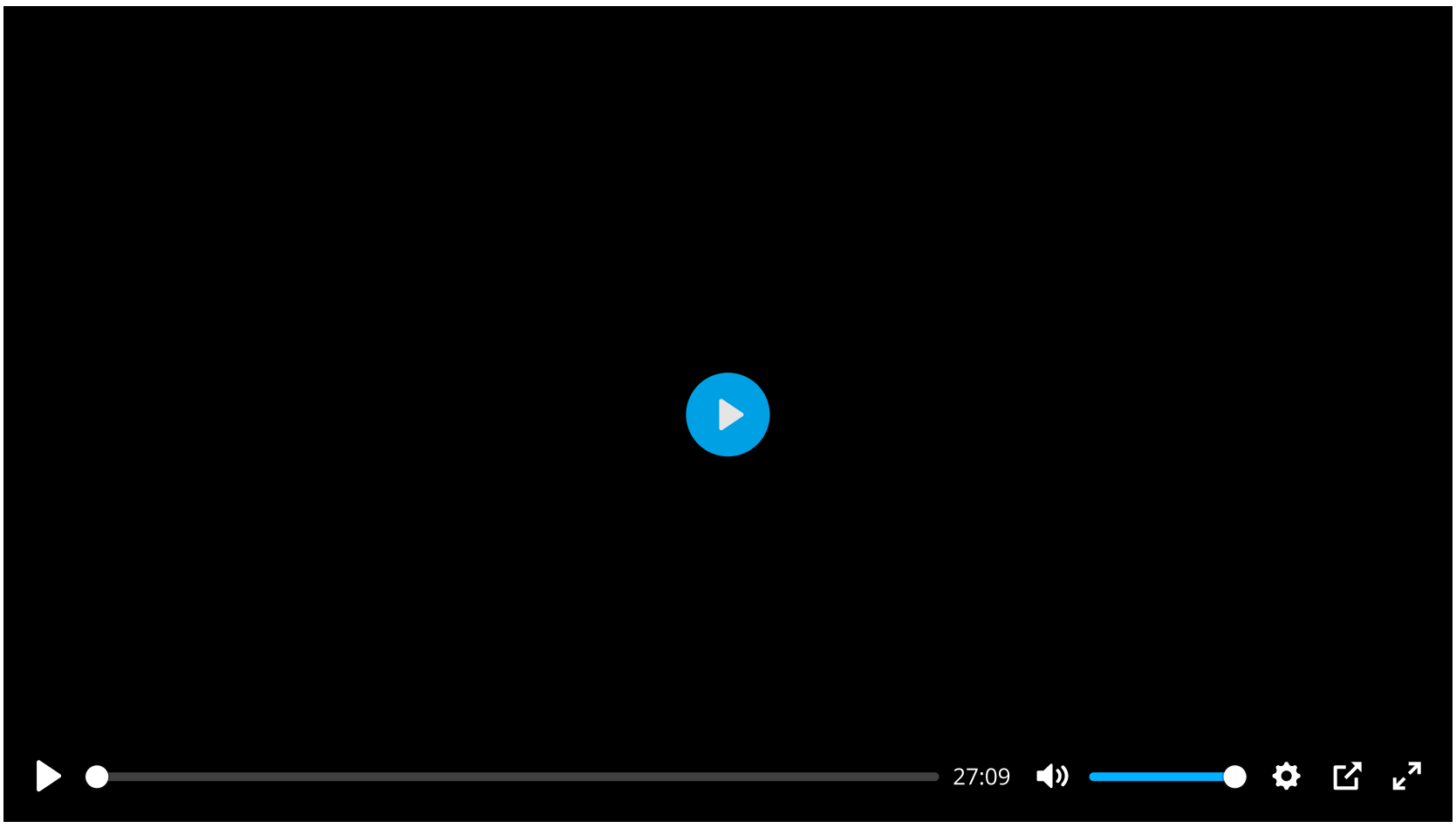
5.3 SQL. Принципы ACID.



Текстовая расшифровка видео:

НАЧАЛО РАБОТЫ С PostgreSQL

План:

- Установка PostgreSQL;
- Установка PostgreSQL в Docker с pgAdmin;
- Типы данных в PostgreSQL;
- Создание таблицы;
- Изменение таблицы;
- Вставка данных в таблицу;
- Анализируем зарплаты;
- Среднее по больнице;
- Лучшие условия;
- Подменяем записи.

Установка PostgreSQL

Поставить PostgreSQL большинство сможет самостоятельно. Если у вас Linux, то вероятнее всего у вас Ubuntu/Debian. Вы можете поставить через менеджера пакетов.

В Windows можно скачать, поставив через инсталлятор.

В macos есть несколько вариантов:

- PostgreSQL Application;
- Установка через Homebrew (данный вариант удобнее).

Установка PostgreSQL В систему (на примере Ubuntu/Debian):

```
~$ sudo apt install postgresql postgresql-contrib
```

```
~$ sudo systemctl start postgresql.service
```

К PostgreSQL можно подключиться через консоль. Консоль запускается командой «**psql**». По умолчанию создастся пользователь, который называется «Postgres». База тоже называется «Postgres».

С помощью команды `~$ sudo -u postgres` вызвать psql, подключиться через юзера Postgres или вызвать Postgres и указать «**psql -U Postgres**».

Для эксперимента мы рекомендуем создать отдельную БД, пользователя, от которого будут запускаться команды и выдать этому пользователю все привилегии на базу.

Базовые команды:

```
~$ sudo -u postgres psql
postgres=# CREATE DATABASE yourdbname;
postgres=# CREATE USER youruser WITH ENCRYPTED PASSWORD 'yourpass';
postgres=# GRANT ALL PRIVILEGES ON DATABASE yourdbname TO youruser;
```

Базовое правило: для приложения, работающего с БД, используйте техническую учетку.

Установка PostgreSQL в Docker с pgAdmin

Мы идем по простому пути.

```
1. version: '3.8'
2. services:
3.   db:
4.     container_name: pg_container
5.     image: postgres
6.    restart: always
7.     environment:
8.       POSTGRES_USER: slurm
9.       POSTGRES_PASSWORD: slurm123
10.       POSTGRES_DB: test_db
11.     ports:
11.     - "5432:5432"
12.   pgadmin:
13.     container_name: pgadmin4_container
14.     image: dpage/pgadmin4
15.     restart: always
16.     environment:
17.       PGADMIN_DEFAULT_EMAIL: admin@admin.com
18.       PGADMIN_DEFAULT_PASSWORD: slurm123
19.     ports:
20.     - "5050:80"
```

Берем файл «Docker Compose», ставим парой Postgres и инструмент «pgAdmin» (веб-интерфейс, через который можно к нему подключаться).

Выглядит файл «Docker Compose» следующим образом:

```yaml
version: '3.8'
services:
  db:
    container_name: pg_container
    image: postgres
    restart: always
    environment:
      POSTGRES_USER: slurm
      POSTGRES_PASSWORD: slurm123
      POSTGRES_DB: test_db
    ports:
    - "5432:5432"
    volumes:
      - "/run/postgresql:/run/postgresql"
      - "/tmp:/tmp"
  pgadmin:
    container_name: pgadmin4_container
    image: dpage/pgadmin4
    restart: always
    environment:
      PGADMIN_DEFAULT_EMAIL: admin@admin.com
      PGADMIN_DEFAULT_PASSWORD: slurm123
    ports:
    - "5050:80"
```

Мы изменили первоначальный вариант, добавив volumes. Это сделано с целью возможности подключения с хоста через инструмент «psql» к консоли. Также мы прокинули tmp. В нем будем создавать временные файлы с данными, на которых будем экспериментировать.

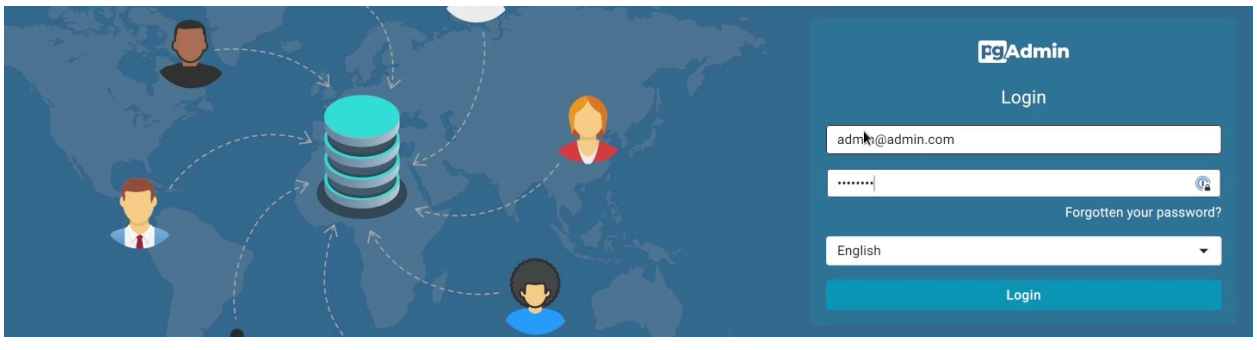
В примере видим, что мы создали двух пользователей:

- Пользователя в самой базе (в Postgres’e) с логином и паролем «slurm123», а также тестовую базу для него.
- Пользователя в pgAdmin’e, где авторизация происходит по email. Email выдуманный, логин и пароль такой же. Поскольку pgAdmin по умолчанию слушает на восьмидесятом порту, мы прокидываем его на другой порт, чтобы с хоста можно было быстро и удобно работать.

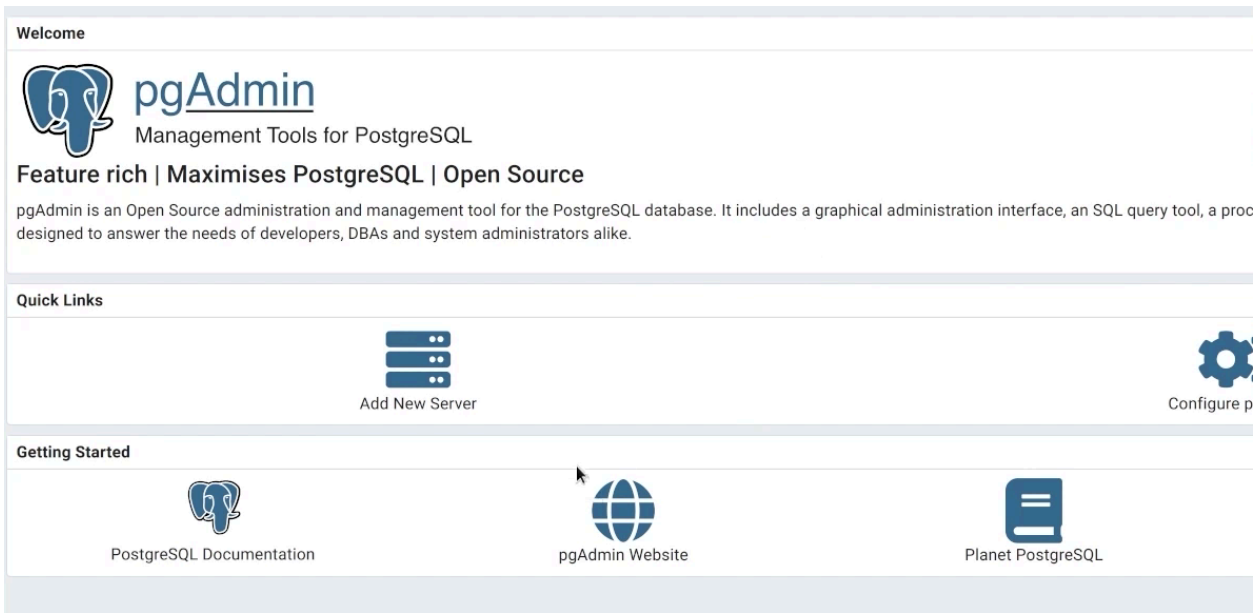
Запустим связку и посмотрим, что получится. Запуск займет какое-то время, мы ожидаем пока поднимутся Postgres и pgAdmin. Чтобы настроить между ними взаимосвязь, потребуется достать IP.

```
pg_container        | 2023-09-07 11:09:07.779 UTC [49] LOG:  checkpoint starting: shutdown immediate
pg_container        | 2023-09-07 11:09:08.029 UTC [49] LOG:  checkpoint complete: wrote 918 buffers (5.6%); 0 WAL file
led; write=0.103 s, sync=0.135 s, total=0.254 s; sync files=301, longest=0.081 s, average=0.001 s; distance=4223 kB,
pg_container        | 2023-09-07 11:09:08.038 UTC [48] LOG:  database system is shut down
pg_container        |  done
pg_container        | server stopped
pg_container        |
pg_container        | PostgreSQL init process complete; ready for start up.
pg_container        |
pg_container        |
pg_container        | 2023-09-07 11:09:08.122 UTC [1] LOG:  starting PostgreSQL 15.4 (Debian 15.4-1.pgdg120+1) on x86
gcc (Debian 12.2.0-14) 12.2.0, 64-bit
pg_container        | 2023-09-07 11:09:08.123 UTC [1] LOG:  listening on IPv4 address "0.0.0.0", port 5432
pg_container        | 2023-09-07 11:09:08.123 UTC [1] LOG:  listening on IPv6 address "::", port 5432
pg_container        | 2023-09-07 11:09:08.127 UTC [1] LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.54
pg_container        | 2023-09-07 11:09:08.138 UTC [64] LOG:  database system was shut down at 2023-09-07 11:09:08 UTC
pg_container        | 2023-09-07 11:09:08.152 UTC [1] LOG:  database system is ready to accept connections
pgadmin4_container  | NOTE: Configuring authentication for SERVER mode.
pgadmin4_container  |
pgadmin4_container  | pgAdmin 4 - Application Initialisation
pgadmin4_container  | ======================================
pgadmin4_container  |
pgadmin4_container  |
pgadmin4_container  | postfix/postlog: starting the Postfix mail system
pgadmin4_container  | [2023-09-07 11:09:20 +0000] [1] [INFO] Starting gunicorn 20.1.0
pgadmin4_container  | [2023-09-07 11:09:20 +0000] [1] [INFO] Listening at: http://[::]:80 (1)
pgadmin4_container  | [2023-09-07 11:09:20 +0000] [1] [INFO] Using worker: gthread
pgadmin4_container  | [2023-09-07 11:09:20 +0000] [91] [INFO] Booting worker with pid: 91
```

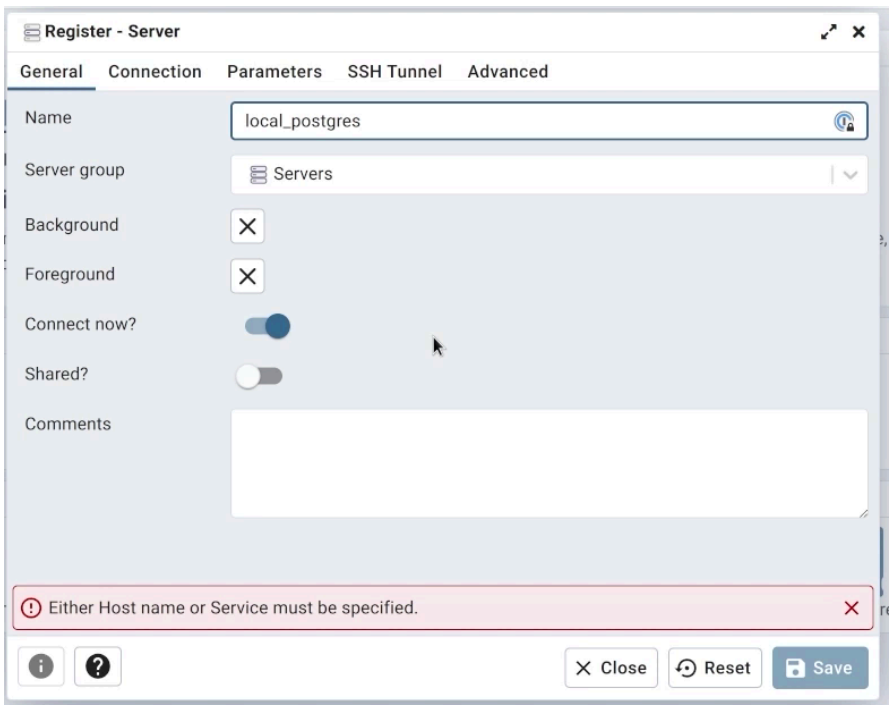
Попробуем подключиться к интерфейсу:



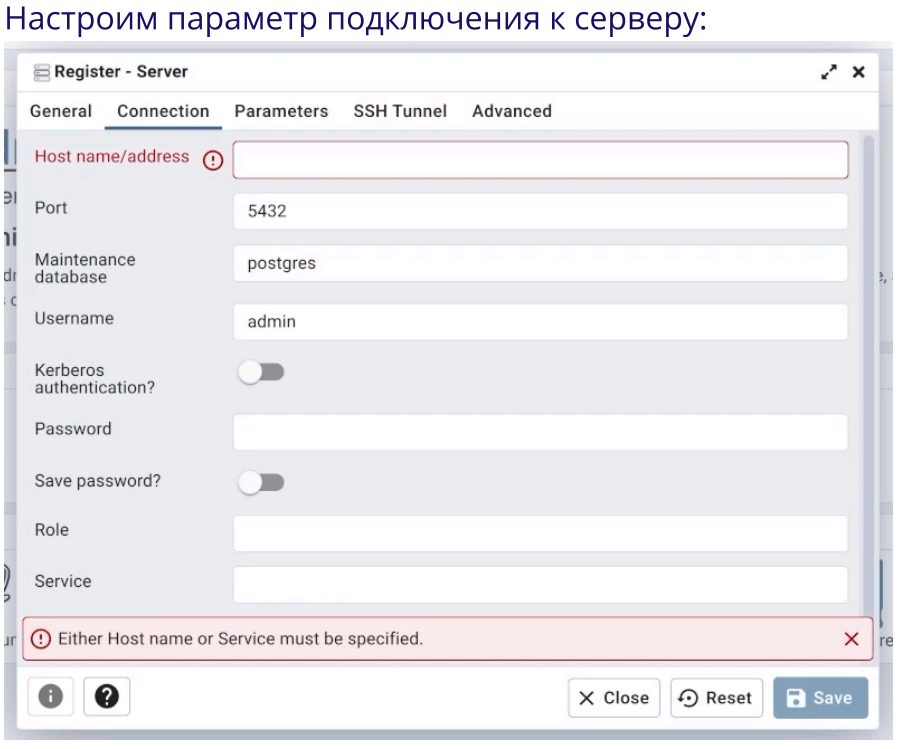
Видим интерфейс Pgadmin’a:



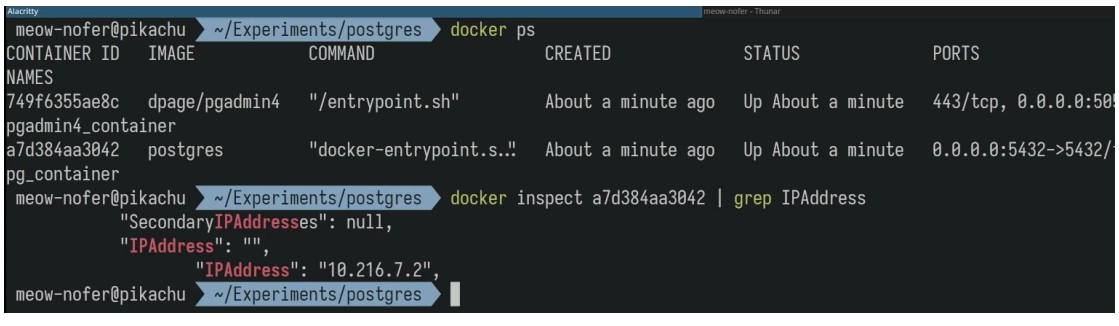
Видим, что никаких подключений нет. Добавим сервер:



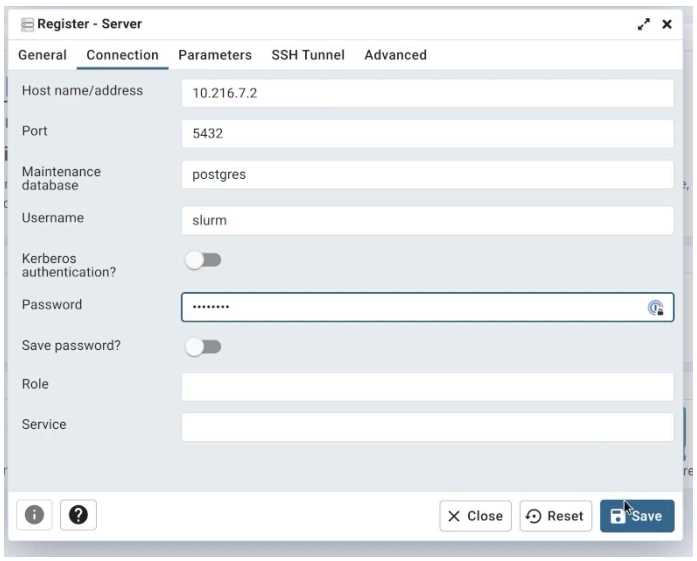
Настроим параметр подключения к серверу:



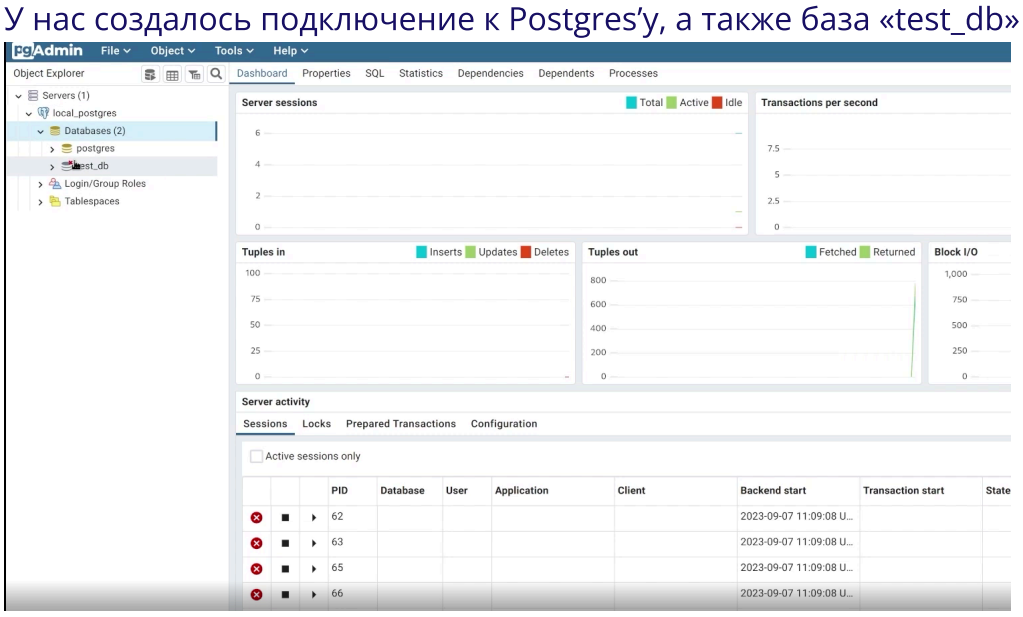
Но для начала достанем IP. Посмотрим контейнер с Postgres’ом и достанем IP-адрес:



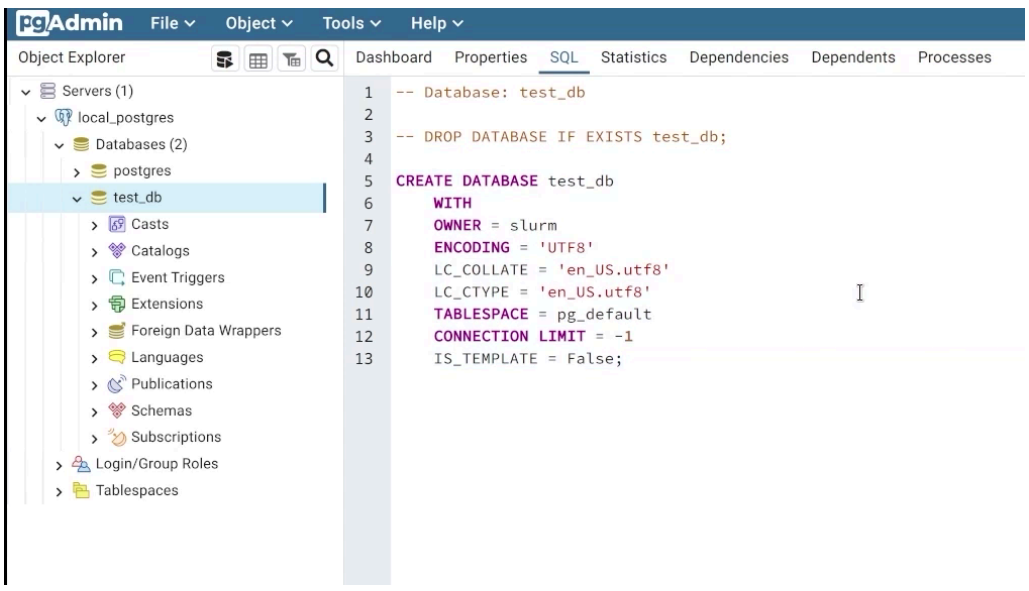
Вводим информацию:



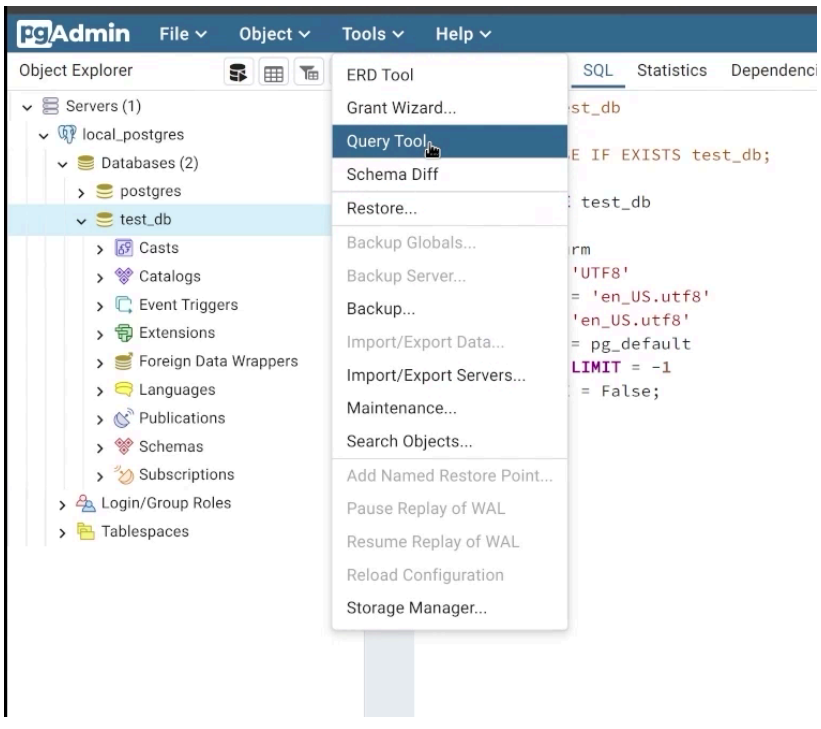
У нас создалось подключение к Postgres'у, а также база «test_db» для пользователя:

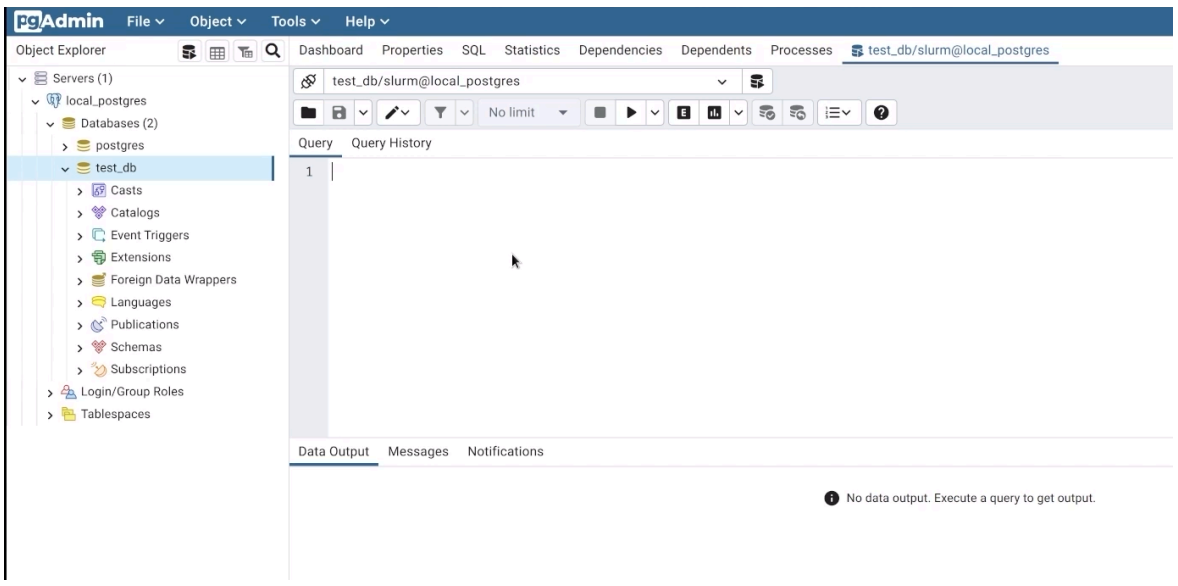


Мы можем посмотреть разные параметры с мониторингом и т.д. Мы можем посмотреть, как именно была создана БД:



Можно зайти в Query Tool, чтобы начать писать SQL-запросы:





Типы данных в PostgreSQL

Чтобы начать с этим всем работать, нужно создать таблицу, набор полей в relation’е и задать всему этому типы.

Типы данных (ранее мы их уже рассматривали):

- **Текстовые** (CHAR, VARCHAR, TEXT)
- **Целочисленные** (INTEGER, SERIAL)
- **Фиксированной точности** (DECIMAL, NUMERIC)
- **С плавающей точкой** (REAL, DOUBLE)
- **Логические** (BOOL)
- **Временные** (DATE, TIME, TIMESTAMP)
- **Бинарные** (BYTEA, JSONB)

Создание таблицы

Вызовем Create Table, зададим ему Primary Key, а также зададим его в виде serial (там будет число, за которым будет следить Postgres и инкрементить его при необходимости).

Мы можем это сделать через интерфейс или с помощью команды:

```
meow-nofer@pikachu ~/Experiments/postgres psql -U slurm -d test_db
```

Обратите внимание, что мы подключаемся пользователем «slurm», а также подключаемся к тестовой базе, которую мы тоже сделали. Видим, что все работает:

```
meow-nofer@pikachu ~/Experiments/postgres psql -U slurm -d test_db
psql (15.4)
Type "help" for help.

test_db=#
```

Вызовем данный код создания таблицы:

```
CREATE TABLE employees (
    employee_id serial PRIMARY KEY,
    name VARCHAR(100),
    salary INTEGER,
    title VARCHAR(100),
    responsibilities TEXT
```

Все отработало:

```
test_db=# CREATE TABLE employees (
        employee_id serial PRIMARY KEY,
        name VARCHAR(100),
    salary INTEGER,
        title VARCHAR(100),
        responsibilities TEXT
);
CREATE TABLE
test_db=#
```

Чтобы посмотреть, какие есть сущности в текущей базе, можно использовать специальные команды Postgres. Есть два варианта:

- Использовать сокращенные варианты со слэшами (\);
- Делать выборки из таблички «Information schema», которая содержит информацию о разных метаданных.

Создалась таблица «Employees», а также создался sequence (отслеживает порядок вставку Primary key):

```
);
CREATE TABLE
test_db=# \d
                 List of relations
 Schema |           Name             |   Type   | Owner
--------+----------------------------+----------+-------
 public | employees                  | table    | slurm
 public | employees_employee_id_seq | sequence | slurm
(2 rows)

test_db=#
```

Существует следующая команда:

```
test_db=# \d+
```

Она показывает более расширенную информацию:

```
test_db=# \d+
                                        List of relations
 Schema |           Name            |   Type   | Owner | Persistence | Access method |    Size    | Description
--------+---------------------------+----------+-------+-------------+---------------+------------+-------------
 public | employees                 | table    | slurm | permanent   | heap          | 8192 bytes |
 public | employees_employee_id_seq | sequence | slurm | permanent   |               | 8192 bytes |
(2 rows)
```

- настройки персистентности;
- формат табличек (по умолчанию – heap).

Можно посмотреть конкретную табличку и ее описание при помощи команды:

```
test_db=# \d employees
```

Мы увидим следующее:

```
test_db=# \d employees
                                    Table "public.employees"
     Column      |          Type          | Collation | Nullable |                   Default
-----------------+------------------------+-----------+----------+---------------------------------------------
 employee_id     | integer                |           | not null | nextval('employees_employee_id_seq'::regclass)
 name            | character varying(100) |           |          |
 salary          | integer                |           |          |
 title           | character varying(100) |           |          |
 responsibilities | text                  |           |          |
Indexes:
    "employees_pkey" PRIMARY KEY, btree (employee_id)
```

В инструменте «psql» мы можем использовать подобные шорткаты:

```
~$ psql -U postgres    # пользователь по умолчанию
~$ psql -U slurm -d test_db    # наш случай

=# \c my_db    # подключиться к базе
=# \d    # посмотреть объекты в базе
=# \d+    # посмотреть объекты в базе подробнее
=# \d my_table    # посмотреть структуру таблицы
```

Однако верхние четыре команды не сработают в pgAdmin, потому что они относятся к выполнению запросов. Данные команды нужны для того, чтобы выполнять их через psql. **Все, что начинается с обратного слэша (\) –команды psql.**

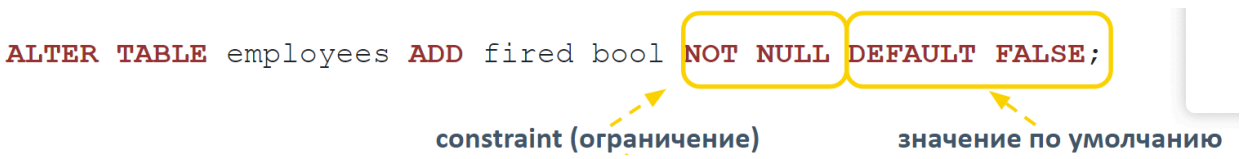
Ссылка на почитать

Изменение таблицы

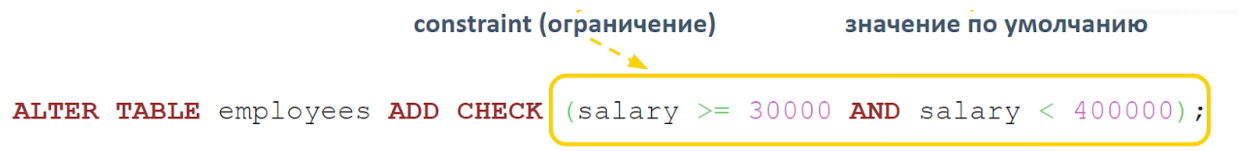
Мы можем модифицировать таблицу, вызвав **Alter Table** и добавив дополнительное поле. Также мы можем добавить дополнительное ограничение (constraint).

Мы можем ограничить диапазон значений, которые конкретная колонка может принимать. Например, типичный вопрос: включать ли туда null?

Мы можем создать колонку, вызвав Alter Table, в которой будет указано «not null», и присвоить ей значение по умолчанию:



Мы можем задать более произвольные проверки, например, используя **Check** (дополнительное ключевое слово):



Можем сказать: «пусть в табличке “Employees” будет ограничение: зарплата должна быть в N-диапазоне». Если мы попытаемся вставить запись, у которой поле «Salary» будет за пределами этого диапазона, то появится ошибка, сообщающая о том, что так нельзя.

Основные положения, о которых необходимо знать:

- Чем **больше данных** у вас в таблице, тем медленнее будет идти операция преобразования.
- Изменение структуры таблиц в боевой базе иногда неизбежно: этот процесс называется **миграцией**.

Ссылка на почитать

Вставка данных в таблицу

Зальем данные в таблицу. Самый простой способ, чтобы это сделать, – вставить запись вручную:

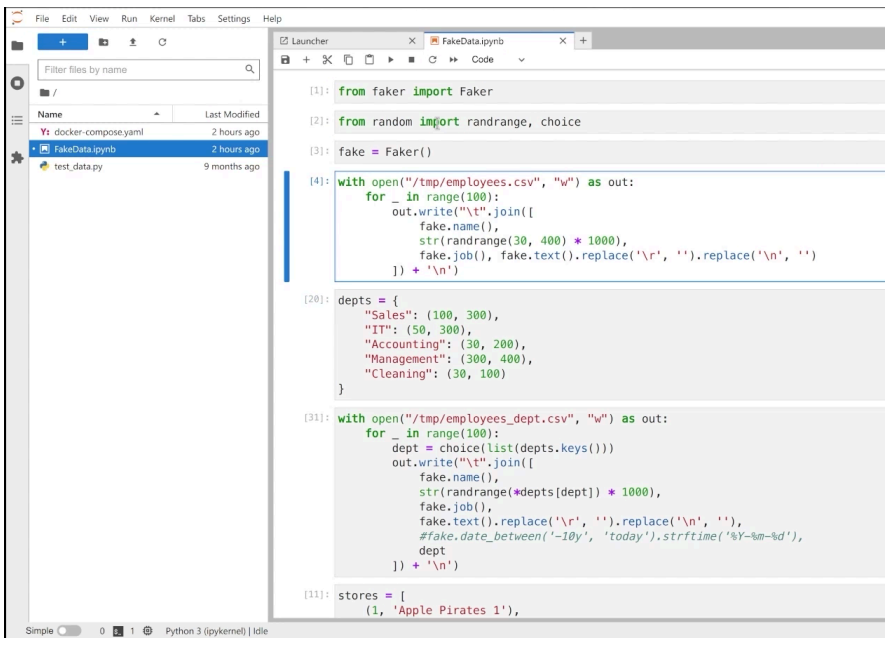
```
INSERT INTO employees
(name, salary, title, responsibilities)
VALUES
('Jon Snow', 500000, 'King of the North', 'Rule Westeros');
```

Поскольку чаще всего данные приходят извне, вполне нормально использовать для вставки файл. Сгенерируем такой файл:

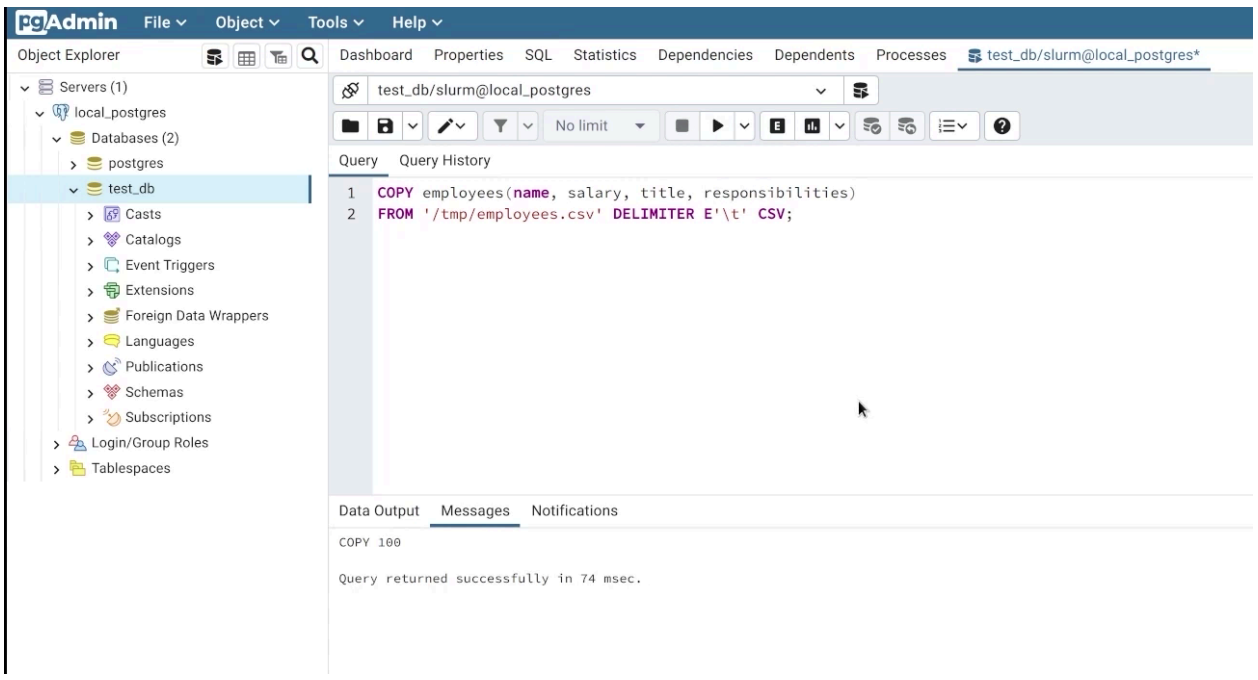
```
\copy employees(name, salary, title, responsibilities)
FROM '/tmp/employees.csv' DELIMITER E'\t' CSV;
```

Обратите внимание, что мы неслучайно сделали акцент на том, что команды, начинающиеся со слэша, – команды psql. Поэтому, если мы данную команду будем выполнять через pgAdmin, нам придется использовать не «\copy», а просто «copy». **Будьте внимательны, чаще всего в этом происходит путаница.**

В качестве примера мы написали фейковый генератор:



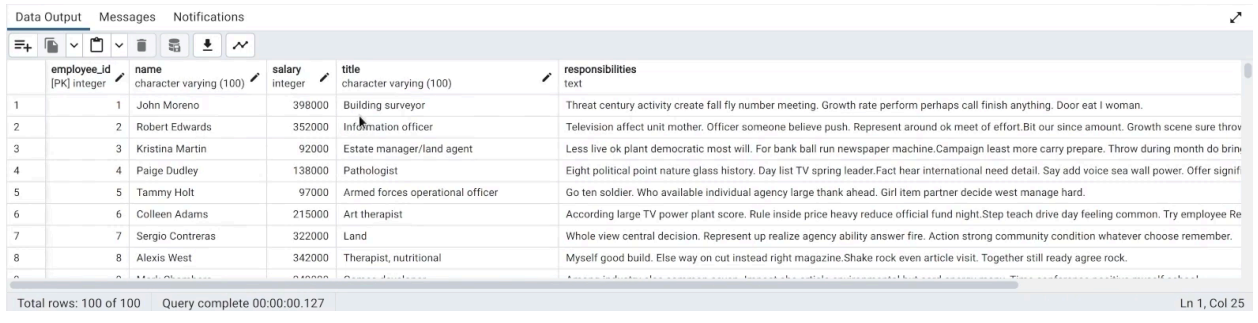
Зальем данные в таблицу:



Все получилось.

Посмотрим, что залилось:

```
1  SELECT * FROM employees;
```



Каждому следующему пользователю присвоен айдишник. Видим, что у них какая-то зарплата, должность и т.д.

Обратите внимание, что в списке полей мы не указывали айдишник, потому что он считается автоматически.

Также мы можем указывать разные разделители, если читаем из CSV. Чтение более «хитрых» форматов, например, Parquet, достигается при помощи дополнительных плагинов.

Анализируем зарплаты

Мы знаем следующее:

Достать все колонки и все записи:

```
SELECT * FROM employees; ← почти всегда антипаттерн!
```

Достать конкретные колонки и 3 записи:

```
SELECT name, salary FROM employees LIMIT 3;
```

Фильтр по условиям:

```
SELECT name, salary FROM employees
WHERE salary > 350000 AND name != 'Jon Snow';
```

Запрос диапазона:

```
SELECT name, salary FROM employees
WHERE salary BETWEEN 30000 AND 100000;
```

Остановимся на важном моменте: когда вы работаете с большими данными, вы можете столкнуться с ситуацией, где таблица получается весьма широкой. Когда вы делаете «SELECT *», вы поднимаете с диска гигантское количество данных несмотря на то, что большинство из них вам не нужно. Эта проблема возникает не только у дата-инженеров, но и у дата-аналитиков, так как они вычитывают для анализа все поля из базы.

Возьмите за правило: используйте явное перечисление набора полей!

В некоторых базах даже есть функция на запрет «SELECT *».

Среднее по больнице

Вспомним агрегирующие функции.

Достать все колонки и все записи:

```
SELECT COUNT(*) FROM employees WHERE salary > 50000;
```

Средняя зарплата:

```
SELECT AVG(salary) FROM employees;
```

Сколько денег тратим на все зарплаты:

```
SELECT SUM(salary) FROM employees;
```

Почитать про COUNT

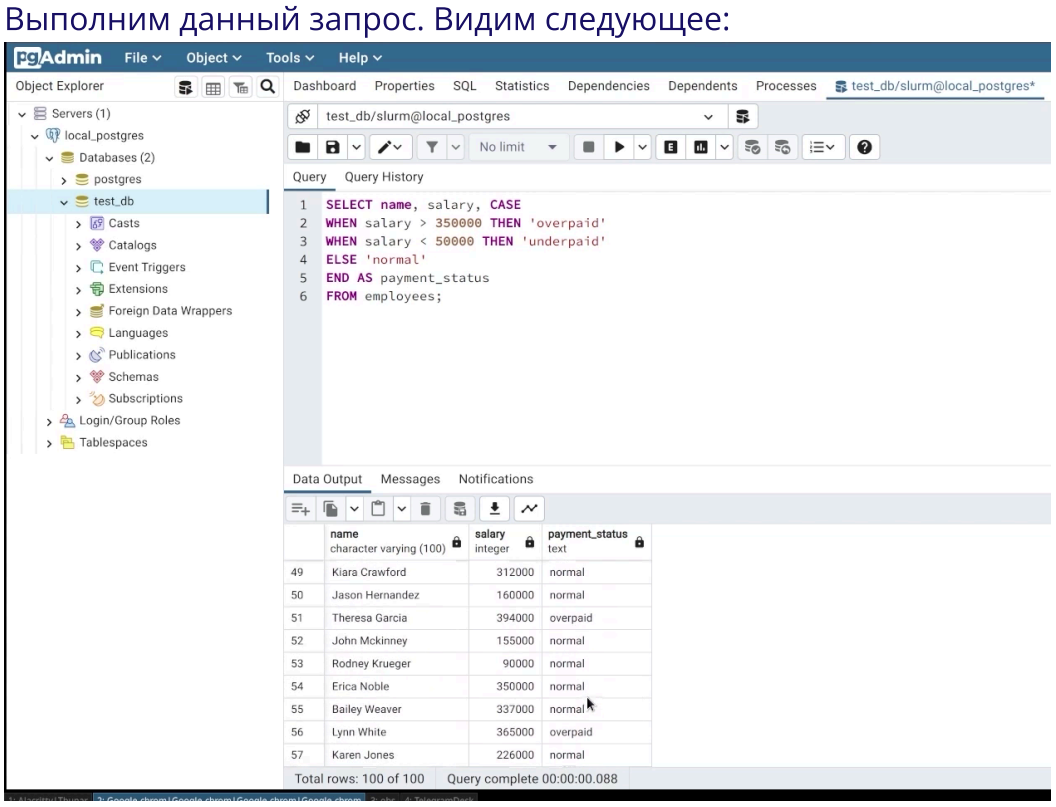
Лучшие условия

Многие знают об «Order by» и о том, что можно создавать порядок сортировки (по возрастанию, по убыванию). Существует также «Case When», «Case Then». Это декларативный «If» в мире SQL.

Мы можем сказать: «Выбери колонки “Имя”, “Зарплата”. Для третьей колонки в случае N-значения в колонке “Зарплата” выведи одно, в случае другого значения выведи другое, в случае третьего значения – третье»:

```
SELECT name, salary, CASE
WHEN salary > 350000 THEN 'overpaid'
WHEN salary < 50000 THEN 'underpaid'
ELSE 'normal'
END AS payment_status
FROM employees;
```

Выполним данный запрос. Видим следующее:



Подменяем записи

С помощью «Case Then» создадим новую колонку и ее заполним. Мы можем выставить ей значение на основе других колонок.

Если создать колонку «Fired», мы можем выставить ее значение и выставить «Fired=True» для тех людей, которым мы в примере переплачиваем.

Выполним данный код (**увольнение других сотрудников**):

```
UPDATE employees SET fired = CASE
WHEN paid_status = 'overpaid' THEN TRUE
ELSE FALSE
END;
```

Добавим поле «Fired»:

```
ALTER TABLE employees ADD fired bool NOT NULL DEFAULT FALSE;
```

И сделаем следующее (обновление значений):

```
UPDATE employees SET paid_status = CASE
WHEN salary > 350000 THEN 'overpaid'
WHEN salary < 50000 THEN 'underpaid'
ELSE 'normal' END;
```

А также добавим колонку «Paid status».

Промежуточно видим следующее:

```
test_db=# \d employees
                                    Table "public.employees"
     Column      |          Type          | Collation | Nullable |                  Default
-----------------+------------------------+-----------+----------+-------------------------------------------
 employee_id     | integer                |           | not null | nextval('employees_employee_id_seq'::regclass)
 name            | character varying(100) |           |          |
 salary          | integer                |           |          |
 title           | character varying(100) |           |          |
 responsibilities | text                  |           |          |
Indexes:
    "employees_pkey" PRIMARY KEY, btree (employee_id)

test_db=# ALTER TABLE employees ADD fired bool NOT NULL DEFAULT FALSE;

ALTER TABLE
test_db=# UPDATE employees SET paid_status = CASE
WHEN salary > 350000 THEN 'overpaid'
WHEN salary < 50000 THEN 'underpaid'
ELSE 'normal' END;
ERROR:  column "paid_status" of relation "employees" does not exist
LINE 1: UPDATE employees SET paid_status = CASE
                             ^
test_db=# ALTER TABLE employees ADD paid_status NOT NULL DEFAULT 'normal';
ERROR:  syntax error at or near "NOT"
LINE 1: ALTER TABLE employees ADD paid_status NOT NULL DEFAULT 'norm...
                                              ^
test_db=# ALTER TABLE employees ADD paid_status VARCHAR(20) NOT NULL DEFAULT 'normal';
ALTER TABLE
test_db=# ALTER TABLE employees ADD paid_status NOT NULL DEFAULT 'normal';
[0] 0:python3.10  1:docker-compose- 2:psql*
```

Теперь заполним:

```
Alacritty                                                                    meow-nofer - Thunar
 name             | character varying(100) |           |          |
 salary           | integer                |           |          |
 title            | character varying(100) |           |          |
 responsibilities | text                   |           |          |
Indexes:
    "employees_pkey" PRIMARY KEY, btree (employee_id)

test_db=# ALTER TABLE employees ADD fired bool NOT NULL DEFAULT FALSE;

ALTER TABLE
test_db=# UPDATE employees SET paid_status = CASE
WHEN salary > 350000 THEN 'overpaid'
WHEN salary < 50000 THEN 'underpaid'
ELSE 'normal' END;
ERROR:  column "paid_status" of relation "employees" does not exist
LINE 1: UPDATE employees SET paid_status = CASE
                             ^
test_db=# ALTER TABLE employees ADD paid_status NOT NULL DEFAULT 'normal';
ERROR:  syntax error at or near "NOT"
LINE 1: ALTER TABLE employees ADD paid_status NOT NULL DEFAULT 'norm...
                                              ^
test_db=# ALTER TABLE employees ADD paid_status VARCHAR(20) NOT NULL DEFAULT 'normal';
ALTER TABLE
test_db=# UPDATE employees SET paid_status = CASE
WHEN salary > 350000 THEN 'overpaid'
WHEN salary < 50000 THEN 'underpaid'
ELSE 'normal' END;
UPDATE 100
test_db#
```

Посмотрим, что получилось:

```
1  SELECT * FROM employees;
```

Data Output | Messages | Notifications

	responsibilities text	fired boolean	paid_status character varying (20)
1	Threat century activity create fall fly number meeting. Growth rate perform perhaps call finish anything. Door eat I woman.	false	overpaid
2	Television affect unit mother. Officer someone believe push. Represent around ok meet of effort.Bit our since amount. Growth scene sure throw. Cup home like watch trial class.	false	overpaid
3	Less live ok plant democratic most will. For bank ball run newspaper machine.Campaign least more carry prepare. Throw during month do bring save worker.	false	normal
4	Eight political point nature glass history. Day list TV spring leader.Fact hear international need detail. Say add voice sea wall power. Offer significant region use.	false	normal
5	Go ten soldier. Who available individual agency large thank ahead. Girl item partner decide west manage hard.	false	normal
6	According large TV power plant score. Rule inside price heavy reduce official fund night.Step teach drive day feeling common. Try employee Republican hospital.	false	normal
7	Whole view central decision. Represent up realize agency ability answer fire. Action strong community condition whatever choose remember.	false	normal
8	Defense attack for cold. Entire hour painting since note.	false	normal

Total rows: 100 of 100 | Query complete 00:00:00.093 | Ln 1, Col 25

Теперь уволим людей, которым мы переплачиваем:

Query | Query History

```
1  UPDATE employees SET fired = CASE
2  WHEN paid_status = 'overpaid' THEN TRUE
3  ELSE FALSE
4  END;
```

Data Output | Messages | Notifications

```
UPDATE 100

Query returned successfully in 100 msec.
```

Мы провели update, выставили значение в колонке на основе условий.

Посмотрим еще раз:

Query | Query History

```
1  SELECT * FROM employees;
```

Data Output | Messages | Notifications

	responsibilities text	fired boolean	paid_status character varying (20)
93	Player list purpose avoid. Western week yard various. Modern expect kind maintain appear hit hundred.Whose ago upon any item surface big. Knowledge long strategy.	true	overpaid
94	Skill have painting ahead. Would decide yes security this always inside. Blue the gas always young dinner. Determine simple majority certain.	false	normal
95	Idea why address. Describe also body specific bank west. Thank story old scientist commercial interest take. Involve guess effort piece fast.	false	normal
96	Nation drive three national. Left material money nothing. Quality crime beat ground course issue interest outside.	false	normal
97	Religious method writer let or begin focus. Positive home author husband certainly. Relationship green onto seven marriage.	false	normal
98	Will floor in benefit owner bag. Learn hundred artist claim admit reality.Official thus either human. Hair off special around ready direction lawyer.	false	normal
99	Floor main story likely me. Continue prepare thus range. Loss suffer bring foreign memory system.Think wonder year nothing during third. Turn billion trouble leg participant decade type student.	true	overpaid
100	Front public current machine. Field if can everything. Professional table bar.Join something trip similar. Position option hope.Religious peace during finish land. Market mean yet table.	true	overpaid

Total rows: 100 of 100 | Query complete 00:00:00.079 | Ln 1, Col 25

Видим, что началось увольнение.

Как вам урок?

Изучил, далее >