15	None	Parallel Computing	None	2023-01-10
16	None	Natural Language Processing	None	2023-09-10
5	100	Algorithms and Data Structures	25	2021-10-01
6	100	Python Programming	133	2021-09-07
7	100	Databases & Data Retrieval	118	2021-11-16

Great! Let's test the `ON DELETE SET NULL` clause too:

```
%%sql

DELETE FROM
    instructor
WHERE
    name = 'Arman'
;
```

```
 * postgresql://postgres:***@localhost:5432/mds
1 rows affected.
```

```
[]
```

Skip to main content

```
%sql SELECT * FROM instructor_course
```

```
 * postgresql://postgres:***@localhost:5432/mds
16 rows affected.
```

id	instructor_id	course	enrollment	begins
1	8	Statistical Inference and Computation I	125	2021-10-01
2	8	Regression II	102	2022-02-05
3	1	Descriptive Statistics and Probability	79	2021-09-10
4	1	Algorithms and Data Structures	25	2021-10-01
8	6	Visualization I	155	2021-10-01
9	6	Privacy, Ethics & Security	148	2022-03-01
10	2	Programming for Data Manipulation	160	2021-09-08
11	7	Data Science Workflows	98	2021-09-15
12	2	Data Science Workflows	98	2021-09-15
13	None	Web & Cloud Computing	78	2022-02-10
14	None	Introduction to Optimization	None	2022-09-01
15	None	Parallel Computing	None	2023-01-10
16	None	Natural Language Processing	None	2023-09-10
5	None	Algorithms and Data Structures	25	2021-10-01
6	None	Python Programming	133	2021-09-07
7	None	Databases & Data Retrieval	118	2021-11-16

Another example is to add a foreign key constraint on the `countrycode` of the `city` table in the `world` database that references the `code` column of the `country` table with the following command:

Skip to main content

```
ADD CONSTRAINT country_code_fk
        FOREIGN KEY (countrycode)
            REFERENCES country(code);
```

Dropping tables

And finally, there comes a (sad) time that we no longer need one of our tables! A table can be dropped using the keyword `DROP` as follows:

```
DROP TABLE [IF EXISTS] table1, table2;
```

Remember that:

- Dropping a table removes the entire table structure and everything associated with it. A dropped table is NOT a table with zero rows, it simply does not exist in the database anymore.
- If you need to delete all rows from a table but keep its structure, you should use `TRUNCATE`.
- Postgres does not allow you to drop a table that is referenced by another table in a foreign key. You should either drop the foreign key constraint from the child table first, or append the the keyword `CASCADE` to your `DROP` clause, so that the parent table can be dropped along with any constraints that depend on it:

```
DROP TABLE parent_table CASCADE;
```

As an example, let's drop the `visiting_instructor` table.

Here is the list of tables at this moment:

```
%%sql

SELECT
```

Skip to main content

```
    information_schema.tables
WHERE
    table_schema ~ 'public'
;
```

```
 * postgresql://postgres:***@localhost:5432/mds
4 rows affected.
```

table_name
course_cohort
visiting_instructor
instructor
instructor_course

Now let's drop the table:

```
%sql DROP TABLE visiting_instructor;
```

```
 * postgresql://postgres:***@localhost:5432/mds
Done.
```

```
[]
```

Retrieve the list of tables again:

```
%%sql

SELECT
    table_name
FROM
    information_schema.tables
WHERE
    table_schema ~ 'public'
;
```

```
 * postgresql://postgres:***@localhost:5432/mds
3 rows affected.
```

Skip to main content

table_name
course_cohort
instructor
instructor_course

If we try to drop the `instructor` table, however:

```
DROP TABLE instructor;
```

we'll get the following error, **because the `instructor` table is now referenced by the `instructor_course` table through a foreign key**:

```
DETAIL:  constraint fk_instructor on table instructor_course depends on table i
HINT:  Use DROP ... CASCADE to drop the dependent objects too.
```

So, we should use the `CASCADE` keyword:

```
%sql DROP TABLE instructor CASCADE;
```

```
 * postgresql://postgres:***@localhost:5432/mds
Done.
```

```
[]
```

We can now make sure that the foreign key constraint referencing the `instructor` table is gone (**but not the `instructor_course` table itself**) using the following command in `psql`:

```
\d instructor_course
```

It is also possible to drop multiple tables at once:

```
%sql DROP TABLE instructor_course, course_cohort, temp_instructor;
```

Skip to main content

```
 * postgresql://postgres:***@localhost:5432/mds
Done.
```

```
[]
```

And check out the list of tables to confirm:

```
%%sql

SELECT
    table_name
FROM
    information_schema.tables
WHERE
    table_schema ~ 'public'
;
```

```
 * postgresql://postgres:***@localhost:5432/mds
0 rows affected.
```

table_name