

Semestral work DBS 2025

Theme: Sports Complex Database

Authors: Oleksii Kolesnichenko, Serhii Zhyltsov

e-mails: **`kolesole@fel.cvut.cz`**, **`zhyltser@fel.cvut.cz`**

Contents

1. CP-0 Topic Statement + Bonus	2
1.1. Theme	2
1.2. Motivation	2
1.3. Short Description	2
1.3. Taxonomic and Operational Scope	2
1.4. Detailed List of Entities and Relationships	2
2. CP-1 Conceptual Model	3
2.1. Description	3
2.2. ER Diagram	3
2.3. Entities	4
2.4. Relationships	4
3. CP-2 Relational Model	5
4. CP-3 Creating a database, querying data	6
4.1. Relational model with ids	6
4.2. ER Model	7
4.3. Creating tables	8
4.4. Python script for generating data	13
4.5. Queries	26
5. CP-4 Advanced Database Technologies	39
5.1. Transaction	39
5.2. View	44
5.3. Trigger	45
5.4. Index	48

CP-0 Topic Statement + Bonus

Theme: Sports Complex Database

Motivation: The motivation is to simplify and accelerate the daily operations of the sports complex, solve issues with scheduling conflicts, payment transactions, and monitoring the use of individual sports facilities.

Short Description: This project focuses on the design and implementation of a database system for managing reservations in the sports complex, which includes group/solo trainings with trainer or without. The goal is to effectively digitize the process of reservations, payments, reservation collisions etc.

Taxonomic and Operational Scope: The proposed system will include several taxonomic levels (records of branches, sports halls, clients, trainers and trainings) and will handle operational data such as attendance at training sessions, payments.

Bonus: Detailed List of Entities and Relationships

Entities(WHO and WHAT):

1. **Branch** - [address](#)(city, street, house number)
2. **Sports Hall** - [name](#), size, specialization
3. **User** - [login](#), password
4. **Client** (derives from User) - preferences in sports
5. **Trainer** (derives from User) - specialization, education
6. **Personal Data** - [name](#), [surname](#), [date of birth](#), [e-mail](#), [phone number](#)
7. **Reservation** - status, reservation date
8. **Payment** - amount, payment date, payment method
9. **Training** - type, training date

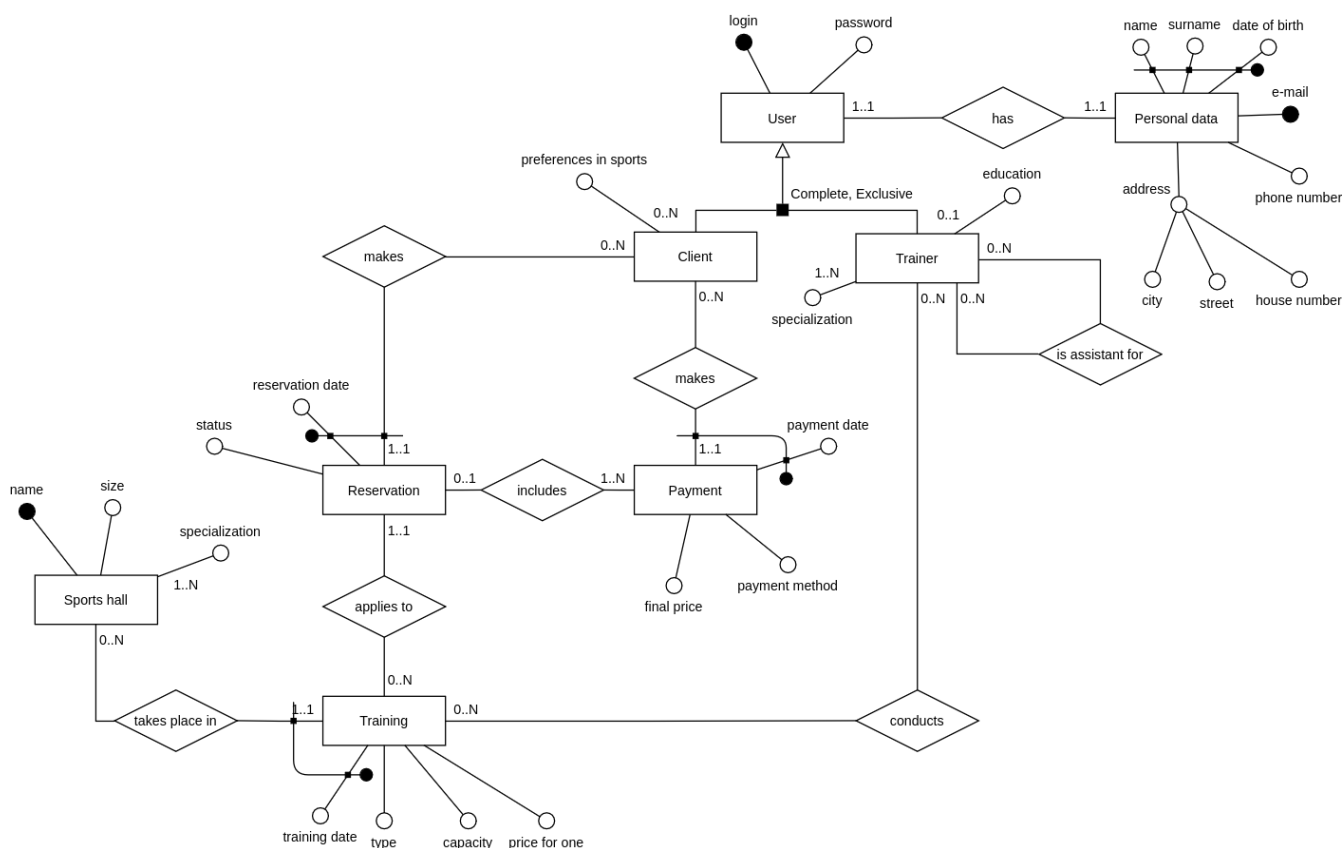
Relationships(What does):

- | | |
|--|---|
| 1. Branch contains Sports Halls | 6. Trainer conducts trainings |
| 2. User has Personal data | 7. Reservation includes Payment |
| 3. Client makes Reservation | 8. Reservation applies to Training |
| 4. Client makes Payment | 9. Training takes place in Sports hall |
| 5. Trainer is assistant for trainer | |

CP-1 Conceptual Model

Description: The Sports Complex Database System allows clients to book trainings, make payments, and manage their training schedules efficiently. Users register with a login and password, with clients selecting their preferences in sports and trainers managing their schedules. Reservations are linked to specific trainings and sports halls, ensuring availability and proper allocation of resources. Clients must complete payment using various methods before confirming their reservations. Trainers conduct sessions based on their specialization and may assist each other. Each training session has a defined training date, type, capacity, and price per participant (which make up the full price in Payment). The system ensures structured scheduling, preventing double bookings and optimizing trainer assignments. Personal data, including information about user (name, surname, date of birth, address, e-mail, phone number). This system provides an organized and seamless experience for managing trainings, payments and reservations.

ER Diagram:



Entities:

1. **Sports Hall** - [name](#), size, specialization(1..N)
2. **User** - [login](#), password
3. **Client** (derives from User) - preferences in sports(0..N)
4. **Trainer** (derives from User) - specialization(1..N), education(0..1)
5. **Personal Data** - [name](#), [surname](#), [date of birth](#), [e-mail](#), phone number, address(city, street, house number)
6. **Reservation**(week entity: identified by Client + reservation date) - status, reservation date
7. **Payment**(week entity: identified by Client + payment date) - final price, payment date, payment method
8. **Training**(week entity: identified by Sports hall + training date) - type, training date, capacity, price for one

Relationships:

1. User **has** Personal data (1..1 - 1..1)
2. Client **makes** Reservation (0..N - 1..1)
3. Client **makes** Payment (0..N - 1..1)
4. Trainer **is assistant for** (recursion relationship) trainer (0..N - 0..N)
5. Trainer **conducts** trainings (0..N - 0..N)
6. Reservation **includes** Payment (0..1 - 1..N)
7. Reservation **applies to** Training (1..1 - 0..N)
8. Training **takes place in** Sports hall (1..1 - 0..N)

CP-2 Relational Model

Sports hall(name, size)

Hall specialization(name, specialization)

FK: (name) \subseteq **Sports hall**(name)

User(login, password)

Personal data(name, surname, date of birth, e-mail, phone number, login)

FK: (login) \subseteq **User**(login)

Address(e-mail, city, street, house number)

FK: (e-mail) \subseteq **Personal data**(e-mail)

Client(login)

FK: (login) \subseteq **User**(login)

Preferences in sports(login, preferences in sports)

FK: (login) \subseteq **Client**(login)

Trainer(login)

FK: (login) \subseteq **User**(login)

Trainer specialization(login, name of sport)

FK: (login) \subseteq **Trainer**(login)

Education(login, education)

FK: (login) \subseteq **Trainer**(login)

Is assistant for(trainer, assistant)

FK: (trainer) \subseteq **Trainer**(login)

FK: (assistant) \subseteq **Trainer**(login)

Training(training date, name, type, capacity, price for one)

FK: (name) \subseteq **Sports hall**(name)

Conducts(login, name, training date)

FK: (login) \subseteq **Trainer**(login)

FK: (name, training date) \subseteq **Training**(name, training date)

Payment(final price, payment method, payment date, login)

FK: (login) \subseteq **Client**(login)

Reservation(reservation date, login1, status, training date, name, payment_date, login2)

FK: (login1) \subseteq **Client**(login)

FK: (payment_date, login2) \subseteq **Payment**(payment_date, login)

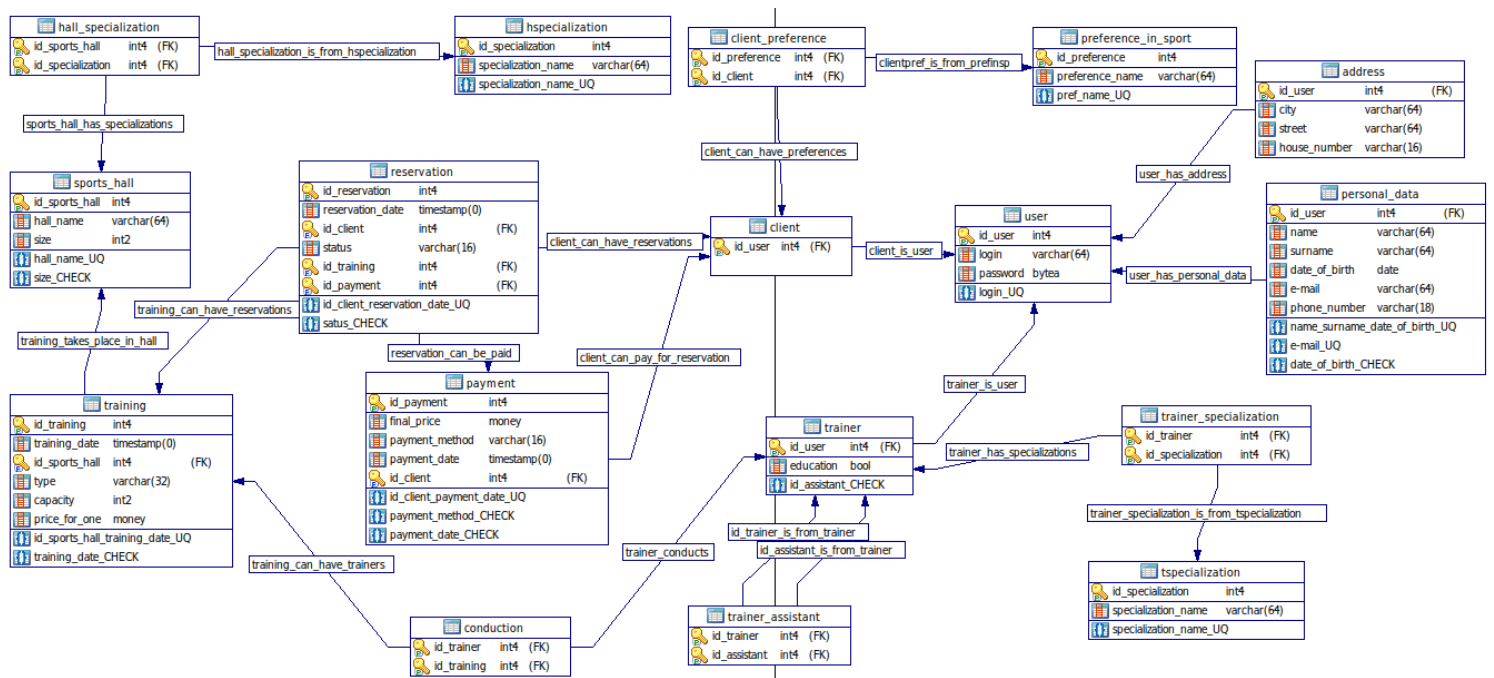
FK: (training date, name) \subseteq **Training**(training date, name)

CP-3 Creating a database, querying data

Relational model with ids:

Sports_hall(id_sports_hall, hall_name, size)
HSpecialization(id_specialization, specialization_name)
Hall_specialization(id_sports_hall, id_specialization)
 FK: (id_sports_hall) \subseteq **Sports hall**(id_sports_hall)
 FK: (id_specialization) \subseteq **HSpecialization**(id_specialization)
User(id_user, login, password)
Personal_data(name, surname, date_of_birth, e-mail, phone_number, id_user)
 FK: (id_user) \subseteq **User**(id_user)
Address(id_user, city, street, house_number)
 FK: (id_user) \subseteq **Personal data**(id_user)
Client(id_user)
 FK: (id_user) \subseteq **User**(id_user)
Preference_in_sports(id_preference, preference_name)
Client_preference(id_client, id_preference)
 FK: (id_client) \subseteq **Client**(id_user)
 FK: (id_preference) \subseteq **Preference_in_sports**(id_preference)
Trainer(id_user, education)
 FK: (id_user) \subseteq **User**(id_user)
TSpecialization(id_specialization, specialization_name)
Trainer_specialization(id_trainer, id_specialization)
 FK: (id_trainer) \subseteq **Trainer**(id_user)
 FK: (id_specialization) \subseteq **TSpecialization**(id_specialization)
Trainer_assistant(id_trainer, id_assistant)
 FK: (id_trainer) \subseteq **Trainer**(id_user)
 FK: (id_assistant) \subseteq **Trainer**(id_user)
Training(id_training, training_date, id_sports_hall, type, capacity, price_for_one)
 FK: (id_sports_hall) \subseteq **Sports hall**(id_sports_hall)
Conduction(id_trainer, id_training)
 FK: (id_trainer) \subseteq **Trainer**(id_user)
 FK: (id_training) \subseteq **Training**(id_training)
Payment(id_payment, final_price, payment_method, payment_date, id_client)
 FK: (id_client) \subseteq **Client**(id_user)
Reservation(id_reservation, reservation_date, id_client, status, training_date, id_sports_hall, id_payment)
 FK: (id_client) \subseteq **Client**(id_user)
 FK: (training date, id_sports_hall) \subseteq **Training**(training date, id_sports_hall)
 FK: (id_payment) \subseteq **Payment**(id_payment)

ER Model:



Creating tables:

```
CREATE TABLE sports_hall (  
    id_sports_hall    serial    PRIMARY KEY,  
    hall_name         varchar(64) NOT NULL UNIQUE,  
    size              smallint  NOT NULL  
                        CHECK (size BETWEEN 1 AND 3)  
                        DEFAULT 1  
);  
  
CREATE TABLE hspecialization (  
    id_specialization serial    PRIMARY KEY,  
    specialization_name varchar(64) NOT NULL UNIQUE  
);  
  
CREATE TABLE hall_specialization (  
    id_sports_hall    int4      NOT NULL REFERENCES sports_hall  
                        ON DELETE CASCADE  
                        ON UPDATE CASCADE,  
    id_specialization int4      NOT NULL REFERENCES hspecialization  
                        ON DELETE CASCADE  
                        ON UPDATE CASCADE,  
    PRIMARY KEY (id_sports_hall, id_specialization)  
);
```

Technical note:

ON DELETE CASCADE ensures that when a sports hall or specialization is deleted, related mappings are also removed.

ON UPDATE CASCADE propagates changes in primary keys to dependent rows.

```
CREATE TABLE "user" (  
    id_user    serial    PRIMARY KEY,  
    login      varchar(64) NOT NULL UNIQUE,  
    password   bytea     NOT NULL  
);  
  
CREATE TABLE client (  
    id_user    int4      PRIMARY KEY REFERENCES "user"  
                        ON DELETE NO ACTION  
                        ON UPDATE CASCADE  
);
```

Technical note:

ON DELETE NO ACTION prevents deleting a user who is also a client (accidental deletion), but if by the end of the transaction the client is deleted, the user will be deleted too.

ON UPDATE CASCADE updates client ID if the user ID changes.

```

CREATE TABLE preference_in_sport (
    id_preference      serial      PRIMARY KEY,
    preference_name    varchar(64) NOT NULL UNIQUE
);

CREATE TABLE client_preference (
    id_client          int4        NOT NULL REFERENCES client (id_user)
                                ON DELETE CASCADE
                                ON UPDATE CASCADE,
    id_preference       int4        NOT NULL REFERENCES preference_in_sport
                                ON DELETE CASCADE
                                ON UPDATE CASCADE,
    PRIMARY KEY (id_client, id_preference)
);

```

Technical note:

ON DELETE CASCADE ensures that when a client or preference is deleted, related mappings are also removed.

ON UPDATE CASCADE propagates changes in primary keys to dependent rows.

```

CREATE TABLE personal_data (
    id_user            int4        PRIMARY KEY REFERENCES "user"
                                ON DELETE NO ACTION
                                ON UPDATE CASCADE,
    name               varchar(64) NOT NULL,
    surname             varchar(64) NOT NULL,
    date_of_birth       date        NOT NULL CHECK (
                                                EXTRACT(year FROM date_of_birth)
                                                BETWEEN EXTRACT(year FROM
                                                    CURRENT_DATE) - 100
                                                AND EXTRACT(year FROM
                                                    CURRENT_DATE)
                                                ),
    email               varchar(64) NOT NULL UNIQUE,
    phone_number        varchar(18) DEFAULT NULL,
    UNIQUE(name, surname, date_of_birth)
);

```

Technical note:

ON DELETE NO ACTION prevents deleting a user if their personal data is still stored (accidental deletion), but if by the end of the transaction personal data are deleted, the user will be deleted too.

ON UPDATE CASCADE ensures user ID updates propagate to personal data.

```
CREATE TABLE address (  
    id_user          int4          PRIMARY KEY REFERENCES "user"  
                                ON DELETE NO ACTION  
                                ON UPDATE CASCADE,  
    city             varchar(64) NOT NULL,  
    street           varchar(64) NOT NULL,  
    house_number     varchar(16) NOT NULL  
);
```

Technical note:

ON DELETE NO ACTION prevents deleting a user if their address is still stored (accidental deletion), but if by the end of the transaction the address is deleted, the user will be deleted too.

ON UPDATE CASCADE ensures address stays linked if user ID changes.

```
CREATE TABLE trainer (  
    id_user          int4          PRIMARY KEY REFERENCES "user"  
                                ON DELETE NO ACTION  
                                ON UPDATE CASCADE,  
    education        bool          NOT NULL DEFAULT false  
);
```

Technical note:

ON DELETE NO ACTION prevents deleting a user who is also a trainer (accidental deletion), but if by the end of the transaction the trainer is deleted, the user will be deleted too.

ON UPDATE CASCADE updates trainer ID if the user ID changes.

```
CREATE TABLE trainer_assistant (  
    id_trainer       int4          NOT NULL REFERENCES trainer (id_user)  
                                ON DELETE CASCADE  
                                ON UPDATE CASCADE,  
    id_assistant     int4          CHECK (id_assistant <> id_trainer)  
                                REFERENCES trainer (id_user)  
                                ON DELETE CASCADE  
                                ON UPDATE CASCADE,  
    PRIMARY KEY (id_trainer, id_assistant)  
);
```

Technical note:

ON DELETE CASCADE removes assistant or trainer relations if one is deleted.

ON UPDATE CASCADE ensures relationships remain valid after ID changes.

```

CREATE TABLE tspecialization (
    id_specialization serial      PRIMARY KEY,
    specialization_name varchar(64) NOT NULL UNIQUE
);

CREATE TABLE trainer_specialization (
    id_trainer          int4      NOT NULL REFERENCES trainer (id_user)
                                ON DELETE CASCADE
                                ON UPDATE CASCADE,
    id_specialization int4      NOT NULL REFERENCES tspecialization
                                ON DELETE CASCADE
                                ON UPDATE CASCADE,
    PRIMARY KEY (id_trainer, id_specialization)
);

```

Technical note:

ON DELETE CASCADE removes trainer's specialization when either the trainer or specialization is deleted.

ON UPDATE CASCADE updates related IDs automatically if they change.

```

CREATE TABLE training (
    id_training      serial      PRIMARY KEY,
    training_date    timestamp(0) NOT NULL CHECK (
                                                EXTRACT(year FROM training_date)
                                                BETWEEN EXTRACT(year FROM
                                                    CURRENT_DATE) - 1
                                                AND EXTRACT(year FROM
                                                    CURRENT_DATE) + 1
                                                ),
    id_sports_hall   int4      REFERENCES sports_hall
                                ON DELETE SET NULL
                                ON UPDATE CASCADE,
    type             varchar(32) NOT NULL,
    capacity         int2      NOT NULL,
    price_for_one    money      NOT NULL,
    UNIQUE (id_sports_hall, training_date)
);

```

Technical note:

ON DELETE SET NULL on id_sports_hall ensures that if the related sports hall is deleted, the training will remain, but its id_sports_hall will be set to NULL (because another hall will be found for training).

ON UPDATE CASCADE automatically updates id_sports_hall if the ID changes in the sports_hall table.

```

CREATE TABLE conduction (
    id_trainer      int4      DEFAULT NULL REFERENCES trainer (id_user)
                        ON DELETE SET DEFAULT
                        ON UPDATE CASCADE,
    id_training     int4      NOT NULL REFERENCES training
                        ON DELETE CASCADE
                        ON UPDATE CASCADE,
    PRIMARY KEY (id_trainer, id_training)
);

```

Technical note:

ON DELETE SET DEFAULT on `id_trainer` sets the trainer to the default value (NULL) if the trainer is deleted, another trainer will be found.

ON DELETE CASCADE on `id_training` deletes conduction records when the corresponding training is deleted.

ON UPDATE CASCADE ensures both `id_trainer` and `id_training` are updated if their referenced IDs change.

```

CREATE TABLE payment (
    id_payment      serial    PRIMARY KEY,
    final_price     money     NOT NULL DEFAULT 0,
    payment_method  varchar(16) NOT NULL
                        CHECK (payment_method IN ('card', 'cash')),
    payment_date    timestamp(0) NOT NULL DEFAULT CURRENT_TIMESTAMP
                        CHECK (payment_date <= CURRENT_TIMESTAMP),
    id_client       int4      NOT NULL REFERENCES client (id_user)
                        ON DELETE RESTRICT
                        ON UPDATE CASCADE,
    UNIQUE (id_client, payment_date)
);

```

Technical note:

ON DELETE RESTRICT on `id_client` prevents deleting a client if there are related payment records, because it can be useful for statistics.

ON UPDATE CASCADE updates `id_client` if the user ID changes in the `client` table.

```

CREATE TABLE reservation (
    id_reservation    serial      PRIMARY KEY,
    reservation_date  timestamp(0) NOT NULL DEFAULT CURRENT_TIMESTAMP
                        CHECK (reservation_date <= CURRENT_TIMESTAMP),
    id_client         int4        NOT NULL REFERENCES client (id_user)
                        ON DELETE RESTRICT
                        ON UPDATE CASCADE,
    status            varchar(16) NOT NULL DEFAULT 'reserved'
                        CHECK (status IN ('reserved', 'in process',
                        'payed')),
    id_training       int4        NOT NULL REFERENCES training
                        ON DELETE RESTRICT
                        ON UPDATE CASCADE,
    id_payment        int4        DEFAULT NULL REFERENCES payment
                        ON DELETE SET DEFAULT
                        ON UPDATE CASCADE,
    UNIQUE (id_client, reservation_date)
);

```

Technical note:

ON DELETE RESTRICT on `id_client` prevents deleting a client if there are related reservation records, because it can be useful for statistics.

ON DELETE RESTRICT on `id_training` prevents, because reservation can be paid.

ON DELETE SET DEFAULT on `id_payment` sets the field to the default (NULL) if the referenced payment is deleted.

ON UPDATE CASCADE updates all foreign keys if the referenced IDs are changed.

Python script for generating data:

`writer.py` defines the `DatabaseWriter` class, which provides inserting and updating methods in the sports complex database. The implementation uses the `psycopg2` library.

Constructor:

- `DatabaseWriter(dbname, user, password, host, port):`

- `dbname` – the name of the database.
- `user` – the username.
- `password` – the password.
- `host` – the address of the database server
- `port` – the port.

Methods:

- `_connect():` used to connect to the database.

```

def _connect(self):
    self.conn = psycopg2.connect(**self.conn_params)
    self.cursor = self.conn.cursor()

```

- `_disconnect()`: used to disconnect from the database

```
def _disconnect(self):
    self.cursor.close()
    self.conn.close()
```

- `_execute_insert(query, values, error_msg)`: A method to execute insert queries with error handling.

```
def _execute_insert(self, query, values, error_msg):
    try:
        self.cursor.execute(sql.SQL(query), values)
        self.num_of_insert += 1
        if(self.num_of_insert > 500):
            self.conn.commit()
            self.num_of_insert = 0
    except Exception as e:
        print(f"{error_msg}: {e}")
```

- `insert_sports_hall(hall_name, size)`: Inserts a sports hall.
- `insert_hspecialization(specialization_name)`: Adds a new hall specialization.
- `insert_hall_specialization(id_sports_hall, id_specialization)`: Links a sports hall to a specialization.
- `insert_user(login, password)`: Creates a new user.
- `insert_client(id_user)`: Registers a new client.
- `insert_preference_in_sport(preference_name)`: Adds a sport preference.
- `insert_client_preference(id_client, id_preference)`: Links a client to their sport preference.
- `insert_personal_data(id_user, name, surname, date_of_birth, email, phone_number)`: Stores personal data.
- `insert_address(id_user, city, street, house_number)`: Adds address.
- `insert_trainer(id_user, education)`: Registers a trainer.
- `insert_trainer_assistant(id_trainer, id_assistant)`: Connects a trainer with their assistant.
- `insert_tspecialization(specialization_name)`: Adds a trainer specialization.
- `insert_trainer_specialization(id_trainer, id_specialization)`: Links a trainer with a specialization.
- `insert_training(training_date, id_sports_hall, type, capacity, price_for_one)`: Adds a training.

- `insert_conduction(id_trainer, id_training)`: Links a trainer with a training.
- `insert_payment(final_price, payment_method, payment_date, id_client)`: Adds a payment.
- `insert_reservation(reservation_date, id_client, status, id_training, id_payment)`: Makes a reservation for a training.

```
def insert_reservation(self, reservation_date, id_client, status,
    id_training, id_payment):
    self._execute_insert("""
        INSERT INTO reservation (reservation_date, id_client, status,
            id_training, id_payment)
        VALUES (%s, %s, %s, %s, %s)
    """, (reservation_date, id_client, status, id_training,
        id_payment), "Error inserting into reservation")
```

- `update_reservation(id_reservation, id_payment)`: Updates a reservation.

```
def update_reservation(self, id_reservation, id_payment):
    self._execute_insert("""
        UPDATE reservation
        SET id_payment = %s
        WHERE id_reservation = %s
    """, (id_payment, id_reservation), "Error updating reservation")
```

Note:

All insert functions are like `insert_reservation`.

`my_faker.py` defines the `MyFaker` class, which provides methods for generating data for the sports complex database. The implementation uses the `Faker` library.

Constructor:

- `MyFaker(db, user, password, host, port, num_users, num_trainers, num_halls):`
 - `db` – the name of the database.
 - `user` – the username.
 - `password` – the password.
 - `host` – the address of the database server.
 - `port` – the port.
 - `num_users` – the number of users to generate.
 - `num_trainers` – the number of trainers to generate.
 - `num_halls` – the number of sports halls to generate.

Methods:

- `gen_password():` generates password.
- `gen_name(gender):` generates Czech name depending on gender.
- `gen_surname(gender):` generates Czech surname depending on gender.
- `gen_date_of_birth():` generates date of birth.
- `gen_name_surname_dob_UQ():` generates unique combination - name, surname, date of birth.
- `gen_email_UQ():` generates unique email.
- `gen_phone_number():` generates Czech phone number.
- `gen_city():` generates Czech city.
- `gen_street():` generates Czech street.
- `gen_house_number():` generates house number.
- `gen_fake_user_inf():` generates `self.num_of_user` fake users, their personal data, addresses, etc.
- `separate_trainers_clients():` chooses `self.num_of_trainers` trainers from users.
- `gen_tspecialization():` generates all possible trainer specializations.
- `gen_trainer_specialization():` generates specializations for all trainers.
- `gen_preference_in_sport():` generates all possible client preferences in sport.
- `gen_client_preference():` generates preferences for all clients.

- `gen_trainer_assistant()`: chooses assistants for some trainers.
- `gen_hall_name_UQ()`: generates unique hall name.
- `gen_sports_hall()`: generates `self.num_halls` sports halls.
- `gen_hspecialization()`: generates all possible hall specializations.
- `gen_hall_specialization()`: generates specializations for all halls.
- `gen_dates()`: generates time slots for trainings on 15 days back and 15 days in future (9 per day).
- `find_trainers(time_slot, training_spec)`: finds trainer for training.
- `gen_reservation_for_training(date_id, price, training_id, capacity, reservation_)`: generates reservations for training.
- `gen_payments()`: generates payments depending on user reservations.
- `gen_training()`: generates trainings.

Note:

The implementation is below.

```

1  from faker import Faker
2  import random
3  from datetime import *
4  from unicode import unicode
5
6  from writer import DatabaseWriter
7
8
9  class MyFaker:
10     usage
11     SPECIALIZATIONS = ['volleyball', 'basketball', 'football', 'tennis', 'judo', 'karate', 'MMA', 'sambo',
12                        'yoga', 'pilates', 'fitness', 'badminton', 'handball', 'futsal', 'squash', 'wrestling',
13                        'gymnastics', 'aerobics', 'zumba', 'piloxing', 'bodybuilding', 'kickboxing', 'table tennis',
14                        'dance', 'cheerleading', 'martial arts', 'taekwondo', 'fencing', 'rowing', 'indoor cycling',
15                        'crossfit', 'weightlifting', 'boxing', 'dodgeball', 'floorball', 'parquet ball', 'indoor archery']
16     SPEC_LEN = len(SPECIALIZATIONS)
17
18     PREFERENCES = ['team sports', 'solo sports', 'with ball', 'without ball', 'indoor sports', 'combat sports',
19                   'fitness and strength training', 'flexibility and balance', 'endurance sports', 'extreme sports',
20                   'aerobic exercises', 'mind-body sports', 'competitive sports']
21     PREF_LEN = len(PREFERENCES)
22
23     PRICES_FOR_ONE = {1 : (250, 500),
24                       2 : (200, 400),
25                       3 : (150, 300)}
26
27     MAX_CAPACITIES = {1 : 14,
28                       2 : 28,
29                       3 : 42}
30
31     TRAINING_TYPES = ['with trainer', 'without trainer']
32     TYPES_LEN = len(TRAINING_TYPES)
33
34     RESERVATIONS_STATUS = ['reserved', 'in process', 'payed']
35
36     PAYMENTS_METHODS = ['cash', 'card']
37
38     def __init__(self, db, user, password, host, port,
39                  num_users, num_trainers, num_halls):
40         self.num_users = num_users
41         self.num_trainers = num_trainers
42         self.num_halls = num_halls
43
44         # fakers
45         self.fake_global = Faker()
46         self.fake_cz = Faker('cz_CZ')
47
48         # database writer
49         self.writer = DatabaseWriter(db, user, password, host, port)
50
51         # PRIMARY KEYS
52         self.trainer_assistant_PK = [[] for _ in range(self.num_trainers)]
53         self.trainer_spec_PK = [[] for _ in range(self.num_trainers)]
54         self.client_pref_PK = [[] for _ in range(self.num_users)]
55         self.hall_spec_PK = [[] for _ in range(self.num_halls)]
56
57         # UNIQUE
58         self.login_UQ = set()
59         self.name_sur_dob_UQ = set()
60         self.email_UQ = set()
61         self.hall_name_UQ = set()

```

```

60     self.hall_name_UQ = set()
61     self.idcl_paymdate_UQ = set()
62
63     self.spec_trainer = [[] for _ in range(self.SPEC_LEN)]
64
65     self.hall_sizes = []
66
67     self.dates = self.gen_dates()
68     self.days_num = 30
69     self.one_day_num_train = 9
70     self.total_slots = self.one_day_num_train * self.days_num
71
72     self.trainer_occupancy = [[False for _ in range(self.total_slots)] for _ in range(self.num_trainers)]
73     self.clients_reservations = set()
74
75     self.clients_reservations_for_pay = [[] for _ in range(self.num_users)]
76
77     # Function to generate password
78     def gen_password(self): 1usage
79         return self.fake_cz.password()
80
81     # Function to generate Czech name depends on gender
82     def gen_name(self, gender): 1usage
83         if gender == 'male':
84             return self.fake_cz.first_name_male()
85         else:
86             return self.fake_cz.first_name_female()
87
88     # Function to generate Czech surname depends on gender
89     def gen_surname(self, gender): 1usage
90         if gender == 'male':
91             return self.fake_cz.last_name_male()
92         else:
93             return self.fake_cz.last_name_female()
94
95     # Function to generate date of birth (interval [-99 years; -18 years] from cur date)
96     def gen_date_of_birth(self): 1usage
97         return self.fake_cz.date_between(start_date="-99y", end_date="-18y")
98
99     # Function to generate unique combination of name, surname, date of birth
100     def gen_name_surname_dob_UQ(self): 1usage
101         gender = random.choice(['male', 'female'])
102
103         while True:
104             name = self.gen_name(gender)
105             surname = self.gen_surname(gender)
106             date_of_birth = self.gen_date_of_birth()
107
108             key = (name, surname, date_of_birth)
109             if key not in self.name_sur_dob_UQ:
110                 self.name_sur_dob_UQ.add(key)
111             return key
112
113     # Function to generate unique email
114     def gen_email_UQ(self):
115         while True:
116             email = self.fake_cz.email()
117             if email not in self.email_UQ:

```

```

117         if email not in self.email_UQ:
118             self.email_UQ.add(email)
119         return email
120
121     # Function to generate phone number
122     def gen_phone_number(self): 1usage
123         phone_number = self.fake_cz.phone_number()
124         if phone_number[0] == '0':
125             phone_number = '+' + phone_number[2:]
126         return phone_number
127
128     # Function to generate Czech city
129     def gen_city(self): 1usage
130         return self.fake_cz.city()
131
132     # Function to generate Czech street
133     def gen_street(self): 1usage
134         return self.fake_cz.street_name()
135
136     # Function to generate house number
137     def gen_house_number(self): 1usage
138         return random.randint(a=1, b=200)
139
140     # Function to generate fake users information
141     def gen_fake_user_inf(self): 1usage
142         for user_id in range(1, self.num_users + 1):
143             name, surname, date_of_birth = self.gen_name_surname_dob_UQ()
144             login = unicode(name + surname).lower() + str(date_of_birth).replace(__old: '-', __new: '')
145             password = self.gen_password()
146             email = login + '@gmail.com'
147             phone_number = self.gen_phone_number()
148             city = self.gen_city()
149             street = self.gen_street()
150             house_number = self.gen_house_number()
151
152             self.writer.insert_user(login, password)
153             self.writer.insert_personal_data(user_id, name, surname, date_of_birth, email, phone_number)
154             self.writer.insert_address(user_id, city, street, house_number)
155
156     # Function to separate trainers and clients (first self.num_trainers users are trainers)
157     def separate_trainers_clients(self): 1usage
158         cur_num_tr = 0
159         for user_id in range(1, self.num_users + 1):
160             if cur_num_tr < self.num_trainers:
161                 education = random.choice(['TRUE', 'FALSE'])
162                 self.writer.insert_trainer(user_id, education)
163                 cur_num_tr += 1
164             else:
165                 self.writer.insert_client(user_id)
166
167     # Function to generate possible trainer specializations
168     def gen_tspecialization(self): 1usage
169         for spec in self.SPECIALIZATIONS:
170             self.writer.insert_tspecialization(spec)
171
172     # Function to gen specializations for all trainers
173     def gen_trainer_specialization(self): 1usage
174         for trainer_id in range(1, self.num_trainers):

```

```

174         for trainer_id in range(self.num_trainers):
175             num_specs = random.randint(a=1, b=6)
176             spec_ids = random.sample(range(self.SPEC_LEN), num_specs)
177
178             for spec_id in spec_ids:
179                 self.writer.insert_trainer_specialization(trainer_id + 1, spec_id + 1)
180                 self.trainer_spec_PK[trainer_id].append(spec_id)
181                 self.spec_trainer[spec_id].append(trainer_id)
182
183     # Function to generate all possible preferences in sports
184     def gen_preference_in_sport(self): 1usage
185         for pref in self.PREFERENCES:
186             self.writer.insert_preference_in_sport(pref)
187
188     # Function to generate preferences for all clients
189     def gen_client_preference(self): 1usage
190         for client_id in range(self.num_trainers, self.num_users):
191             num_prefs = random.randint(a=0, b=5)
192             if num_prefs == 0:
193                 continue
194
195             pref_ids = random.sample(range(self.PREF_LEN), min(num_prefs, self.PREF_LEN))
196
197             for pref_id in pref_ids:
198                 if pref_id not in self.client_pref_PK[client_id]:
199                     self.writer.insert_client_preference(client_id + 1, pref_id + 1)
200                     self.client_pref_PK[client_id].append(pref_id)
201
202     # Function to generate assistants for trainers
203     def gen_trainer_assistant(self): 1usage
204         for trainer_id in range(self.num_trainers):
205             num_assistants = random.randint(a=0, b=5)
206             if num_assistants == 0:
207                 continue
208
209             possible_assistants = []
210             trainer_specs = set(self.trainer_spec_PK[trainer_id])
211
212             for assistant_id in range(self.num_trainers):
213                 if assistant_id == trainer_id:
214                     continue
215                 assistant_specs = set(self.trainer_spec_PK[assistant_id])
216                 if trainer_specs & assistant_specs:
217                     possible_assistants.append(assistant_id)
218
219             if not possible_assistants:
220                 continue
221
222             assistants = random.sample(possible_assistants, min(num_assistants, len(possible_assistants)))
223
224             for assistant_id in assistants:
225                 if assistant_id not in self.trainer_assistant_PK[trainer_id]:
226                     self.writer.insert_trainer_assistant(trainer_id + 1, assistant_id + 1)
227                     self.trainer_assistant_PK[trainer_id].append(assistant_id)
228
229     # Function to generate unique halls name
230     def gen_hall_name_UQ(self): 1usage
231         while True:

```

```

231         while True:
232             hall_name = self.fake_global.first_name_female()
233             if hall_name not in self.hall_name_UQ:
234                 self.hall_name_UQ.add(hall_name)
235             return hall_name
236
237     # Function to generate sport halls
238     def gen_sports_hall(self): 1usage
239         for _ in range(self.num_halls):
240             size = random.randint(a=1, b=3)
241             hall_name = self.gen_hall_name_UQ()
242             self.hall_sizes.append(size)
243             self.writer.insert_sports_hall(hall_name, size)
244
245     # Function to generate all possible hall specializations
246     def gen_hspecialization(self): 1usage
247         for spec in self.SPECIALIZATIONS:
248             self.writer.insert_hspecialization(spec)
249
250     # Function to generate specializations for all halls
251     def gen_hall_specialization(self): 1usage
252         for hall_id in range(self.num_halls):
253             num_specializations = random.randint(a=1, b=6)
254
255             spec_ids = random.sample(range(self.SPEC_LEN), num_specializations)
256
257             for spec_id in spec_ids:
258                 if spec_id not in self.hall_spec_PK[hall_id]:
259                     self.writer.insert_hall_specialization(hall_id + 1, spec_id + 1)
260                     self.hall_spec_PK[hall_id].append(spec_id)
261
262     # Function to generate dates slots on 15 days back and 15 days in future
263     def gen_dates(self): 1usage
264         past_days = 15
265         future_days = 15
266         interval = timedelta(minutes=90)
267         slots = []
268
269         start_date = datetime.today().date() - timedelta(days=past_days)
270         total_days = past_days + future_days + 1
271
272         for day_offset in range(total_days):
273             current_date = start_date + timedelta(days=day_offset)
274             time = timedelta(hours=9)
275             while time <= timedelta(hours=22, minutes=30):
276                 dt = datetime.combine(current_date, datetime.min.time()) + time
277                 slots.append(dt)
278                 time += interval
279
280         return slots
281
282     # Function to find trainers for training
283     def find_trainers(self, time_slot, training_spec): 1usage
284         trainers = []
285         for trainer_id in self.spec_trainer[training_spec]:
286             if self.trainer_occupancy[trainer_id][time_slot] == False:
287                 trainers.append(trainer_id)
288                 assist_num = min(1, random.randint(a=0, b=100/self.trainer_assistant_PK[trainer_id]))

```

```

288         assist_num = min(1, random.randint(0, len(self.trainer_assistant_PK[trainer_id])))
289         if assist_num > 0:
290             for assist_id in self.trainer_assistant_PK[trainer_id]:
291                 if training_spec in self.trainer_spec_PK[assist_id] and self.trainer_occupancy[assist_id][time_slot] == False\
292                     and assist_id != trainer_id:
293                     trainers.append(assist_id)
294                     break
295             return trainers
296         return None
297
298     # Function to generate reservations for training
299     def gen_reservation_for_training(self, date_id, price, training_id, capacity, reservation_id): 1usage
300         for _ in range(capacity):
301             client_id = random.randint(self.num_trainers, self.num_users - 1)
302             reservation_date = self.dates[date_id] if self.dates[date_id] < datetime.today() else datetime.today()
303             delta = timedelta(days=random.randint(0, 1, 9), hours=random.randint(0, 23), minutes = random.randint(0, 59))
304             reservation_date += delta
305             key = (client_id, reservation_date.month, reservation_date.day, reservation_date.hour, reservation_date.minute)
306             if key in self.clients_reservations:
307                 continue
308
309             reservation_status = None
310             if self.dates[date_id] < datetime.today():
311                 reservation_status = 'payed'
312             else:
313                 reservation_status = random.choice(self.RESERVATIONS_STATUS)
314
315             if reservation_status != 'reserved':
316                 new_client_id = client_id
317                 if random.random() >= 0.5:
318                     new_client_id = random.randint(num_trainers, self.num_users - 1)
319                 self.clients_reservations_for_pay[new_client_id].append((reservation_id, price, reservation_date, date_id))
320
321             self.writer.insert_reservation(reservation_date, client_id + 1, reservation_status, training_id, id_payment=None)
322             reservation_id += 1
323             self.clients_reservations.add(key)
324
325     # Function to generate payments
326     def gen_payments(self): 1usage
327         payment_id = 1
328         for client_id in range(self.num_trainers, self.num_users):
329             len_res_for_pay = len(self.clients_reservations_for_pay[client_id])
330             for res_for_pay in range(len_res_for_pay):
331                 reservations = []
332                 reservation_id, price, reservation_date, date_id = self.clients_reservations_for_pay[client_id][res_for_pay]
333                 reservations.append(reservation_id)
334
335                 max_res_date = reservation_date
336                 min_train_date = self.dates[date_id]
337
338                 if res_for_pay < len_res_for_pay - 1:
339                     next_reservation_id, next_price, next_reservation_date, next_date_id = self.clients_reservations_for_pay[client_id][res_for_pay + 1]
340                     if reservation_date < self.dates[next_date_id] and next_reservation_date < self.dates[date_id]:
341                         reservations.append(next_reservation_id)
342                         max_res_date = max(max_res_date, next_reservation_date)
343                         min_train_date = min(min_train_date, self.dates[next_date_id])
344                     price += next_price
345                 res_for_pay += 1

```



```

342         max_res_date = max(max_res_date, next_reservation_date)
343         min_train_date = min(min_train_date, self.dates[next_date_id])
344         price += next_price
345         res_for_pay += 1
346
347         payment_method = random.choice(self.PAYMENTS_METHODS)
348         payment_date = datetime.fromtimestamp((max_res_date.timestamp() + min_train_date.timestamp()) / 2)
349         if payment_date >= datetime.today():
350             payment_date = datetime.fromtimestamp((max_res_date.timestamp() + datetime.today().timestamp()) / 2)
351
352         while (client_id, payment_date.month, payment_date.day, payment_date.hour, payment_date.minute) in self.idcl_paymdate:
353             payment_date += timedelta(minutes=1)
354
355         self.writer.insert_payment(price, payment_method, payment_date, client_id + 1)
356
357         for reserv_id in reservations:
358             self.writer.update_reservation(reserv_id, payment_id)
359
360         self.idcl_paymdate_UQ.add((client_id, payment_date.month, payment_date.day, payment_date.hour, payment_date.minute))
361         payment_id += 1
362
363     # Function to generate trainings
364     def gen_training(self):
365         training_id = 1
366         reservation_id = 1
367         for time_slot in range(self.one_day_num_train * self.days_num):
368             for hall_id in range(self.num_halls):
369                 training_type = random.choice(self.TRAINING_TYPES)
370                 training_spec = random.choice(self.hall_spec_PK[hall_id])
371                 hall_size = self.hall_sizes[hall_id]
372                 capacity = self.MAX_CAPACITIES[hall_size]
373                 price_for_one = None
374                 if training_type == 'without trainer':
375                     price_for_one = self.PRICES_FOR_ONE[hall_size][0]
376                     self.writer.insert_training(self.dates[time_slot], hall_id + 1, self.SPECIALIZATIONS[training_spec] + " " + training_type)
377                 else:
378                     trainers = self.find_trainers(time_slot, training_spec)
379                     if trainers is None:
380                         training_type = 'without trainer'
381                         price_for_one = self.PRICES_FOR_ONE[hall_size][0]
382                         self.writer.insert_training(self.dates[time_slot], hall_id + 1, self.SPECIALIZATIONS[training_spec] + " " + training_type)
383                     else:
384                         price_for_one = self.PRICES_FOR_ONE[hall_size][1]
385                         self.writer.insert_training(self.dates[time_slot], hall_id + 1, self.SPECIALIZATIONS[training_spec] + " " + training_type)
386                         for trainer_id in trainers:
387                             self.trainer_occupancy[trainer_id][time_slot] = True
388                             self.writer.insert_conduction(trainer_id + 1, training_id)
389                 self.gen_reservation_for_training(time_slot, price_for_one, training_id, capacity, reservation_id)
390                 reservation_id += capacity
391             training_id += 1
392

```

```

393 if __name__ == '__main__':
394     num_users = 32000
395     num_trainers = 100
396     num_halls = 10
397
398     faker = MyFaker(db: 'kolesole', user: 'username', password: 'password', host: 'slon.felk.cvut.cz', port: '5432',
399                    num_users, num_trainers, num_halls)
400     # faker = MyFaker('postgres', 'postgres', 'Razryvakal', 'localhost', '5432',
401                     # num_users, num_trainers, num_halls)
402     faker.writer.connect()
403     faker.gen_fake_user_inf()
404     print('fake users generated')
405     faker.separate_trainers_clients()
406     print('trainers and clients separated')
407     faker.gen_tspecialization()
408     print('tspecializations generated')
409     faker.gen_trainer_specialization()
410     print('trainer_specializations generated')
411     faker.gen_preference_in_sport()
412     print('preference_in_sport generated')
413     faker.gen_client_preference()
414     print('client_preference generated')
415     faker.gen_trainer_assistant()
416     print('trainer_assistant generated')
417     faker.gen_sports_hall()
418     print('sports_hall generated')
419     faker.gen_hspecialization()
420     print('hspecialization generated')
421     faker.gen_hall_specialization()
422     print('hall_specialization generated')
423     faker.gen_training()
424     print('trainings generated')
425     faker.gen_payments()
426     print('payments generated')
427     faker.writer.conn.commit()
428     faker.writer.disconnect()
429

```

Note:

32000 users, 100 trainers, 10 sports halls, 2700 trainings, etc. are now in the database.

Queries:

```
GRANT CONNECT, TEMP ON DATABASE kolesole TO zhyltser;
GRANT ALL ON ALL TABLES IN SCHEMA public TO zhyltser;
GRANT ALL ON ALL SEQUENCES IN SCHEMA public TO zhyltser;
GRANT ALL ON ALL FUNCTIONS IN SCHEMA public TO zhyltser;
```

```
ALTER DEFAULT PRIVILEGES IN SCHEMA public
    GRANT ALL ON TABLES TO zhyltser;
```

```
ALTER DEFAULT PRIVILEGES IN SCHEMA public
    GRANT ALL ON SEQUENCES TO zhyltser;
```

Note:

Grants full access rights to the partner zhyltser for the database.

```
GRANT CONNECT, TEMP ON DATABASE kolesole TO prokoyul;
GRANT ALL ON ALL TABLES IN SCHEMA public TO prokoyul;
GRANT ALL ON ALL SEQUENCES IN SCHEMA public TO prokoyul;
GRANT ALL ON ALL FUNCTIONS IN SCHEMA public TO prokoyul;
```

```
ALTER DEFAULT PRIVILEGES IN SCHEMA public
    GRANT ALL ON TABLES TO prokoyul;
```

```
ALTER DEFAULT PRIVILEGES IN SCHEMA public
    GRANT ALL ON SEQUENCES TO prokoyul;
```

Note:

Grants full access rights to the tutor prokoyul for the database.

```
SELECT DISTINCT name || ' ' || surname AS trainer_name,  
                specialization_name  
FROM trainer t  
    JOIN personal_data USING(id_user)  
    JOIN trainer_specialization tr_spec ON t.id_user = tr_spec.id_trainer  
    JOIN tspecialization tspec ON tr_spec.id_specialization =  
        tspec.id_specialization  
LIMIT 10  
;
```

Note:

Returns the first 10 trainer names and one of their specializations by joining trainer, personal_data, trainer_specialization, and tspecialization.

Output:

	☐ trainer_name ▼	⚡	☐ specialization_name ▼	⚡
1	Božena Bártová		volleyball	
2	Jana Navrátilová		zumba	
3	Renáta Němcová		tennis	
4	Dalibor Král		judo	
5	Adam Dušek		football	
6	Radim Sýkora		taekwondo	
7	Pavlina Žáková		squash	
8	Pavlina Žáková		indoor archery	
9	Dana Němcová		indoor cycling	
10	Jiřina Procházková		futsal	

```
SELECT city,
       count(*) AS num_of_users
FROM "user"
     JOIN address addr USING(id_user)
GROUP BY city
ORDER BY num_of_users DESC
LIMIT 10
;
```

Note:

Counts the number of users in each city by using `GROUP BY city`. Sorts the cities by the number of users in descending order and returns only the top 10 results and their counts.

Output:

	city	num_of_users
1	Zruč nad Sázavou	89
2	Netolice	87
3	Oslavany	85
4	Všeruby	85
5	Rejštejn	85
6	Hulín	84
7	Zákupy	84
8	Ronov nad Doubravou	83
9	Újezd u Brna	82
10	Polička	82

```

SELECT pd_trainer.name || ' ' || pd_trainer.surname AS trainer_name,
       pd_assistant.name || ' ' || pd_assistant.surname AS assistant_name
FROM trainer t
     JOIN personal_data pd_trainer USING(id_user)
     LEFT JOIN trainer_assistant train_assist ON t.id_user =
         train_assist.id_trainer
     LEFT JOIN personal_data pd_assistant ON train_assist.id_assistant =
         pd_assistant.id_user
ORDER BY assistant_name DESC
LIMIT 30
;

```

Note:

Shows the first 30 trainer–assistant pairs, including trainers without an assistant.

Output:

	☐ trainer_name ▼	÷	☐ assistant_name ▼	÷
1	Žofie Ježková		<null>	
2	Richard Kopecký		<null>	
3	Jiřina Procházková		<null>	
4	Peter Fišer		<null>	
5	Klára Křížová		<null>	
6	Božena Bártová		<null>	
7	Tomáš Staněk		<null>	
8	Hubert Hruška		<null>	
9	Igor Mach		<null>	
10	Kristýna Pavlíková		<null>	
11	Alena Mašková		<null>	
12	Jana Navrátilová		<null>	
13	Milan Kopecký		<null>	
14	Marcela Stejskalová		<null>	
15	Alena Navrátilová		<null>	
16	Nela Beranová		<null>	
17	Ivo Kašpar		<null>	
18	Slavěna Kadlecová		<null>	
19	Olivie Němcová		<null>	
20	Věra Fialová		<null>	
21	Vratislav Hrubý		<null>	
22	Erika Šimková		<null>	
23	Františka Pospíšilová		<null>	
24	Slavěna Novotná		<null>	
25	Dalibor Král		<null>	
26	Anastázie Vávrová		Žofie Růžicková	
27	Pavel Doležal		Vlasta Mašek	
28	Radim Sýkora		Vlasta Mašek	
29	Radovan Šimek		Vlasta Mašek	
30	Kamila Vaňková		Vladislav Sýkora	

```
SELECT DISTINCT name || ' ' || surname AS trainer_name,  
               email  
FROM trainer t  
   JOIN personal_data USING(id_user)  
   JOIN trainer_specialization tr_spec ON t.id_user = tr_spec.id_trainer  
   JOIN tspecialization tspec USING(id_specialization)  
WHERE tspec.specialization_name IN ('volleyball')  
;
```

Note:

Shows trainers names and emails whose specializations include volleyball.

Output:

	trainer_name	email
1	Anastázie Vávrová	anastazievavrova19471029@gmail.com
2	Božena Bártová	bozenabartova20110429@gmail.com
3	Dalibor Král	daliborkral19600704@gmail.com
4	Eva Dušková	evaduskova19960702@gmail.com
5	Olivie Němcová	olivienemcova19620106@gmail.com
6	Petra Marešová	petramaresova19840314@gmail.com
7	Slavěna Kadlecová	slavenakadlecova20121016@gmail.com

```
SELECT name || ' ' || surname AS trainer_name,  
       count(*) AS num_of_conducted_trainings  
FROM trainer t  
     JOIN personal_data pd USING(id_user)  
     JOIN conduction c ON c.id_trainer = t.id_user  
GROUP BY trainer_name  
ORDER BY num_of_conducted_trainings  
LIMIT 10  
;
```

Note:

Returns the 10 trainers (full name) with the fewest conducted and future trainings, sorted in ascending order.

Output:

	trainer_name	num_of_conducted_trainings
1	Leoš Dostál	1
2	Judita Beránková	1
3	Pavel Doležal	1
4	Radim Jelínek	1
5	Horymír Müller	2
6	Martin Slavík	2
7	Kristýna Pavlíková	3
8	Miluše Holubová	4
9	Luděk Moravec	7
10	Božena Bártová	8

```

SELECT login,
       count(r.id_reservation) AS num_of_reservations,
       sum(p.final_price) AS sum_of_payments
FROM "user"
     JOIN client c USING(id_user)
     JOIN reservation r ON r.id_client = c.id_user
     JOIN payment p ON p.id_client = c.id_user
GROUP BY login
HAVING sum(p.final_price::numeric) < 5000
ORDER BY num_of_reservations DESC, sum_of_payments
LIMIT 10
;

```

Note:

Returns top 10 clients whose sum of payments is less than 5000 and shows their reservation counts.

Output:

	login	num_of_rese...	sum_of_paym...
1	aloisslavik20010527	21	\$3,500.00
2	romanmarek19341225	20	\$3,000.00
3	nataliejarosova1952...	20	\$4,000.00
4	milenasykorova19311...	20	\$4,750.00
5	alesmares19300104	20	\$4,750.00
6	kamilanovakova19931...	20	\$4,750.00
7	johanastastna195308...	18	\$3,600.00
8	oliviekuceroval19980...	18	\$3,600.00
9	bohumilmatousek1948...	18	\$3,600.00
10	romanvavra20080208	18	\$3,900.00

```

SELECT hall_name,
       count(CASE WHEN tr.type LIKE '% with trainer' THEN 1 END) AS
         num_of_trainings_with_trainer,
       count(CASE WHEN tr.type LIKE '% without trainer' THEN 1 END) AS
         num_of_trainings_without_trainer
FROM sports_hall
     JOIN training tr USING(id_sports_hall)
GROUP BY hall_name
ORDER BY num_of_trainings_with_trainer DESC,
         num_of_trainings_without_trainer DESC
LIMIT 3
;

```

Note:

Counts trainings with and without a trainer per hall; returns top 3 by number of trainings with a trainer.

Output:

	hall_name	num_of_trainings_with_trainer	num_of_trainings_without_trainer
1	Haley	148	122
2	Kelly	148	122
3	Brenda	147	123

```

SELECT hall_name,
       count(CASE WHEN tr.type LIKE '% with trainer' THEN 1 END) AS
         num_of_trainings_with_trainer,
       count(CASE WHEN tr.type LIKE '% without trainer' THEN 1 END) AS
         num_of_trainings_without_trainer
FROM sports_hall
     JOIN training tr USING(id_sports_hall)
GROUP BY hall_name
ORDER BY num_of_trainings_with_trainer DESC,
         num_of_trainings_without_trainer DESC
OFFSET 3
;

```

Note:

Same as above, but skips top 3 to return the next halls.

Output:

	hall_name	num_of_trainings_with_trainer	num_of_trainings_without_trainer
1	Tamara	146	124
2	Erin	143	127
3	Catherine	138	132
4	Karen	138	132
5	Jennifer	126	144
6	Cynthia	122	148
7	Julia	112	158

```

SELECT name || ' ' || surname AS client_name
  FROM client c
    JOIN personal_data USING(id_user)
    JOIN client_preference cl_pref ON cl_pref.id_client = c.id_user
    JOIN preference_in_sport pref USING(id_preference)
  WHERE preference_name = 'combat sports'

UNION

SELECT name || ' ' || surname AS client_name
  FROM client c
    JOIN personal_data USING(id_user)
    JOIN client_preference cl_pref ON cl_pref.id_client = c.id_user
    JOIN preference_in_sport pref USING(id_preference)
  WHERE preference_name = 'competitive sports'
LIMIT 10
;

```

Note:

Returns the first ten clients names who prefer either combat sports or competitive sports, removing duplicates.

Output:

	client_name
1	Tadeáš Kříž
2	Františka Brožová
3	Ján Slavík
4	Kristýna Kopecká
5	Blanka Blažková
6	Ferdinand Horáček
7	Ján Toman
8	Drahomíra Beranová
9	Sabina Veselá
10	Vlastimil Svoboda

```
SELECT name || ' ' || surname AS name_surname
      FROM client
      JOIN personal_data USING(id_user)

INTERSECT

SELECT name || ' ' || surname AS name_surname
      FROM trainer
      JOIN personal_data USING(id_user)
ORDER BY name_surname
LIMIT 10
;
```

Note:

Returns the first ten names and surnames that appear both among clients and among trainers.

Output:

	name_surname
1	Agáta Slavíková
2	Alena Mašková
3	Alexandr Dostál
4	Alice Šmídová
5	Andrea Švecová
6	Bedřich Holub
7	Bohumil Bílek
8	Bohumír Slavík
9	Ctírad Ježek
10	Dalibor Havlíček

```

SELECT login AS client_login,
       name || ' ' || surname AS client_name
FROM client
     JOIN "user" USING(id_user)
     JOIN personal_data USING(id_user)

EXCEPT

SELECT login AS client_login,
       name || ' ' || surname AS client_name
FROM client AS c
     JOIN "user" USING(id_user)
     JOIN personal_data USING(id_user)
     JOIN client_preference AS cl_pref ON cl_pref.id_client = c.id_user
LIMIT 10
;

```

Note:

Returns the first ten clients logins and names who have no preferences assigned.

Output:

	client_login	client_name
1	zorabartova20110220	Zora Bártová
2	artursykora19720724	Artur Sýkora
3	nikolbilkova19661208	Nikol Bílková
4	denisakuceroval9880927	Denisa Kučerová
5	anetajandova19280406	Aneta Jandová
6	bohuslavmoravec20120122	Bohuslav Moravec
7	martinapospisilova20030205	Martina Pospíšilová
8	monikapetrova19700910	Monika Petrová
9	alicehruskova20120713	Alice Hrušková
10	miroslavurban19511224	Miroslav Urban

```

SELECT name || ' ' || surname AS client_name,
       count (id_reservation) AS num_of_reservations
FROM client c
     JOIN personal_data USING(id_user)
     JOIN reservation r ON c.id_user = r.id_client
GROUP BY client_name
HAVING count (id_reservation) > (
    SELECT AVG (res_count)
    FROM (
        SELECT count (id_reservation) AS res_count
        FROM reservation
        GROUP BY id_client
    ) AS num_of_reservations)
ORDER BY num_of_reservations DESC
LIMIT 10
;

```

Note:

Returns the first ten clients who have more reservations than the average number of reservations per client.

Output:

	client_name	num_of_reservations
1	Tadeáš Čermák	20
2	Oldřich Tichý	19
3	Věra Krausová	19
4	Květoslava Čermáková	19
5	Vladimíra Kovářová	18
6	René Mach	18
7	Šimon Kolář	18
8	Štěpánka Bednářová	18
9	Jakub Pokorný	18
10	Viktorie Hrubá	18

CP-4 Advanced Database Technologies

Transaction:

- **Idea:**

The main idea of this transaction is to update the status of a reservation, which corresponds to adding a payment for that reservation.

- **Conflict:**

Our database contains users who have reservations that can be paid for. A reservation belongs to one user, but it can be paid for by a different user. The payment process is designed so that it first checks whether the reservation has not been paid yet (i.e., the status is **'reserved'**), and only then creates a new payment entry in the **payment** table. A conflict may occur if two users attempt to pay for the same reservation at the same time. Both may see the reservation status as **'reserved'**, and each may proceed to create a new payment. As a result, two payments could be created for the same reservation, leading to a double payment.

- **Isolation level:**

In this situation, the best-suited isolation level is **SERIALIZABLE**, because it ensures full isolation between concurrent transactions. By using **SERIALIZABLE**, we ensure that only one transaction can successfully verify the reservation's unpaid status and proceed to create the payment, the other will be rolled back or retried.

- **Function:**

The function **update_reservation_status** handles the logic for processing a reservation payment. It first checks the current status of the reservation and retrieves the price. If the reservation is already paid or the payment method is invalid, an exception is raised. Otherwise, it inserts a new payment into the **payment** table and updates the reservation's status to **'payed'** by assigning the corresponding **id_payment**.

```

CREATE FUNCTION update_reservation_status(id_reserv INTEGER, id_user
    INTEGER, pay_method varchar(16), pay_date TIMESTAMP)
    RETURNS VOID AS $$
DECLARE
    reserv_status VARCHAR(16);
    price MONEY;
    id_paym INTEGER;
BEGIN
    reserv_status := (SELECT status
                      FROM reservation r
                      WHERE r.id_reservation = id_reserv);

    price := (SELECT price_for_one
              FROM training
              JOIN reservation r USING(id_training)
              WHERE r.id_reservation = id_reserv);

    IF reserv_status NOT IN ('reserved') THEN
        RAISE EXCEPTION 'Reservation is already payed';
    END IF;

    IF pay_method NOT IN ('cash', 'card') THEN
        RAISE EXCEPTION 'Wrong payment method';
    END IF;

    INSERT INTO payment (final_price, payment_method, payment_date,
                        id_client)
    VALUES (price, pay_method, pay_date, id_user)
    RETURNING id_payment INTO id_paym;

    UPDATE reservation r
    SET id_payment = id_paym,
        status = 'payed'
    WHERE id_reservation = id_reserv;
END;
$$ LANGUAGE plpgsql;

```

- Usage:

To demonstrate how the transaction works, a new client was created along with their personal data, a training, and a reservation.

```
WITH inserted_user AS (  
    INSERT INTO "user" (login, password)  
        VALUES ('coolSem', '12345')  
    RETURNING id_user  
)  
,  
inserted_personal_data AS (  
    INSERT INTO personal_data (id_user, name, surname, date_of_birth,  
        email, phone_number)  
    SELECT id_user, 'Sem', 'Smith', '2015-01-01', 'ssmith@gmail.com',  
        '+420000000001'  
    FROM inserted_user  
)  
,  
inserted_client AS (  
    INSERT INTO client (id_user)  
    SELECT id_user FROM inserted_user  
)  
,  
inserted_training AS (  
    INSERT INTO training (training_date, id_sports_hall, type, capacity,  
        price_for_one)  
    SELECT CURRENT_DATE + INTERVAL '2 months' + TIME '09:30', 1,  
        'volleyball without trainer', 12, 150  
    RETURNING id_training  
)  
INSERT INTO reservation (reservation_date, id_client, status,  
    id_training, id_payment)  
VALUES (CURRENT_TIMESTAMP,  
    (SELECT id_user FROM inserted_user), 'reserved',  
    (SELECT id_training FROM inserted_training), NULL)  
;
```

– **Transaction with an invalid payment method:**

The following transaction attempts to make a payment using an unsupported payment method ('car').

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SELECT update_reservation_status((SELECT id_reservation FROM
    reservation ORDER BY id_reservation DESC LIMIT 1),
                                (SELECT id_user FROM "user" ORDER BY
    id_user DESC LIMIT 1),
                                'car'::VARCHAR(16),
                                CURRENT_TIMESTAMP::TIMESTAMP - TIME
                                '00:30');

COMMIT TRANSACTION;
;
```



```
81
82 BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
83 SELECT update_reservation_status( id_reservation (SELECT id_reservation FROM reservation ORDER BY id_reservation DESC LIMIT 1),
84                                id_user (SELECT id_user FROM "user" ORDER BY id_user DESC LIMIT 1),
85                                pay_method 'car'::VARCHAR(16), pay_date CURRENT_TIMESTAMP::TIMESTAMP - TIME '00:30');
86 COMMIT TRANSACTION;
87
```

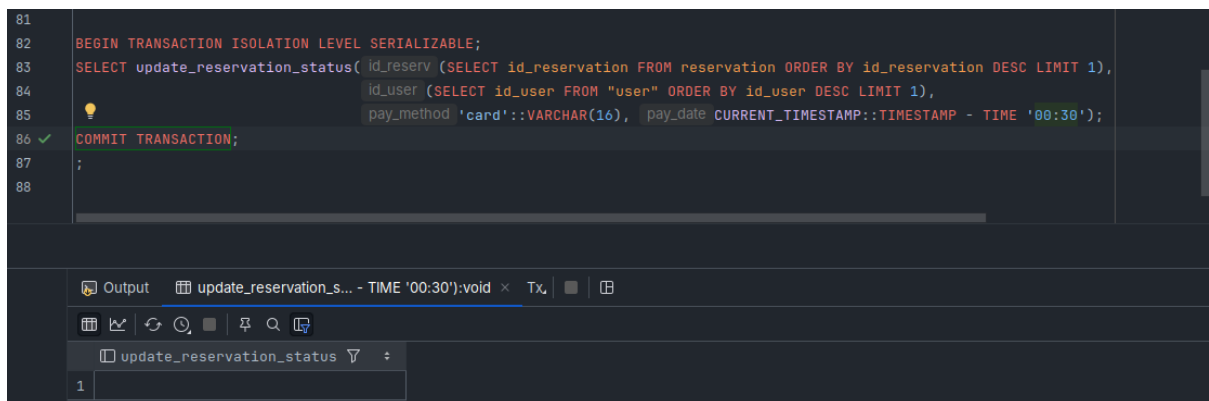
[P0001] ERROR: Wrong payment method
Where: PL/pgSQL function update_reservation_status(integer,integer,character varying,timestamp without time zone) line 21 at RAISE

[Explain with AI](#) [Fix with AI](#)

- **Correct transaction (reservation not yet paid, valid payment method):**
This transaction uses a valid payment method ('card'), successfully creates a new payment, and updates the reservation status.

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SELECT update_reservation_status((SELECT id_reservation FROM
    reservation ORDER BY id_reservation DESC LIMIT 1),
                                (SELECT id_user FROM "user" ORDER BY
    id_user DESC LIMIT 1),
                                'card'::VARCHAR(16),
                                CURRENT_TIMESTAMP::TIMESTAMP - TIME
                                '00:30');

COMMIT TRANSACTION;
;
```



The screenshot shows a SQL IDE with a dark theme. The editor displays a SQL script with line numbers 81 to 88. The script is:
81 BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
82
83 SELECT update_reservation_status((SELECT id_reservation FROM reservation ORDER BY id_reservation DESC LIMIT 1),
84 id_user (SELECT id_user FROM "user" ORDER BY id_user DESC LIMIT 1),
85 pay_method 'card'::VARCHAR(16), pay_date CURRENT_TIMESTAMP::TIMESTAMP - TIME '00:30');
86 COMMIT TRANSACTION;
87 ;
88
Below the editor, there is a toolbar with icons for output, update, refresh, and search. A tab labeled 'update_reservation_s... - TIME '00:30':void x Tx' is active. Below the toolbar, a table with one row and one column is visible, showing the value '1'.

- **Second attempt to pay for the same reservation:**
A second attempt to pay for the same reservation results in an exception, as the reservation status is no longer 'reserved'.



The screenshot shows a SQL IDE with a dark theme. The editor displays a SQL script with line numbers 81 to 87. The script is:
81 BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
82
83 SELECT update_reservation_status((SELECT id_reservation FROM reservation ORDER BY id_reservation DESC LIMIT 1),
84 id_user (SELECT id_user FROM "user" ORDER BY id_user DESC LIMIT 1),
85 pay_method 'card'::VARCHAR(16), pay_date CURRENT_TIMESTAMP::TIMESTAMP - TIME '00:30');
86 COMMIT TRANSACTION;
87
Below the editor, there is a toolbar with icons for output, update, refresh, and search. A tab labeled 'update_reservation_s... - TIME '00:30':void x Tx' is active. Below the toolbar, a table with one row and one column is visible, showing the value '1'. At the bottom of the IDE, a red error message is displayed:
[P0001] ERROR: Reservation is already paid
Where: PL/pgSQL function update_reservation_status(integer,integer,character varying,timestamp without time zone) line 17 at RAISE

View:

This view identifies trainers who have conducted a low number of training sessions (10 or fewer). It helps monitor trainer engagement and identify potentially inactive personnel.

- **Creating:**

```
CREATE VIEW trainers_at_risk AS
  SELECT id_user AS id_trainer,
         name || ' ' || surname AS trainer_name,
         email,
         count(*) AS num_of_conducted_trainings
  FROM trainer t
       JOIN personal_data USING(id_user)
       JOIN conduction c ON c.id_trainer = t.id_user
  GROUP BY id_user, name, surname, email
  HAVING count(*) <= 10
  ORDER BY count(*), id_user;
```

- **Usage:**

```
SELECT * FROM trainers_at_risk;
```

- **Output:**

id_trainer	trainer_name	email	num_of_conducted_trainings
22	Pavel Doležal	paveldolezal19750823@gmail.com	1
24	Judita Beránková	juditaberankova19951023@gmail.com	1
53	Radim Jelinek	radimjelinek19490421@gmail.com	1
63	Leoš Dostál	leosdostal19560429@gmail.com	1
42	Horymír Müller	horymirmuller19560305@gmail.com	2
52	Martin Slavík	martinslavik19580307@gmail.com	2
27	Kristýna Pavlíková	kristynapavlikova19790514@gmail.com	3
41	Milúše Holubová	miluseholubova19631113@gmail.com	4
55	Luděk Moravec	ludekmoravec19891201@gmail.com	7
9	Božena Bártová	bozenabartova20110429@gmail.com	8

Trigger:

This trigger prevents the insertion of a trainer who is younger than 13 years old. The condition is verified through a BEFORE INSERT trigger.

- **Trigger function:**

```
CREATE FUNCTION check_trainer_age()
  RETURNS TRIGGER AS $$
DECLARE
  dob timestamp(0);
BEGIN
  dob := (SELECT date_of_birth
          FROM personal_data
          WHERE (id_user = NEW.id_user));

  IF EXTRACT (YEAR FROM age(dob)) < 13 THEN
    RAISE EXCEPTION 'Trainer must be at least 13 years old';
  END IF;

  RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

- **Trigger declaration:**

```
CREATE TRIGGER trainer_trigger
  BEFORE INSERT ON trainer
  FOR EACH ROW
  EXECUTE FUNCTION check_trainer_age();
```

- Usage:

- Inserting a trainer with age < 13 (should fail):

```
WITH inserted_user AS (  
    INSERT INTO "user" (login, password)  
        VALUES ('coolJohn', '12345')  
    RETURNING id_user  
)  
  
INSERT INTO personal_data (id_user, name, surname, date_of_birth,  
    email, phone_number)  
SELECT id_user, 'John', 'Smith', '2015-01-01',  
    'jsmith@gmail.com', '+420000000002'  
FROM inserted_user  
;  
  
INSERT INTO trainer (id_user, education)  
VALUES ((SELECT id_user FROM "user" ORDER BY id_user DESC LIMIT  
    1), FALSE)  
;
```



```
1 WITH inserted_user AS (  
2     INSERT INTO "user" (login, password)  
3         VALUES (login 'coolJohn', password '12345')  
4     RETURNING id_user  
5 )  
6  
7 INSERT INTO personal_data (id_user, name, surname, date_of_birth, email, phone_number)  
8     SELECT id_user, name 'John', surname 'Smith', date_of_birth '2015-01-01', email 'jsmith@gmail.com', phone_number '+420000000002'  
9     FROM inserted_user  
10 ;  
11  
12 INSERT INTO trainer (id_user, education)  
13     VALUES ((SELECT id_user FROM "user" ORDER BY id_user DESC LIMIT 1), education FALSE)  
14 ;  
15
```

[P0001] ERROR: Trainer must be at least 13 years old
Where: PL/pgSQL function check_trainer_age() line 10 at RAISE

[Explain with AI](#) [Fix with AI](#)

- Inserting a trainer with age ≥ 13 (should succeed):

```
WITH inserted_user AS (  
    INSERT INTO "user" (login, password)  
        VALUES ('coolJohn', '12345')  
    RETURNING id_user  
)  
  
INSERT INTO personal_data (id_user, name, surname, date_of_birth,  
    email, phone_number)  
    SELECT id_user, 'John', 'Smith', '2010-01-01',  
        'jsmith@gmail.com', '+420000000002'  
    FROM inserted_user  
;  
  
INSERT INTO trainer (id_user, education)  
VALUES ((SELECT id_user FROM "user" ORDER BY id_user DESC LIMIT 1),  
    FALSE)  
;  
;
```

```
1 WITH inserted_user AS (  
2     INSERT INTO "user" (login, password)  
3     VALUES ('coolJohn', '12345')  
4     RETURNING id_user  
5 )  
6  
7 INSERT INTO personal_data (id_user, name, surname, date_of_birth, email, phone_number)  
8     SELECT id_user, 'John', 'Smith', '2010-01-01', 'jsmith@gmail.com', '+420000000002'  
9     FROM inserted_user  
10  
11 ;  
12  
13 INSERT INTO trainer (id_user, education)  
14 VALUES ((SELECT id_user FROM "user" ORDER BY id_user DESC LIMIT 1), FALSE)  
15 ;  
16 ;
```

Tx. [X] [Y]

[2025-05-03 16:52:48] Connected
kolesole> WITH inserted_user AS (
 INSERT INTO "user" (login, password)
 VALUES ('coolJohn', '12345')
 RETURNING id_user
)

 INSERT INTO personal_data (id_user, name, surname, date_of_birth, email, phone_number)
 SELECT id_user, 'John', 'Smith', '2010-01-01', 'jsmith@gmail.com', '+420000000002'
 FROM inserted_user
[2025-05-03 16:52:48] 1 row affected in 17 ms
kolesole.public> INSERT INTO trainer (id_user, education)
 VALUES ((SELECT id_user FROM "user" ORDER BY id_user DESC LIMIT 1), FALSE)
[2025-05-03 16:52:51] 1 row affected in 15 ms

Index:

This index is created on the first character of the `name` column in the `personal_data` table to speed up queries filtering by the first letter.

- **Without index:**

```
EXPLAIN ANALYZE
SELECT name || ' ' || surname AS user_name
FROM personal_data
WHERE LEFT(name, 1) = 'Y';
```

QUERY PLAN
Seq Scan on personal_data (cost=0.00..898.81 rows=160 width=32) (actual time=25.710..25.711 rows=0 loops=1)
Filter: ("left"((name)::text, 1) = 'Y'::text)
Rows Removed by Filter: 32001
Planning Time: 1.415 ms
Execution Time: 25.765 ms

- **With index:**

```
CREATE INDEX idx_name_first_char ON personal_data (LEFT(name, 1));

EXPLAIN ANALYZE
SELECT name || ' ' || surname AS user_name
FROM personal_data
WHERE LEFT(name, 1) = 'Y';
```

QUERY PLAN
Bitmap Heap Scan on personal_data (cost=5.53..318.57 rows=160 width=32) (actual time=0.078..0.079 rows=0 loops=1)
Recheck Cond: ("left"((name)::text, 1) = 'Y'::text)
-> Bitmap Index Scan on idx_name_first_char (cost=0.00..5.49 rows=160 width=0) (actual time=0.074..0.074 rows=0 loops=1)
Index Cond: ("left"((name)::text, 1) = 'Y'::text)
Planning Time: 0.504 ms
Execution Time: 0.128 ms

- **Conclusion:**

Without the index, a sequential scan is used, with a planning time of 1.415 ms and an execution time of 25.765 ms.

With the index, a bitmap index scan is used, reducing the planning time to 0.504 ms and the execution time to 0.128 ms.

This demonstrates that indexing significantly improves query performance in this case, and the created index is effectively used in the query execution plan.