Database | SQL | PostgreSQL

Session 2

- Relationship types
- GUI tools
- psycopg2
- Advance topics

by Mohammad Amin H.B. Tehrani

www.maktabsharif.ir

Relationship types

Intro

A relational database collects different types of data sets that use tables, records, and columns. It is used to create a well-defined relationship between database tables so that relational databases can be easily stored. For example of relational databases such as Microsoft SQL Server, Oracle Database, MYSQL, PostgreSQL, etc.

1. **One to One relationship (1:1)**
2. **One to many or many to one relationship (1:M)**
3. **Many to many relationships (M:N)**

Relationship types

One to One Relationship (1:1, 0:1)

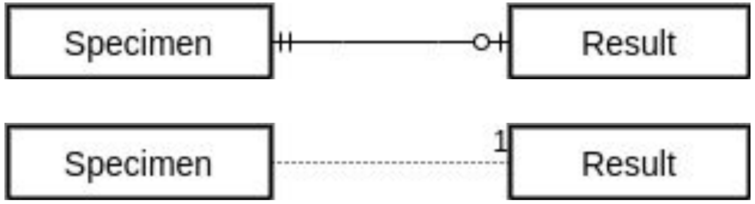
It is used to create a relationship between two tables in which **a single row of the first table can only be related to one and only one records of a second table.** Similarly, the row of a second table can also be related to anyone row of the first table.

Specimen		
1	Akbar	blood
2	Mamad	Sugar
	...	
11	Asqar	Covid

Result
Positive
Negative
...
Positive

Other examples:

- Man/Woman partnership
- Car, Driver
- Country, Capital
- Person, Passport

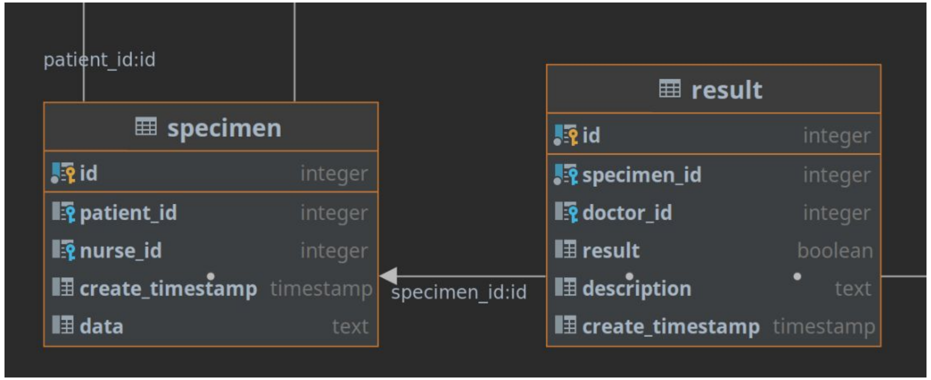


Relationship types

Example of One to One Relationship

```
CREATE TABLE specimen (
   id SERIAL PRIMARY KEY,
   ...
);
```

```
CREATE TABLE result (
   ...,
   specimen id INT UNIQUE
     REFERENCES specimen(id)
     ON DELETE RESTRICT
   ...,
);
```



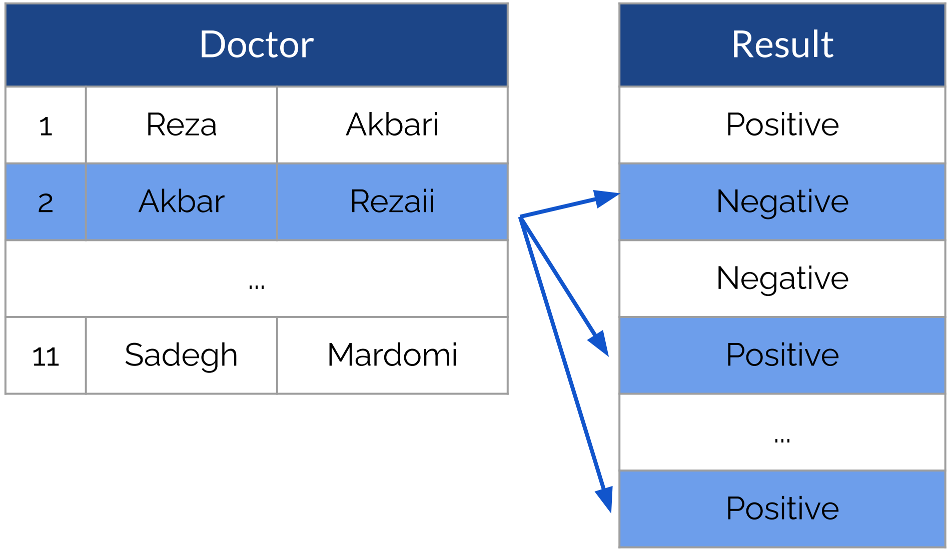
specimen	
id	integer
patient_id	integer
nurse_id	integer
create_timestamp	timestamp
data	text

result	
id	integer
specimen_id	integer
doctor_id	integer
result	boolean
description	text
create_timestamp	timestamp

Relationship types

One to Many Relationship (1:N, 0:N)

It is used to create a relationship between two tables. **Any single rows of the first table can be related to one or more rows of the second tables, but the rows of second tables can only relate to the only row in the first table.** It is also known as a many to one relationship.

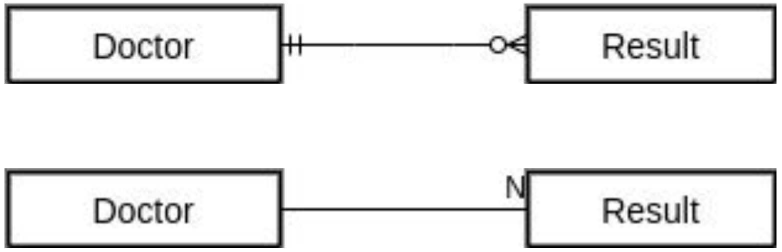


Doctor		
1	Reza	Akbari
2	Akbar	Rezaii
...		
11	Sadegh	Mardomi

Result
Positive
Negative
Negative
Positive
...
Positive

Other examples:

- Card, CardItems
- Car, Passengers
- Mother, Childs
- Role, Users
- User, addresses

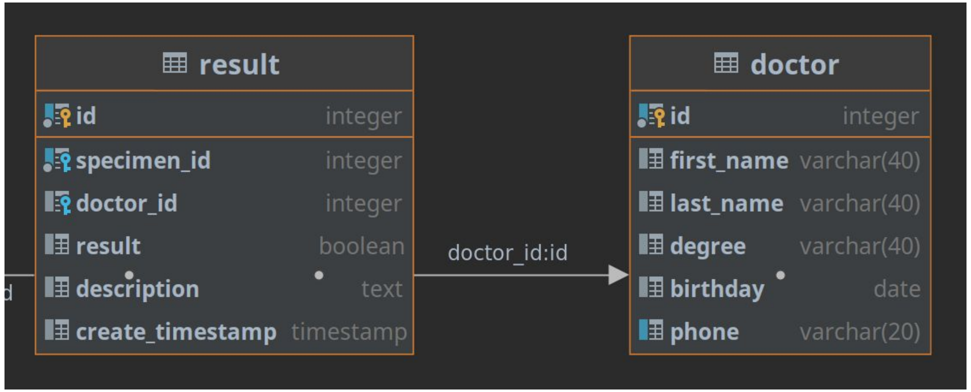


Relationship types

Example of One to Many Relationship

```
CREATE TABLE doctor (
    id SERIAL PRIMARY KEY,
    ...
);
```

```
CREATE TABLE result (
    ...,
    doctor id INT
      REFERENCES doctor(id)
      ON DELETE CASCADE
    ...,
);
```



result	
id	integer
specimen_id	integer
doctor_id	integer
result	boolean
description	text
create_timestamp	timestamp

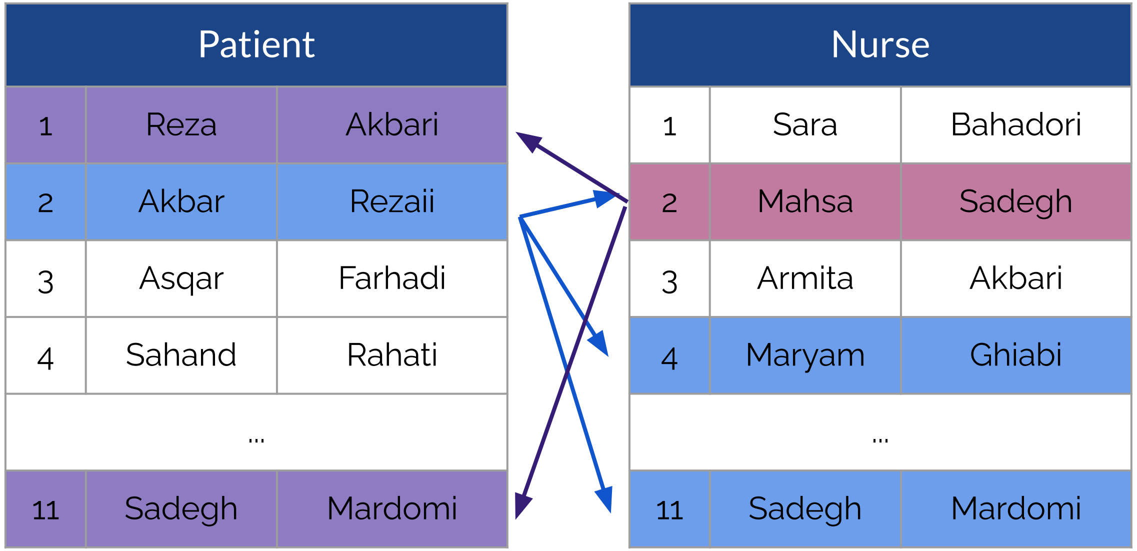
doctor	
id	integer
first_name	varchar(40)
last_name	varchar(40)
degree	varchar(40)
birthday	date
phone	varchar(20)

What is the difference from One to One Relationship table structure?!

Relationship types

Many to Many Relationship (M:N)

It is **many to many** relationships that create a relationship between two tables. **Each record of the first table can relate to any records (or no records) in the second table. Similarly, each record of the second table can also relate to more than one record of the first table.** It is also represented an M:N relationship.

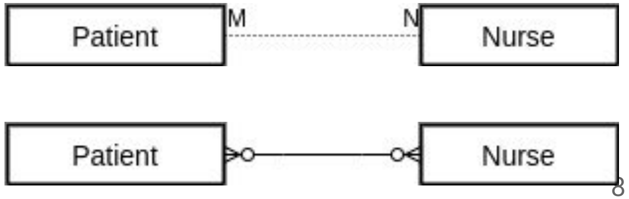


Patient		
1	Reza	Akbari
2	Akbar	Rezaii
3	Asqar	Farhadi
4	Sahand	Rahati
...		
11	Sadegh	Mardomi

Nurse		
1	Sara	Bahadori
2	Mahsa	Sadegh
3	Armita	Akbari
4	Maryam	Ghiabi
...		
11	Sadegh	Mardomi

Other examples:

- Book, Author
- Person, Group
- Student, Course
- Friends
- ...



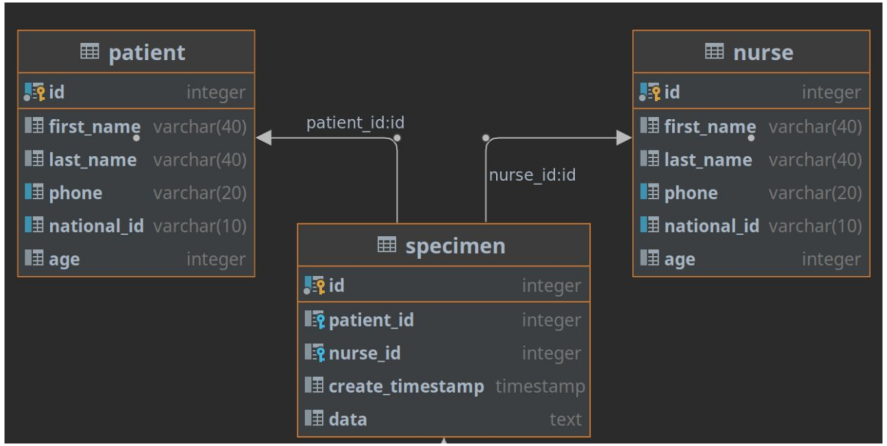
Relationship types

Example of Many to Many Relationship

```
CREATE TABLE patient (
   id SERIAL PRIMARY KEY,
   ...
);
```

```
CREATE TABLE nurse (
   id SERIAL PRIMARY KEY,
   ...,
);
```

```
CREATE TABLE specimen (
   ...,
   patient_id INT REFERENCES patient (id),
   nurse_id INT REFERENCES nurse (id),
   ...
);
```



In these cases with Many to Many Relationships, We **ALWAYS** need a Middle table, which connects both tables through itself.

GUI tools

Top PostgreSQL GUI tools

PostgreSQL graphical user interface (GUI) tools help open source database users to manage, manipulate, and visualize their data.

Many still prefer CLIs over GUIs, but this set is ever so shrinking. I believe anyone who comes into programming after 2010 will tell you GUI tools increase their productivity over a CLI solution.

Top PostgreSQL GUI tools:

1. pgAdmin
2. DBeaver
3. OmniDB
4. DataGrip
5. Navicat
6. HeidiSQL

GUI tools

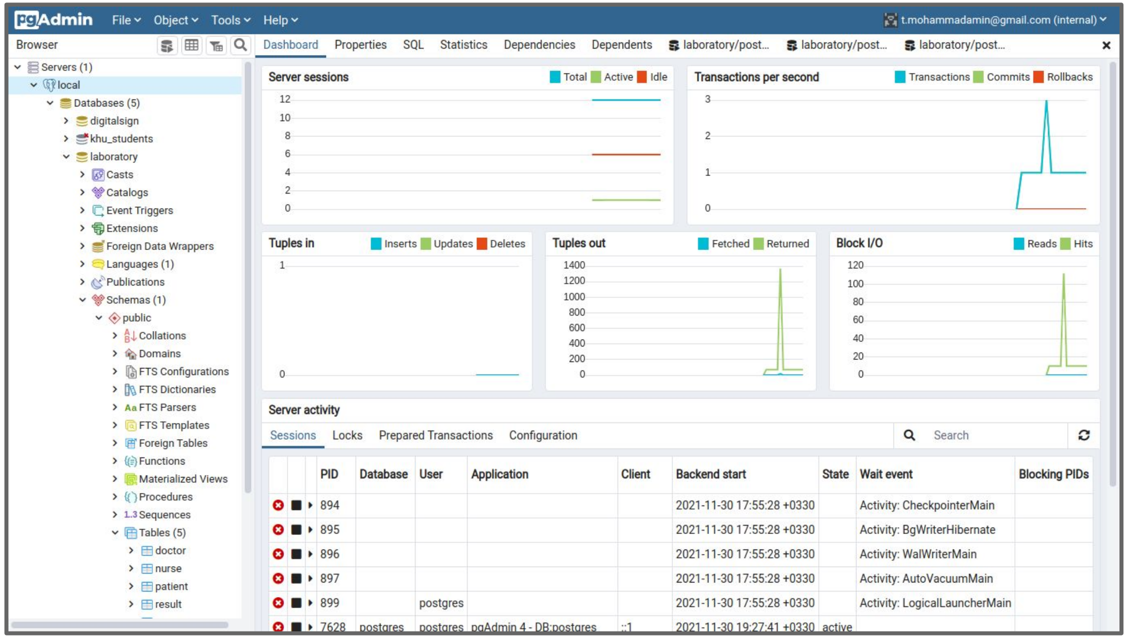
PgAdmin

pgAdmin is the most popular and feature rich Open Source administration and development platform for PostgreSQL, the most advanced Open Source database in the world.

Download Page

Official supported platforms:

- Python
- Docker Container
- macOS
- APT (Debian/Ubuntu)
- RPM
- Windows

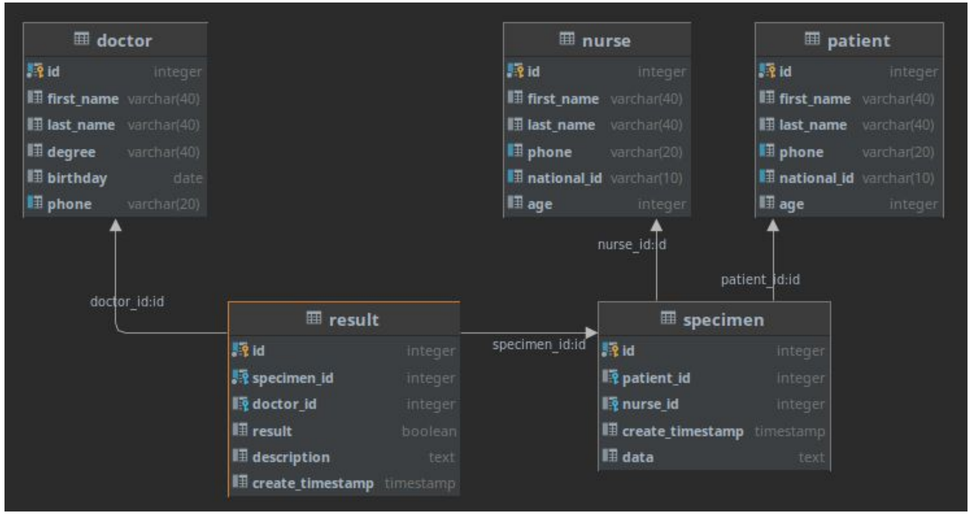


PID	Database	User	Application	Client	Backend start	State	Wait event	Blocking PIDs
894					2021-11-30 17:55:28 +0330		Activity: CheckpointerMain	
895					2021-11-30 17:55:28 +0330		Activity: BgWriterHibernate	
896					2021-11-30 17:55:28 +0330		Activity: WalWriterMain	
897					2021-11-30 17:55:28 +0330		Activity: AutoVacuumMain	
899		postgres			2021-11-30 17:55:28 +0330		Activity: LogicalLauncherMain	
7628	postgres	postgres	pgAdmin 4 - DB:postgres	::1	2021-11-30 19:27:41 +0330	active		

GUI tools

PyCharm Professional Database tool

In the **Database** tool window (**View | Tool Windows | Database**), you can work with databases and DDL data sources. You can view and modify data structures in your databases, and perform other associated tasks. To view a table, double-click the table. For more information about different viewing modes, see View data.



doctor

id	integer
first_name	varchar(40)
last_name	varchar(40)
degree	varchar(40)
birthday	date
phone	varchar(20)

nurse

id	integer
first_name	varchar(40)
last_name	varchar(40)
phone	varchar(20)
national_id	varchar(10)
age	integer

patient

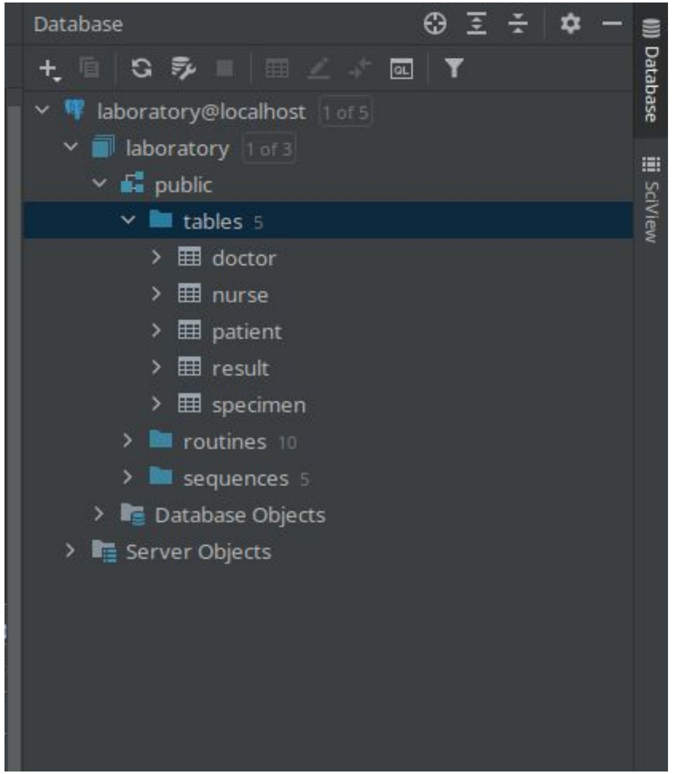
id	integer
first_name	varchar(40)
last_name	varchar(40)
phone	varchar(20)
national_id	varchar(10)
age	integer

result

id	integer
specimen_id	integer
doctor_id	integer
result	boolean
description	text
create_timestamp	timestamp

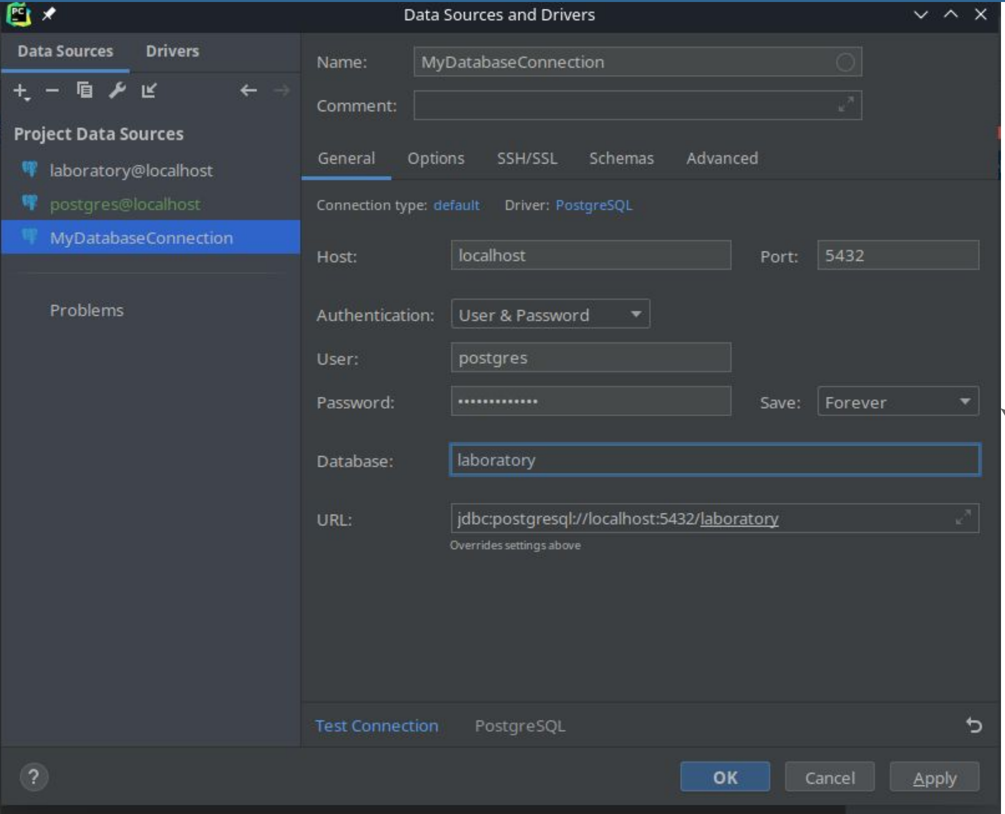
specimen

id	integer
patient_id	integer
nurse_id	integer
create_timestamp	timestamp
data	text



GUI tools

Start a Connection using Pycharm Database tool



Data Sources and Drivers

Data Sources | Drivers

Project Data Sources

- laboratory@localhost
- postgres@localhost
- MyDatabaseConnection

Problems

Name:	MyDatabaseConnection		
Comment:			

General | Options | SSH/SSL | Schemas | Advanced

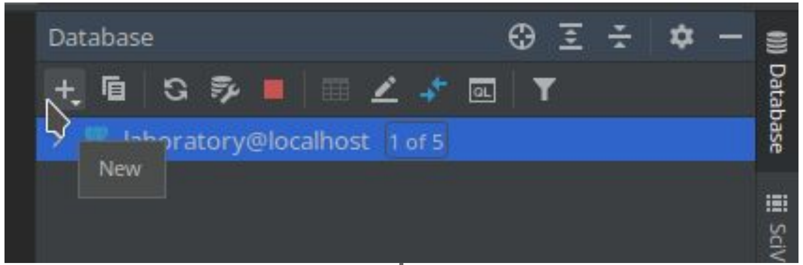
Connection type: default Driver: PostgreSQL

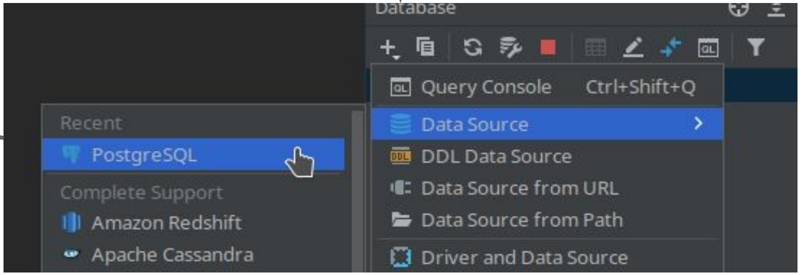
Host:	localhost	Port:	5432
Authentication:	User & Password		
User:	postgres		
Password:	•••••••••••••	Save:	Forever
Database:	laboratory		
URL:	jdbc:postgresql://localhost:5432/laboratory		

Overrides settings above

Test Connection PostgreSQL

OK Cancel Apply





psycopg2

psycopg2

Introduction

Psycopg is the most popular PostgreSQL database adapter for the Python programming language. Its main features are the complete implementation of the Python DB API 2.0 specification and the thread safety (several threads can share the same connection). It was designed for heavily multi-threaded applications that create and destroy lots of cursors and make a large number of concurrent “INSERT”s or “UPDATE”s.

Pypi: https://pypi.org/project/psycopg2/

Official Document: https://www.psycopg.org/docs/index.html

Installation: https://www.psycopg.org/docs/install.html#install-from-source

psycopg2

Installation

For most operating systems, the quickest way to install Psycopg is using the wheel package available on PyPI:

```
pip install psycopg2-binary
```

Example:

```python
import psycopg2

# Connect to your postgres DB
conn = psycopg2.connect("dbname=test user=postgres")

# Open a cursor to perform database operations
cur = conn.cursor()

# Execute a query
cur.execute("SELECT * FROM my_data")

# Retrieve query results
records = cur.fetchall()
```

psycopg2

1. Connecting to the DB

*psycopg2.**connect**(dsn=None, connection_factory=None, cursor_factory=None, async=False, **kwargs)*

>>> Create a new database session and return a new **connection** object.

The connection parameters can be specified as a libpq connection string using the **dsn** parameter:

```
conn = psycopg2.connect("dbname=test user=postgres password=secret")
```

or using a set of keyword arguments:

```
conn = psycopg2.connect(dbname="test", user="postgres", password="secret")
```

- **dbname** – the database name (database is a deprecated alias)
- **user** – user name used to authenticate
- **password** – password used to authenticate
- **host** – database host address (defaults to UNIX socket if not provided)
- **port** – connection port number (defaults to 5432 if not provided)

2. Opening a cursor

connection.cursor(name=None, cursor_factory=None, scrollable=None, withhold=False)

>>> Return a new cursor object using the connection.

In order to query a database, a cursor object through the programing level is always required. All commands will be sent through the opened cursor.

```
import psycopg2
from psycopg2._psycopg import connection, cursor  # For typing

conn: connection = psycopg2.connect(dbname='laboratory', user='postgres')

# Open a cursor to perform database operations
cur: cursor = conn.cursor()

...

# Close the cursor
cur.close()
```

psycopg2

3. Executing commands

execute(query, vars=None)

>>> Execute a database operation (query or command).

Parameters may be provided as sequence or mapping and will be bound to variables in the operation. Variables are specified either with positional (%s) or named (%(name)s) placeholders. See Passing parameters to SQL queries.

The method returns None. If a query was executed, the returned values can be retrieved using fetch*() methods.

```
import psycopg2
from psycopg2._psycopg import connection, cursor  # For typing

conn: connection = psycopg2.connect(dbname='laboratory', user='postgres')

# Open a cursor to perform database operations
cur: cursor = conn.cursor()

# Execute a query
cur.execute("""SELECT table_name FROM information_schema.tables WHERE table_schema = 'public'""")
```

psycopg2

4. Fetching records

1. ***fetchone()***
   Fetch the next row of a query result set, returning a single tuple, or None when no more data is available
2. ***fetchmany([size=cursor.arraysize])***
   Fetch the next set of rows of a query result, returning a list of tuples. An empty list is returned when no more rows are available.
3. ***fetchall()***
   Fetch all (remaining) rows of a query result, returning them as a list of tuples. An empty list is returned if there is no more record to fetch.

```
>>> cur.execute("SELECT * FROM test WHERE id = %s", (3,))
>>> cur.fetchone()
(3, 42, 'bar')
```

```
>>> cur.execute("SELECT * FROM test;")
>>> cur.fetchall()
[(1, 100, "abc'def"), (2, None, 'dada'), (3, 42, 'bar')]
```

```
>>> cur.execute("SELECT * FROM test;")
>>> cur.fetchmany(2)
[(1, 100, "abc'def"), (2, None, 'dada')]
>>> cur.fetchmany(2)
[(3, 42, 'bar')]
>>> cur.fetchmany(2)
[]
```

psycopg2

Example: fetching tables

```python
import psycopg2
from psycopg2._psycopg import connection, cursor  # For typing

conn: connection = psycopg2.connect(dbname='laboratory', user='postgres')

# Open a cursor to perform database operations
cur: cursor = conn.cursor()

# Execute a query
cur.execute("""SELECT table_name FROM information_schema.tables
      WHERE table_schema = 'public'""")

# Retrieve query results
tables = cur.fetchall()
print(tables)

# Close the cursor
cur.close()
```

psycopg2

Passing parameters to SQL queries

See full tutorial Here

Psycopg converts Python variables to SQL values using their types: the Python type determines the function used to convert the object into a string representation suitable for PostgreSQL. Many standard Python types are already adapted to the correct SQL representation.

Passing parameters to an SQL statement happens in functions such as cursor.execute() by using %s placeholders in the SQL statement, and passing a sequence of values as the second argument of the function. For example the Python function call:

```
>>> cur.execute("""
...     INSERT INTO some_table (an_int, a_date, a_string)
...     VALUES (%s, %s, %s);
...     """,
...     (10, datetime.date(2005, 11, 18), "O'Reilly"))
```

```
>>> cur.execute("""
...     INSERT INTO some_table (an_int, a_date, another_date, a_string)
...     VALUES (%(int)s, %(date)s, %(date)s, %(str)s);
...     """,
...     {'int': 10, 'str': "O'Reilly", 'date': datetime.date(2005, 11, 18)})
```

psycopg2

Cursor and Connection context manager

Connections can be used as context managers. Note that a context wraps a transaction: if the context exits with success the transaction is committed, if it exits with an exception the transaction is rolled back. **Note that the connection is not closed by the context and it can be used for several contexts.**

Cursors can be used as context managers: leaving the context will close the cursor.

```
conn = psycopg2.connect(DSN)

with conn:
   with conn.cursor() as curs:
       curs.execute(SQL1)
       ...
   # the cursor is now closed

with conn:
   with conn.cursor() as curs:
       curs.execute(SQL2)
       ...
   # the cursor is now closed

# leaving contexts doesn't close the connection
conn.close()
```

Exercise

Consider a class **"Specimen"** with attributes matching with the table ***'specimen'*** and write the methods below**:**

1. Class method **create_specimen(...)** to insert a new record of specimen into the table.
2. Instance method **delete()** to delete the record from the table.
3. Instance method **update()** to update the record with the instance values
4. Class method **count()** to update: returns count of all records
5. Class method **all_specimen():** returns all records of the table
6. Class method **get_specimen_by_id():** returns a **Specimen** object fetching from the table.
7. Class method **filter_specimen(where=...):** returns a list of **Specimen** objects, filtered by **where conditions.**

Advance Topics

- Logical delete
- Self-Relation
- DB indexes
- Normalizations
- Database Views
- Database Functions
- Database Procedure
- Database Triggers

Advance topics

Self-Relation

This particular type of relationship does not exist between a pair of tables, which is why it isn't mentioned at the beginning of this section. It is instead a relationship that exists between the records within a table. Ironically, you'll still regard this throughout the design process as a table relationship.

A table bears a self-referencing relationship (also known as a recursive relationship) to itself when a given record in the table is related to other records within the table. Similar to its dual-table counterpart,

a self-referencing relationship can be:

- **one-to-one**
- **one-to-many**
- **many-to-many**

Employees

id	name	...	supervisor	...
1	Akbar	...	-	...
2	Asqar	...	-	...
3	Reza	...	1	...
4	Darab	...	2	...
...	...	...	...	...
551	Mohammad	...	4	...
552	Sadeq	...	3	...
553	Ali	...	552	...

Advance topics

Logical delete

Physical (Hard) delete

A physical deletion is an actual deletion in SQL, and is also deleted from the database.
Therefore, you can’t restore or refer to the deleted data.

Logical (Soft) delete

Logical deletion is the process of not actually deleting the data,
but setting a column called “flag” to make it appear as if the data has been deleted to the user.

id	name	…	is_delete	…
1	Akbar	…	False	…
2	Asqar	…	True	…
3	Reza	…	False	…
4	Darab	…	False	…
…	…	…	…	…
551	Mohammad	…	True	…
552	Sadeq	…	False	…
553	Ali	…	False	…