|  |  |
| --- | --- |
| Gerb-BMSTU_01 | **Министерство науки и высшего образования Российской Федерации**  **Федеральное государственное бюджетное образовательное учреждение**  **высшего образования**  **«Московский государственный технический университет**  **имени Н.Э. Баумана**  **(национальный исследовательский университет)»**  **(МГТУ им. Н.Э. Баумана)** |

ФАКУЛЬТЕТ **Информатика и системы управления**

КАФЕДРА **Компьютерные системы и сети (ИУ6)**

НАПРАВЛЕНИЕ ПОДГОТОВКИ **09.03.01**

**Отчет**

**по лабораторной работе №** **3**

**Название:** Создание БД для аналитики

**Дисциплина:** Базы данных

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Студент | ИУ6-32Б |  |  | Н. И. Белоусов |
|  | (Группа) |  | (Подпись, дата) | (И.О. Фамилия) |
|  |  |  |  |  |
| Преподаватель |  |  |  | Д. А. Миков |
|  |  |  | (Подпись, дата) | (И.О. Фамилия) |

Москва, 2022

**Вариант 5**

**Часть 1**

**Задание:**

Результатом данного задания является схема базы данных, скрипты создания базы данных и ее обладающие следующими:

Как минимум одна таблица должна содержать не меньше 100 млн. записей, которые со временем теряют актуальность.

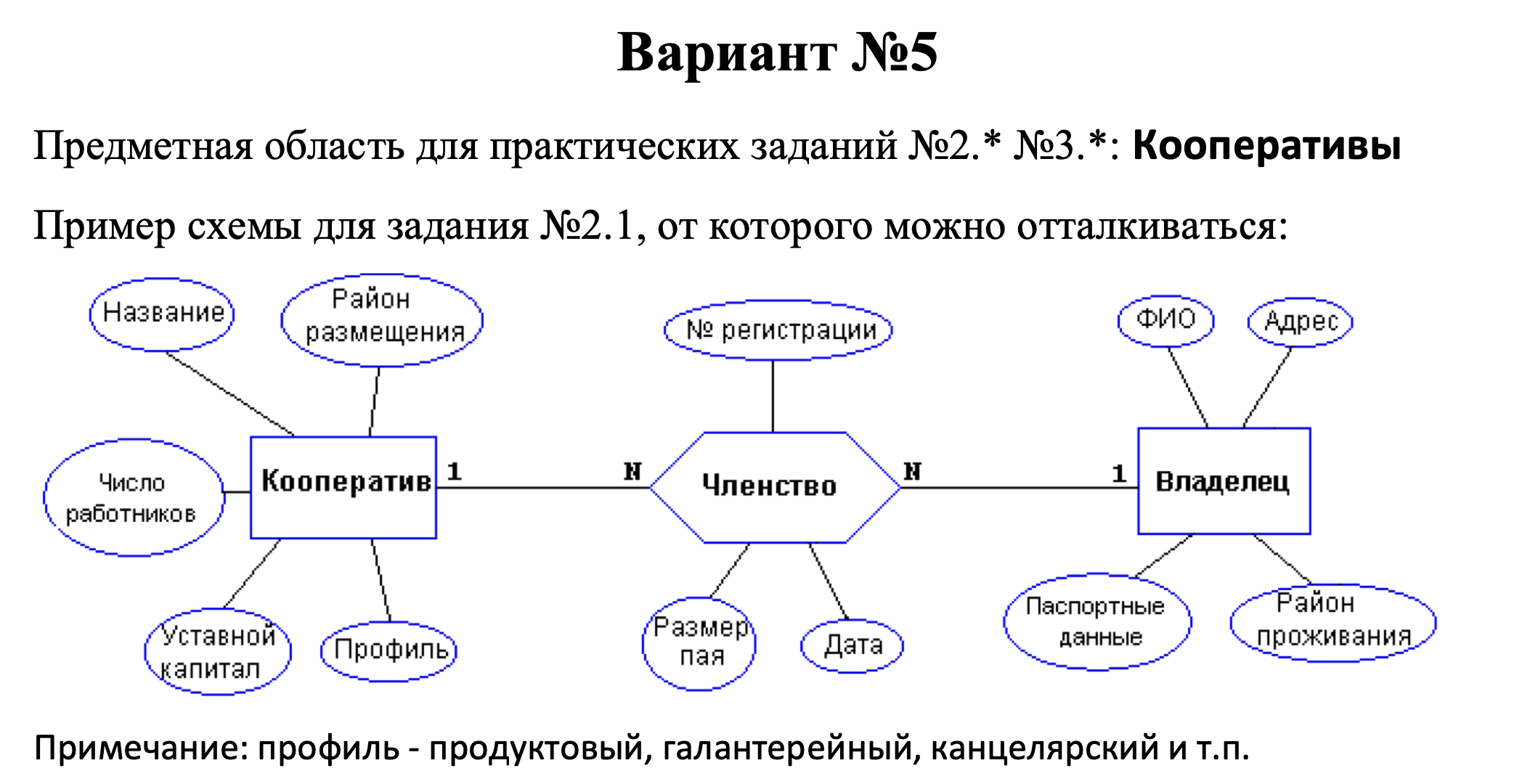
Другая таблица, связанная с первой, должна содержать не меньше 1 млн. записей.

В одной из таблиц с количество записей больше 1 млн. должна быть колонка с текстом, по которой будет необходимо настроить полнотекстовый поиск.

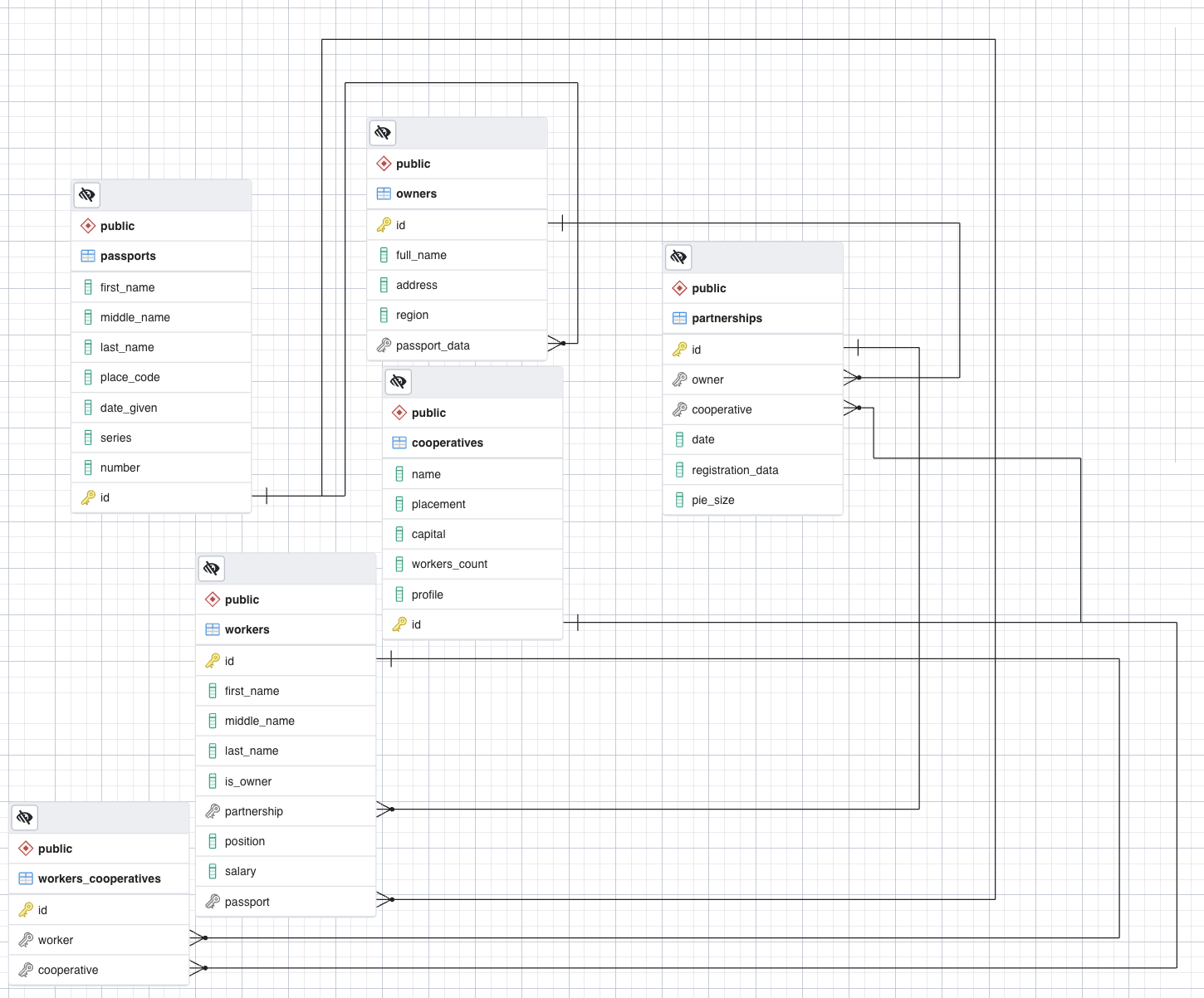
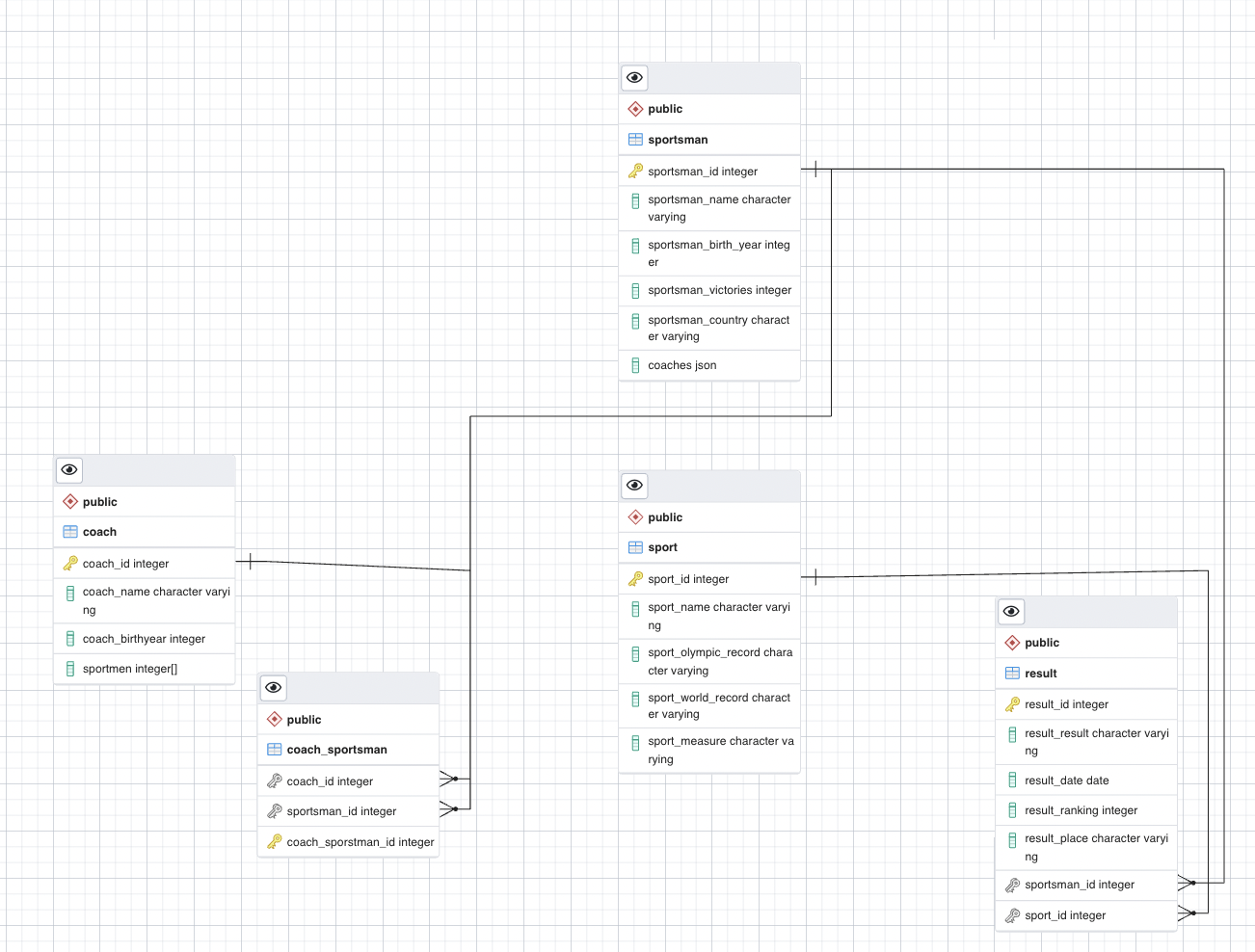
В одной из таблиц с количество записей больше 1 млн. должна быть колонка с данными в json формате.

В одной из таблиц с количество записей больше 1 млн. должна быть колонка с массивом.

Изначальная схема:

****

Доделанная схема:

****

Скрипт для создания этих таблиц:

*-- Table: public.cooperatives*

*-- DROP TABLE IF EXISTS public.cooperatives;*

*CREATE* *TABLE* *IF* *NOT* *EXISTS* public.cooperatives

(

*name* *character* varying COLLATE pg\_catalog."default" *NOT* *NULL*,

placement *character* varying COLLATE pg\_catalog."default" *NOT* *NULL*,

capital *character* varying COLLATE pg\_catalog."default" *NOT* *NULL*,

workers\_count *character* varying COLLATE pg\_catalog."default" *NOT* *NULL*,

*profile* *character* varying COLLATE pg\_catalog."default" *NOT* *NULL*,

id *bigint* *NOT* *NULL*,

*CONSTRAINT* cooperatives\_pkey *PRIMARY* *KEY* (id)

)

TABLESPACE pg\_default;

*ALTER* *TABLE* *IF* *EXISTS* public.cooperatives

*OWNER* *to* postgres;

*-- Table: public.owners*

*-- DROP TABLE IF EXISTS public.owners;*

*CREATE* *TABLE* *IF* *NOT* *EXISTS* public.owners

(

id *bigint* *NOT* *NULL*,

full\_name *character* varying COLLATE pg\_catalog."default" *NOT* *NULL*,

*address* *character* varying COLLATE pg\_catalog."default" *NOT* *NULL*,

region *character* varying COLLATE pg\_catalog."default" *NOT* *NULL*,

passport\_data *bigint*,

*CONSTRAINT* owners\_pkey *PRIMARY* *KEY* (id),

*CONSTRAINT* passport *FOREIGN KEY* (passport\_data)

*REFERENCES* public.passports (id) MATCH *SIMPLE*

*ON* *UPDATE* *NO* *ACTION*

*ON DELETE* *NO* *ACTION*

NOT VALID

)

TABLESPACE pg\_default;

ALTER TABLE IF EXISTS public.owners

OWNER to postgres;

*-- Table: public.partnerships*

*-- DROP TABLE IF EXISTS public.partnerships;*

*CREATE* *TABLE* *IF* *NOT* *EXISTS* public.partnerships

(

id *bigint* *NOT* *NULL*,

*owner* *bigint* *NOT* *NULL*,

cooperative *bigint* *NOT* *NULL*,

*date* *timestamp* *without* *time* *zone* *NOT* *NULL*,

registration\_data *integer* *NOT* *NULL*,

pie\_size *double precision* *NOT* *NULL*,

*CONSTRAINT* partnerships\_pkey *PRIMARY* *KEY* (id),

*CONSTRAINT* cooperative *FOREIGN KEY* (cooperative)

*REFERENCES* public.cooperatives (id) MATCH *SIMPLE*

*ON* *UPDATE* *NO* *ACTION*

*ON DELETE CASCADE*

*NOT* VALID,

*CONSTRAINT* *owner* *FOREIGN KEY* (*owner*)

*REFERENCES* public.owners (id) MATCH *SIMPLE*

*ON* *UPDATE* *NO* *ACTION*

*ON DELETE CASCADE*

*NOT* VALID

)

TABLESPACE pg\_default;

*ALTER* *TABLE* *IF* *EXISTS* public.partnerships

*OWNER* *to* postgres;

*-- Table: public.passports*

*-- DROP TABLE IF EXISTS public.passports;*

*CREATE* *TABLE* *IF* *NOT* *EXISTS* public.passports

(

first\_name *character* varying COLLATE pg\_catalog."default" *NOT* *NULL*,

middle\_name *character* varying COLLATE pg\_catalog."default" *NOT* *NULL*,

last\_name *character* varying COLLATE pg\_catalog."default" *NOT* *NULL*,

place\_code *character* varying COLLATE pg\_catalog."default" *NOT* *NULL*,

date\_given *timestamp* *without* *time* *zone* *NOT* *NULL*,

series *integer* *NOT* *NULL*,

"number" *integer* *NOT* *NULL*,

id *bigint* *NOT* *NULL*,

*CONSTRAINT* passports\_pkey *PRIMARY* *KEY* (id)

)

TABLESPACE pg\_default;

ALTER TABLE IF EXISTS public.passports

OWNER to postgres;

*-- Table: public.workers\_cooperatives*

*-- DROP TABLE IF EXISTS public.workers\_cooperatives;*

*CREATE* *TABLE* *IF* *NOT* *EXISTS* public.workers\_cooperatives

(

id *bigint* *NOT* *NULL*,

worker *bigint* *NOT* *NULL*,

cooperative *bigint* *NOT* *NULL*,

*CONSTRAINT* workers\_cooperatives\_pkey *PRIMARY* *KEY* (id),

*CONSTRAINT* cooperative *FOREIGN KEY* (cooperative)

*REFERENCES* public.cooperatives (id) MATCH *SIMPLE*

*ON* *UPDATE* *NO* *ACTION*

*ON DELETE* *NO* *ACTION*

*NOT* VALID,

*CONSTRAINT* worker *FOREIGN KEY* (worker)

*REFERENCES* public.workers (id) MATCH *SIMPLE*

*ON* *UPDATE* *NO* *ACTION*

*ON DELETE* *NO* *ACTION*

*NOT* VALID

)

TABLESPACE pg\_default;

*ALTER* *TABLE* *IF* *EXISTS* public.workers\_cooperatives

*OWNER* *to* postgres;

*-- Table: public.workers*

*-- DROP TABLE IF EXISTS public.workers;*

*CREATE* *TABLE* *IF* *NOT* *EXISTS* public.workers

(

id *bigint* *NOT* *NULL*,

first\_name *character* varying COLLATE pg\_catalog."default" *NOT* *NULL*,

middle\_name *character* varying COLLATE pg\_catalog."default" *NOT* *NULL*,

last\_name *character* varying COLLATE pg\_catalog."default" *NOT* *NULL*,

is\_owner *boolean* *NOT* *NULL*,

partnership *bigint*,

"position" *character* varying COLLATE pg\_catalog."default" *NOT* *NULL*,

salary *character* varying COLLATE pg\_catalog."default" *NOT* *NULL*,

passport *bigint* *NOT* *NULL*,

*CONSTRAINT* workers\_pkey *PRIMARY* *KEY* (id),

*CONSTRAINT* partnership *FOREIGN KEY* (partnership)

*REFERENCES* public.partnerships (id) MATCH *SIMPLE*

*ON* *UPDATE* *NO* *ACTION*

*ON DELETE* *NO* *ACTION*,

*CONSTRAINT* passport *FOREIGN KEY* (passport)

*REFERENCES* public.passports (id) MATCH *SIMPLE*

*ON* *UPDATE* *NO* *ACTION*

*ON DELETE* *NO* *ACTION*

*NOT* VALID

)

TABLESPACE pg\_default;

*ALTER* *TABLE* *IF* *EXISTS* public.workers

Заполнение происходило с помощью скрипта на python:

*from* faker *import* Faker

*from* rich.console *import* Console

*from* rich.table *import* Table

*from* rich.style *import* Style

*import* random

*from* datetime *import* datetime

*from* rich.progress *import* track

*import* json

*import* psycopg2

*from* os.path *import* isfile, join

*from* os *import* listdir

currencies *=* ("рубль(ей)", "доллар(ов)", "евро", "юань(и)")

profiles *=* ("продуктовый", "галантерейный", "канцелярский",

"сапожный", "машиностроительный", "фермерский")

positions *=* ("директор", "уборщик", "менеджер")

OWNER\_COUNT *=* 1000

COOPERATIVE\_COUNT *=* 1000

PARTNERSHIP\_COUNT *=* 10000

WORKER\_COUNT *=* 10000

PASSPORT\_COUNT *=* OWNER\_COUNT *+* WORKER\_COUNT

*class* FakerWrapper:

*def* \_\_init\_\_(*self*):

*self*.faker *=* Faker(*locale=*'ru-RU')

*def* generate\_money(*self*):

currency *=* random.choice(currencies)

*return* str(random.randint(1, 200000000)) *+* *f*" {currency}"

*def* generate\_location(*self*):

*return* *self*.faker.address()

*def* generate\_region(*self*): *return* *self*.faker.administrative\_unit()

*def* generate\_company\_name(*self*): *return* *self*.faker.company()

*def* generate\_name(*self*): *return* *self*.faker.name()

*def* generate\_date\_time(*self*): *return* *self*.faker.date\_object()

*def* generate\_job(*self*): *return* *self*.faker.job()

*class* DBWorker:

*def* \_\_init\_\_(*self*, *host*, *port*, *database\_name*, *user*) -> None:

*self*.connection *=* psycopg2.connect(

*dbname=database\_name*, *user=user*, *port=port*, *host=host*)

*self*.cursor *=* *self*.connection.cursor()

*self*.connection.autocommit *=* True

*def* drop\_all(*self*):

*try*:

*self*.cursor.execute("""drop table public.cooperatives cascade;""")

*self*.cursor.execute("""drop table public.passports cascade;""")

*self*.cursor.execute("""drop table public.workers cascade;""")

*self*.cursor.execute("""drop table public.owners cascade;""")

*self*.cursor.execute(

"""drop table public.workers\_cooperatives cascade;""")

*self*.cursor.execute("""drop table public.partnerships cascade;""")

*except*:

Console().log("[orange] all tables are already deleted [/orange]")

*def* create\_all(*self*):

*self*.drop\_all()

create\_files *=* [f *for* f *in* listdir(

"./scripts") *if* isfile(join("./scripts", f))]

*with* open("./scripts/create\_passports.sql", 'r') *as* f:

*self*.cursor.execute(f.read())

f.close()

*with* open("./scripts/create\_cooperatives.sql", 'r') *as* f:

*self*.cursor.execute(f.read())

f.close()

*with* open("./scripts/create\_owners.sql", 'r') *as* f:

*self*.cursor.execute(f.read())

f.close()

*with* open("./scripts/create\_partnerships.sql", 'r') *as* f:

*self*.cursor.execute(f.read())

f.close()

*with* open("./scripts/create\_workers.sql", 'r') *as* f:

*self*.cursor.execute(f.read())

f.close()

*with* open("./scripts/create\_worker\_cooperatives.sql", 'r') *as* f:

*self*.cursor.execute(f.read())

f.close()

*# for i in track(range(len(create\_files))):*

*# with open(f'./scripts/{create\_files[i]}', 'r') as f:*

*# if (f.name.find("insert") == -1):*

*# self.cursor.execute(f.read())*

*def* fill\_passports(*self*):

t *=* datetime.now()

*for* i *in* track(range(PASSPORT\_COUNT), *description=*"Generating passports"):

tmp *=* Passport()

*self*.cursor.execute(

"""INSERT INTO public.passports(

first\_name, middle\_name, last\_name, place\_code, date\_given, series, "number", id)

VALUES (%s, %s, %s, %s, %s, %s, %s, %s);""", (tmp.name, tmp.middle\_name, tmp.last\_name, tmp.place\_code, tmp.data\_given, tmp.series, tmp.number, i))

Console().log(*f*"[green] {*self*.fill\_passports} done [/green] in [yellow] {datetime.now() *-* t} [/yellow] ")

*def* fill\_cooperatives(*self*):

t *=* datetime.now()

*for* i *in* track(range(COOPERATIVE\_COUNT), *description=*"Generating cooperatives"):

tmp *=* Cooperative()

*self*.cursor.execute(

"""INSERT INTO public.cooperatives(

name, placement, capital, workers\_count, profile, id)

VALUES (%s, %s, %s, %s, %s, %s);""", (tmp.name, tmp.placement, tmp.capital, tmp.workers\_count, tmp.profile, i)

)

Console().log(*f*"done in [yellow] {datetime.now() *-* t}")

*def* fill\_owners(*self*):

t *=* datetime.now()

*for* i *in* track(range(OWNER\_COUNT), *description=*"Generating owners"):

tmp *=* Owner()

*self*.cursor.execute("""INSERT INTO public.owners(

id, full\_name, address, region, passport\_data)

VALUES (%s, %s, %s, %s, %s);""", (i, tmp.name, tmp.adress, tmp.region, tmp.passport\_data))

Console().log(*f*"done in [yellow] {datetime.now() *-* t}")

*def* fill\_workers(*self*):

t *=* datetime.now()

*for* i *in* track(range(WORKER\_COUNT), *description=*"Generating workers"):

tmp *=* Worker()

*self*.cursor.execute("""INSERT INTO public.workers(

id, first\_name, middle\_name, last\_name, is\_owner, partnership, "position", salary, passport)

VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s);

""", (i, tmp.first\_name, tmp.middle\_name, tmp.last\_name, tmp.is\_owner, tmp.cooperative, tmp.position, tmp.salary, tmp.passport))

Console().log(*f*"done in [yellow] {datetime.now() *-* t}")

*def* fill\_partnerships(*self*):

t *=* datetime.now()

*for* i *in* track(range(PARTNERSHIP\_COUNT), *description=*"Generating partnerships"):

tmp *=* Partnership()

*self*.cursor.execute("""INSERT INTO public.partnerships(

id, owner, cooperative, date, registration\_data, pie\_size)

VALUES (%s, %s, %s, %s, %s, %s);""", (i, tmp.owner, tmp.cooperative, tmp.date, tmp.registration\_data, tmp.pie\_size))

Console().log(*f*"done in [yellow] {datetime.now() *-* t}")

*def* fill\_worker\_cooperatives(*self*):

t *=* datetime.now()

*for* i *in* track(range(WORKER\_COUNT)):

tmp *=* WorkerCooperative()

*self*.cursor.execute("""INSERT INTO public.workers\_cooperatives(

id, worker, cooperative)

VALUES (%s, %s, %s);""", (i, tmp.worker, tmp.cooperative))

Console().log(*f*"done in [yellow] {datetime.now() *-* t}")

*def* generate\_all(*self*):

t *=* datetime.now()

*self*.fill\_passports()

*self*.fill\_cooperatives()

*self*.fill\_owners()

*self*.fill\_partnerships()

*self*.fill\_workers()

*self*.fill\_worker\_cooperatives()

Console().log(*f*"done generating in [yellow] {datetime.now() *-* t}")

*class* Worker:

*def* \_\_init\_\_(*self*) -> None:

*self*.fakerWrapper *=* FakerWrapper()

*self*.generate\_worker()

*def* generate\_worker(*self*):

fullname: str *=* *self*.fakerWrapper.generate\_name()

*self*.first\_name *=* fullname.split(" ")[0]

*self*.middle\_name *=* fullname.split(" ")[1]

*self*.last\_name *=* fullname.split(" ")[2]

*self*.is\_owner *=* True *if* random.randint(0, 10) *%* 2 *==* 0 *else* False

*if* (*self*.is\_owner):

*self*.cooperative *=* random.randint(1, PARTNERSHIP\_COUNT *-* 1)

*else*:

*self*.cooperative *=* None

*self*.job *=* *self*.fakerWrapper.generate\_job()

*self*.position *=* random.choice(positions)

*self*.salary *=* *self*.fakerWrapper.generate\_money()

*self*.passport *=* random.randint(1, PASSPORT\_COUNT *-* 1)

*class* Cooperative:

*def* \_\_init\_\_(*self*):

*self*.fakerWrapper *=* FakerWrapper()

*self*.generate\_cooperative()

"""генерирует кооператив

"""

*def* generate\_cooperative(*self*):

*self*.name *=* *self*.fakerWrapper.generate\_company\_name()

*self*.placement *=* *self*.fakerWrapper.generate\_region()

*self*.capital *=* *self*.fakerWrapper.generate\_money()

*self*.workers\_count *=* random.randint(1, 255)

*self*.profile *=* random.choice(profiles)

*def* generate\_places(*self*):

*pass*

*def* to\_string(*self*) -> str:

*return* *f*"{*self*.name}\n {*self*.placement}"

*class* WorkerCooperative:

*def* \_\_init\_\_(*self*) -> None:

*self*.generate\_connection()

*self*.generate\_connection()

*def* generate\_connection(*self*):

*self*.worker *=* random.randint(1, WORKER\_COUNT *-* 1)

*self*.cooperative *=* random.randint(1, COOPERATIVE\_COUNT *-* 1)

*class* Passport:

*def* \_\_init\_\_(*self*) -> None:

*self*.faker *=* FakerWrapper()

*self*.generate\_passport()

*def* generate\_passport(*self*):

fullname *=* *self*.faker.generate\_name()

*self*.name *=* fullname.split(" ")[0]

*self*.middle\_name *=* fullname.split(" ")[1]

*self*.last\_name *=* fullname.split(" ")[2]

*self*.place\_code *=* str(random.randint(100, 999)) *+* \

"-" *+* str(random.randint(100, 999))

*self*.data\_given *=* *self*.faker.generate\_date\_time()

*self*.series *=* str(random.randint(10, 99)) *+* str(random.randint(10, 99))

*self*.number *=* random.randint(100000, 999999)

*def* to\_map(*self*):

*return* *self*.\_\_dict\_\_

*class* Owner:

*def* \_\_init\_\_(*self*) -> None:

*self*.faker *=* FakerWrapper()

*self*.generate\_owner()

*def* generate\_owner(*self*):

*self*.name *=* *self*.faker.generate\_name()

*self*.adress *=* *self*.faker.generate\_location()

*self*.region *=* *self*.faker.generate\_region()

*self*.passport\_data *=* random.randint(1, PASSPORT\_COUNT *-* 1)

*def* to\_map(*self*):

d *=* *self*.\_\_dict\_\_

d['passport\_data'] *=* *self*.passport\_data.to\_map()

*return* d

*class* Partnership:

*def* \_\_init\_\_(*self*) -> None:

*self*.cooperative *=* random.randint(1, COOPERATIVE\_COUNT *-* 1)

*self*.owner *=* random.randint(1, OWNER\_COUNT *-* 1)

*self*.registration\_data *=* random.randint(1, 1000000)

*self*.date *=* FakerWrapper().generate\_date\_time()

*self*.pie\_size *=* random.randint(1, 1000) */* 1000

*class* Interface:

*def* \_\_init\_\_(*self*):

*self*.console *=* Console()

*self*.db\_worker *=* DBWorker(

*host=*"127.0.0.1", *port=*"8014", *user=*"postgres", *database\_name=*"labs")

*def* start(*self*):

*self*.select\_variant()

"""

(a, b, c) = interface.ask\_for\_input("a","b","c")

"""

*def* ask\_for\_input(*self*, *\*vars*) -> list:

target *=* []

*for* i *in* *vars*:

*self*.console.log(*f*'provide {i*=*}')

target.append(input())

*return* target

"""prints menu for generator"""

*def* print\_menu(*self*):

table\_of\_choice *=* Table(*title=*"Generator menu")

table\_of\_choice.add\_column("hotkey", *style=*Style(*color=*"green"))

table\_of\_choice.add\_column("description")

*with* open("./data\_generator/configs/menu\_config.json", 'r')*as* f:

*try*:

*self*.hotkeys *=* json.load(*fp=*f)

print(*self*.hotkeys)

*for* i *in* list(*self*.hotkeys.keys()):

table\_of\_choice.add\_row(*f*"{i}", str(*self*.hotkeys[i]))

*self*.console.print(table\_of\_choice)

*except*:

*self*.console.log("print\_menu method [red] error [/red]")

*def* select\_variant(*self*):

*self*.print\_menu()

variant *=* *self*.ask\_for\_input("variant")

variant *=* variant[0]

*while* variant *not* *in* list(*self*.hotkeys.keys()):

*self*.console.clear()

*self*.console.print("[orange] wrong input [/orange]")

*self*.print\_menu()

variant *=* *self*.ask\_for\_input("variant")

*match* variant:

*case* "0":

*self*.shutdown()

*case* "1":

*self*.db\_worker.generate\_all()

input()

*self*.console.clear()

*self*.select\_variant()

*case* "2":

*self*.db\_worker.create\_all()

input()

*self*.console.clear()

*self*.select\_variant()

*case* "3":

*self*.db\_worker.fill\_owners()

input()

*self*.console.clear()

*self*.select\_variant()

*case* "4":

*self*.db\_worker.fill\_partnerships()

input()

*self*.console.clear()

*self*.select\_variant()

*case* "5":

*self*.db\_worker.fill\_workers()

input()

*self*.console.clear()

*self*.select\_variant()

*case* "6":

*self*.db\_worker.fill\_worker\_cooperatives()

input()

*self*.console.clear()

*self*.select\_variant()

*def* shutdown(*self*):

*self*.console.print("[green] shutting down [/green]")

exit()

""" to display table

vars: list(n, m+1) - table with \*\*column names\*\*

example:

```python

interface.debug\_output\_table([["name", a, b], ["age", 12, 24]])

```

| name | age |

| a | 12 |

| b | 24 |

"""

*def* debug\_output\_table(*self*, *vars*: list):

*self*.console.log("print\_debud\_table method: [green] start [/green]")

target *=* Table()

*if* (len(*vars*)) *>* 10:

*for* i *in* range(5, len(*vars*) *-* 5):

*vars*.remove(i)

*for* i *in* range(len(*vars*)):

target.add\_column(*f*'{*vars*[i][0]}')

*# что я высрал еб твою мать*

*for* i *in* range(1, len(*vars*)):

target.add\_row(*\**[str(*vars*[\_][i]) *for* \_ *in* range(len(*vars*[i]))])

*self*.console.print(target)

*self*.console.log("print\_debud\_table method: [green] end [/green]\n")

*def* debug\_output\_obj(*self*, *value*: object):

*try*:

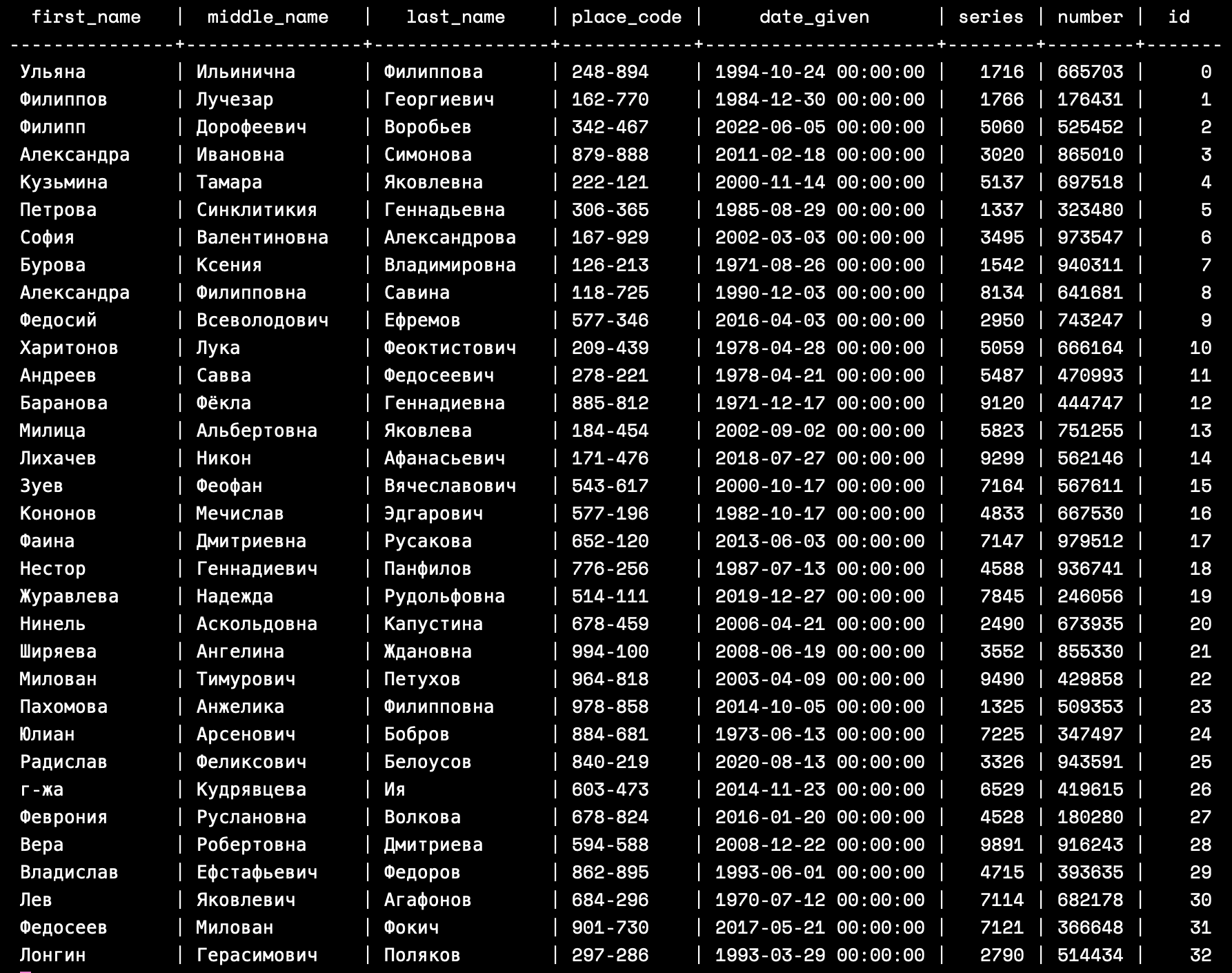
*self*.console.print(str(*value*))

*except*:

*self*.console.log(*self*.console.log(

*f*"debug\_output\_obj method: [red] error [/red]\nno convert to string for object of type {type(*value*)}"))

Пример заполнения таблицы спортсменов:



**Часть 2**

**Задание:**

• Создать пользователя test и выдать ему доступ к базе данных.

• Составить и выполнить скрипты присвоения новому пользователю прав доступа к таблицам, созданным в практическом задании

1. При этом права доступа к различным таблицам должны быть различными, а именно: o По крайней мере, для одной таблицы новому пользователю присваиваются права SELECT, INSERT, UPDATE в полном объеме. По крайней мере, для одной таблицы новому пользователю присваиваются права SELECT и UPDATE только избранных столбцов.

o По крайней мере, для одной таблицы новому пользователю присваивается только право SELECT.

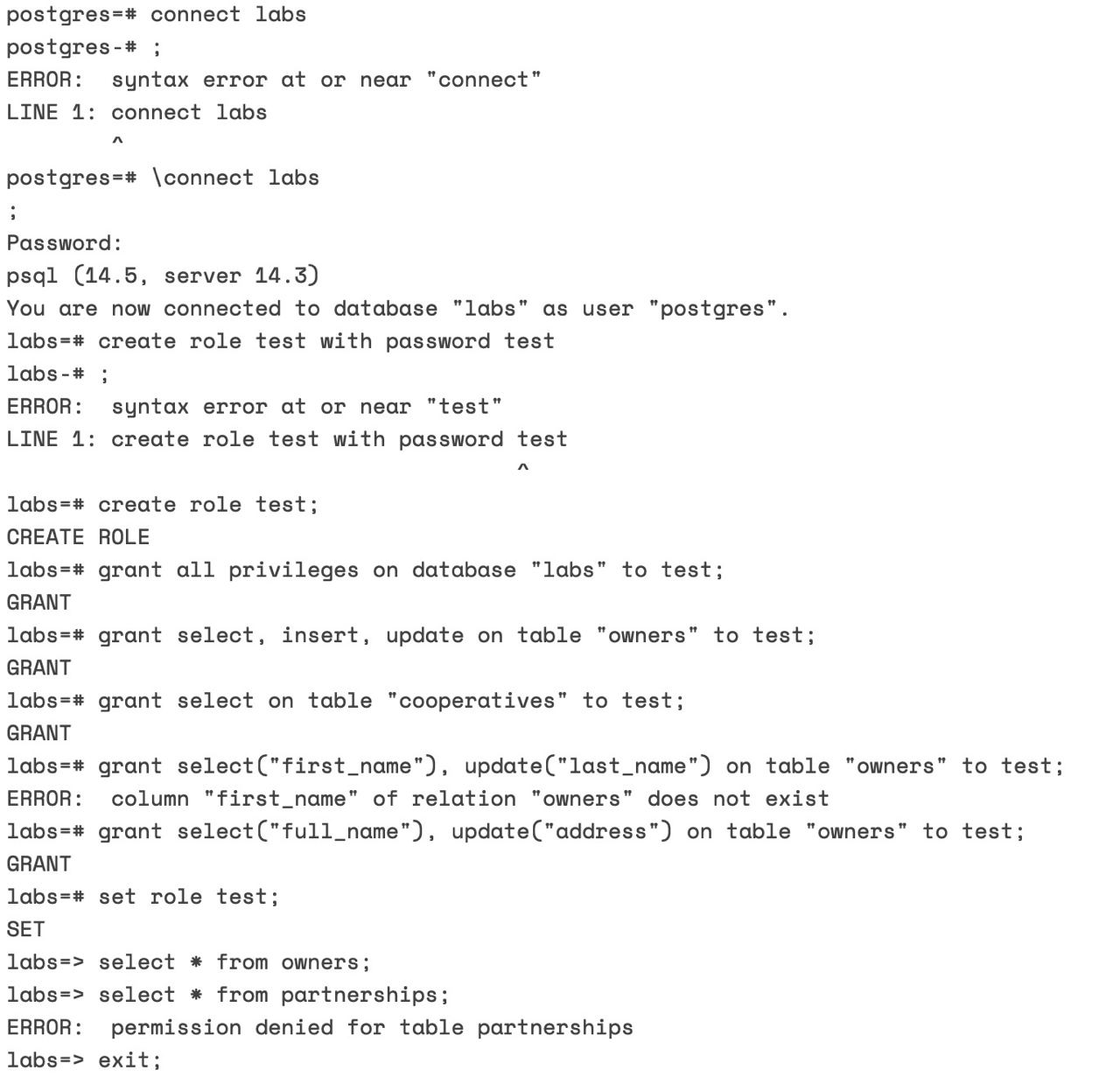
• Создать стандартную роль уровня базы данных, присвоить ей право доступа (UPDATE на некоторые столбцы) к представлению, созданному в практическом задании №3.3, назначить новому пользователю созданную роль.

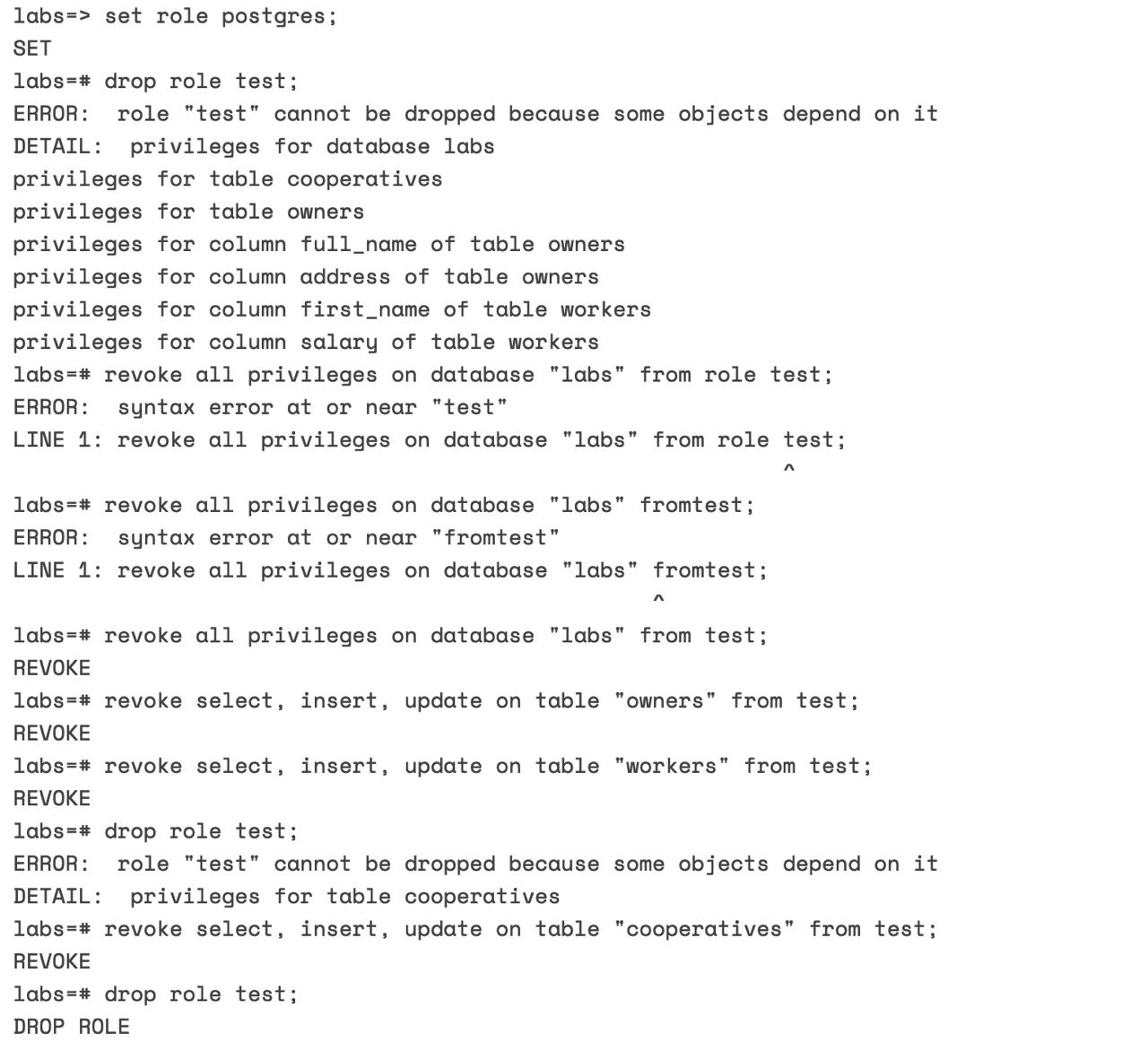
• Выполнить от имени нового пользователя некоторые выборки из таблиц и представления. Убедиться в правильности контроля прав доступа.

• Выполнить от имени нового пользователя операторы изменения таблиц с ограниченными правами доступа. Убедиться в правильности контроля прав доступа.

• Составить SQL-скрипты для создания нескольких представлений, которые позволяли бы упростить манипуляции с данными или позволяли бы ограничить доступ к данным, предоставляя только необходимую информацию

Создаём роль, даем ей права и проверяем:



Попытка удалить роль:

Поскольку данной роли был дан ряд прав, чтобы ее удалить, необходимо сначала отозвать все права у данной роли. После того, как мы это сделаем, роль можно удалить.

**Контрольные вопросы:**

• Для чего нужны роли?

Роль можно рассматривать как пользователя базы данных, в зависимости от настроек роли. Роли могут владеть объектами базы данных и выдавать другим ролям разрешения на доступ к этим объектам, управляя тем, кто имеет доступ и к каким объектам

• Что такое схема?

Схема - дополнительный уровень структуризации объектов. База данных содержит одну или несколько именованных схем, которые в свою очередь содержат таблицы. Схемы также содержат именованные объекты других видов, включая типы данных, функции и операторы. Схемы не ограничивают доступ к данным: пользователь с достаточными правами может обращаться к объектам в любой схеме текущей базы данных. Применение схем: использование БД несколькими пользователями, независимо друг от друга, объединение объектов БД в логические группы, а также для сосуществования разные приложения без конфликтов имён.

• Рассказать про директивы GRANT и REVOKE.

GRANT — определить права доступа. Команда GRANT имеет две основные разновидности: первая назначает права для доступа к объектам баз данных - вторая назначает одни роли членами других.

1) GRANT для объектов баз данных. Эта разновидность команды GRANT даёт одной или нескольким ролям определённые права для доступа к объекту базы данных. Эти права добавляются к списку имеющихся, если роль уже наделена какими-то правами. Также можно дать роли некоторое право для всех объектов одного типа в одной или нескольких схемах. Ключевое слово PUBLIC означает, что права даются всем ролям, включая те, что могут быть созданы позже. PUBLIC можно воспринимать как неявно определённую группу, в которую входят все роли. Любая конкретная роль получит в сумме все права, данные непосредственно ей и ролям, членом которых она является, а также права, данные роли PUBLIC.

2) GRANT для ролей. Эта разновидность команды GRANT включает роль в члены одной или нескольких других ролей. Членство в ролях играет важную роль, так как права, данные роли, распространяются и на всех её членов. Получивший членство в роли с указанием WITH ADMIN OPTION сможет, в свою очередь, включать в члены этой роли, а также исключать из неё другие роли. Без этого указания обычные пользователи не могут это делать. Считается, что роль не имеет права WITH ADMIN OPTION для самой себя, но ей позволено управлять своими членами из сеанса, в котором пользователь сеанса соответствует данной роли. Суперпользователи баз данных могут включать или исключать любые роли из любых ролей. Роли с правом CREATEROLE могут управлять членством в любых ролях, кроме ролей суперпользователей.

3) REVOKE — отозвать права доступа. Команда REVOKE лишает одну или несколько ролей прав, назначенных ранее. Ключевое слово PUBLIC обозначает неявно определённую группу всех ролей.

• Как добавить нового пользователя в текущую базу данных?

В PostgreSQL для начала перейдем к шаблону

$ psql -Upostgres template1

Затем создадим пользователя example с паролем

password =# CREATE USER example WITH PASSWORD 'password';

Потом займемся назначением прав на пользование БД database и подключимся к ней

=# GRANT ALL PRIVILEGES ON DATABASE "database" to example;

=# \c database

Добавим все права на использование всех таблиц (определенной таблицы с именем table\_name) в базе database для example

database=# GRANT ALL PRIVILEGES ON ALL TABLES (table\_name) IN SCHEMA public TO "example";

Выходим из SQL-оболочки

database=# \q

• Как позволить пользователю заходить на сервер?

CREATE ROLE name LOGIN;

CREATE USER name WITH PASSWORD 'password';

• Какие существуют права?

Типы прав: SELECT, INSERT, UPDATE, DELETE, TRUNCATE, REFERENCES, TRIGGER, CREATE, CONNECT, TEMPORARY, EXECUTE и USAGE. Набор прав, применимых к определённому объекту, зависит от типа объекта

• Рассказать о CHECK OPTION.

Это указание управляет поведением автоматически изменяемых представлений. Если оно присутствует, при выполнении операций INSERT и UPDATE с этим представлением будет проверяться, удовлетворяют ли новые строки условию, определяющему представление. Если они не удовлетворяют условию, операция не будет выполнена. Если указание CHECK OPTION отсутствует, команды INSERT и UPDATE смогут создавать в этом представлении строки, которые не будут видны в нём. CASCADED (по умолчанию) Новые строки проверяются по условиям данного представления и всех нижележащих базовых. LOCAL Новые строки проверяются только по условиям, определённым непосредственно в самом представлении. Любые условия, определённые в нижележащих базовых представлениях, не проверяются.

**Вывод**: в ходе выполнения лабораторной работы научился создавать скрипты для наполнения базы большими объемами тестовых данных, а также познакомился с ролями в БД.