Digital Career Institute

Python Course - Database - Basic Usage

Digital Career Institute

Goal of the Submodule

The goal of this submodule is to help the learners use databases in Python. By the end of this submodule, the learners should be able to understand how to:

- Create tables with some data types
- Create relationships between tables and perform simple queries on multiple tables.
- Define views.

Topics

- Basic column data types
- Table relationships
  - Primary and foreign keys
- Views

Columns & Data Types

PostgreSQL Data Types

PostgreSQL has a variety of data types available.

- bigint
- bigserial
- bit
- bit varying
- boolean
- box
- bytea
- character
- character varying
- cidr
- circle
- date
- double precision
- inet
- integer
- interval
- json
- jsonb
- line
- lseg
- macaddr
- macaddr8
- money
- numeric
- path
- pg_lsn
- pg_snapshot
- point
- polygon
- real
- smallint
- smallserial
- serial
- text
- time
- time with time zone
- timestamp
- timestamp with time zone
- tsquery
- tsvector
- txid_snapshot
- uuid
- xml

PostgreSQL Data Types

Digital Career Institute

In this submodule we will focus on:

Boolean Type

Numeric Types

Text Types

Values vs. No-Values

All types allow the data to be unset, with no value.

This state is named `NULL`.

Sometimes it is called *NULL value*,
but it is technically not a value.

`NULL` represents the absence of a value.

Retrieve No-Values

```
personal=# SELECT first_name
personal-# FROM friends
personal-# WHERE phone = NULL;
 first_name
------------
(0 rows)
```

```
personal=# SELECT first_name
personal-# FROM friends
personal-# WHERE phone IS NULL;
 first_name
------------
 Maria
 Karen
 Lidia
 James
(4 rows)
```

To check if a row has no value we cannot do `column = NULL` because the `=` operator works only with values.

Instead, the query must be defined as `column IS NULL`.

Define Columns Without No-Values

```
CREATE TABLE private.friends (
  first_name            varchar(20) NOT
NULL,
  last_name       varchar(50),
  phone                 varchar(12),
  age                   integer
);
```

The **NOT NULL** construct will not allow
NULL values in the column.

The Boolean Type

```
CREATE TABLE friends (
  first_name
      varchar(20),
  last_name         varchar(50),
  age
      integer,
  from_school  boolean
);
```

A boolean column will accept any of the following states:

- TRUE
- FALSE
- NULL

A **boolean** column may contain a boolean value, or no value at all. Therefore, it is a **three-state switch**.

The Boolean Type

```
UPDATE friends
SET from_school = TRUE;
UPDATE friends
SET from_school = 'yes';
UPDATE friends
SET from_school = 'on';
UPDATE friends
SET from_school = 1;
```

A boolean column may be set to `TRUE` with any of these values:

- TRUE
- yes
- on
- 1

The Boolean Type

```
UPDATE friends
SET from_school = FALSE;
UPDATE friends
SET from_school = 'no';
UPDATE friends
SET from_school = 'off';
UPDATE friends
SET from_school = 0;
```

A boolean column may be set to `FALSE` with any of these values:

- FALSE
- no
- off
- 0

The Numeric Types

There is a variety of numeric types
that can be grouped into:

Integer Types

Decimal Types

The Numeric Types: Integers

Different integer types are provided
to optimize the database.

	SMALLINT	INTEGER	BIGINT
STORAGE	2 bytes	4 bytes	8 bytes
MIN. VALUE	-32768	-2147483648	-9223372036854775808
MAX. VALUE	+32767	+2147483647	+9223372036854775807

The Numeric Types: Integers

PostgreSQL validates against each type.

```
CREATE TABLE friends (
   first_name
        varchar(20),
   last_name
        varchar(50),
   age
        smallint
);
```

```
=# INSERT INTO friends(age)
-# VALUES(50000);
ERROR:  smallint out of range
```

The Numeric Types: Serial Integers

Serial types are
auto-incrementing integers.

	SMALLSERIAL	SERIAL	BIGSERIAL
STORAGE	2 bytes	4 bytes	8 bytes
MIN. VALUE	1	1	1
MAX. VALUE	32767	2147483647	9223372036854775807

The Numeric Types: Serial Integers

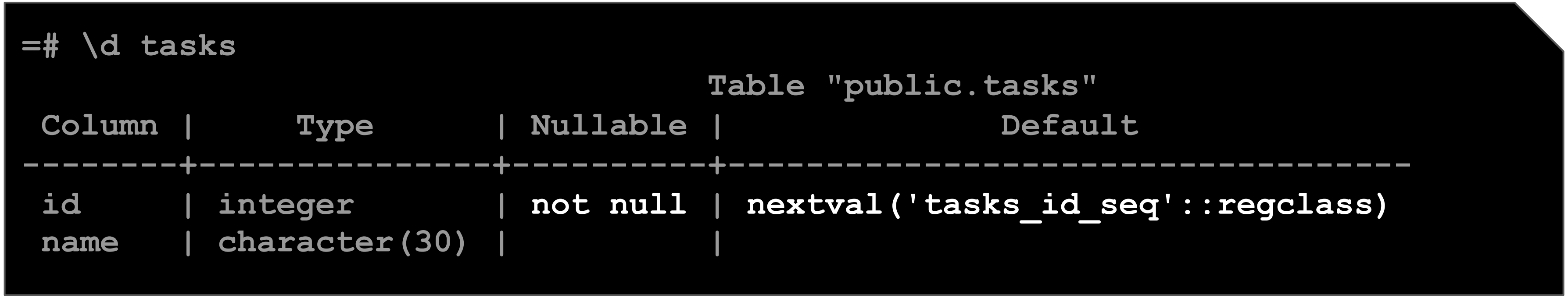
Inserting data will auto populate the serial column.

```
CREATE TABLE tasks (
  id        serial,
  name      varchar(30)
);
```

```
=# INSERT INTO tasks(name)
-# VALUES('Iron'),('Clean'),
-#       ('Study'),('Cook');
INSERT 0 4
=# SELECT * FROM tasks;
 id | name
----+-------
  1 | Iron
  2 | Clean
  3 | Study
  4 | Cook
(4 rows)
```

The Numeric Types: Serial Integers

A serial sets the column to `not null` and defines a default value.

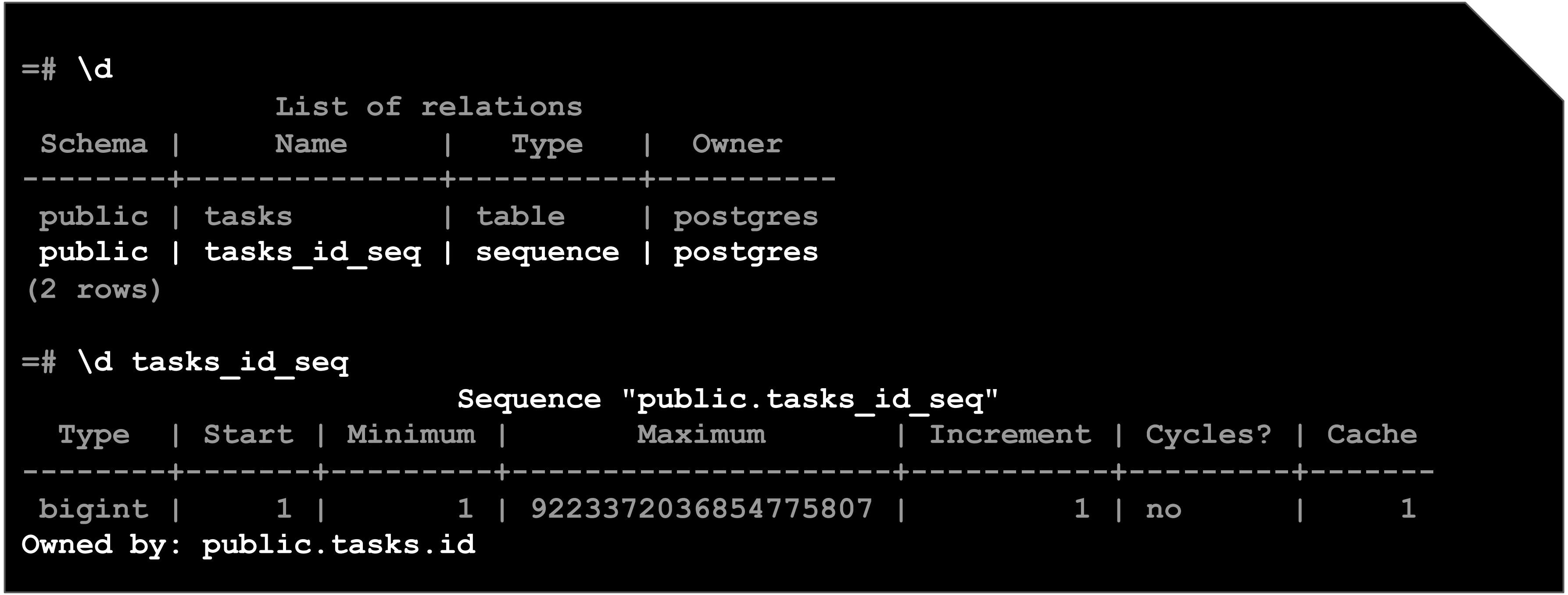


```
=# \d tasks
                                   Table "public.tasks"
 Column |     Type      | Nullable |               Default
--------+---------------+----------+------------------------------------
 id     | integer       | not null | nextval('tasks_id_seq'::regclass)
 name   | character(30) |          |
```

The default value is the next value (`nextval`) in the sequence `tasks_id_seq`.

The Numeric Types: Serial Integers

The `tasks_id_seq` relation is a sequence of type bigint.

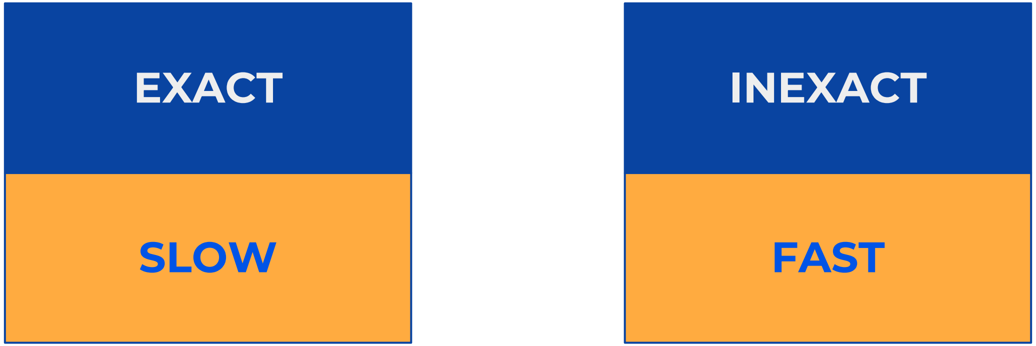


```
=# \d
             List of relations
 Schema |     Name     |   Type   |  Owner
--------+--------------+----------+----------
 public | tasks        | table    | postgres
 public | tasks_id_seq | sequence | postgres
(2 rows)

=# \d tasks_id_seq
                          Sequence "public.tasks_id_seq"
  Type  | Start | Minimum |       Maximum       | Increment | Cycles? | Cache
--------+-------+---------+---------------------+-----------+---------+-------
 bigint |     1 |       1 | 9223372036854775807 |         1 | no      |     1
Owned by: public.tasks.id
```

The Numeric Types: Decimals

Decimal types can be divided into **exact** and **inexact** decimals.



EXACT

SLOW

INEXACT

FAST

Exact types produce exact results when used in calculations.

The Numeric Types: Exact Decimals

There are two exact types, but they are equivalent.

DECIMAL = **NUMERIC**

The Numeric Types: Exact Decimals

The numeric type has two parameters:

```
NUMERIC(<precision>, <scale>);
```

`<precision>` is the total amount of digits (to both the right and left of the comma) that can be stored for each value.

`<scale>` is the total amount of decimal digits the column may store for each value. That is, the amount of digits to the right of the comma.

The Numeric Types: Exact Decimals

```
CREATE TABLE people (
  id      serial,
  height  numeric(3, 2)
);
```

Valid values:

- 1.62
- 2.32
- 9.99
- 0.01
- 1.00
- -3.50

Invalid values:

- 21.29
- 1.12345

The Numeric Types: Exact Decimals

The numeric type can also be used with only one parameter:

```
NUMERIC(<precision>);
```

The `<scale>` will be set to 0. So the field will only accept integer values.

The Numeric Types: Exact Decimals

The numeric type can even be used without any parameter:

```
NUMERIC;
```

The column will accept any value of any `<precision>` and `<scale>`.

There will be no limitation to the amount of digits that can be stored.

The Numeric Types: Inexact Decimals

There are two inexact types.

	REAL	DOUBLE PRECISION
STORAGE	4 bytes	8 bytes
PRECISION	6	15

The Text Types

There are 3 types of text columns:

	CHARACTER	CHARACTER VARYING	TEXT
LENGTH	FIXED*	VARIABLE	VARIABLE
LIMIT	YES	YES	NO
ALIAS	CHAR	VARCHAR	-

* The fixed-length type will fill up the remaining characters with white spaces.

The Text Types

```
CREATE TABLE people (
  id           serial,
  name         varchar(50),
  id_card      char(10),
  description  text
);
```

Different situations may require different text types.

Column Constraints

Constraints are a basic form of validation.

They are used to define some rules any value in a column should follow.

If the value that is being inserted does not match the rules of the column, the engine produces an error.

Column Constraints

```
CREATE TABLE people (
  username   varchar(20) UNIQUE,
  name       varchar(100) NOT NULL,
  age        integer CHECK(age > 17)
);
```

`UNIQUE` will only accept one same value in the entire column. Repeated values will produce an error.

`NOT NULL` will make the column required. A value must be provided.

`CHECK` will execute a logical expression to validate each value.

We learned …

- That PostgreSQL has a variety of types, including booleans and a variety of integer and text types.
- That booleans can be defined in many ways: true/false, yes/no, on/off and 1/0.
- That there are three types of integers that will use more or less storage space.
- That there are exact and inexact decimal types .
- That exact types are slow in performance as compared to inexact types.
- That all data types allow, by default, an additional state named **`NULL`**, which means it holds no value.
- That we can enforce different constraints on the columns.

Keys

What are Keys?

Keys are columns in a table whose values can be used to **uniquely identify** a row in the same or another table.

One may need to do an operation on any single row in a table, so there has to be a way to identify that row.

Primary Keys

- They are the columns in a table that can be used to uniquely identify any record **on that same table**.

- The values in that column **must be unique**. No two different rows may have the same value in that column.

- Although PostgreSQL does not enforce it, almost all tables should have a primary key.

Primary Keys

Any type can be set as a primary key.

```
CREATE TABLE people (
  full_name    varchar(150) PRIMARY KEY,
  description  text
);
```

This example assumes no two people in the database will have the same full name.

If that is true, this is called a **natural primary key**.

Natural vs. Artificial Primary Keys

Natural primary keys are those attributes in our user data set that can be used to identify a row (for instance, the social security number).

Often, the data does not have such combination of fields, then we have to create a **surrogate primary key**.

```
CREATE TABLE people (
  id            serial PRIMARY KEY,
  ...
);
```

Multi-Column Primary Keys

Primary keys can be declared
on multiple columns at once.

```
CREATE TABLE city (
  name          varchar(30),
  region        varchar(30),
  country       varchar(30),
  PRIMARY KEY(name, region, country)
);
```

Foreign Keys

- They are the columns in a table that can be used to uniquely identify any record **on a different table**.

- The values in that column **are not unique**. They should refer to a column in a different table where values are unique, usually the primary key in that table.

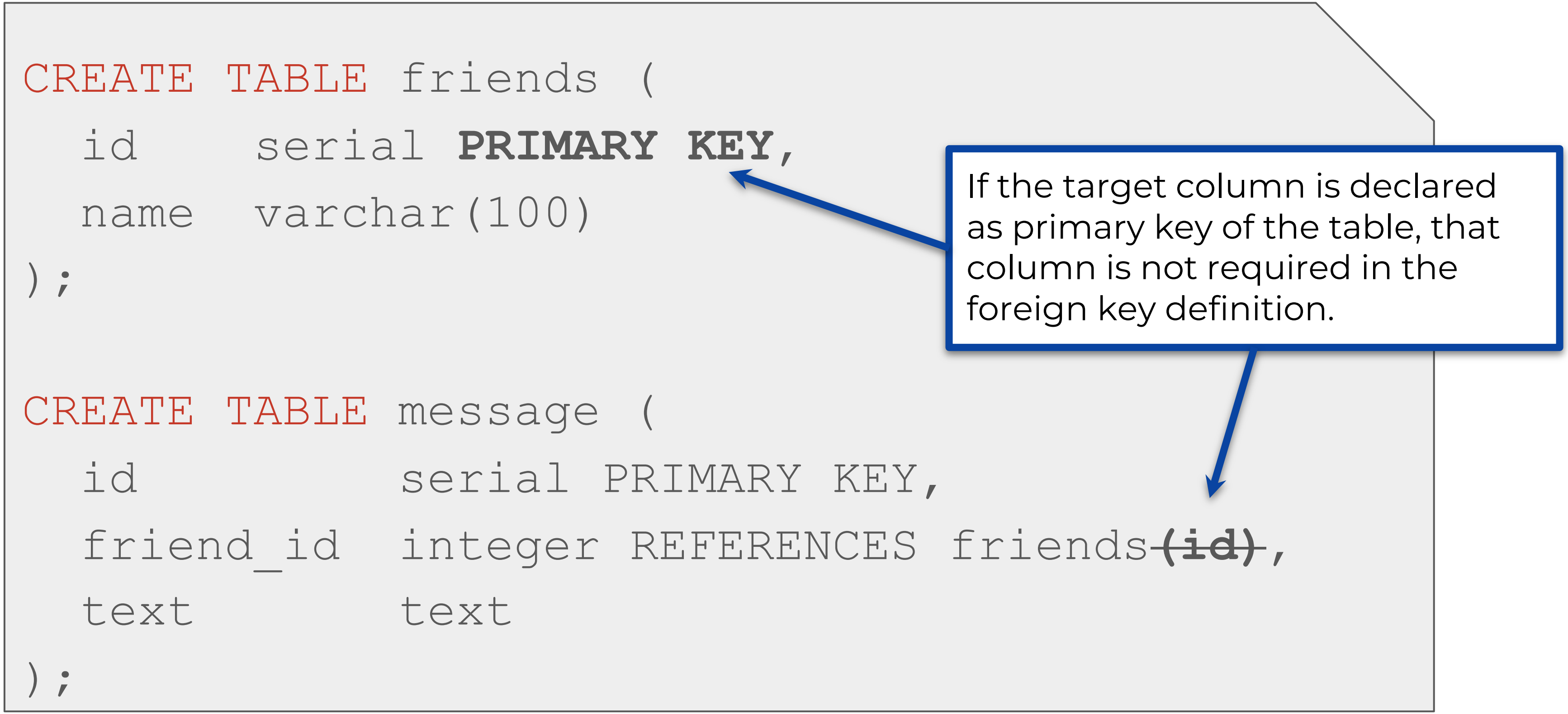
- These keys are used to define relationships between tables.

Foreign Keys

```
CREATE TABLE friends (
  id    serial,
  name  varchar(100)
);

CREATE TABLE message (
  id           serial PRIMARY KEY,
  friend_id    integer REFERENCES friends(id),
  text         text
);
```

Foreign Keys



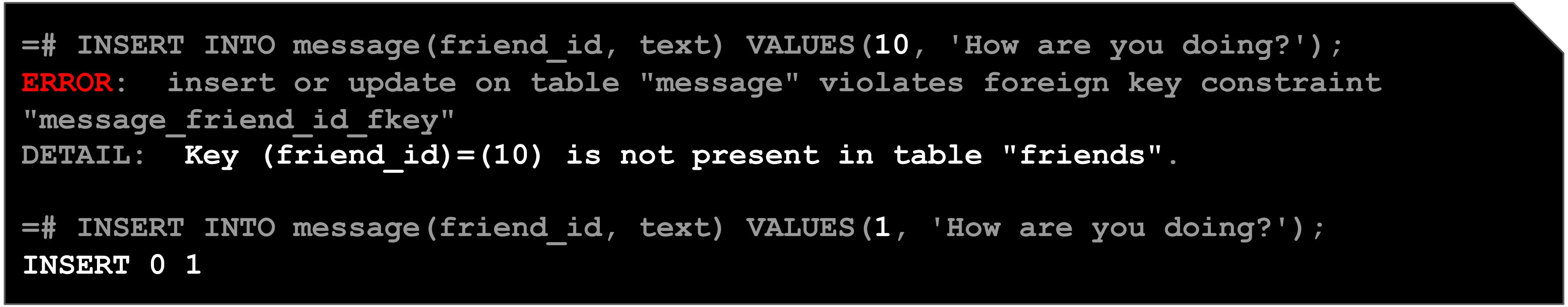
```
CREATE TABLE friends (
   id     serial PRIMARY KEY,
   name   varchar(100)
);

CREATE TABLE message (
   id          serial PRIMARY KEY,
   friend_id   integer REFERENCES friends~~(id)~~,
   text        text
);
```

If the target column is declared as primary key of the table, that column is not required in the foreign key definition.

Populating Foreign Keys

```
INSERT INTO message(friend_id, text)
VALUES(10, 'How are you doing?');
```



```
=# INSERT INTO message(friend_id, text) VALUES(10, 'How are you doing?');
ERROR:  insert or update on table "message" violates foreign key constraint
"message_friend_id_fkey"
DETAIL:  Key (friend_id)=(10) is not present in table "friends".

=# INSERT INTO message(friend_id, text) VALUES(1, 'How are you doing?');
INSERT 0 1
```

Querying Related Tables

```
SELECT friends.name, message.text
FROM friends, message
WHERE friends.id = message.friend_id;
```

```
=# SELECT friends.name, message.text FROM friends, message WHERE friends.id =
message.friend_id;
    name     |          text
-------------+--------------------
 Lisa Klepp | How are you doing?
(1 row)
```

Deleting Related Rows

```
DELETE FROM friends WHERE id = 1;
```

```
=# DELETE FROM friends WHERE id = 1;
ERROR:  update or delete on table "friends" violates foreign key constraint
"message_friend_id_fkey" on table "message"
DETAIL:  Key (id)=(1) is still referenced from table "message".
```

Deleting Related Rows: On Delete

```
CREATE TABLE message (
  id         serial   PRIMARY KEY,
  friend_id  integer  REFERENCES friends
                      ON DELETE SET NULL,
  text       text
);
```

The two most common modes for `ON DELETE` are `SET NULL` and `CASCADE`.

`SET NULL` will set the referencing value to `NULL`.

`CASCADE` will delete the referencing row.

Deleting Related Rows with SET NULL

```
DELETE FROM friends WHERE id = 1;
```

```
=# DELETE FROM friends WHERE id = 1;
DELETE 1
=# SELECT * FROM message;
 id | friend_id |        text
----+-----------+--------------------
  1 |           | How are you doing?
(1 row)
```

Deleting Related Rows with CASCADE

```
DELETE FROM friends WHERE id = 1;
```

```
=# DELETE FROM friends WHERE id = 1;
DELETE 1
=# SELECT * FROM message;
 id | friend_id | text
----+-----------+------
(0 rows)
```

We learned …

- That every table must have a combination of columns that can be used to uniquely identify a row.
- That primary keys are unique columns to identify each row.
- That foreign keys are used to reference the primary keys in different tables.
- That these keys are used to define relationships between tables in the database.
- That we can control what happens when a row in a table is deleted and there are rows in another table referring to the missing primary key.

Views

Views

- In SQL, a **view** is a statement that has been given a name.
- It works like a function. It can be executed later.
- Only **`SELECT`** statements are used in Views.
- Every time a view is called/executed, the underlying statement is executed.

Define a View

```
CREATE VIEW <name> AS <statement>;
```

```
CREATE VIEW friend_messages AS
SELECT friends.name, message.text
FROM friends, message
WHERE friends.id = message.friend_id;
```

Use a View

```
personal=# SELECT * FROM friend_messages;
      name       |             text
-----------------+------------------------------
 Lisa Klepp      | How are you doing?
 Maria Schmidt   | Will you come to the party?
 Karen O'Mailey  | Have you seen my wallet?
(3 rows)
```

The view returns a temporary table. This table can be used to perform additional queries.

```
personal=# SELECT * FROM friend_messages WHERE text LIKE 'H%';
      name       |             text
-----------------+------------------------------
 Lisa Klepp      | How are you doing?
 Karen O'Mailey  | Have you seen my wallet?
(2 rows)
```

Rename and Remove a View

```
ALTER VIEW [IF EXISTS] friend_messages
RENAME TO full_name_messages;
```

```
DROP VIEW [IF EXISTS] full_name_messages;
```

Change a View

```
CREATE OR REPLACE VIEW <name> AS <statement>;
```

```
CREATE OR REPLACE VIEW friend_messages AS
SELECT friends.name, friends.age, message.text
FROM friends, message
WHERE friends.id = message.friend_id;
```

Updatable Views

```sql
INSERT INTO teenage_friends(name, age)
VALUES('Amina', 30);
```

INSERT, **UPDATE** and **DELETE** can be used on a view, only if the view is defined with one single table and the columns modified are present in the view.

```sql
CREATE OR REPLACE VIEW teenage_friends AS
SELECT friends.name, friends.age
FROM friends
WHERE friends.age BETWEEN 13 AND 19;
```

The new record will be added to the **friends** table. The values inserted do not need to match the query's conditions.

Updatable Views

```
INSERT INTO teenage_friends(name, age)
VALUES('Amina', 30);
```

The values inserted do not need to match the conditions in the view.

```
CREATE OR REPLACE VIEW teenage_friends AS
SELECT friends.name, friends.age
FROM friends
WHERE friends.age BETWEEN 13 AND 19;
```

Updatable Views

Adding `WITH CHECK OPTION` will require the inserted values to match the conditions in the query defined in the view.

```
CREATE OR REPLACE VIEW teenage_friends AS
SELECT friends.name, friends.age
FROM friends
WHERE friends.age BETWEEN 13 AND 19
WITH CHECK OPTION;
```

```
personal=# INSERT INTO teenage_friends(name, age) VALUES('Amina', 30);
ERROR:  new row violates WITH CHECK OPTION for view "teenage_friends"
DETAIL:  Failing row contains (null, null, null, 30, null, null, 7, Amina).
```

Materialized Views

- A materialized view is a view that has been made persistent by storing its results in a temporary table.
- Subsequent calls to the view, will not process the underlying query, but will return the previously stored data.
- The query will not be executed unless the materialized view is refreshed (re-evaluated).

Define a Materialized View

```sql
CREATE MATERIALIZED VIEW friend_messages AS
SELECT friends.name, message.text
FROM friends, message
WHERE friends.id = message.friend_id;
```

The usage of a materialized view is the same as a standard view.

Refresh a Materialized View

```
REFRESH MATERIALIZED VIEW friend_messages;
```

Refreshing the materialized view will execute again the query and store the results.

We learned ...

- That a query can be given a name.
- That named queries are called views and can be reused many times.
- That calling a view executes the underlying query.
- That there are special views, who store the results of the query and do not get executed again every time.
- That these views are called materialized views and can be refreshed when required.
- That materialized views can be used to cache complex queries and improve the user experience.

Documentation

Documentation

General PostgreSQL Documentation

- https://www.postgresql.org/docs/current/index.html
- https://www.postgresql.org/docs/current/tutorial.html

SQL

- https://en.wikipedia.org/wiki/SQL
- https://www.w3schools.com/sql/sql_intro.asp
- https://www.postgresql.org/docs/current/sql.html

Data types

- https://www.postgresql.org/docs/current/datatype.html
- https://www.tutorialspoint.com/postgresql/postgresql_data_types.htm

Primary & Foreign Keys

- https://www.postgresqltutorial.com/postgresql-primary-key/
- https://www.postgresql.org/docs/current/ddl-constraints.html

Views

- https://www.postgresql.org/docs/current/sql-createview.html
- https://www.postgresql.org/docs/current/rules-materializedviews.html

THANK YOU

Contact Details
DCI Digital Career Institute gGmbH

Digital Career Institute