Базы данных для индустриального программирования

Пальчевский Евгений Владимирович

Кандидат технических наук

Доцент кафедры индустриального программирования

О курсе

Группы в потоке: ЭФБО-01-24 – ЭФБО-06-24

Что ждёт в осеннем семестре 2025/26 учебного года?

1. **Лекции (8 штук) и семинары (16 штук)**
2. **Выполнение индивидуальных заданий в виде контрольных работ + их защита. Работаем в группе по 4-5 человек**
3. **За всё получаем баллы в соответствии с балльно-рейтинговой системой (БРС)**
4. **Зачет (сдаем, будут вопросы из опросника на сайте)**

Посещения – 30 баллов. Зачет – 30 баллов. Текущий контроль (то, что будем делать в течение семестра):

Наименование	Формат + защита	Даты проведения	Баллы
Контрольная работа 1	Очно (на занятиях)	01.09.2025 – 30.09.2025	4
Контрольная работа 2	Очно (на занятиях)	01.10.2025 – 31.10.2025	5
Контрольная работа 3	Очно (на занятиях)	01.11.2025 – 30.11.2025	6
Контрольная работа 4	Очно (на занятиях)	01.11.2025 – 30.11.2025	7
Контрольная работа 5	Очно (на занятиях)	01.12.2025 – 20.12.2025	8
Вопросы из опросника	Очно (на занятиях)	01.12.2025 – 20.12.2025	10
Итого за семестр			**40**

Если сумма «Посещения + Текущий контроль» < 10, то студент до зачета не допускается. Зачтено – > 40, менее или равно 40 – не зачтено

Базы данных для индустриального программирования

Пальчевский Евгений Владимирович

Кандидат технических наук

Доцент кафедры индустриального программирования

Инструкции и материалы

Ответы на контрольные загружаем сюда (проекты либо ссылки на гит):

Максимальная сумма баллов по мероприятиям текущего контроля по дисциплине в течение учебного семестра			40
Наименование	Формат	Даты проведения	Баллы
Контрольная работа 1	Очно (на занятиях)	01.09.2025 - 30.09.2025	4
Контрольная работа 2	Очно (на занятиях)	01.10.2025 - 31.10.2025	5
Контрольная работа 3	Очно (на занятиях)	01.11.2025 - 30.11.2025	6
Контрольная работа 4	Очно (на занятиях)	01.11.2025 - 30.11.2025	7
Контрольная работа 5	Очно (на занятиях)	01.12.2025 - 20.12.2025	8
Тест	СДО	01.12.2025 - 20.12.2025	10

После загрузки ответов показываем работоспособность программы

Социальная сеть / мессенджер / почта	Контакт
Электронная почта	teelxp@inbox.ru
VK	https://vk.com/teelxp
WhatsApp	+7-937-485-80-48
YouTube-канал с видеолекциями	https://www.youtube.com/@teelxp
Лекции в ВК (альтернатива ютубчику)	https://vk.com/finkablog

Базы данных для индустриального программирования

Пальчевский Евгений Владимирович

Кандидат технических наук

Доцент кафедры индустриального программирования

Софт и языки программирования

Логотип	Название	Группы
	IntelliJ IDEA Ultimate	https://palchevsky.ru/materials.php
	Visual Studio Code	https://code.visualstudio.com/
	PostgreSQL	https://www.postgresql.org/download/
	PyCharm	https://palchevsky.ru/materials.php

Язык программирования + SQL	Для чего нужна связка языка программирования и SQL?
Python + SQL	Создание графического приложения для работы с информацией через базу данных. Пример на Python: GitHub
Java + SQL	
C# + SQL	
C++ + SQL	

Лекция №1

Тема: «Модели организации данных и PostgreSQL»

Модель данных представляет собой способ описания и структурирования данных, а также операций над ними.

Основные модели данных:

1. Иерархическая модель
2. Сетевая модель
3. Реляционная (основная) модель
4. Объектно-ориентированная модель

Лекция №1

Тема: «Модели организации данных и PostgreSQL»

Подтема: «Иерархическая модель данных»

Иерархическая модель данных

Основная суть модели заключается в том, что она основана на древовидной структуре: каждая запись (узел) имеет ровно одного родителя и множество потомков.

Если говорить об иерархической модели данных как об отдельной модели БД, а не о приёме реализации структуры хранения данных в реляционных СУБД, то исторически и сегодня её чаще всего применяют там, где данные изначально деревообразные. Т.е. в NoSQL (нереляционных) базах данных.

1.JSON-ориентированные БД, например, MongoDB

2. XML-базы данных, например, BaseX

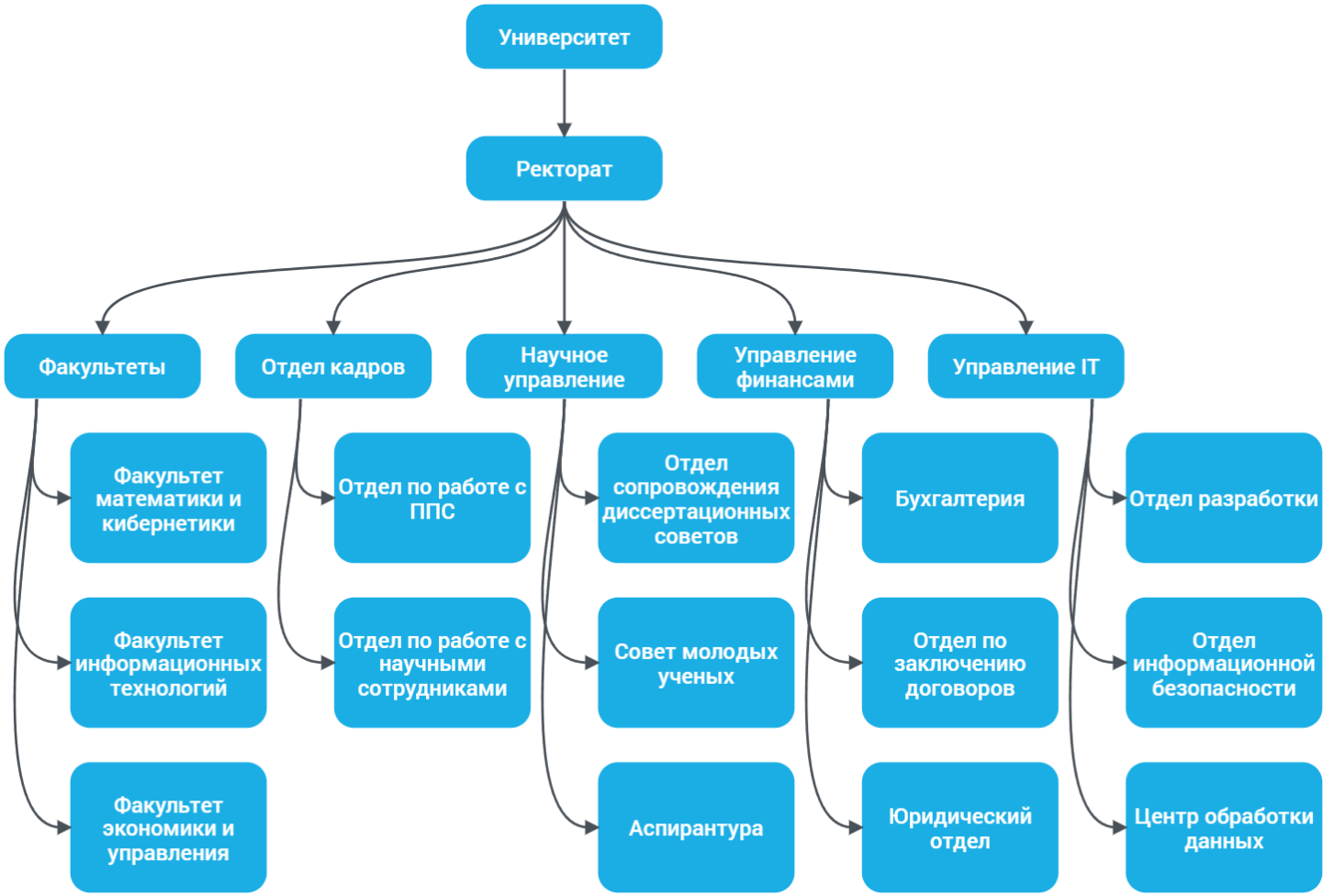
3. LDAP-каталоги (Active Directory (Microsoft), OpenLDAP), файловые системы (NTFS и др.), ERP/CRM-системы

Где применяются? Различные компании, классификаторы, каталоги, документы (JSON/XML)

Лекция №1

Тема: «Модели организации данных и PostgreSQL»

Подтема: «Иерархическая модель данных»



Лекция №1

Тема: «Модели организации данных и PostgreSQL»

Подтема: «Иерархическая модель данных»

Как выглядит в MongoDB

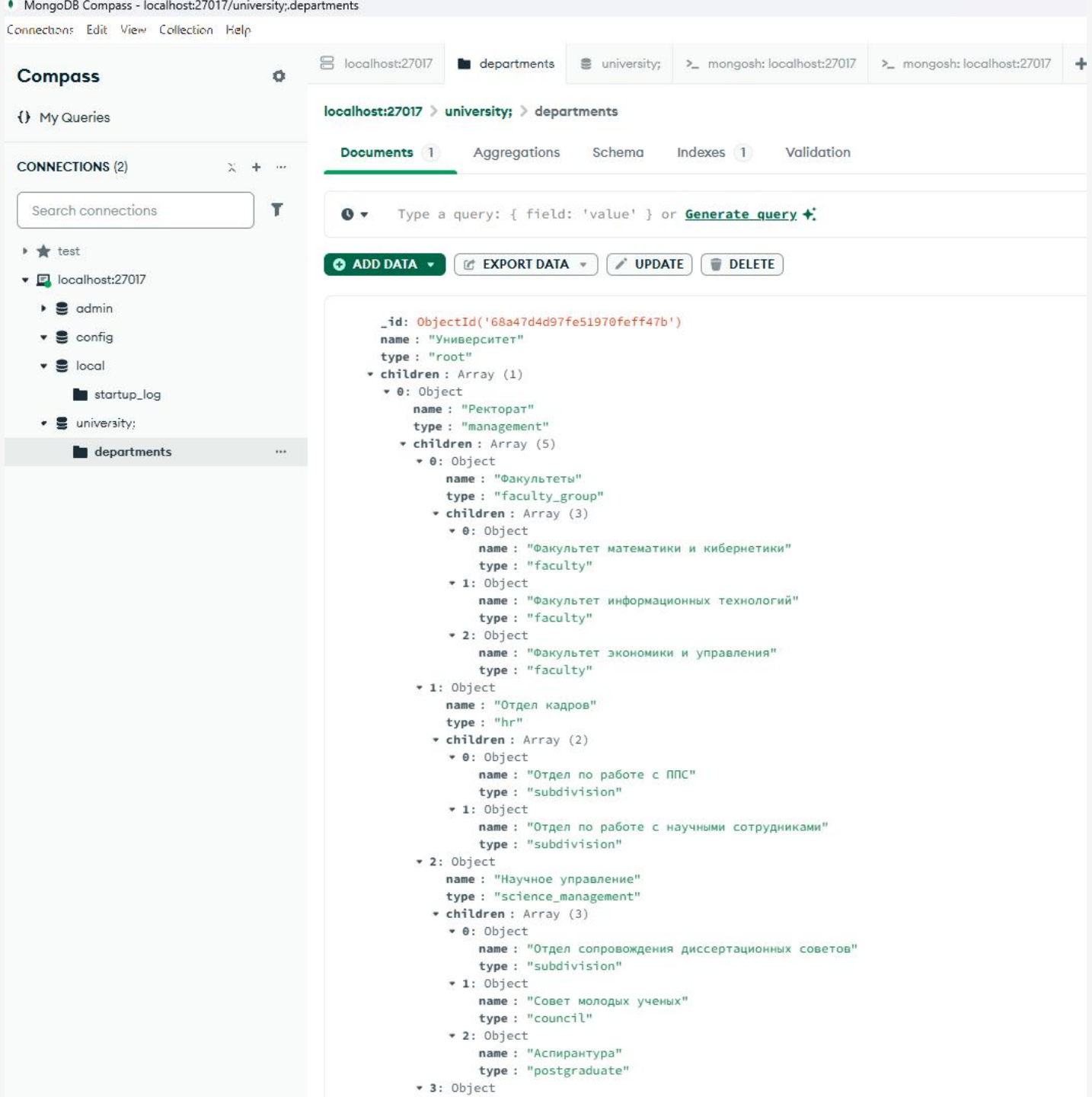
```
use university;

db.departments.insertOne({ name: "Университет", type: "root", children: [
 { name: "Ректорат", type: "management", children: [
  { name: "Факультеты", type: "faculty_group", children: [
   { name: "Факультет математики и кибернетики", type: "faculty" },
   { name: "Факультет информационных технологий", type: "faculty" },
   { name: "Факультет экономики и управления", type: "faculty" }
  ]},
  { name: "Отдел кадров", type: "hr", children: [
   { name: "Отдел по работе с ППС", type: "subdivision" },
   { name: "Отдел по работе с научными сотрудниками", type: "subdivision" }
  ]},
  { name: "Научное управление", type: "science_management", children: [
   { name: "Отдел сопровождения диссертационных советов", type: "subdivision" },
   { name: "Совет молодых ученых", type: "council" },
   { name: "Аспирантура", type: "postgraduate" }
  ]},
  { name: "Управление финансами", type: "finance", children: [
   { name: "Бухгалтерия", type: "finance_subdivision" },
   { name: "Отдел по заключению договоров", type: "legal" },
   { name: "Юридический отдел", type: "legal" }
  ]},
  { name: "Управление IT", type: "it_management", children: [
   { name: "Отдел разработки", type: "it_subdivision" },
   { name: "Отдел информационной безопасности", type: "it_security" },
   { name: "Центр обработки данных", type: "data_center" }
  ]}
 ]}
]});
```

Лекция №1

Тема: «Модели организации данных и PostgreSQL»

Подтема: «Иерархическая модель данных»

Как выглядит в MongoDB



Лекция №1

Тема: «Модели организации данных и PostgreSQL»

Подтема: «Иерархическая модель данных»

Как выглядит в PostgreSQL

```
-- Удаляем схему university, если она существует, вместе со всеми объектами внутри неё
DROP SCHEMA IF EXISTS university CASCADE;

-- Создаем новую пустую схему university
CREATE SCHEMA university;

-- Устанавливаем search_path, чтобы все таблицы и запросы по умолчанию относились к схеме university
SET search_path TO university;

-- ------------------------------------------------------------
-- Создаем таблицу departments для хранения иерархии подразделений
-- ------------------------------------------------------------
CREATE TABLE departments (
    id        SERIAL PRIMARY KEY,          -- уникальный идентификатор строки (генерируется автоматически)
    name      TEXT NOT NULL,               -- название подразделения (обязательно для заполнения)
    type      TEXT,                        -- тип подразделения (например: root, faculty, hr, management и т.п.)
    parent_id INT REFERENCES departments(id) -- ссылка на родительский элемент (идентификатор другого подразделения)
              ON DELETE CASCADE            -- если удалить родителя, то автоматически удаляются все дочерние элементы
);

-- ------------------------------------------------------------
-- 1. Вставляем корневой элемент "Университет" без родителя
-- 2. Создаем "Ректорат" как дочерний элемент "Университета"
-- 3. Создаем группу "Факультеты" как дочерний элемент "Ректората"
-- 4. Добавляем три факультета внутрь этой группы
-- ------------------------------------------------------------
WITH root AS (  -- временный набор данных, содержащий id корневого элемента
    INSERT INTO departments (name, type, parent_id)
    VALUES ('Университет', 'root', NULL)   -- вставляем корень "Университет"
    RETURNING id                           -- возвращаем его id для дальнейшего использования
),
rectorate AS (  -- временный набор данных для "Ректората"
    INSERT INTO departments (name, type, parent_id)
    SELECT 'Ректорат', 'management', id    -- вставляем "Ректорат" как дочерний элемент "Университета"
    FROM root
    RETURNING id                           -- сохраняем id "Ректората"
),
fac_group AS (  -- временный набор данных для "Факультетов"
    INSERT INTO departments (name, type, parent_id)
    SELECT 'Факультеты', 'faculty_group', id  -- создаем группу факультетов внутри "Ректората"
    FROM rectorate
    RETURNING id                           -- сохраняем id группы факультетов
)
INSERT INTO departments (name, type, parent_id) -- добавляем сами факультеты
SELECT
    unnest(ARRAY[                          -- разворачиваем массив названий факультетов в строки
        'Факультет математики и кибернетики',
        'Факультет информационных технологий',
        'Факультет экономики и управления'
    ]),
    'faculty',                             -- для всех указываем тип "faculty"
    id                                     -- связываем их с id группы факультетов
FROM fac_group;
```

Лекция №1

Тема: «Модели организации данных и PostgreSQL»

Подтема: «Иерархическая модель данных»

Как выглядит в PostgreSQL

```
-- ------------------------------------------------------------
-- Добавляем HR-блок в Ректорат и его подразделения
-- ------------------------------------------------------------
WITH rectorate AS (                    -- создаём временный набор данных rectorate
    SELECT id
    FROM departments
    WHERE name = 'Ректорат'            -- выбираем id подразделения "Ректорат"
),
hr AS (                                -- создаём временный набор данных hr
    INSERT INTO departments (name, type, parent_id)
    SELECT 'Отдел кадров',             -- название подразделения
           'hr',                       -- тип: hr (кадровый отдел)
           id                          -- parent_id = id "Ректората"
    FROM rectorate
    RETURNING id                       -- возвращаем id созданного отдела кадров
)
INSERT INTO departments (name, type, parent_id)  -- вставляем подчинённые подразделения HR
SELECT
    unnest(ARRAY[                      -- разворачиваем массив названий в строки
        'Отдел по работе с ППС',       -- первый подраздел HR
        'Отдел по работе с научными сотрудниками' -- второй подраздел HR
    ]),
    'subdivision',                     -- общий тип для этих подразделений
    id                                 -- parent_id = id отдела кадров
FROM hr;

-- ------------------------------------------------------------
-- Добавляем Научное управление и его подразделения
-- ------------------------------------------------------------
WITH rectorate AS (                    -- создаём временный набор данных rectorate
    SELECT id
    FROM departments
    WHERE name = 'Ректорат'            -- выбираем id подразделения "Ректорат"
),
sci AS (                               -- создаём временный набор данных sci
    INSERT INTO departments (name, type, parent_id)
    SELECT 'Научное управление',       -- название: Научное управление
           'science_management',       -- тип: управление наукой
           id                          -- parent_id = id "Ректората"
    FROM rectorate
    RETURNING id                       -- возвращаем id Научного управления
)
INSERT INTO departments (name, type, parent_id)  -- вставляем подчинённые подразделения науки
SELECT
    unnest(ARRAY[                      -- массив названий
        'Отдел сопровождения диссертационных советов',
        'Совет молодых ученых',
        'Аспирантура'
    ]),
    unnest(ARRAY[                      -- массив типов для каждого названия
        'subdivision',                 -- тип для отдела сопровождения диссертационных советов
        'council',                     -- тип для Совета молодых ученых
        'postgraduate'                 -- тип для Аспирантуры
    ]),
    id                                 -- parent_id = id Научного управления
FROM sci;
```

Лекция №1

Тема: «Модели организации данных и PostgreSQL»

Подтема: «Иерархическая модель данных»

Как выглядит в PostgreSQL

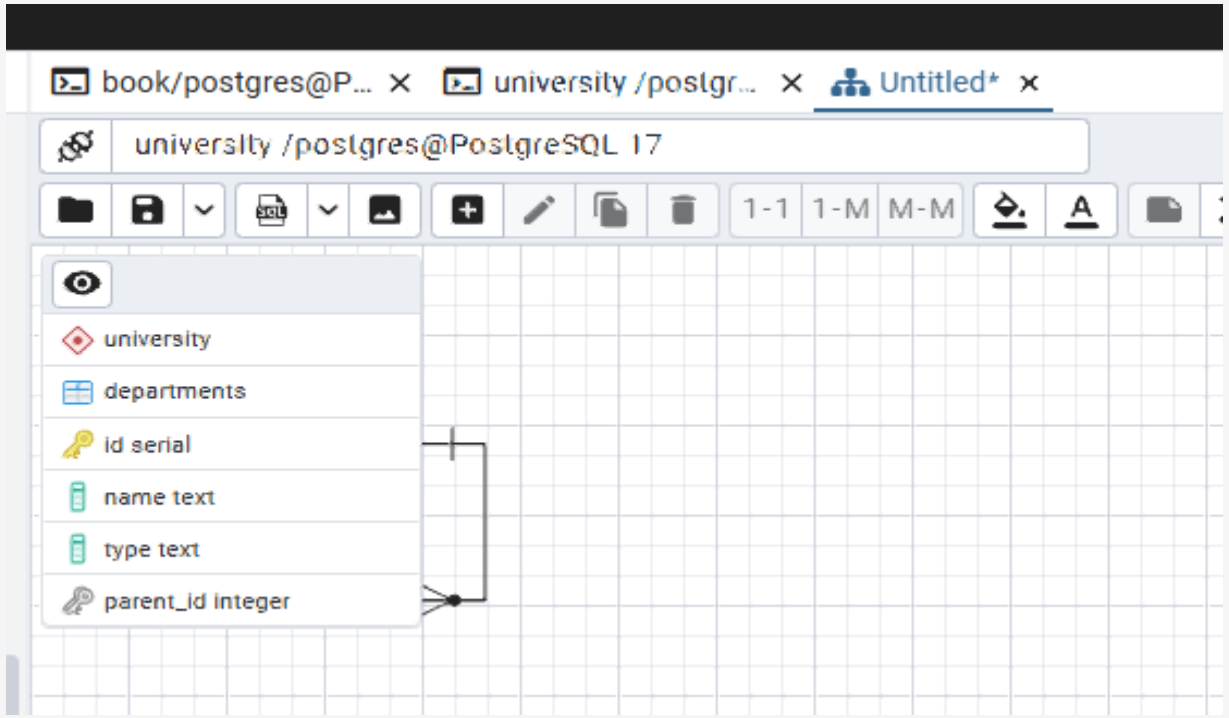
```
-- ------------------------------------------------------------
-- Добавляем Финансовое управление и его подразделения
-- ------------------------------------------------------------
WITH rectorate AS (                    -- создаём временный набор rectorate
   SELECT id
   FROM departments
   WHERE name = 'Ректорат'             -- выбираем id подразделения "Ректорат"
),
fin AS (                               -- создаём временный набор fin
   INSERT INTO departments (name, type, parent_id)
   SELECT 'Управление финансами',      -- название подразделения
          'finance',                   -- тип = управление финансами
          id                           -- parent_id = id "Ректората"
   FROM rectorate
   RETURNING id                        -- сохраняем id созданного Управления финансами
)
INSERT INTO departments (name, type, parent_id) -- вставляем подразделения финансов
SELECT
   unnest(ARRAY[                       -- массив названий подразделений
      'Бухгалтерия',                   -- бухгалтерия
      'Отдел по заключению договоров', -- договорной отдел
      'Юридический отдел'              -- юр. отдел
   ]),
   unnest(ARRAY[                       -- массив типов для каждого названия
      'finance_subdivision',           -- тип для бухгалтерии
      'legal',                         -- тип для договорного отдела
      'legal'                          -- тип для юридического отдела
   ]),
   id                                  -- parent_id = id Управления финансами
FROM fin;
-- ------------------------------------------------------------
-- Добавляем IT-управление и его подразделения
-- ------------------------------------------------------------
WITH rectorate AS (                    -- создаём временный набор rectorate
   SELECT id
   FROM departments
   WHERE name = 'Ректорат'             -- выбираем id подразделения "Ректорат"
),
it AS (                                -- создаём временный набор it
   INSERT INTO departments (name, type, parent_id)
   SELECT 'Управление IT',             -- название подразделения
          'it_management',             -- тип = управление IT
          id                           -- parent_id = id "Ректората"
   FROM rectorate
   RETURNING id                        -- сохраняем id Управления IT
)
INSERT INTO departments (name, type, parent_id) -- вставляем подразделения IT
SELECT
   unnest(ARRAY[                       -- массив названий подразделений
      'Отдел разработки',              -- отдел разработки
      'Отдел информационной безопасности',  -- отдел ИБ
      'Центр обработки данных'         -- ЦОД
   ]),
   unnest(ARRAY[                       -- массив типов для каждого
      'it_subdivision',                -- тип для отдела разработки
      'it_security',                   -- тип для отдела ИБ
      'data_center'                    -- тип для ЦОД
   ]),
   id                                  -- parent_id = id Управления IT
FROM it;
```

Лекция №1

Тема: «Модели организации данных и PostgreSQL»

Подтема: «Иерархическая модель данных»

Как выглядит в PostgreSQL




```
university =# SELECT id, name, type, parent_id FROM departments ORDER BY id;
 id |                    name                    |        type          | parent_id
----+--------------------------------------------+----------------------+-----------
  1 | Университет                                | root                 |
  2 | Ректорат                                   | management           |         1
  3 | Факультеты                                 | faculty_group        |         2
  4 | Факультет математики и кибернетики         | faculty              |         3
  5 | Факультет информационных технологий        | faculty              |         3
  6 | Факультет экономики и управления           | faculty              |         3
  7 | Отдел кадров                               | hr                   |         2
  8 | Отдел по работе с ППС                      | subdivision          |         7
  9 | Отдел по работе с научными сотрудниками    | subdivision          |         7
 10 | Научное управление                         | science_management   |         2
 11 | Отдел сопровождения диссертационных советов | subdivision          |        10
 12 | Совет молодых ученых                       | council              |        10
 13 | Аспирантура                                | postgraduate         |        10
 14 | Управление финансами                       | finance              |         2
 15 | Бухгалтерия                                | finance_subdivision  |        14
 16 | Отдел по заключению договоров              | legal                |        14
 17 | Юридический отдел                          | legal                |        14
 18 | Управление IT                              | it_management        |         2
 19 | Отдел разработки                           | it_subdivision       |        18
 20 | Отдел информационной безопасности          | it_security          |        18
 21 | Центр обработки данных                     | data_center          |        18
(21 rows)
```

Лекция №1

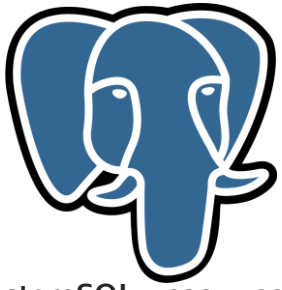
Тема: «Модели организации данных и PostgreSQL»

Подтема: «Сетевая модель данных»

Сетевая модель данных

Основная суть модели заключается в том, что она основана на графе (ориентированном или неориентированном). Записи называются узлами (nodes/records). Связи между записями — рёбра (links/edges). Каждая запись может иметь несколько родителей и несколько потомков (в отличие от иерархической).
Иерархическая: одно дерево, 1 родитель → много наследников (потомков).
Сетевая: граф, возможны произвольные связи.

1.Neo4J – графовая система управления базами данных с открытым исходным кодом, реализованная на Java.



2. PostgreSQL + расширения:
-pgRouting (графы + маршруты);
-Apache AGE (поддержкаязыка запрсоов openCypher прямо в PostgreSQL);
-«ручное моделирование графа» через таблицы «nodes» и «edges».

3. ArangoDB (Multi-Model: документная + графовая)

Сетевая модель данных сегодня широко применяется в поисковых системах (граф ссылок интернета, на основе которого работает PageRank, и графы знаний, связывающие сущности вроде людей, компаний и мест), в маркетплейсах и рекомендательных системах (графы «пользователь–товар–категория», по которым строятся рекомендации), а также в социальных сетях (графы друзей и подписок), транспорте и логистике (сети дорог и маршрутов), телекоммуникациях (сети соединений) и финансовой сфере (графы транзакций для анализа мошенничества).

Лекция №1

Тема: «Модели организации данных и PostgreSQL»

Подтема: «Сетевая модель данных»

Как выглядит в Neo4J (запрос на языке Cypher)

```
CREATE
(u:Department {name:'Университет', type:'root'}),
(r:Department {name:'Ректорат', type:'management'}),
(u)-[:HAS_CHILD]->(r),
(fg:Department {name:'Факультеты', type:'faculty_group'}),
(r)-[:HAS_CHILD]->(fg),
(f1:Department {name:'Ф-т математики и кибернетики', type:'faculty'}),
(f2:Department {name:'Ф-т информ. технологий', type:'faculty'}),
(f3:Department {name:'Ф-т экономики и управления', type:'faculty'}),
(fg)-[:HAS_CHILD]->(f1),
(fg)-[:HAS_CHILD]->(f2),
(fg)-[:HAS_CHILD]->(f3),
(hr:Department {name:'Отдел кадров', type:'hr'}),
(r)-[:HAS_CHILD]->(hr),
(hr1:Department {name:'Работа с ППС', type:'subdivision'}),
(hr2:Department {name:'Работа с науч. сотрудниками', type:'subdivision'}),
(hr)-[:HAS_CHILD]->(hr1),
(hr)-[:HAS_CHILD]->(hr2),
(sci:Department {name:'Научное управление', type:'science_management'}),
(r)-[:HAS_CHILD]->(sci),
(s1:Department {name:'Дисс. советы', type:'subdivision'}),
(s2:Department {name:'Совет молодых ученых', type:'council'}),
(s3:Department {name:'Аспирантура', type:'postgraduate'}),
(sci)-[:HAS_CHILD]->(s1),
(sci)-[:HAS_CHILD]->(s2),
(sci)-[:HAS_CHILD]->(s3),
(fin:Department {name:'Управление финансами', type:'finance'}),
(r)-[:HAS_CHILD]->(fin),
(fin1:Department {name:'Бухгалтерия', type:'finance_subdivision'}),
(fin2:Department {name:'Договоры', type:'legal'}),
(fin3:Department {name:'Юр. отдел', type:'legal'}),
(fin)-[:HAS_CHILD]->(fin1),
(fin)-[:HAS_CHILD]->(fin2),
(fin)-[:HAS_CHILD]->(fin3),
(it:Department {name:'Управление IT', type:'it_management'}),
(r)-[:HAS_CHILD]->(it),
(it1:Department {name:'Разработка', type:'it_subdivision'}),
(it2:Department {name:'Инф. безопасность', type:'it_security'}),
(it3:Department {name:'ЦОД', type:'data_center'}),
(it)-[:HAS_CHILD]->(it1),
(it)-[:HAS_CHILD]->(it2),
(it)-[:HAS_CHILD]->(it3);
```

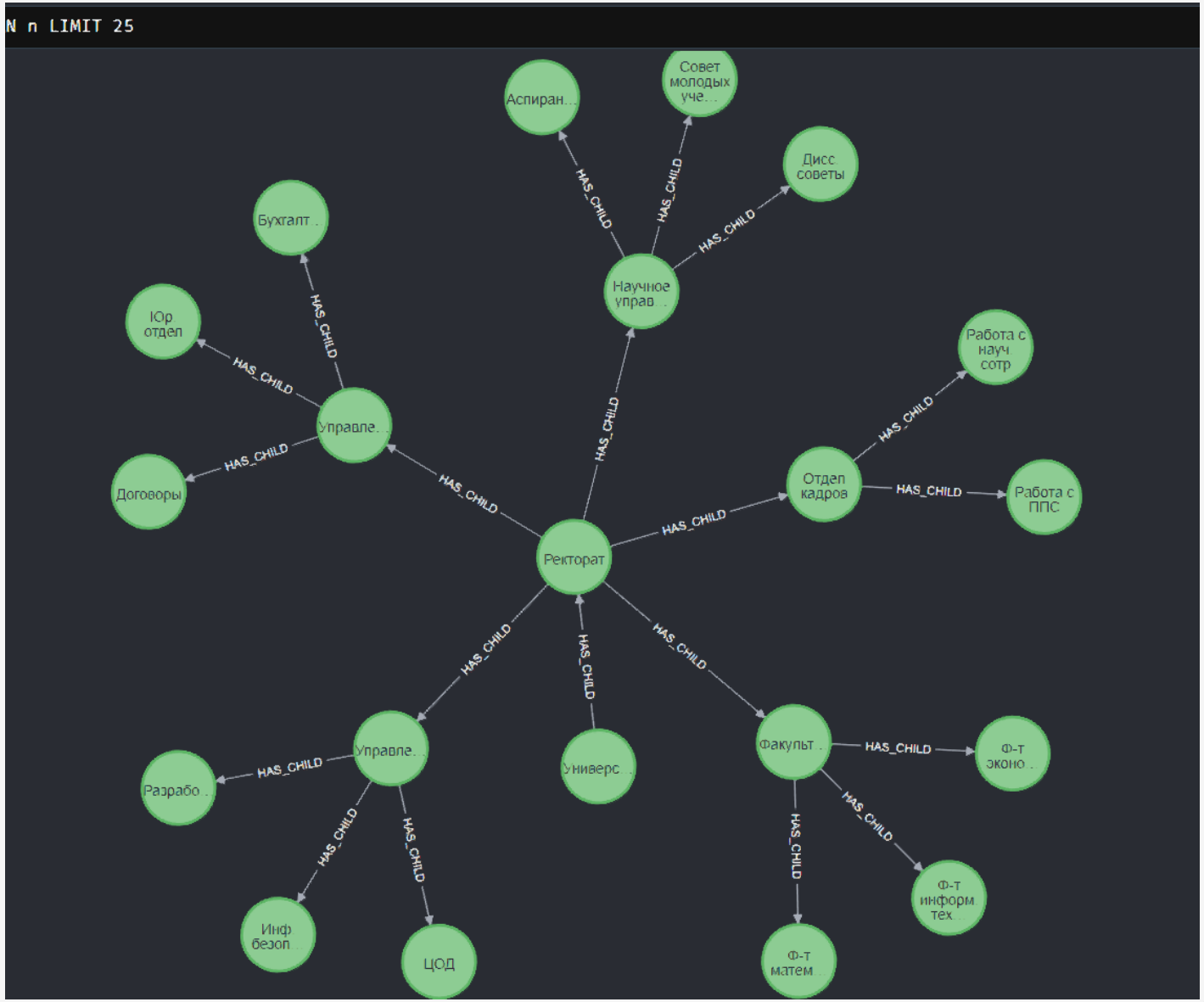
Та же самая структура университета, взятая из иерархической модели данных, но уже представленная в виде ориентированного графа на языке запросов Cypher

Лекция №1

Тема: «Модели организации данных и PostgreSQL»

Подтема: «Сетевая модель данных»

Как выглядит в Neo4J



Та же самая структура университета, взятая из иерархической модели данных, но уже представленная в виде ориентированного графа.

Лекция №1

Тема: «Модели организации данных и PostgreSQL»

Подтема: «Сетевая модель данных»

Как выглядит в PostgreSQL

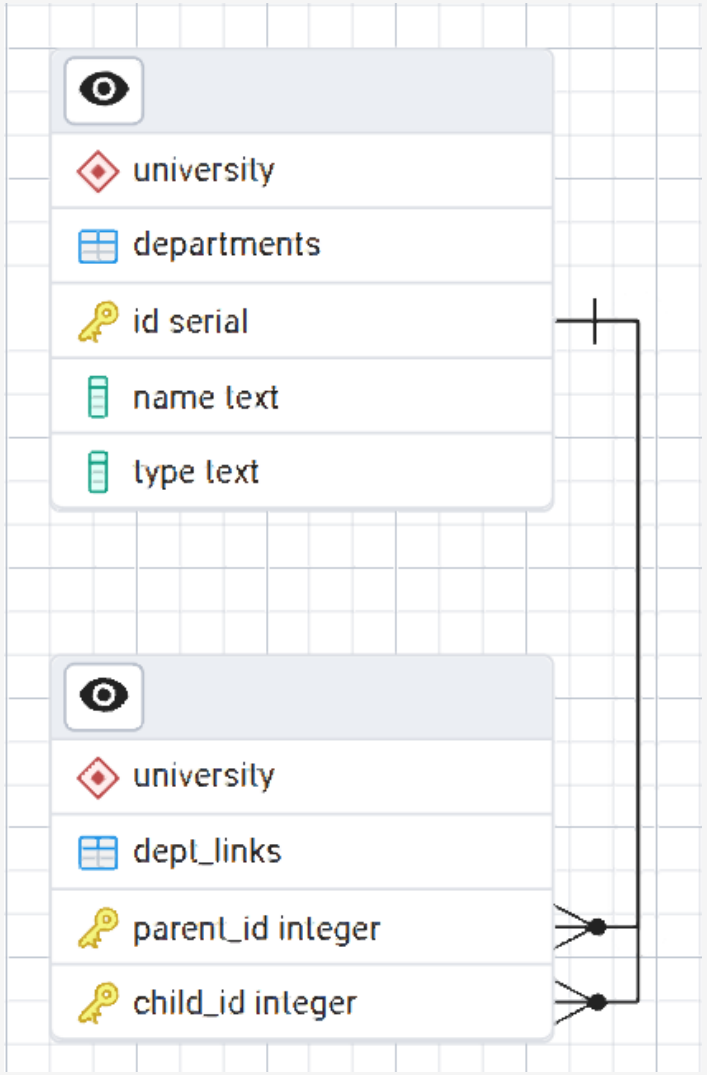
```
DROP SCHEMA IF EXISTS university CASCADE;
CREATE SCHEMA university;
SET search_path TO university;

-- Узлы
CREATE TABLE departments (
   id SERIAL PRIMARY KEY,
   name TEXT NOT NULL,
   type TEXT
);

-- Связи (сетевая модель)
CREATE TABLE dept_links (
   parent_id INT REFERENCES departments(id) ON DELETE CASCADE,
   child_id INT REFERENCES departments(id) ON DELETE CASCADE,
   PRIMARY KEY (parent_id, child_id)
);

-- Вставка узлов
INSERT INTO departments (id, name, type) VALUES
(1,'Университет','root'),
(2,'Ректорат','management'),
(3,'Факультеты','faculty_group'),
(4,'Ф-т математики и кибернетики','faculty'),
(5,'Ф-т информ. технологий','faculty'),
(6,'Ф-т экономики и управления','faculty'),
(7,'Отдел кадров','hr'),
(8,'Работа с ППС','subdivision'),
(9,'Работа с науч. сотрудниками','subdivision'),
(10,'Научное управление','science_management'),
(11,'Дисс. советы','subdivision'),
(12,'Совет молодых ученых','council'),
(13,'Аспирантура','postgraduate'),
(14,'Управление финансами','finance'),
(15,'Бухгалтерия','finance_subdivision'),
(16,'Договоры','legal'),
(17,'Юр. отдел','legal'),
(18,'Управление IT','it_management'),
(19,'Разработка','it_subdivision'),
(20,'Инф. безопасность','it_security'),
(21,'ЦОД','data_center');

-- Вставка связей (сетевые дуги)
INSERT INTO dept_links VALUES
(1,2),
(2,3),(3,4),(3,5),(3,6),
(2,7),(7,8),(7,9),
(2,10),(10,11),(10,12),(10,13),
(2,14),(14,15),(14,16),(14,17),
(2,18),(18,19),(18,20),(18,21);
```

Лекция №1

Тема: «Модели организации данных и PostgreSQL»

Подтема: «Сетевая модель данных»

Как выглядит в PostgreSQL



Вся структура находится внутри отдельной схемы **university** в PostgreSQL.
Это удобно для логической группировки таблиц, относящихся к одному проекту (университетская БД).

Таблица «**departments**» хранит **подразделения университета** (узлы графа). Поля:

1) id – первичный ключ, уникальный идентификатор подразделения (автоинкремент);
2) name – название подразделения (например, *«Ректорат»*, *«Факультет ИТ»*, *«Отдел кадров»*);
3) type – тип подразделения, следующее от, например, ректората.

Таблица «**dept_links**» хранит **связи между подразделениями** (рёбра графа). Благодаря ей модель становится **сетевой**: один узел может быть связан с несколькими родителями или потомками. Поля:

1) parent_id – внешний ключ, указывающий на departments.id (родительский узел);
2) child_id – внешний ключ, указывающий на departments.id (дочерний узел).

Каждая строка в этой таблице описывает одно отношение «родитель → потомок».

Примеры связей:

1. (1, 2) = Университет → Ректорат.
2. (2, 3) = Ректорат → Факультеты.
3. (3, 4) = Факультеты → Факультет информационных технологий

Departments.id связывается с dept_links.parent_id (родительская сторона)
Departments.id связывается с child_links.parent_id (дочерняя сторона)
Т.е. Departments **вершины графа** (узлы, подразделения), а dept_links = **рёбра графа** (сети связей между подразделениями).

Лекция №1

Тема: «Модели организации данных и PostgreSQL»

Подтема: «Реляционная модель данных»

Реляционная модель данных

Основная суть модели заключается в том, что она основана на отношениях (таблицах).

Основная терминология:

1) **Отношение** → логическая таблица, т.е. таблица в БД;
2) **Кортеж** → строка таблицы;
3) **Атрибут** → столбец таблицы;
4) **Домен** → множество допустимых значений атрибута (тип + ограничения).

Ключевые свойства отношения:

1. Строки и столбцы **логически неупорядочены** (важны значения и имена).
2. **Атомарность**: в ячейке одно значение (1NF) по умолчанию. Есть возможность сохранять несколько значений, но это при помощи массивов, но это нарушает свойство атомарности.
3. **Отсутствие дубликатов строк** (по схеме отношения).
4. БД (схема) строго определяет **типы данных** и **ограничения ячейки**.

Лекция №1

Тема: «Модели организации данных и PostgreSQL»

Подтема: «Реляционная модель данных»

Реляционная модель данных. Основные операторы в PostgreSQL –Часть 1

Конструкция	Что делает
CREATE DATABASE	Создаёт новую базу данных.
CREATE TABLE	Создаёт таблицу: столбцы и их типы.
ALTER TABLE	Меняет структуру таблицы: добавить/удалить столбец, индекс, ограничение, тип и т.п.
CREATE VIEW	Создаёт представление — «сохранённый запрос» как виртуальную таблицу.
CREATE MATERIALIZED VIEW	Создаёт материализованное представление — результат запроса хранится физически; обновляется командой REFRESH.
CREATE TYPE	Определяет пользовательский тип (например, перечисление ENUM или составной тип).
PRIMARY KEY	Главный ключ строки: уникален и не NULL.
FOREIGN KEY … REFERENCES	Внешний ключ — обеспечивает связь с родительской таблицей.
NOT NULL	Запрещает пропуск значения (обязательный столбец).
CHECK (условие)	Собственное правило допустимых значений (напр. age >= 0).
UNIQUE	Запрещает дубли в столбце/наборе столбцов.
ON DELETE/UPDATE CASCADE	**Каскадно** удаляет/обновляет связанные дочерние строки при удалении/изменении родительской.
ON DELETE/UPDATE RESTRICT	**Запрещает** удаление/изменение родительской строки, если есть дочерние.
ON DELETE/UPDATE SET NULL	При удалении/изменении родителя в дочерних строках поле-ссылка **ставится в NULL**.

Лекция №1

Тема: «Модели организации данных и PostgreSQL»

Подтема: «Реляционная модель данных»

Реляционная модель данных. Основные операторы в PostgreSQL –Часть 2

Конструкция	Что делает
INSERT INTO …	Вставляет новые строки (вручную или результатом SELECT).
SELECT … FROM …	Выбирает данные из таблиц/представлений.
WHERE	Фильтрует строки по условию.
ORDER BY	Сортирует результат (возрастающе/убывающе).
LIMIT / OFFSET	Возвращает порцию строк: сколько и с какого места (пагинация).
JOIN	Соединяет таблицы: INNER — **внутреннее**; LEFT — **левое внешнее**; RIGHT — **правое внешнее**; FULL — **полное внешнее**; CROSS — **декартово произведение**.
ON / USING	Условие/столбцы, по которым соединяем таблицы.
LIKE	Поиск по шаблону: % — любые символы, _ — один символ.
~, ~*, !~, !~*	Поиск по **регулярным выражениям** (чувствительно/нечувствительно к регистру; «не совпадает»).
SIMILAR TO	Шаблонный поиск в SQL-стиле (аналог регулярок/LIKE).
GROUP BY	Группирует строки для расчётов (итоги по признаку).
HAVING	Фильтрует **группы** после GROUP BY.
WITH (CTE)	Общее табличное выражение — временно именованный подзапрос для читабельности сложных запросов.
UNION / UNION ALL	Объединяет результаты SELECT: UNION — **убирает дубли**, UNION ALL — **оставляет**.
ANY / ALL	Сравнение с подзапросом/массивом: «хотя бы с одним» / «со всеми».
Подзапрос (SELECT внутри другого запроса)	Вложенный SELECT внутри другого запроса (как временная таблица/значение).
Оконные функции … OVER (…)	Подсчёты «по окну» без склейки строк (скользящие суммы, сравнение соседей и т.п.).
RANK / DENSE_RANK	Ранги в окне: с пропусками / без пропусков.
LAG / LEAD	Значение из предыдущей / следующей строки в окне.
ROLLUP / GROUPING SETS / CUBE	Многоуровневые итоги (OLAP): промежуточные и общие суммы по нескольким измерениям.
COALESCE(a, b, …)	Возвращает первый **не NULL** аргумент (удобно для значений по умолчанию).
NULLIF(a, b)	Возвращает NULL, если a = b (полезно, напр., чтобы не делить на ноль).

Лекция №1

Тема: «Модели организации данных и PostgreSQL»

Подтема: «Реляционная модель данных»

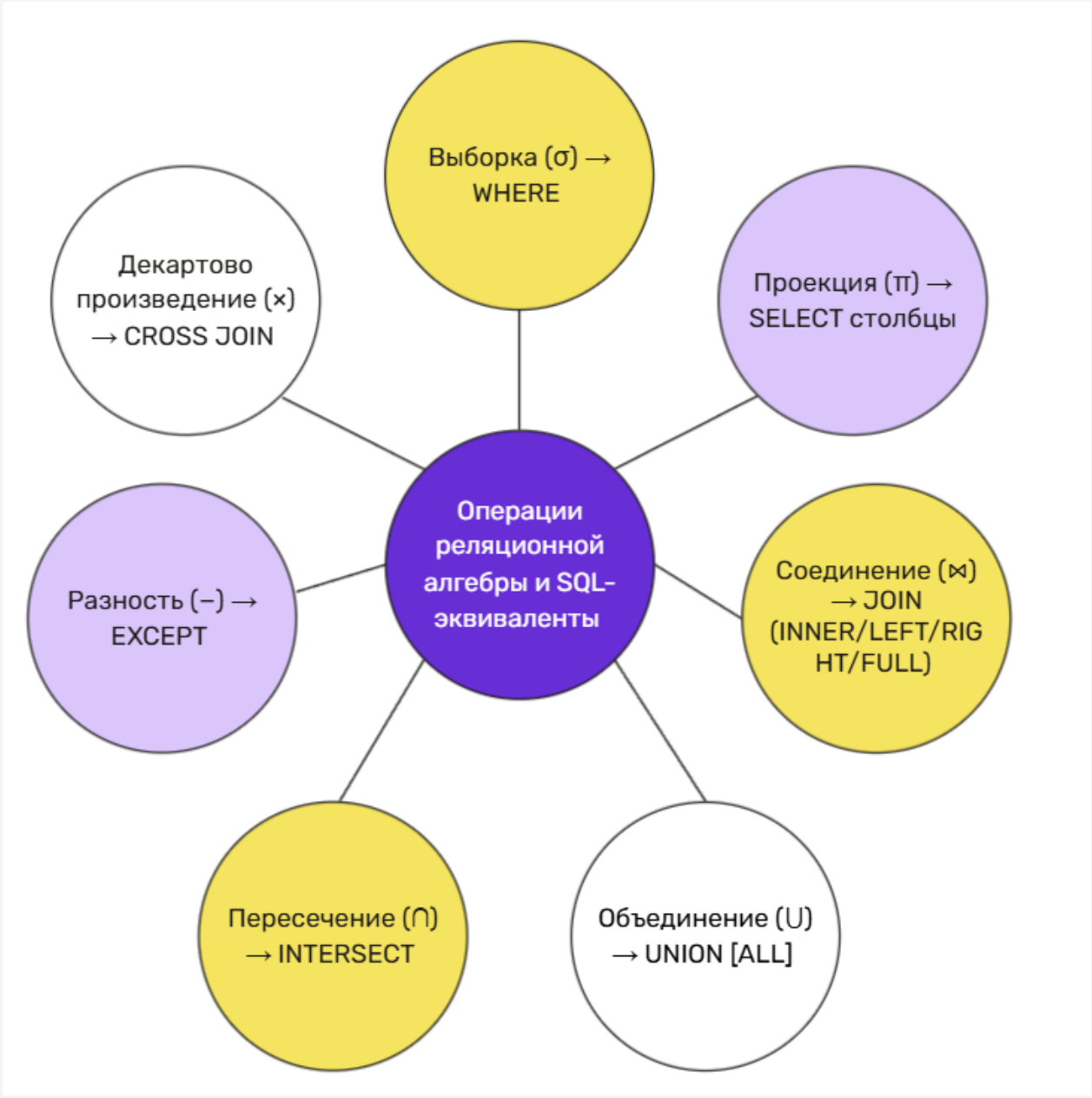
Реляционная модель данных – связи между таблицами

Лекция №1

Тема: «Модели организации данных и PostgreSQL»

Подтема: «Реляционная модель данных»

Реляционная модель данных – операции реляционной алгебры и SQL-эквиваленты



Лекция №1

Тема: «Модели организации данных и PostgreSQL»

Подтема: «Реляционная модель данных»

Реляционная модель данных – свойства ACID

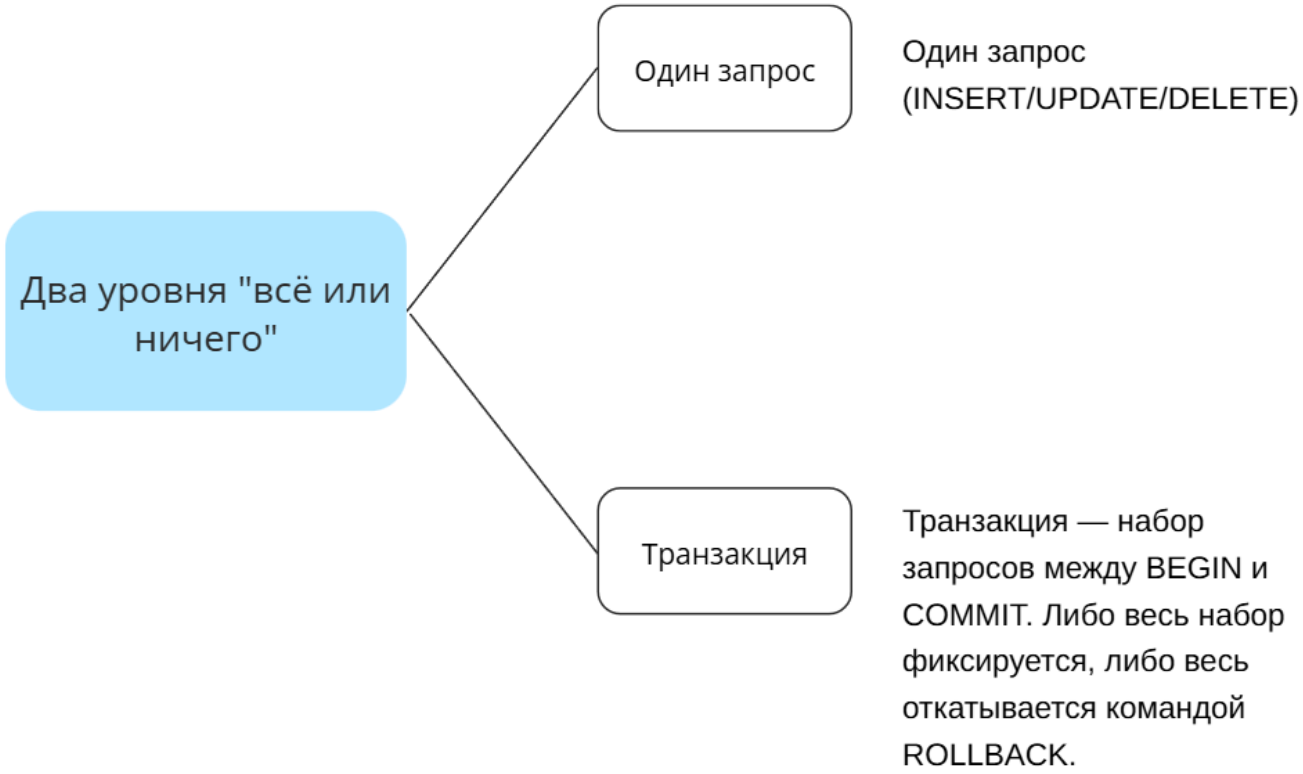
Лекция №1

Тема: «Модели организации данных и PostgreSQL»

Подтема: «Реляционная модель данных»

Реляционная модель данных – свойство (требование) Атомарности

Атомарность = **делаем всё сразу или не делаем ничего**. Пример с заказом: положили в коробку ноутбук, зарядку и чек **и** отправили всё сразу, **или же** при возникновении проблемы коробку (заказ) вообще не отправили.



Лекция №1

Тема: «Модели организации данных и PostgreSQL»

Подтема: «Реляционная модель данных»

Реляционная модель данных – свойство (требование) Атомарности. Примеры запросов

```
BEGIN;  -- старт операции (транзакции)

INSERT INTO students(full_name) VALUES ('Иван Иванов');  -- шаг 1
-- допустим, id нового студента = 101

INSERT INTO enrollments(student_id, course_id, year, term)  -- шаг 2
VALUES (101, 1, 2025, 'autumn');

COMMIT;  -- применили и отправили запрос, закрытие транзакции
```

```
BEGIN;

SAVEPOINT s1;                -- поставили «метку»
INSERT INTO enrollments VALUES (101, 2, 2025, 'autumn'); -- ой, лишнее
ROLLBACK TO SAVEPOINT s1;        -- отменили только этот кусок, условно говоря – это как ctrl+z

-- продолжаем другие действия...
COMMIT;
```

Лекция №1

Тема: «Модели организации данных и PostgreSQL»

Подтема: «Реляционная модель данных»

Реляционная модель данных – свойство (требование) Согласованности

Идея: база не позволит записать «чушь». Нарушил правило — вся операция отменяется.

Например, у нас есть 5 правил (ограничений):

И мы попытаемся сослаться на несуществующего студента с ID 999:

```
-- Родительская таблица (students)
-- Здесь хранятся студенты. Каждый студент имеет уникальный идентификатор student_id.
CREATE TABLE students (
  student_id INT PRIMARY KEY   -- PRIMARY KEY: уникальное значение, обязательное для каждой строки
);

-- Дочерняя таблица (enrollments)
-- Здесь хранятся записи о зачислении студентов на курсы.
-- В колонке student_id указываем, какой студент (из таблицы students) зачислен.
CREATE TABLE enrollments (
  student_id INT REFERENCES students(student_id),  -- FOREIGN KEY: ссылка на students(student_id)
  course_id  INT,  -- курс, на который зачислен студент
  year       INT,  -- год обучения
  term       TEXT  -- семестр (например, 'autumn', 'spring')
);

-- Попробуем вставить запись в enrollments:
-- говорим, что студент с id=999 зачислен на курс 1 в 2025 году.
-- Но в таблице students такого студента НЕТ!
INSERT INTO enrollments(student_id, course_id, year, term)
VALUES (999, 1, 2025, 'autumn');

-- PostgreSQL проверяет ограничение FOREIGN KEY:
-- enrollments.student_id должен существовать в students.student_id.
-- Так как student_id=999 отсутствует в родительской таблице,
-- система выдаёт ошибку и вставка не происходит.

-- ОШИБКА:
-- ERROR:  insert or update on table "enrollments" violates foreign key constraint ...
-- DETAIL: Key (student_id)=(999) is not present in table "students".
```

Лекция №1

Тема: «Модели организации данных и PostgreSQL»

Подтема: «Реляционная модель данных»

Реляционная модель данных – свойство (требование) изолированности

Идея: параллельные пользователи не мешают друг другу. Каждый видит **свою** устойчивую картинку данных на время запроса/транзакции.
Например, вы сделали «фото-снимок» таблиц — смотрите по нему, пока выполняете запросы/транзакцию. Уровни изолированности:

Лекция №1

Тема: «Модели организации данных и PostgreSQL»

Подтема: «Реляционная модель данных»

Реляционная модель данных – свойство (требование) изолированности. Пример запроса READ COMMITTED

Идея: каждый запрос видит самые свежие **зафиксированные** данные на момент выполнения.

Поведение: два одинаковых SELECT подряд в транзакции могут дать **разные числа**, если параллельно кто-то успел закоммитить изменения.

Когда использовать: обычные web-CRUD, короткие операции.

```
-- По умолчанию (уровень изоляции READ COMMITTED в PostgreSQL)
-- Каждая команда внутри транзакции видит только зафиксированные (committed) изменения
-- на момент начала самой команды, а не на момент начала всей транзакции.

BEGIN;
-- Начинаем транзакцию

-- Считаем, сколько записей есть в таблице enrollments для course_id = 1
SELECT COUNT(*) FROM enrollments WHERE course_id=1;
-- Результат: 10 (в таблице сейчас 10 строк)

-- !!! В это время параллельная сессия (другая программа/пользователь)
-- вставляет новую строку в enrollments (course_id=1) и делает COMMIT.

-- Повторяем тот же запрос в нашей транзакции
SELECT COUNT(*) FROM enrollments WHERE course_id=1;
-- Результат: 11 (мы уже видим изменения, которые закоммитила другая сессия,
-- потому что в READ COMMITTED каждый SELECT заново «смотрит» в таблицу)

COMMIT;
-- Завершаем транзакцию
```

Лекция №1

Тема: «Модели организации данных и PostgreSQL»

Подтема: «Реляционная модель данных»

Реляционная модель данных – свойство (требование) изолированности. Пример запроса REPEATABLE READ

Идея: один «снимок» на всю транзакцию — данные для вас **заморожены**.
Поведение: повторные SELECT показывают **тот же набор** строк, что и в начале транзакции.
Риск: логическое правило на множество строк может быть нарушено.
Когда использовать: отчёт «на момент времени», длинные чтения.

```
-- В этой транзакции мы явно задаём уровень изоляции REPEATABLE READ.
-- Это значит: все SELECT внутри транзакции будут работать с "снимком" (snapshot)
-- базы данных, сделанным в момент начала транзакции.
-- Даже если кто-то параллельно добавит новые данные и закоммитит,
-- мы их не увидим до завершения нашей транзакции.

BEGIN ISOLATION LEVEL REPEATABLE READ;
-- Начинаем транзакцию с уровнем REPEATABLE READ

-- Считаем количество записей для course_id=1
SELECT COUNT(*) FROM enrollments WHERE course_id=1;
-- Результат: 10 (в момент старта транзакции в таблице было 10 строк)

-- !!! Параллельная сессия (другой пользователь) вставляет новую строку
-- в enrollments (course_id=1) и делает COMMIT.
-- Но наша транзакция "заморожена" на снимке начала.

-- Повторяем тот же запрос в нашей транзакции
SELECT COUNT(*) FROM enrollments WHERE course_id=1;
-- Результат: всё ещё 10 (мы НЕ видим изменения другой сессии,
-- потому что наш snapshot остался таким же, как в момент BEGIN)

COMMIT;
-- Завершаем транзакцию
```

Лекция №1

Тема: «Модели организации данных и PostgreSQL»

Подтема: «Реляционная модель данных»

Реляционная модель данных – свойство (требование) изолированности. Пример запроса SERIALIZABLE

Идея: СУБД гарантирует результат как при **последовательном** выполнении транзакций. Т.е. все запросы идут по очереди. Если данное условие нарушится, одну транзакцию отменят и её надо повторить.

Поведение: если ваша и чужая транзакции вместе дали бы «нелогичный» эффект, одна получит ошибку «could not serialize» → **повторите** транзакцию.

Когда использовать: при наличии строгих лимитов или ограничений.

```
-- ============================================================
-- Таблица счетов
-- ============================================================
DROP TABLE IF EXISTS accounts;
CREATE TABLE accounts (
    id      INT PRIMARY KEY,      -- уникальный номер счёта
    balance NUMERIC(12,2) NOT NULL -- баланс счёта (например, в рублях)
);

-- Начальные данные: два счёта по 1000
INSERT INTO accounts (id, balance) VALUES
(1, 1000.00),
(2, 1000.00);

-- ============================================================
-- Шаблон retry-loop для уровня SERIALIZABLE (psql-скрипт)
-- ============================================================

\set try 0          -- переменная-счётчик попыток
\set max_try 5      -- максимальное количество повторов (чтобы не зациклиться)

\echo 'Начало цикла'

\while :try < :max_try            -- цикл: пока не исчерпаны попытки
\set try :try + 1                 -- увеличиваем счётчик
  \echo 'Попытка:' :try           -- выводим номер попытки

  BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
    -- Начинаем транзакцию с уровнем SERIALIZABLE
    -- ---------------------------------------------------------
    -- Здесь бизнес-логика (например, перевод 100 со счёта 1 на счёт 2)
    UPDATE accounts SET balance = balance - 100 WHERE id = 1;
    UPDATE accounts SET balance = balance + 100 WHERE id = 2;
    -- ---------------------------------------------------------
  COMMIT;                         -- пытаемся зафиксировать транзакцию

  \if :ERROR                      -- проверяем: была ли ошибка
    \echo 'Ошибка сериализации, повторяем...'
    ROLLBACK;                     -- откат транзакции и повтор
  \else
    \echo 'Успех: транзакция закоммичена'
    \quit                         -- завершаем цикл досрочно
  \endif
\endwhile
```

Лекция №1

Тема: «Модели организации данных и PostgreSQL»

Подтема: «Реляционная модель данных»

Реляционная модель данных – свойство (требование) надежности

Идея: если БД дана команда «Commit», то данные не потеряются даже при сбое.

Как это достигается?

1. Перед тем как «окончательно положить» данные, Postgres пишет «чек» в журнал (WAL).
2. Если свет выключился: при старте сервер читает журнал и **восстанавливает** всё, что было зафиксировано.

Что делать/не делать:

1. Не использовать UNLOGGED-таблицы для важных данных (они быстрее, но после сбоя не сохраняются).
2. Если включили «ускорители» вроде «synchronous_commit=off» нужно помнить, что последние записи можно потерять.

Пример запроса:

```
-- Обычная таблица (с журналированием Write-Ahead Log)
-- WAL — это последовательный журнал (файлы на диске), который хранит информацию об изменениях данных.
-- Пример: /var/lib/postgresql/16/main/pg_wal/ (путь зависит от установки).
-- Файлы WAL находятся в каталоге pg_wal (раньше назывался pg_xlog) внутри data-директории PostgreSQL.

CREATE TABLE notes_safe(
   id  serial PRIMARY KEY,  -- уникальный идентификатор (автоинкремент)
   txt text                 -- текст заметки
);

-- Вставляем строку
INSERT INTO notes_safe(txt) VALUES ('сохранится навсегда');

-- После COMMIT запись попадёт в WAL (журнал предзаписи).
-- Это значит:
-- 1. Если сервер внезапно "упадёт" (сбой питания, авария),
--    то при перезапуске PostgreSQL использует WAL, чтобы восстановить данные.
-- 2. Поэтому строки в обычных таблицах "переживут" аварию.
-- 3. Минус: запись в WAL немного замедляет вставки (есть дополнительные расходы по вычислительным ресурсам).
```

```
-- Быстрая, но «хрупкая» таблица (UNLOGGED)

CREATE UNLOGGED TABLE notes_fast(
   id  serial PRIMARY KEY,  -- автоинкрементный id
   txt text                 -- текст заметки
);

-- Вставляем строку
INSERT INTO notes_fast(txt) VALUES ('может исчезнуть после аварии');

-- Особенности UNLOGGED:
-- 1. Данные НЕ пишутся в WAL (журнал предзаписи).
-- 2. За счёт этого INSERT/UPDATE/DELETE быстрее (меньше дисковых операций).
-- 3. Но при сбое питания или аварийном завершении работы сервера
--    данные из UNLOGGED-таблиц будут утеряны.
-- 4. При рестарте PostgreSQL такие таблицы автоматически очищаются (TRUNCATE).
-- 5. Подходит только для временных/кэшевых данных, которые не жалко потерять.
--    Например: сессии, буферы, промежуточные результаты вычислений.
```

Лекция №1

Тема: «Модели организации данных и PostgreSQL»

Подтема: «Реляционная модель данных»

Реляционная модель данных – разница между БД, схемой и таблицей

Объект	Обозначение	Определение	Где находится в иерархии	Пример имени	Когда использовать	Ключевое отличие
База данных (Database)	**DB**, 𝔻	Отдельный контейнер данных проекта; единица подключения и бэкапа	На самом верху; содержит схемы	university	Нужна полная изоляция (разные проекты, prod/test, отдельные настройки/расширения)	В PostgreSQL обычные JOIN использовать **между БД нельзя** (нужны сторонние механизмы)
Схема (Schema)	**S**	«Папка» внутри БД: пространство имён для объектов (таблиц, типов, функций)	Внутри БД; содержит таблицы и др. объекты	hr, finance	Разделить модули/команды/тенантов, разграничить права, упорядочить имена	JOIN **между схемами** одной БД — можно; работает search_path
Таблица (Table)	**T** или **R**	Структура из строк и столбцов, где лежат данные	Внутри схемы	hr.employees	Хранить конкретную сущность (студенты, выплаты и т.д.)	Здесь данные и ограничения (PK, FK, UNIQUE, CHECK)

Лекция №2

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Типы данных. Язык SQL. Операторы CREATE и INSERT. Ограничения CHECK, NOT NULL, UNIQUE. Первичный и внешний ключи. Обеспечение ссылочной целостности»

Основы синтаксиса SQL в PostgreSQL – основные типы данных. Числа

Числовой тип в (PostgreSQL)	Диапазон / точность	Когда брать (применение)	Примеры полей	Замечания
SMALLINT	−32 768…32 767	Совсем маленькие счётчики, коды, флаги	room_floor, semester_no	Быстро «закончится». По умолчанию лучше **INT**
INT / INTEGER	−2 147 483 648…2 147 483 647	Тип «по умолчанию»: ID, количества, годы	student_id, year_admit, credits	Дешёвый по памяти/индексам, хватает для 90% задач
BIGINT	−9.22e18…9.22e18	Очень большие ID/счётчики, логи, аналитика	order_id, views, event_id	8 байт (больше индекс), берите, если реально ожидаются миллиарды+
NUMERIC(p,s)	Точно фиксированные десятичные (например, NUMERIC(10,2))	**Деньги, оценки, проценты**, где важна точность	price NUMERIC(10,2), grade NUMERIC(3,1)	Точен, но тяжелее/медленнее, задавайте CHECK (диапазон)
REAL / DOUBLE PRECISION	Приблизительные числа (примерно 6/15 знаков)	Датчики, телеметрия, статистика, ML	temperature DOUBLE PRECISION, probability REAL	Не для денег/оценок, больше для научно-прикладных исследований

Лекция №2

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Типы данных. Язык SQL. Операторы CREATE и INSERT. Ограничения CHECK, NOT NULL, UNIQUE. Первичный и внешний ключи. Обеспечение ссылочной целостности»

Основы синтаксиса SQL в PostgreSQL – основные типы данных. Числа. SQL-запрос на создание таблицы с числами

Ограничения задаем оператором CHECK (не менее 10 и не более 100)

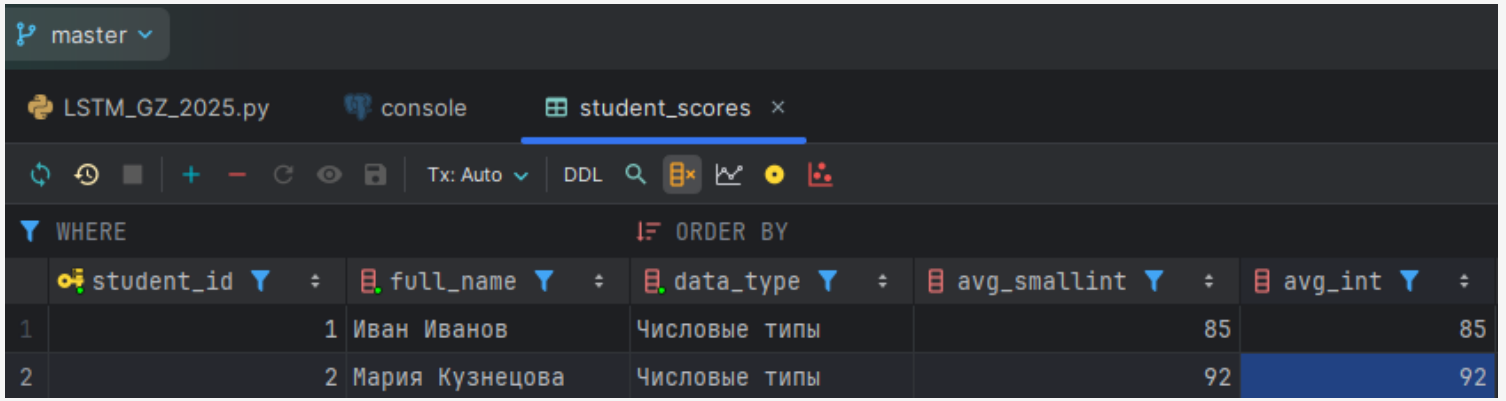
```
CREATE TABLE student_scores (
   student_id      SERIAL PRIMARY KEY,              -- авто-ID (INT)
   full_name       VARCHAR(100) NOT NULL,           -- ФИО

   -- для наглядности сохраняем тип данных отдельной строкой
   data_type       VARCHAR(30) NOT NULL,

   -- варианты хранения среднего балла в разных числовых типах
   avg_smallint    SMALLINT     CHECK (avg_smallint BETWEEN 10 AND 100),
   avg_int         INT          CHECK (avg_int BETWEEN 10 AND 100),
   avg_bigint      BIGINT       CHECK (avg_bigint BETWEEN 10 AND 100),
   avg_numeric     NUMERIC(5,2) CHECK (avg_numeric BETWEEN 10 AND 100),
   avg_real        REAL         CHECK (avg_real BETWEEN 10 AND 100),
   avg_double      DOUBLE PRECISION CHECK (avg_double BETWEEN 10 AND 100)
);

INSERT INTO student_scores(full_name, data_type,
   avg_smallint, avg_int, avg_bigint, avg_numeric, avg_real, avg_double)
VALUES
('Иван Иванов', 'Числовые типы',
   85, 85, 85, 85.50, 85.5, 85.5),

('Мария Кузнецова', 'Числовые типы',
   92, 92, 92, 92.25, 92.3, 92.25);
```

Лекция №2

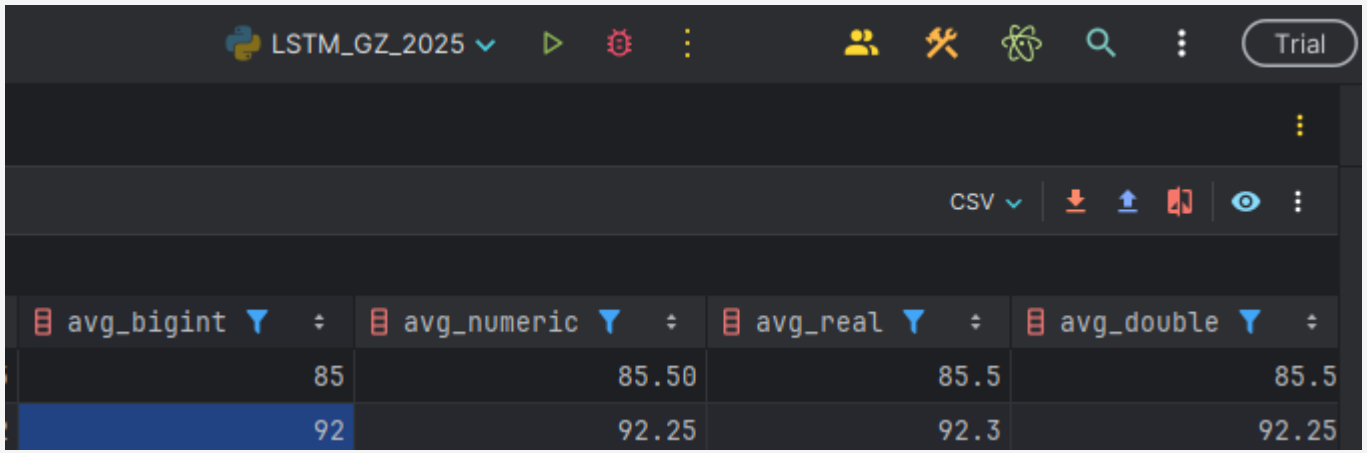
Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Типы данных. Язык SQL. Операторы CREATE и INSERT. Ограничения CHECK, NOT NULL, UNIQUE. Первичный и внешний ключи. Обеспечение ссылочной целостности»

Основы синтаксиса SQL в PostgreSQL – основные типы данных. Числа. Результат выполнения SQL-запроса



	student_id	full_name	data_type	avg_smallint	avg_int
1	1	Иван Иванов	Числовые типы	85	85
2	2	Мария Кузнецова	Числовые типы	92	92



avg_bigint	avg_numeric	avg_real	avg_double
85	85.50	85.5	85.5
92	92.25	92.3	92.25

Лекция №2

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Типы данных. Язык SQL. Операторы CREATE и INSERT. Ограничения CHECK, NOT NULL, UNIQUE. Первичный и внешний ключи. Обеспечение ссылочной целостности»

Основы синтаксиса SQL в PostgreSQL – основные типы данных. Строки

Тип данных	Диапазон длины при указании n	Если n не указан	Что важно знать
TEXT	—	0 … ~1 ГБ текста	Максимальная длина определяется общим лимитом на значение (~1 ГБ); фактическое число символов зависит от кодировки
VARCHAR(n)	0 … n символов, причём 1 ≤ n ≤ 10 485 760	0 … ~1 ГБ (как TEXT)	При превышении n будет ошибка; длина считается в символах, не в байтах
CHAR(n)	ровно n символов, причём 1 ≤ n ≤ 10 485 760	CHAR без n = CHAR(1)	Строка дополняется пробелами до n; хвостовые (т.е. в конце текста) пробелы в сравнении игнорируются

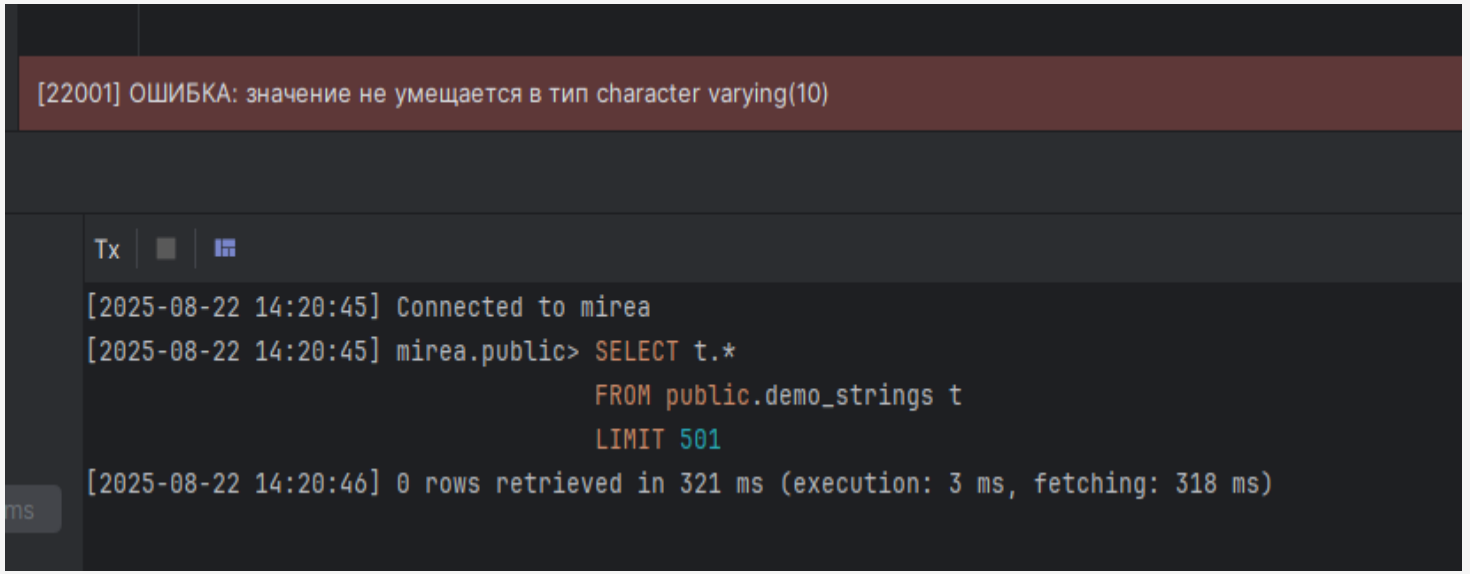
Лекция №2

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Типы данных. Язык SQL. Операторы CREATE и INSERT. Ограничения CHECK, NOT NULL, UNIQUE. Первичный и внешний ключи. Обеспечение ссылочной целостности»

Основы синтаксиса SQL в PostgreSQL – основные типы данных. Пример SQL-запроса и результат

```sql
CREATE TABLE demo_strings (
   id SERIAL PRIMARY KEY,
   fio TEXT,              -- любое ФИО любой длины
   group_code VARCHAR(10),   -- максимум 10 символов
   country_code CHAR(2)      -- всегда ровно 2 символа
);

INSERT INTO demo_strings(fio, group_code, country_code) VALUES
('Иван Иванов', 'ИВТ-101', 'RU'),
('Мария Петровна-Сидорова-Лебедева', 'SUPER-LONG-CODE', 'US'), -- ❌ ошибка по VARCHAR(10)
('Анна Смирнова', 'DB-202', 'A');  -- сохранится 'A ' (с пробелом)
```



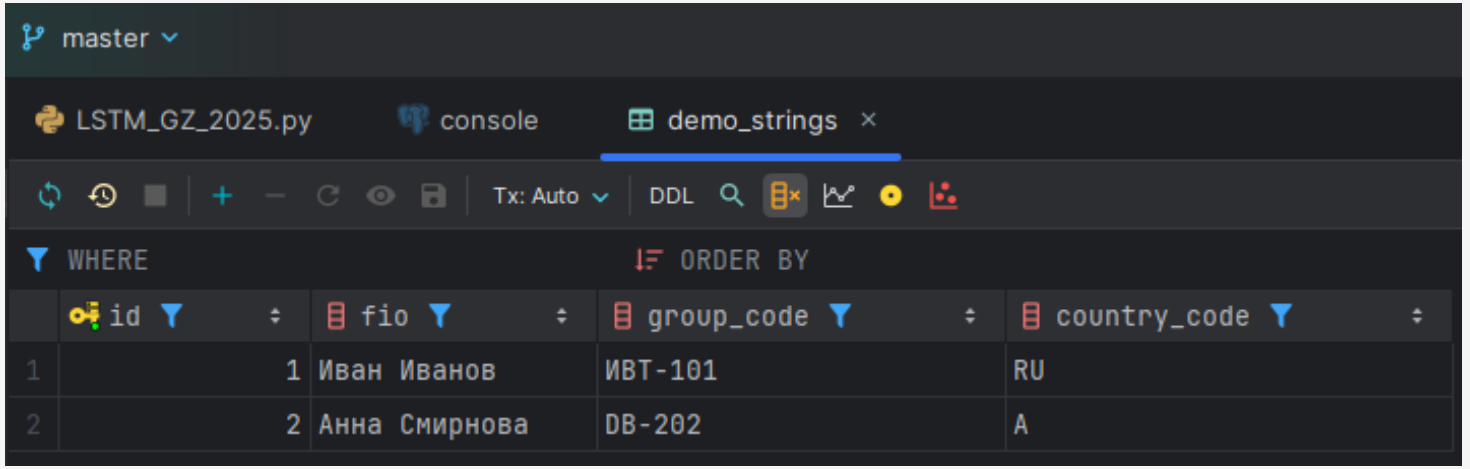
Лекция №2

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Типы данных. Язык SQL. Операторы CREATE и INSERT. Ограничения CHECK, NOT NULL, UNIQUE. Первичный и внешний ключи. Обеспечение ссылочной целостности»

Основы синтаксиса SQL в PostgreSQL – основные типы данных. Пример SQL-запроса и результат

```
CREATE TABLE IF NOT EXISTS demo_strings (
   id SERIAL PRIMARY KEY,
   fio TEXT,              -- любое ФИО любой длины
   group_code VARCHAR(10),   -- максимум 10 символов
   country_code CHAR(2)      -- всегда ровно 2 символа
);

INSERT INTO demo_strings(fio, group_code, country_code) VALUES
('Иван Иванов', 'ИВТ-101', 'RU'),
('Анна Смирнова', 'DB-202', 'A');  -- сохранится 'A ' (с пробелом из-за того, что указали 2 символа (CHAR(2))
```



Лекция №2

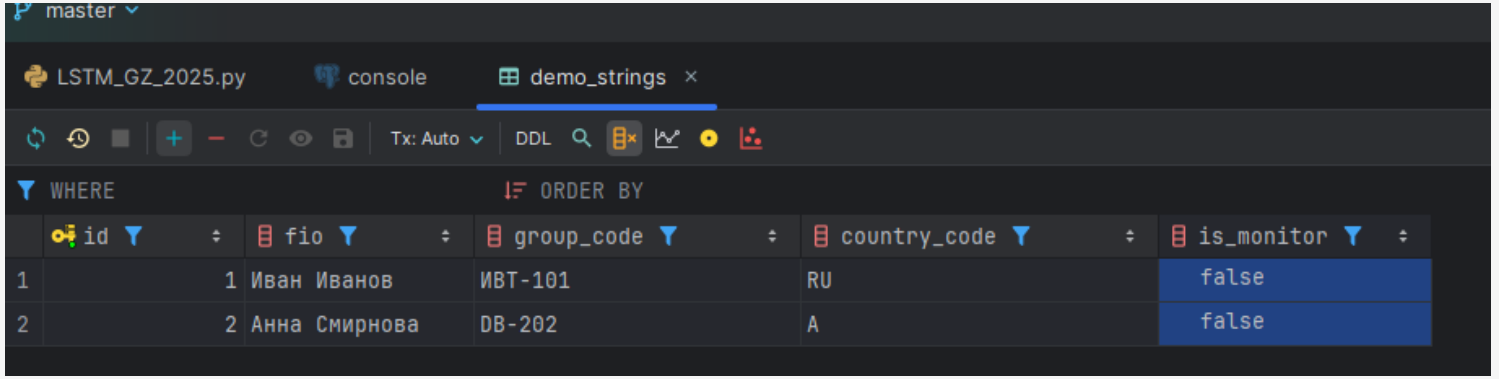
Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Типы данных. Язык SQL. Операторы CREATE и INSERT. Ограничения CHECK, NOT NULL, UNIQUE. Первичный и внешний ключи. Обеспечение ссылочной целостности»

Основы синтаксиса SQL в PostgreSQL – основные типы данных. Логический тип данных

BOOLEAN – логический тип данных. Пример запроса:

```
ALTER TABLE demo_strings ADD COLUMN is_monitor BOOLEAN DEFAULT FALSE;
```



	id	fio	group_code	country_code	is_monitor
1	1	Иван Иванов	ИВТ-101	RU	false
2	2	Анна Смирнова	DB-202	A	false

```
[2025-08-22 14:32:00] mirea.public> ALTER TABLE demo_strings ADD COLUMN is_monitor BOOLEAN DEFAULT FALSE
[2025-08-22 14:32:00] completed in 2 ms
[2025-08-22 14:33:06] mirea.public> ALTER TABLE demo_strings ADD COLUMN is_monitor BOOLEAN DEFAULT FALSE
[2025-08-22 14:33:06] [42701] ОШИБКА: столбец "is_monitor" отношения "demo_strings" уже существует
```

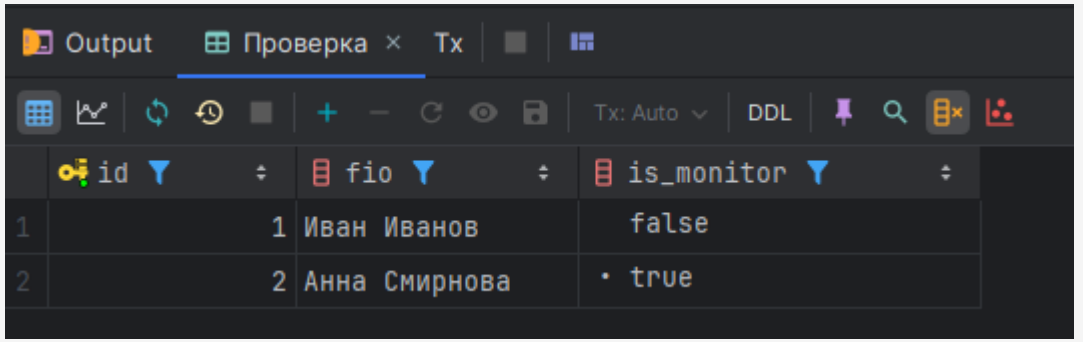
Лекция №2

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Типы данных. Язык SQL. Операторы CREATE и INSERT. Ограничения CHECK, NOT NULL, UNIQUE. Первичный и внешний ключи. Обеспечение ссылочной целостности»

Если хотим, чтобы только к определенному студенту применилось значение TRUE:

```sql
-- 1) Добавим колонку (если её нет)
ALTER TABLE demo_strings
  ADD COLUMN IF NOT EXISTS is_monitor BOOLEAN;

-- 2) По умолчанию для новых строк ставим FALSE
ALTER TABLE demo_strings
  ALTER COLUMN is_monitor SET DEFAULT FALSE;

-- 3) Проставим значения всем существующим строкам:
--    только id=2 → TRUE, остальные → FALSE
UPDATE demo_strings
SET is_monitor = (id = 2);

-- Проверка
SELECT id, fio, is_monitor
FROM demo_strings
ORDER BY id;
```



is_monitor = (student_id = 2) — это короткая запись: выражение в скобках дает TRUE только для id=2, иначе FALSE.

Лекция №2

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Типы данных. Язык SQL. Операторы CREATE и INSERT. Ограничения CHECK, NOT NULL, UNIQUE. Первичный и внешний ключи. Обеспечение ссылочной целостности»

Основы синтаксиса SQL в PostgreSQL – основные типы данных. Дата и время

Тип	Что хранит	Пример литерала	Когда брать/пояснение
DATE	Только дату (без времени)	DATE '2025-09-01'	Обычная дата. Дни рождения, академические даты, дедлайны
TIME	Только время дня (без часового пояса)	TIME '10:30:00'	Обычное время. Локальное расписание внутри дня (одних суток)
TIME WITH TIME ZONE (*TIMETZ*)	Время + смещение TZ	TIME '10:30+02'	Время со смещением во времени. Редко нужно, только если важно указать смещение для времени
TIMESTAMP	Дата+время без TZ (наивное)	TIMESTAMP '2025-09-01 10:30'	Время без часовых поясов
TIMESTAMPTZ	Дата+время со смещением/TZ (момент)	TIMESTAMPTZ '2025-09-01 10:30+02'	Время со смещением (часовыми поясами)
INTERVAL	Длительность/промежуток	INTERVAL '90 min'	Временной интервал. Например, пары по 90 мин, длительность аренды и т. п.

Лекция №2

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Типы данных. Язык SQL. Операторы CREATE и INSERT. Ограничения CHECK, NOT NULL, UNIQUE. Первичный и внешний ключи. Обеспечение ссылочной целостности»

Основы синтаксиса SQL в PostgreSQL – основные типы данных. Даты и время – пример SQL-запроса

```
-- ========================
-- 1) Создаём временную таблицу для демонстрации типов даты/времени
-- ========================
-- TEMP (или TEMPORARY) — таблица живёт только в текущей сессии подключения;
-- при закрытии соединения исчезает автоматически.
CREATE TEMP TABLE demo_times (
 -- Идентификатор строки. INT — 4-байтовое целое.
 -- GENERATED BY DEFAULT AS IDENTITY — автоинкремент по стандарту SQL (1,2,3...),
 -- но при желании можно задать своё значение вручную.
 -- PRIMARY KEY — уникальность + NOT NULL + индекс.
 id            INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,

 -- Только календарная дата без времени и часовых поясов.
 -- Примеры применений: день мероприятия, дедлайн, дата рождения.
 event_date    DATE          NOT NULL,          -- e.g. 2025-09-01

 -- Только локальное «стенное» время без привязки к часовому поясу.
 -- Это не момент на шкале времени, а просто время суток (09:30, 10:00).
 -- Не учитывает часовой пояс и переходы на летнее/зимнее время.
 start_time    TIME          NOT NULL,          -- e.g. 10:00

 -- TIME WITH TIME ZONE (тип TIMETZ) — время суток + числовой сдвиг к UTC.
 -- Это тоже НЕ абсолютный момент: хранится «10:00 +03:00» как пара (время, смещение).
 -- Зональные имена (Europe/Moscow) при вводе превращаются в числовой offset.
 -- Обычно этот тип редко нужен; чаще достаточно TIME или TIMESTAMPTZ.
 start_time_tz TIME WITH TIME ZONE NOT NULL,      -- e.g. 10:00+03

 -- TIMESTAMP (без TZ) — локальная дата+время без часового пояса.
 -- Интерпретация зависит от того, какой часовой пояс вы подразумеваете снаружи.
 -- Подходит для «стенного» расписания внутри одной зоны; НЕ хранит «момент».
 ts_local      TIMESTAMP     NOT NULL,          -- e.g. 2025-09-01 10:00

 -- TIMESTAMPTZ — дата+время с часовым поясом (абсолютный момент).
 -- Внутри хранится как UTC, при выводе конвертируется в текущий TimeZone сеанса.
 -- Это предпочтительный тип для событий в реальном мире (встречи, логи).
 ts_moment     TIMESTAMPTZ   NOT NULL,          -- e.g. 2025-09-01 10:00+03  ≡ 07:00Z

 -- INTERVAL — длительность/промежуток (минуты, часы, дни и т.п.).
 -- Применяется для прибавления/вычитания от TIMESTAMP/DATE/TIME.
 duration      INTERVAL      NOT NULL           -- e.g. '90 min', '2 hours'
);
```

```
-- ========================
-- 2) Вставляем демонстрационные строки
-- ========================
-- Обратите внимание на литералы с явным указанием типа слева
(DATE/TIME/TIMESTAMP/TIMESTAMPTZ/INTERVAL).
-- Это самая читаемая и надёжная форма записи для учебных примеров.

INSERT INTO demo_times
 (event_date,    start_time,    start_time_tz,    ts_local,              ts_moment,              duration)
VALUES
 -- Строка №1
 (
  DATE '2025-09-01',          -- 1 сентября 2025 (только дата)
  TIME '10:00',               -- локальное 10:00 без TZ
  TIME '10:00+03',            -- время 10:00 в зоне UTC+03 (тип TIMETZ)
  TIMESTAMP  '2025-09-01 10:00',  -- локальный timestamp (без TZ)
  TIMESTAMPTZ '2025-09-01 10:00+03', -- абсолютный момент: 07:00 UTC (хранится как UTC, вывод по
TimeZone сеанса)
  INTERVAL '90 min'           -- длительность 1 час 30 минут
 ),

 -- Строка №2
 (
  DATE '2025-12-15',          -- 15 декабря 2025
  TIME '09:30',               -- локальное 09:30 без TZ
  TIME '09:30+01',            -- 09:30 при смещении +01 (например, CET зимой)
  TIMESTAMP  '2025-12-15 09:30',  -- локальный timestamp (без TZ)
  TIMESTAMPTZ '2025-12-15 09:30+01', -- абсолютный момент: 08:30 UTC
  INTERVAL '2 hours'          -- длительность 2 часа
 );

-- ========================
-- 3) Смотрим, что хранится/отображается
-- ========================
-- Порядок строк — по id (1, затем 2...). TIMESTAMPTZ будет отформатирован в TimeZone текущего сеанса.
-- Узнать текущую зону можно командой: SHOW TIME ZONE;
SELECT
 id,
 event_date,    -- только дата
 start_time,    -- «стенное» время без TZ
 start_time_tz, -- время + смещение (TIMETZ)
 ts_local,      -- локальный timestamp без TZ
 ts_moment,     -- абсолютный момент (TIMESTAMPTZ), выводится в зоне сеанса
 duration       -- интервал
FROM demo_times
ORDER BY id;
```

Output | Смотрим результат (всё в одной таблице) | Tx

CSV

	id	event_date	start_time	start_time_tz	ts_local	ts_moment	duration
1	1	2025-09-01	10:00:00	10:00:00 +00:00	2025-09-01 10:00:00.000000	2025-09-01 07:00:00.000000 +00:00	0 years 0 mons 0 days 1 hours 30 mins 0.0 secs
2	2	2025-12-15	09:30:00	09:30:00 +00:00	2025-12-15 09:30:00.000000	2025-12-15 08:30:00.000000 +00:00	0 years 0 mons 0 days 2 hours 0 mins 0.0 secs

Подсказка: TIMESTAMPTZ хранит текущее указанное время (в UTC). Если сделать SET TIMEZONE TO 'Europe/Brussels'; и затем SELECT ts_moment;, вы увидите такое же время, но отформатированное под текущую часовую зону (часовой пояс).

Лекция №2

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Типы данных. Язык SQL. Операторы CREATE и INSERT. Ограничения CHECK, NOT NULL, UNIQUE. Первичный и внешний ключи. Обеспечение ссылочной целостности»

Основы синтаксиса SQL в PostgreSQL – основные типы данных. JSON/JSONB

JSON и JSONB — это два типа для хранения структурированных данных в PostgreSQL. JSON сохраняет текст в исходном виде, включая пробелы и дублирующиеся ключи. JSONB (Binary JSON) использует бинарное представление, не сохраняя пробелы или порядок, но гарантируя уникальность ключей. В таблице будет проще понять, когда и как их использовать для эффективного хранения данных.

Что такое JSON (формат)
JSON состоит из:
1. **Объектов** — фигурные скобки { }, пара **"ключ": значение, т.е. словарь**.
2. **Массивов** — квадратные скобки [ ], список значений.
3. **Значений** — строка, число, true/false, null, объект или массив.

Правила формата:
4. **Строки и ключи** — **в двойных кавычках** "...".
5. **Запятых в конце** списков быть **не должно**.
6. **Комментарии** (// или /* */) **нельзя**.

```
{
  "studentId": 101,
  "fullName": "Иван Иванов",
  "isMonitor": false,
  "email": null
}
```

Лекция №2

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Типы данных. Язык SQL. Операторы CREATE и INSERT. Ограничения CHECK, NOT NULL, UNIQUE. Первичный и внешний ключи. Обеспечение ссылочной целостности»

Основы синтаксиса SQL в PostgreSQL – основные типы данных. JSON/JSONB – основные отличия

Критерий	JSON	JSONB	Комментарии
Как хранится	Как текст (строка JSON)	Как бинарная структура	JSONB уже «разобран», его проще и быстрее искать
Порядок ключей, пробелы	Сохраняет как ввели	Не сохраняет (порядок/пробелы не важны)	У JSON можно увидеть «оригинальную» запись; у JSONB порядок может поменяться
Дубликаты ключей в объекте	Остаются как есть	Убираются, остаётся последнее значение	В {"a":1,"a":2} у JSONB будет только "a":2
Индексы и скорость поиска	Нет GIN-индексов (по умолчанию медленно)	Есть GIN/GiST индексы (быстро)	Для фильтров по JSON-полям выбирайте JSONB
Операторы/функции	Базовый доступ ->, ->>	Плюс: @> (содержит), ?, `?	, ?&, jsonb_set,
Сравнение/группировка	Нельзя сравнивать = и использовать в GROUP BY	Можно (=, GROUP BY, DISTINCT)	JSONB можно агрегировать и сравнивать
Типичное применение	Хранить «как пришло» (сырые логи, оригинал)	Рабочий формат для запросов и аналитики	В проде почти всегда JSONB

Лекция №2

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Типы данных. Язык SQL. Операторы CREATE и INSERT. Ограничения CHECK, NOT NULL, UNIQUE. Первичный и внешний ключи. Обеспечение ссылочной целостности»

Основы синтаксиса SQL в PostgreSQL – основные типы данных. JSON/JSONB – основные отличия на практике (SQL-запрос с выводом данных) – Часть 1

```
-- ============================
-- 0) Пересоздаём «песочницу» для лабораторной по JSONB
-- ============================
DROP SCHEMA IF EXISTS lab_json CASCADE;   -- удаляем схему со всем, что внутри
CREATE SCHEMA lab_json;                   -- создаём чистую схему
SET search_path TO lab_json;              -- чтобы не писать lab_json. перед именами

-- ============================
-- 1) Базовые сущности: студенты и их гибкий профиль в JSONB
-- ============================

-- Таблица «жёсткой» структуры: только id и ФИО.
CREATE TABLE students (
  student_id INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,  -- автоинкрементный PK
  full_name  TEXT NOT NULL                                      -- имя студента
);

-- Таблица «гибкой» структуры: произвольные поля анкеты в JSONB.
-- Логика: один профиль на одного студента (1:1), удаляем студента — удалится и профиль.
CREATE TABLE student_profile (
  student_id INT PRIMARY KEY
          REFERENCES students(student_id) ON DELETE CASCADE, -- каскадное удаление
  extra      JSONB NOT NULL                                 -- JSONB: быстрый поиск, индексы GIN
  -- JSONB автоматически нормализует порядок ключей, удаляет дубли.
);

-- Данные студентов
INSERT INTO students(full_name) VALUES
  ('Иван Иванов'),
  ('Мария Кузнецова'),
  ('Анна Смирнова');

-- Профили: обращаем внимание на строгий JSON-формат (двойные кавычки у ключей/строк).
-- Здесь демонстрируются:
--  • простые поля ("tg"),
--  • массивы ("skills"),
--  • вложенные объекты ("address"),
--  • массив объектов ("contacts").
INSERT INTO student_profile(student_id, extra) VALUES
(1, '{
  "tg": "@ivan",
  "skills": ["sql","python"],
  "address": {"city":"Minsk","zip":"220000"},
  "contacts": [
    {"type":"mobile","value":"+375-29-000-00-00"},
    {"type":"home","value":"+375-17-123-45-67"}
  ]
}'),
(2, '{
  "tg": "@maria",
  "skills": ["ml","sql"],
  "address": {"city":"Warsaw","zip":"00-001"}
}'),
(3, '{
  "skills": ["python","docker"]
}');
```

Лекция №2

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Типы данных. Язык SQL. Операторы CREATE и INSERT. Ограничения CHECK, NOT NULL, UNIQUE. Первичный и внешний ключи. Обеспечение ссылочной целостности»

Основы синтаксиса SQL в PostgreSQL – основные типы данных. JSON/JSONB – основные отличия на практике (SQL-запрос с выводом данных) – Часть 2

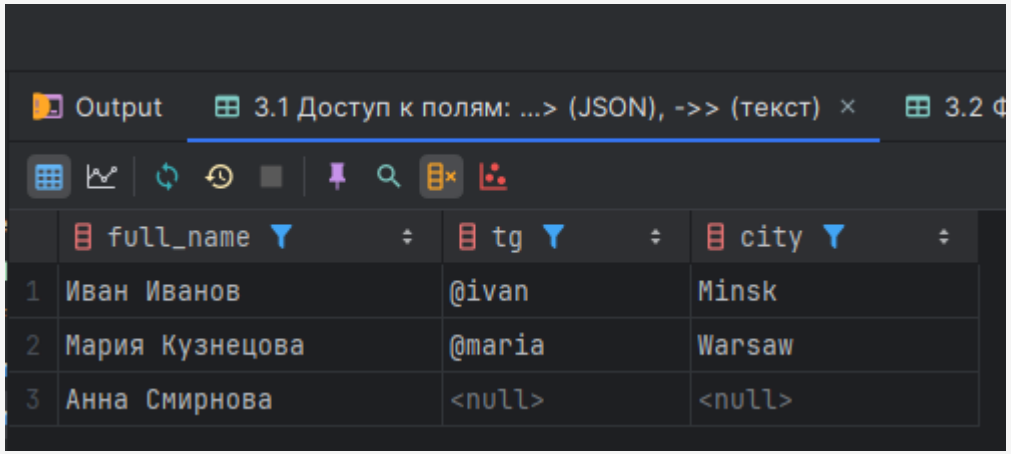
```
-- Индекс GIN для JSONB.
-- Класс операторов jsonb_path_ops — компактный индекс, хорошо ускоряет операции «содержит» (@>).
-- Если нужен более широкий спектр ускоряемых операторов (? ?& ?| и др.), используйте дефолтный класс (jsonb_ops):
--   CREATE INDEX ... USING GIN (extra);   -- без указания jsonb_path_ops
CREATE INDEX idx_student_profile_extra
  ON student_profile USING GIN (extra jsonb_path_ops);

-- ========================
-- 2) Короткая демонстрационная таблица (для простых примеров @> по JSONB)
-- ========================

CREATE TABLE profiles (
  uid  INT PRIMARY KEY,
  data JSONB NOT NULL
);

INSERT INTO profiles(uid, data) VALUES
  (1, '{"skills":["sql","python"],"city":"Minsk"}'),
  (2, '{"skills":["ml","sql"],"city":"Warsaw"}');

CREATE INDEX idx_profiles_data
  ON profiles USING GIN (data jsonb_path_ops);

-- ========================
-- 3) Примеры запросов
-- ========================

-- 3.1 Доступ к полям JSONB:
--   ->   : извлечь как JSON/JSONB (тип остаётся jsonb)
--   ->>  : извлечь как текст (text) — удобно для сравнений/вывода
-- Вложенные поля: p.extra->'address'->>'city'
SELECT
  s.full_name,
  p.extra->>'tg'            AS tg,    -- вытащили текст из поля "tg" (или NULL, если нет)
  p.extra->'address'->>'city' AS city   -- спустились в объект "address" и взяли "city" как текст
FROM student_profile p
JOIN students s USING (student_id)
ORDER BY s.student_id;

-- 3.2 Фильтр «содержит» (@>): у кого среди skills есть "sql"
-- ВАЖНО: справа — корректный JSON, а не текст: '{"skills":["sql"]}'
-- Этот запрос сопоставляет поддокумент и учитывает массивы.
SELECT s.full_name
FROM student_profile p
JOIN students s USING (student_id)
WHERE p.extra @> '{"skills":["sql"]}';

-- 3.3 Проверка наличия ключа (оператор ?):
-- p.extra ? 'tg'  → TRUE, если в корне профиля есть ключ "tg".
SELECT
  s.full_name,
  (p.extra ? 'tg') AS has_tg
FROM student_profile p
JOIN students s USING (student_id)
ORDER BY s.student_id;
```

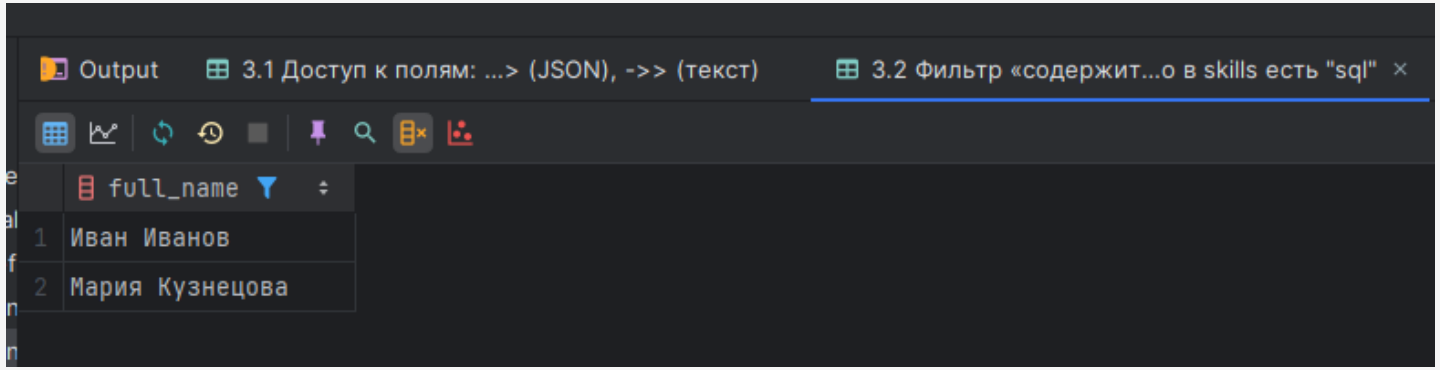
Лекция №2

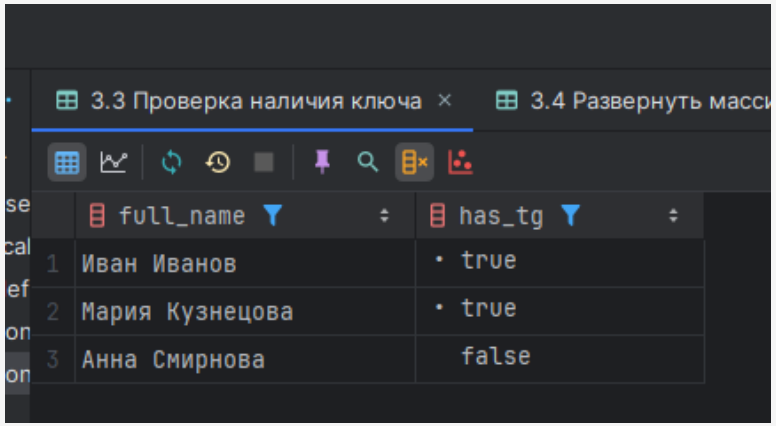
Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Типы данных. Язык SQL. Операторы CREATE и INSERT. Ограничения CHECK, NOT NULL, UNIQUE. Первичный и внешний ключи. Обеспечение ссылочной целостности»

Основы синтаксиса SQL в PostgreSQL – основные типы данных. JSON/JSONB – основные отличия на практике (SQL-запрос с выводом данных) – Часть 2 (результаты)







Лекция №2

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Типы данных. Язык SQL. Операторы CREATE и INSERT. Ограничения CHECK, NOT NULL, UNIQUE. Первичный и внешний ключи. Обеспечение ссылочной целостности»

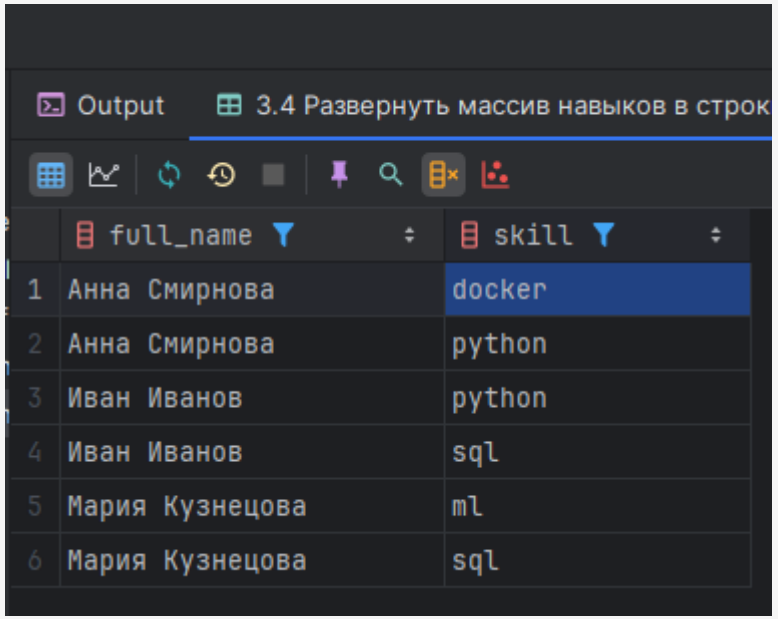
Основы синтаксиса SQL в PostgreSQL – основные типы данных. JSON/JSONB – основные отличия на практике (SQL-запрос с выводом данных) – Часть 3

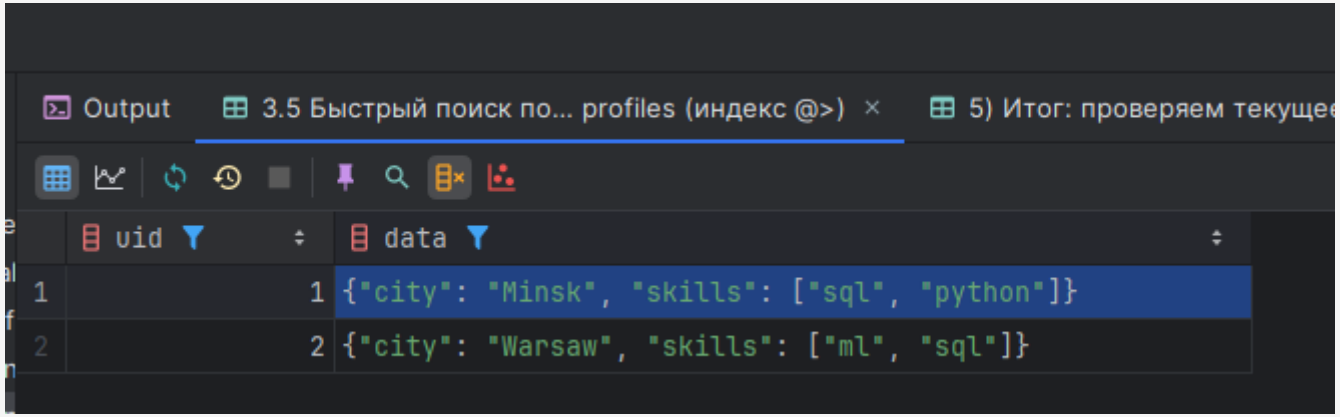
```sql
-- 3.4 Развернуть массив навыков в строки
SELECT s.full_name, skill
FROM student_profile p
JOIN students s USING (student_id)
CROSS JOIN LATERAL jsonb_array_elements_text(p.extra->'skills') AS t(skill)
ORDER BY s.full_name, skill;

-- 3.5 Быстрый поиск по profiles (индекс @>)
SELECT uid, data
FROM profiles
WHERE data @> '{"skills":["sql"]}';

-- 4) Обновления внутри JSONB

-- 4.1 Поменять tg у Ивана
UPDATE student_profile
SET extra = jsonb_set(extra, '{tg}', '"@ivan_new"', true)
WHERE student_id = 1;

-- 4.2 Добавить навык "docker" у Марии (аккуратно к массиву)
UPDATE student_profile
SET extra = jsonb_set(
        extra,
        '{skills}',
        COALESCE(extra->'skills','[]'::jsonb) || '"docker"'::jsonb,
        true)
WHERE student_id = 2;

-- 4.3 Удалить ключ "contacts" у Ивана
UPDATE student_profile
SET extra = extra - 'contacts'
WHERE student_id = 1;
```

Лекция №2

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Типы данных. Язык SQL. Операторы CREATE и INSERT. Ограничения CHECK, NOT NULL, UNIQUE. Первичный и внешний ключи. Обеспечение ссылочной целостности»





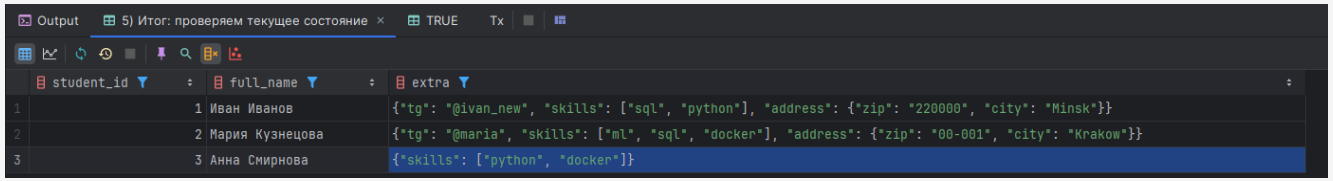
Лекция №2

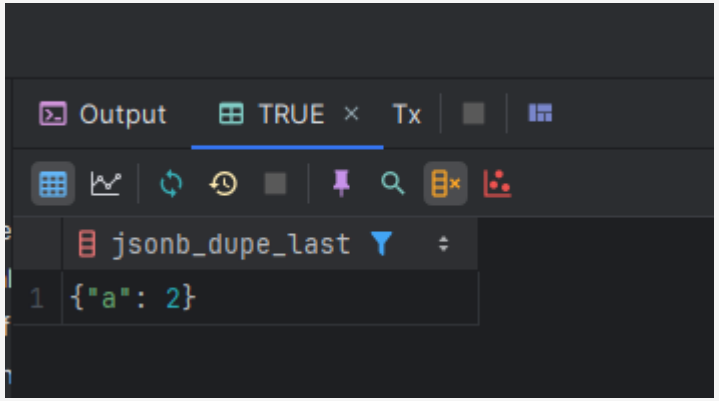
Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Типы данных. Язык SQL. Операторы CREATE и INSERT. Ограничения CHECK, NOT NULL, UNIQUE. Первичный и внешний ключи. Обеспечение ссылочной целостности»

Основы синтаксиса SQL в PostgreSQL – основные типы данных. JSON/JSONB – основные отличия на практике (SQL-запрос с выводом данных) – Часть 4

```sql
-- 4.4 Обновить вложенное поле (город) у Марии
UPDATE student_profile
SET extra = jsonb_set(extra, '{address,city}', '"Krakow"', true)
WHERE student_id = 2;

-- 5) Итог: проверяем текущее состояние
SELECT s.student_id, s.full_name, p.extra
FROM students s
LEFT JOIN student_profile p USING (student_id)
ORDER BY s.student_id;

-- 6) Мини-разница JSON vs JSONB
-- Порядок ключей у JSON сохраняется, у JSONB — нет, равенство структурно:
SELECT ('{"a":1,"b":2}'::jsonb = '{"b":2,"a":1}'::jsonb) AS jsonb_equal;  -- TRUE, так как нет дубликата, а в строке ниже – есть дубликат ключа
SELECT '{"a":1,"a":2}'::jsonb AS jsonb_dupe_last;
```





Лекция №2

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Типы данных. Язык SQL. Операторы CREATE и INSERT. Ограничения CHECK, NOT NULL, UNIQUE. Первичный и внешний ключи. Обеспечение ссылочной целостности»

Основы синтаксиса SQL в PostgreSQL – основные типы данных. Массивы

Массив в SQL (в PostgreSQL) — это тип данных столбца, который хранит упорядоченный список значений **одного типа**; поддерживает обращение по индексу и спецоперации над элементами/подмножествами. Уместен для небольших вспомогательных списков, но не для моделирования связей между таблицами.

Особенности в PostgreSQL:

1. Любой скалярный тип можно сделать массивом: text[], int[], uuid[], timestamptz[] и др.
2. В одной ячейке хранится упорядоченный список значений одного типа.
3. Индексация с 1: arr[1] — первый элемент; есть срезы arr[2:4].

Когда уместно и неуместно применять? 2 гарантированных случая:

1. УМЕСТНО: если список короткий (обычно ≤ 10–20 элементов), редко меняется, нужно «наличие/пересечение», и на элементы не нужны внешние ключи и ограничения FK/UNIQUE → массивы подходят.
2. НЕУМЕСТНО: если элементы — сущности со своей жизнью (нужны PK/FK/UNIQUE, отчёты, частые операции объединения (join), поиск по подстроке, триггеры на один элемент) → нормализуйте в отдельную таблицу.

Индексация

В PostgreSQL индексация массивов по умолчанию начинается с 1: arr[1] — первый элемент.
arr[0] обычно дает ошибку «array subscript out of range».
В PostgreSQL можно задать другую нижнюю границу (редко используется):

```
SELECT ('[0:2]={10,20,30}'::int[])[0];  -- 10
```

Тогда индексация будет начинаться с 10.

Лекция №2

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Типы данных. Язык SQL. Операторы CREATE и INSERT. Ограничения CHECK, NOT NULL, UNIQUE. Первичный и внешний ключи. Обеспечение ссылочной целостности»

Основы синтаксиса SQL в PostgreSQL – основные типы данных. Массивы – когда уместно и неуместно использовать

Сценарий	Пример данных	Типичные операции	Массивы подходят? Почему?
Теги/метки без ссылок	tags text[]	Проверяем, есть ли тег/пересечение (@>, &&, ANY())	✔ **Да.** Короткий однотипный список, быстро проверить наличие.
Кэш для витрин/поиска	phones_cache text[]	Быстрый @> по «белому списку»	✔ **Да.** Храним вычисленный список, не ядро.
Маленькие списки параметров	«быстрые фильтры», «разрешённые опции в text[]»	Проверка принадлежности значений	✔ **Да.** Простое хранение, если элементов мало и редко применяются.
Порядок элементов имеет смысл	Приоритеты ['high', 'low'], т.е. высокий или низкий	Берем элементы массива и выводим их вместе с их порядковыми номерами, в том же порядке, как они лежат в массиве (arr[n], WITH ORDINALITY)	✔ **Да.** Массив хранит порядок, удобно для коротких списков.
Хранение координат/векторов фикс. длины	[x,y], [r,g,b]	Читаем целиком как один объект	✔ **Да.** Маленькие фиксированные наборы значений.
Большие/часто меняющиеся наборы	«тысячи телефонов», «история событий», т.е. большие данные	Частые вставки/удаления по одному	✘ **Нет.** Переписывается весь массив → «дорого по времени и ресурсам».
Нужны ссылки/уникальность на элемент	Ссылка на справочник кодов или коды из справочника	Гарантия целостности, т.е. Требуется FOREIGN KEY, UNIQUE, валидация каждого элемента	✘ **Нет.** На элементы массива нельзя повесить FK/UNIQUE.
Поиск по подстроке внутри элементов	Маски телефонов, т.е. шаблоны для поиска номеров по образцу (префиксу/формату).	LIKE, регулярки, триграммы	✘ **Нет.** GIN по массиву **не** ускорит LIKE.
Неоднородные/вложенные структуры	«параметры разного типа»	Поиск по ключам/пути	✘ **Нет.** Массив типизирован и плоский. Лучше JSONB.

Лекция №2

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Типы данных. Язык SQL. Операторы CREATE и INSERT. Ограничения CHECK, NOT NULL, UNIQUE. Первичный и внешний ключи. Обеспечение ссылочной целостности»

Основы синтаксиса SQL в PostgreSQL – основные типы данных. Массивы – операторы/конструкции

Оператор / конструкция	Что делает / операция	Пример (возвращает TRUE?)	Работает с GIN-индексом на массиве	Комментарий
@>	**Левый массив содержит правый** (все элементы правого есть в левом, порядок неважен)	ARRAY['a','b','c'] @> ARRAY['a','c'] → TRUE	**Да**	Самый частый. Быстро с CREATE INDEX ... USING GIN(arr);
<@	**Левый массив содержится в правом**	ARRAY['a','c'] <@ ARRAY['a','b','c'] → TRUE	**Да**	Обратный к @>.
&&	**Есть пересечение** (хотя бы один общий элемент)	ARRAY['a','x'] && ARRAY['y','a'] → TRUE	**Да**	Проверка «есть ли что-то общее», т.е. проверка на наличие общих элементов.
=	**Полное равенство массивов** (и состав, и порядок, и длина)	ARRAY['a','b'] = ARRAY['a','b'] → TRUE	Обычно **нет**	Редко индексируется. Для равенства чаще не делают индекс.
<>	**Не равны**	ARRAY['a'] <> ARRAY['a','b'] → TRUE	**Нет**	Логическое отрицание равенства.
`		`	**Склейка массивов**	`ARRAY[1,2]
arr[i]	**Элемент по индексу** (индексы с **1**)	ARRAY['a','b','c'][2] → 'b'	**Нет**	Ошибка, если i вне границ.
arr[i:j]	**Срез** (подмассив)	ARRAY[1,2,3,4][2:3] → {2,3}	**Нет**	Позиции включительно.
X = ANY(arr)	**Есть элемент X в массиве**	'a' = ANY(ARRAY['b','a']) → TRUE	**Да**, если переписать как arr @> ARRAY['a']	Для надёжного использования GIN лучше писать @> ARRAY['X'].
X <> ALL(arr)	**X не равен ни одному элементу**	'z' <> ALL(ARRAY['a','b']) → TRUE	**Нет**	«Ни один из».
cardinality(arr)	**Длина массива**	cardinality(ARRAY[10,20]) → 2	**Нет**	Аналог array_length(arr,1).
array_position(arr, X)	**Первая позиция X**	array_position(ARRAY['a','b','a'],'a') → 1	**Нет**	Для всех позиций: array_positions.
unnest(arr)	**Развернуть в строки**	unnest(ARRAY['a','b']) → строки 'a','b'	**Нет**	С порядком: WITH ORDINALITY.

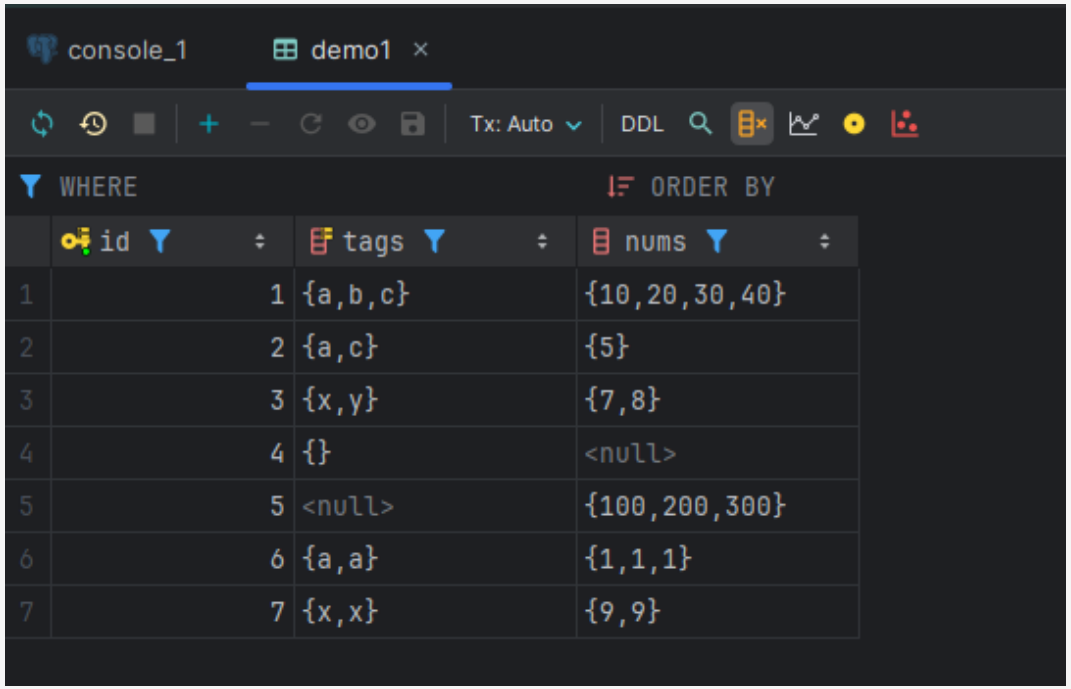
Лекция №2

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Типы данных. Язык SQL. Операторы CREATE и INSERT. Ограничения CHECK, NOT NULL, UNIQUE. Первичный и внешний ключи. Обеспечение ссылочной целостности»

Основы синтаксиса SQL в PostgreSQL – основные типы данных. Массивы – примеры SQL-запросов и результаты

```sql
-- 0) Создаём учебную таблицу с массивами
CREATE TABLE demo1 (
  id   int PRIMARY KEY,
  tags text[],    -- массив тегов
  nums int[]      -- массив чисел
);

INSERT INTO demo1(id, tags, nums) VALUES
(1, ARRAY['a','b','c'], ARRAY[10,20,30,40]),
(2, ARRAY['a','c'],     ARRAY[5]),
(3, ARRAY['x','y'],     ARRAY[7,8]),
(4, ARRAY[]::text[],    NULL),          -- пустой массив тегов
(5, NULL,               ARRAY[100,200,300]), -- NULL в tags
(6, ARRAY['a','a'],     ARRAY[1,1,1]),
(7, ARRAY['x','x'],     ARRAY[9,9]);

-- Вывод всех данных (массивов)
SELECT FROM demo1;
```



	id	tags	nums
1	1	{a,b,c}	{10,20,30,40}
2	2	{a,c}	{5}
3	3	{x,y}	{7,8}
4	4	{}	<null>
5	5	<null>	{100,200,300}
6	6	{a,a}	{1,1,1}
7	7	{x,x}	{9,9}

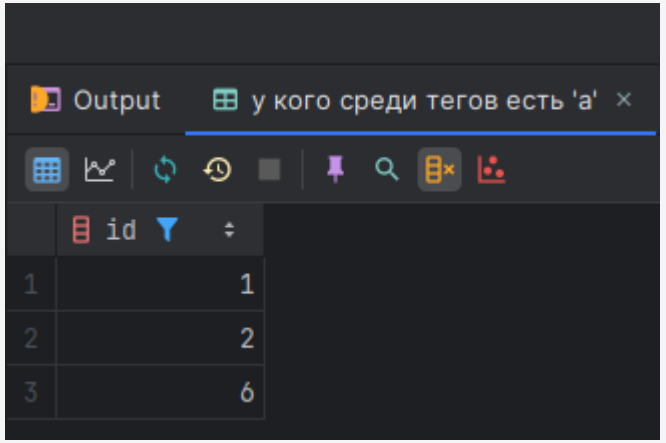
Лекция №2

Тема: «Основы синтаксиса SQL в PostgreSQL»

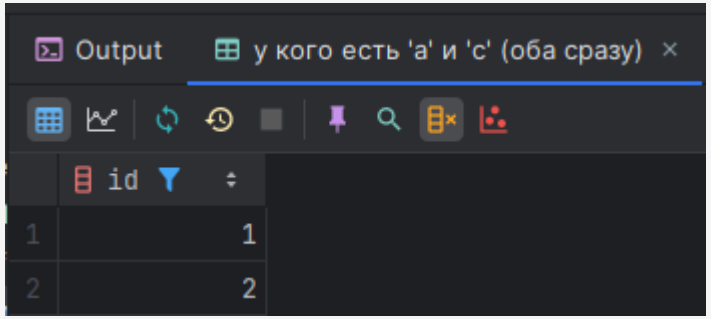
Подтема: «Типы данных. Язык SQL. Операторы CREATE и INSERT. Ограничения CHECK, NOT NULL, UNIQUE. Первичный и внешний ключи. Обеспечение ссылочной целостности»

Основы синтаксиса SQL в PostgreSQL – основные типы данных. Массивы – примеры SQL-запросов и результаты

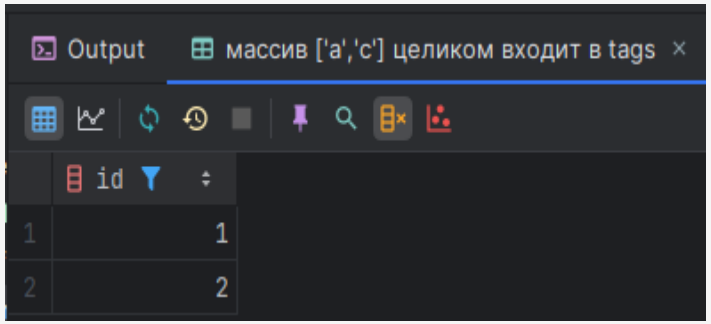
```sql
-- 1) @>  — "левый массив СОДЕРЖИТ правый (все его элементы)"
-- у кого среди тегов есть 'a'
SELECT id
FROM demo1
WHERE tags @> ARRAY['a'];
-- ожидаем: id = 1,2,6
```




```sql
-- у кого есть 'a' и 'c' (оба сразу)
SELECT id
FROM demo1
WHERE tags @> ARRAY['a','c'];
-- ожидаем: id = 1,2
```




```sql
-- 2) <@ — "левый массив СОДЕРЖИТСЯ в правом"
-- массив ['a','c'] целиком входит в tags
SELECT id
FROM demo1
WHERE ARRAY['a','c'] <@ tags;
-- ожидаем: id = 1,2
```



Лекция №2

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Типы данных. Язык SQL. Операторы CREATE и INSERT. Ограничения CHECK, NOT NULL, UNIQUE. Первичный и внешний ключи. Обеспечение ссылочной целостности»

Основы синтаксиса SQL в PostgreSQL – основные типы данных. Массивы – примеры SQL-запросов и результаты

```sql
-- 3) && — "есть ПЕРЕСЕЧЕНИЕ (хотя бы один общий элемент)"
-- есть ли пересечение с {'y','z'}
SELECT id
FROM demo1
WHERE tags && ARRAY['y','z'];
-- ожидаем: id = 3 (у него есть 'y')
```

Output | есть ли пересечение с {'y','z'}

	id
1	3

```sql
-- 4) = — "полное РАВЕНСТВО массивов (состав, порядок, длина)"
SELECT id
FROM demo1
WHERE tags = ARRAY['a','c'];
-- ожидаем: id = 2 (строго ['a','c'], а не ['c','a'])
```

Output | 4) = — "полное РАВЕН...тав, порядок, длина)"

	id
1	2

```sql
-- 5) <> — "массивы НЕ равны"
SELECT id
FROM demo1
WHERE tags <> ARRAY['a','b','c'];
-- ожидаем: id = 2,3,4,6,7   (строка с NULL (id=5) не вернётся — сравнение с NULL даёт NULL)
```

Output | 5) <> — "массивы НЕ равны"

	id
1	2
2	3
3	4
4	6
5	7

Лекция №2

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Типы данных. Язык SQL. Операторы CREATE и INSERT. Ограничения CHECK, NOT NULL, UNIQUE. Первичный и внешний ключи. Обеспечение ссылочной целостности»

Основы синтаксиса SQL в PostgreSQL – основные типы данных. Массивы – примеры SQL-запросов и результаты

```sql
-- 6) || — "СКЛЕИТЬ массивы"
-- просто покажем результат склейки без изменения таблицы
SELECT id, tags, tags || ARRAY['z'] AS tags_plus_z
FROM demo1
ORDER BY id;
-- у каждой строки справа добавится 'z'
```

Output | просто покажем резул...без изменения таблицы

	id	tags	tags_plus_z
1	1	{a,b,c}	{a,b,c,z}
2	2	{a,c}	{a,c,z}
3	3	{x,y}	{x,y,z}
4	4	{}	{z}
5	5	<null>	{z}
6	6	{a,a}	{a,a,z}
7	7	{x,x}	{x,x,z}

```sql
-- 7) arr[i] — "элемент по номеру" (нумерация с 1)
SELECT id,
    nums[1] AS первый,
    nums[2] AS второй
FROM demo1
WHERE nums IS NOT NULL
ORDER BY id;
-- возьмём 1-й и 2-й элементы там, где nums не NULL
```

Output | 7) arr[i] — "элемент...меру" (нумерация с 1)

	id	первый	второй
1	1	10	20
2	2	5	<null>
3	3	7	8
4	5	100	200
5	6	1	1
6	7	9	9

```sql
-- 8) arr[i:j] — "срез (подмассив) с i по j, включительно"
SELECT id, nums[2:3] AS срез_2_3
FROM demo1
WHERE cardinality(nums) >= 3
ORDER BY id;
-- покажем 2..3 элементы только там, где их достаточно
```

Output | 8) arr[i:j] — "срез ...i по j, включительно"

	id	срез_2_3
1	1	{20,30}
2	5	{200,300}
3	6	{1,1}

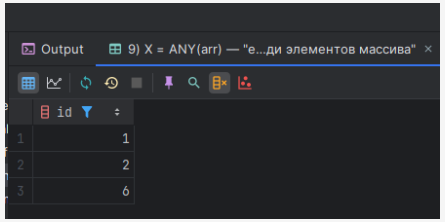
Лекция №2

Тема: «Основы синтаксиса SQL в PostgreSQL»

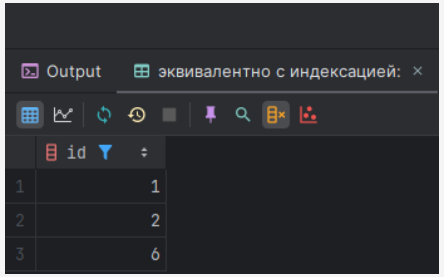
Подтема: «Типы данных. Язык SQL. Операторы CREATE и INSERT. Ограничения CHECK, NOT NULL, UNIQUE. Первичный и внешний ключи. Обеспечение ссылочной целостности»

Основы синтаксиса SQL в PostgreSQL – основные типы данных. Массивы – примеры SQL-запросов и результаты

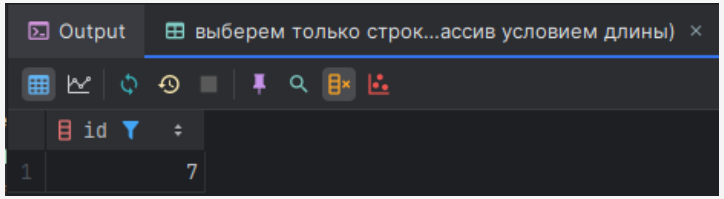
```sql
-- 9) X = ANY(arr) — "есть ли X среди элементов массива"
SELECT id
FROM demo1
WHERE 'a' = ANY(tags);
-- ожидаем: id = 1,2,6   (в пустом массиве и при NULL это FALSE/NULL)
```



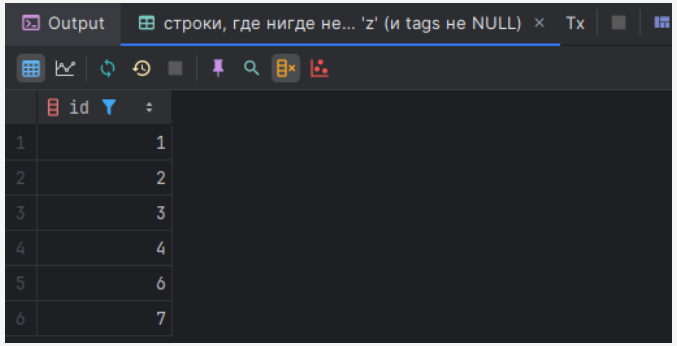

```sql
-- (хитрость: для задействования GIN лучше писать так же, как @> с одиночным элементом)
-- эквивалентно с индексацией:
SELECT id
FROM demo1
WHERE tags @> ARRAY['a'];
```




```sql
-- 10) X = ALL(arr) — "X равен КАЖДОМУ элементу массива"
-- выберем только строки, где все теги — 'x' (и исключим пустой массив условием длины)
SELECT id
FROM demo1
WHERE cardinality(tags) > 0
  AND 'x' = ALL(tags);
-- ожидаем: id = 7   (у него ['x','x'])
```




```sql
-- 11) X <> ALL(arr) — "X не равен НИ одному элементу массива"
-- строки, где нигде нет 'z' (и tags не NULL)
SELECT id
FROM demo1
WHERE tags IS NOT NULL
  AND 'z' <> ALL(tags);
-- ожидаем: id = 1,2,3,4,6,7   (везде нет 'z'; для пустого массива условие TRUE)
```



Лекция №2

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Типы данных. Язык SQL. Операторы CREATE и INSERT. Ограничения CHECK, NOT NULL, UNIQUE. Первичный и внешний ключи. Обеспечение ссылочной целостности»

Основы синтаксиса SQL в PostgreSQL – основные типы данных. ENUM (перечисления) – результаты выполнения SQL-запроса

ENUM — это «перечень допустимых значений» для колонки. Вы заранее перечисляете варианты (строки), и база гарантирует, что в поле окажется только один из них. Идея: меньше ошибок в данных, удобная сортировка «как задумано», лаконичный код.

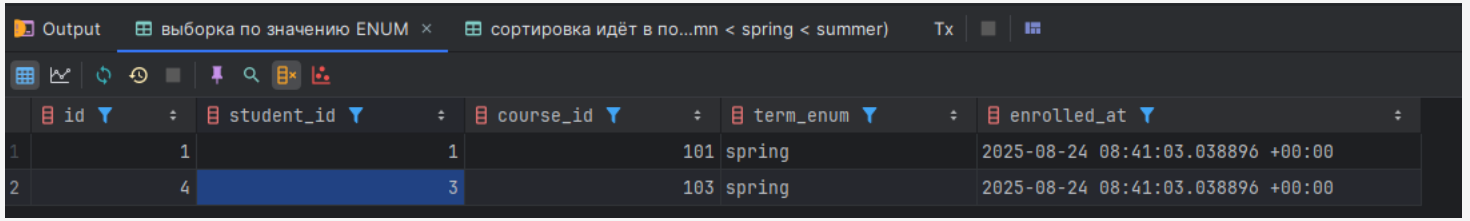
```
-- 1) Чистый старт (без ошибок, если уже было)
DROP TABLE IF EXISTS enrollments;
DROP TYPE  IF EXISTS term; -- term - это имя вашего собственного типа данных (ENUM)

-- 2) ENUM-тип со списком допустимых значений
CREATE TYPE term AS ENUM ('autumn', 'spring', 'summer');

-- 3) Таблица с использованием ENUM
CREATE TABLE enrollments (
  id          bigserial PRIMARY KEY,
  student_id  integer     NOT NULL,
  course_id   integer     NOT NULL,
  term_enum   term        NOT NULL DEFAULT 'autumn',
  enrolled_at  timestamptz NOT NULL DEFAULT now(),
  -- чтобы один студент не записывался на один и тот же курс в один и тот же срок дважды
  UNIQUE (student_id, course_id, term_enum)
);

-- 4) Индексы (частые фильтры/сортировки)
CREATE INDEX idx_enrollments_term        ON enrollments(term_enum);
CREATE INDEX idx_enrollments_course_term  ON enrollments(course_id, term_enum);

-- 5) Демо-данные
INSERT INTO enrollments (student_id, course_id, term_enum) VALUES
  (1, 101, 'spring'),
  (1, 102, 'autumn'),
  (2, 101, 'summer'),
  (3, 103, 'spring');

-- 6) Примеры запросов

-- выборка по значению ENUM
SELECT * FROM enrollments
WHERE term_enum = 'spring';

-- сортировка идёт в порядке объявления ENUM (autumn < spring < summer)
SELECT student_id, course_id, term_enum
FROM enrollments
ORDER BY term_enum, student_id;

-- 7) (опционально) Добавить новое значение и указать место в порядке
-- ALTER TYPE term ADD VALUE IF NOT EXISTS 'winter' BEFORE 'spring';
```

Лекция №2

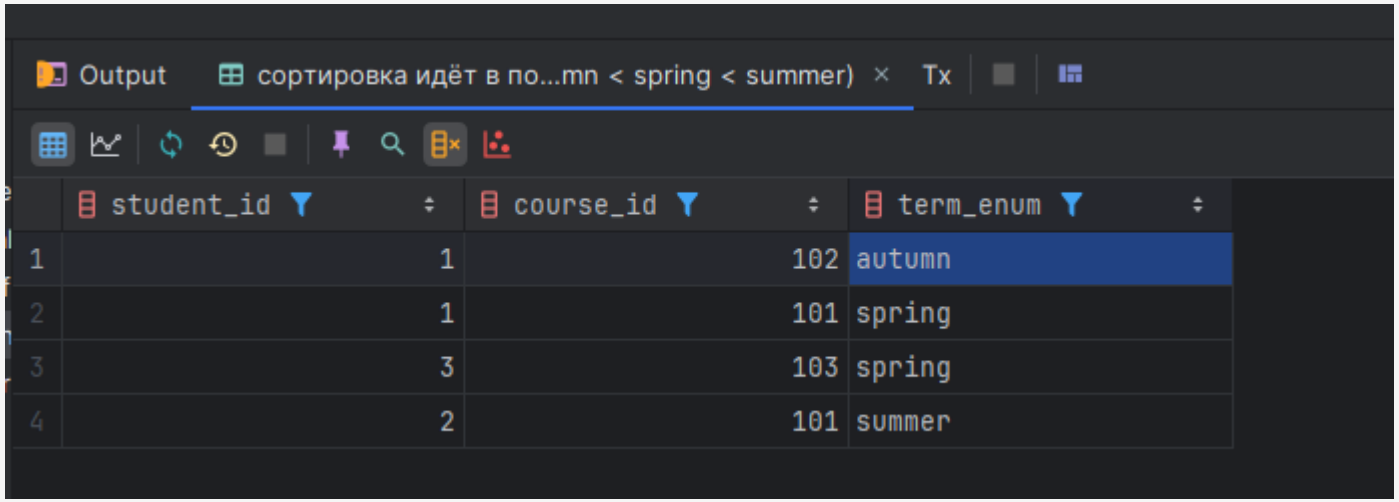
Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Типы данных. Язык SQL. Операторы CREATE и INSERT. Ограничения CHECK, NOT NULL, UNIQUE. Первичный и внешний ключи. Обеспечение ссылочной целостности»

Основы синтаксиса SQL в PostgreSQL – основные типы данных. ENUM (перечисления) – результаты выполнения SQL-запроса



	id	student_id	course_id	term_enum	enrolled_at
1	1	1	101	spring	2025-08-24 08:41:03.038896 +00:00
2	4	3	103	spring	2025-08-24 08:41:03.038896 +00:00



	student_id	course_id	term_enum
1	1	102	autumn
2	1	101	spring
3	3	103	spring
4	2	101	summer

Лекция №2

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Типы данных. Язык SQL. Операторы CREATE и INSERT. Ограничения CHECK, NOT NULL, UNIQUE. Первичный и внешний ключи. Обеспечение ссылочной целостности»

Основы синтаксиса SQL в PostgreSQL – основные типы данных. BYTEA

BYTEA – это «мешок байтов». В колонку этого типа можно класть любые небольшие бинарные данные: иконку PNG, подпись в виде изображения, PDF-фрагмент, бинарный ключ и т. п. Это **не текст**, а «сырые» байты.
Правило: для мелочи (иконки, превью, подписи ≤100–500 КБ) — BYTEA ок. Для больших файлов (фото в полном размере, видео, документы на мегабайты) — храним файл в объектном хранилище/файловой системе, а в БД — только ссылку + метаданные.

Почему так удобно?

1. Значение хранится вместе со строкой — простые транзакции и бэкапы.
2. TOAST в PostgreSQL сам сжимает и «выносит» большие BYTEA за пределы основной страницы, так что мелкие кусочки работают быстро.
3. В драйверах (JDBC, psycopg и т. д.) BYTEA — это просто массив байтов/поток.

Ограничения и нюансы:

1. Теоретически поле до ~1 ГБ, но практически держим сотни килобайт, максимум единицы мегабайт на значение — иначе страдает память/сеть/бэкапы.
2. Индексировать сами байты почти никогда не надо. Если нужно искать дубликаты, храните хеш (например, SHA-256) и индексируйте его.
3. Текстовые «представления» бинарника:

3.1. decode('...','base64' | 'hex') — превратить строку в байты при вставке.
3.2. encode(bytea,'base64' | 'hex') — получить строку для вывода/экспорта.

Лекция №2

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Типы данных. Язык SQL. Операторы CREATE и INSERT. Ограничения CHECK, NOT NULL, UNIQUE. Первичный и внешний ключи. Обеспечение ссылочной целостности»

Основы синтаксиса SQL в PostgreSQL – основные типы данных. BYTEA – пример SQL-запроса

```
-- 0) Расширение для digest() (хешей), по желанию
CREATE EXTENSION IF NOT EXISTS pgcrypto;

-- 1) Минимальная таблица студентов (для примера)
DROP TABLE IF EXISTS student1s CASCADE;
CREATE TABLE student1s (
  student1_id   INT PRIMARY KEY,
  full_name     TEXT NOT NULL
);

INSERT INTO student1s(student1_id, full_name) VALUES
  (1, 'Иван Иванов'),
  (2, 'Мария Петрова');

-- 2) Таблица с аватарками (иконками)
DROP TABLE IF EXISTS avatars;
CREATE TABLE avatars (
  student1_id   INT PRIMARY KEY
           REFERENCES student1s(student1_id) ON DELETE CASCADE,
  mime        TEXT NOT NULL CHECK (mime IN ('image/png','image/jpeg','image/webp')),
  filename     TEXT NOT NULL,
  img         BYTEA NOT NULL,
  byte_size    INT  NOT NULL CHECK (byte_size BETWEEN 1 AND 200*1024),  -- лимит: 200 КБ
  sha256       BYTEA NOT NULL,                                  -- хеш для контроля/дедупликации
  created_at   TIMESTAMPTZ NOT NULL DEFAULT now(),
  UNIQUE (sha256)                                               -- предотвращаем дубликаты
);
```

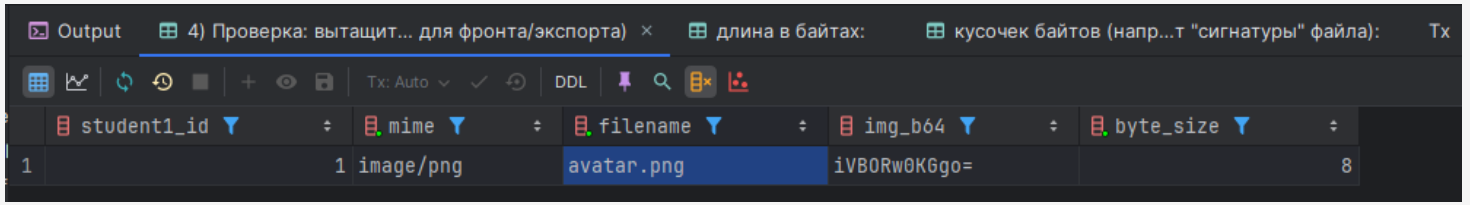
```
-- 3) Вставка аватарки (пример на base64)
--   Представим, что у нас есть короткая PNG-строка в base64 (для учебных целей).
--   На практике клиент/приложение даст длинную base64-строку.
WITH img_base64 AS (
  SELECT 'iVBORw0KGgo='::text AS b64         -- это просто заглушка/обрезок для примера
)
INSERT INTO avatars(student1_id, mime, filename, img, byte_size, sha256)
SELECT
  1,
  'image/png',
  'avatar.png',
  decode(b64, 'base64')                          AS img,
  length(decode(b64, 'base64'))                  AS byte_size,
  digest(decode(b64, 'base64'), 'sha256')        AS sha256
FROM img_base64;

-- 4) Проверка: вытащить картинку в base64 (удобно для фронта/экспорта)
SELECT student1_id,
     mime,
     filename,
     encode(img, 'base64') AS img_b64,
     byte_size
FROM avatars
WHERE student1_id = 1;

-- 5) Полезные функции для bytea
-- длина в байтах:
SELECT octet_length(img) AS bytes FROM avatars WHERE student1_id = 1;

-- кусочек байтов (например, первые 8 байт "сигнатуры" файла):
SELECT encode(substring(img FROM 1 FOR 8), 'hex') AS magic
FROM avatars
WHERE student1_id = 1;
```

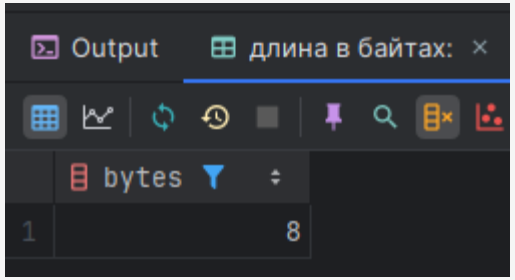
Лекция №2

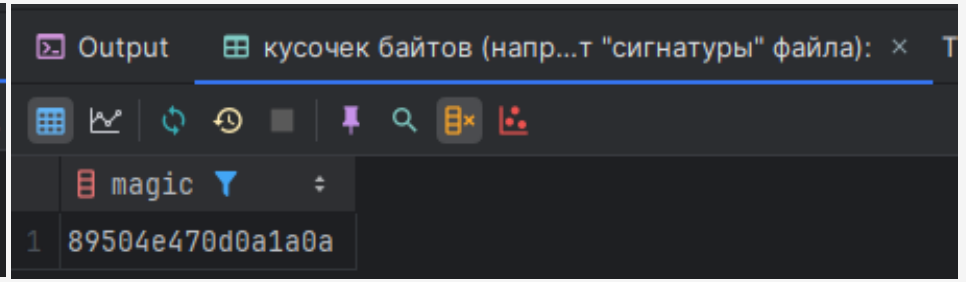
Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Типы данных. Язык SQL. Операторы CREATE и INSERT. Ограничения CHECK, NOT NULL, UNIQUE. Первичный и внешний ключи. Обеспечение ссылочной целостности»

Основы синтаксиса SQL в PostgreSQL – основные типы данных. BYTEA – результаты выполнения SQL-запроса



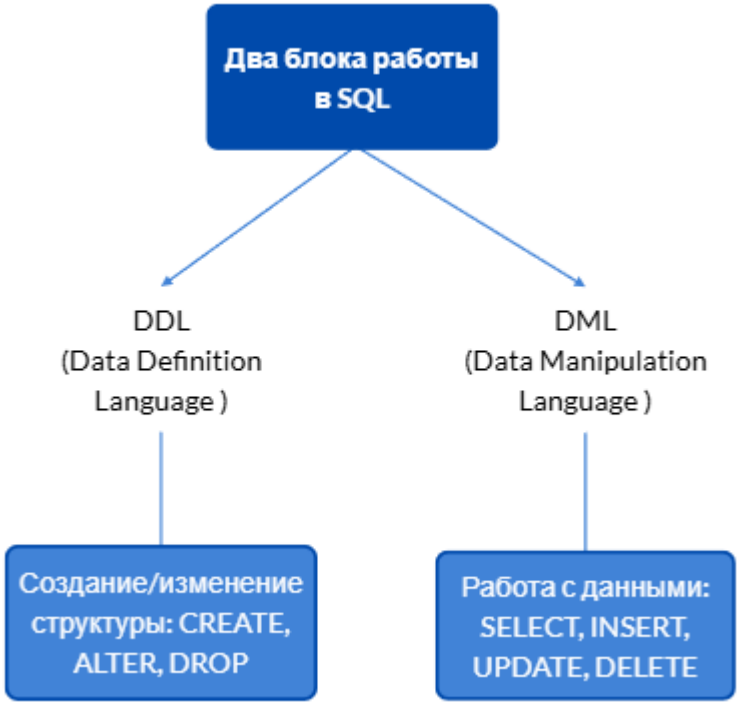




Лекция №3

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Оператор ALTER TABLE. Оператор SELECT. Предложения оператора SELECT. Оператор LIKE. Регулярные выражения POSIX. SQL для работы со строками. Соединение таблиц»



Лекция №3

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Оператор ALTER TABLE. Оператор SELECT. Предложения оператора SELECT. Оператор LIKE. Регулярные выражения POSIX. SQL для работы со строками. Соединение таблиц»

Основы синтаксиса SQL в PostgreSQL – Операторы ALTER и SELECT. Типы соединений (JOIN) – Создание таблиц и внесение данных

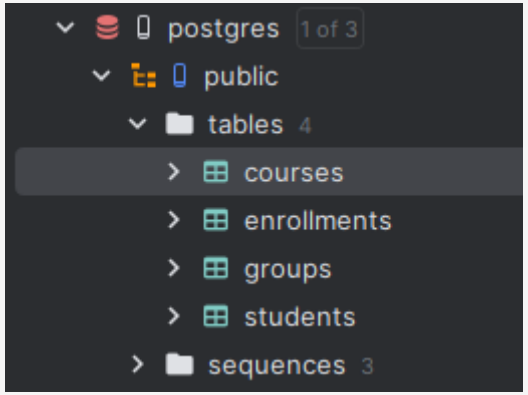
```
-- Группы студентов
CREATE TABLE groups (
group_id SERIAL PRIMARY KEY,
group_name TEXT NOT NULL UNIQUE
);

-- Студенты
CREATE TABLE students (
student_id SERIAL PRIMARY KEY,
full_name TEXT NOT NULL,
email TEXT,
group_id INT REFERENCES groups(group_id)
);

-- Курсы
CREATE TABLE courses (
course_id SERIAL PRIMARY KEY,
title TEXT NOT NULL UNIQUE
);

-- Записи на курсы (многие-ко-многим)
CREATE TABLE enrollments (
student_id INT REFERENCES students(student_id),
course_id INT REFERENCES courses(course_id),
grade NUMERIC(3,1), -- оценка, например 82.5
PRIMARY KEY (student_id, course_id)
);

-- Немного данных (пример)
INSERT INTO groups (group_name) VALUES ('IKBO-01-23'), ('IKBO-02-23');
INSERT INTO students (full_name, email, group_id) VALUES
('Иван Иванов', 'ivan@mail.com', 1),
('Анна Смирнова','anna@uni.ru', 1),
('Пётр Петров', 'p.petrov@uni.ru', 2);
INSERT INTO courses (title) VALUES ('SQL basics'), ('Python intro');
INSERT INTO enrollments (student_id, course_id, grade) VALUES
(1, 1, 85.0), (1, 2, 90.0), (2, 1, 78.5);
```

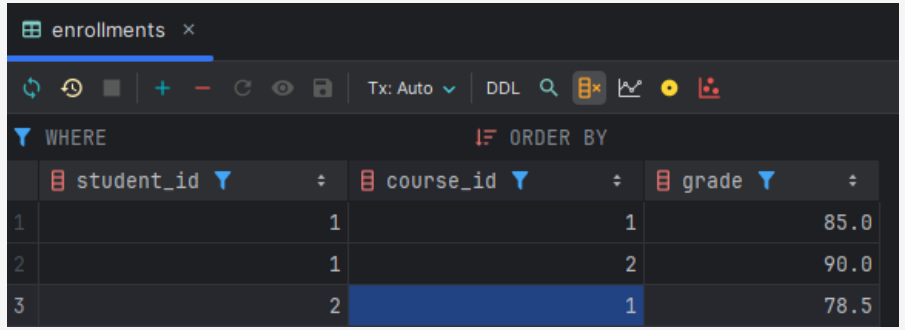
Лекция №3

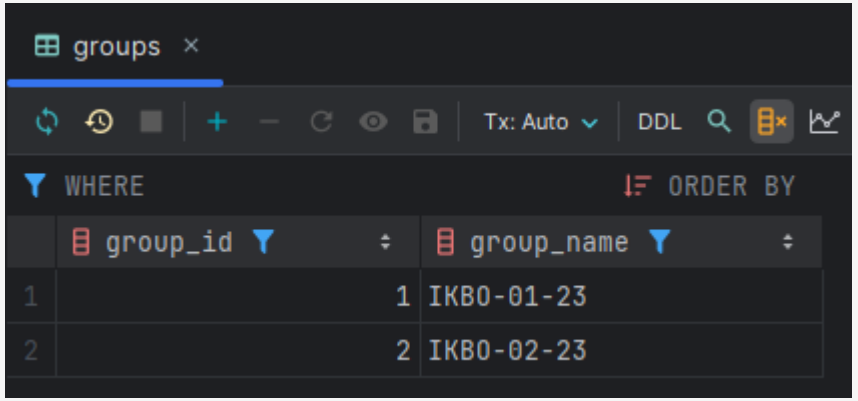
Тема: «Основы синтаксиса SQL в PostgreSQL»

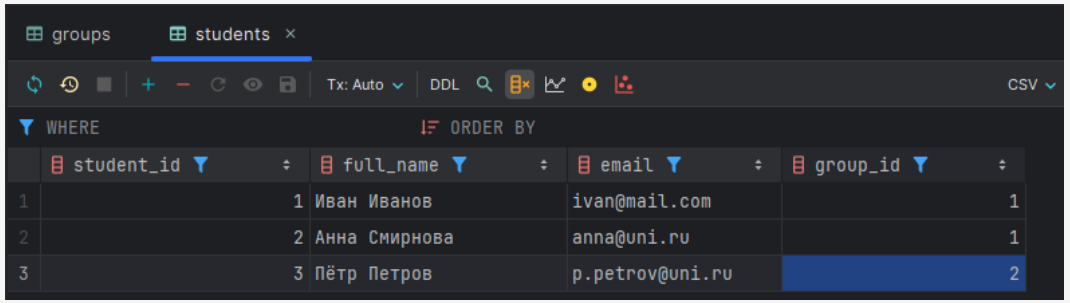
Подтема: «Оператор ALTER TABLE. Оператор SELECT. Предложения оператора SELECT. Оператор LIKE. Регулярные выражения POSIX. SQL для работы со строками. Соединение таблиц»

Основы синтаксиса SQL в PostgreSQL – Операторы ALTER и SELECT. Типы соединений (JOIN) – Создание таблиц и внесение данных. Результат









Лекция №3

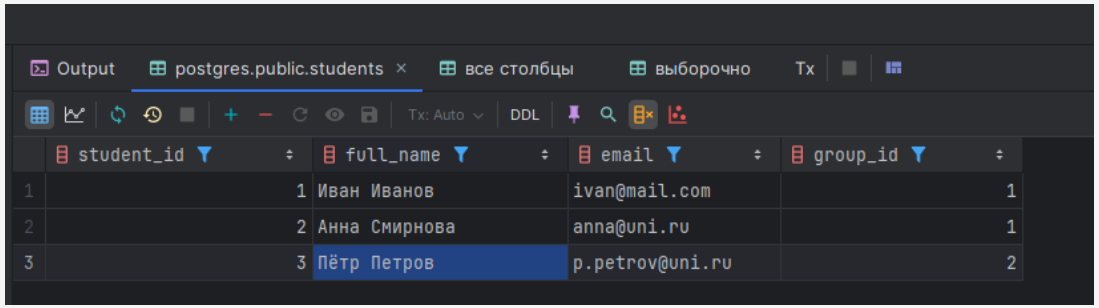
Тема: «Основы синтаксиса SQL в PostgreSQL»

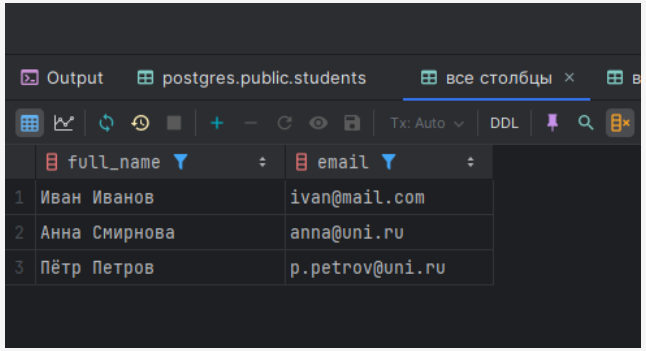
Подтема: «Оператор ALTER TABLE. Оператор SELECT. Предложения оператора SELECT. Оператор LIKE. Регулярные выражения POSIX. SQL для работы со строками. Соединение таблиц»

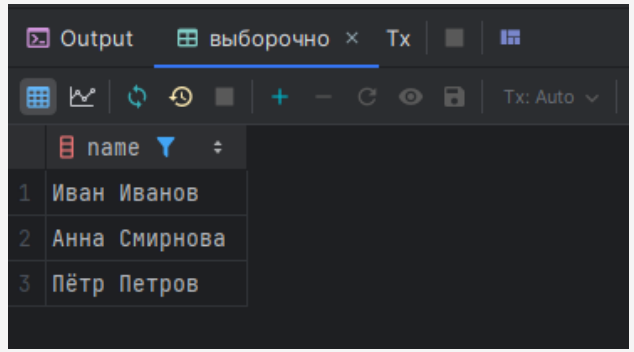
Основы синтаксиса SQL в PostgreSQL – Операторы ALTER и SELECT. Типы соединений (JOIN). SELECT

SELECT — основной запрос для чтения (вывода) данных из БД.

```
SELECT * FROM students; -- все столбцы
SELECT full_name, email FROM students; -- выборочно
SELECT full_name AS name FROM students;-- псевдоним столбца
```







Лекция №3

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Оператор ALTER TABLE. Оператор SELECT. Предложения оператора SELECT. Оператор LIKE. Регулярные выражения POSIX. SQL для работы со строками. Соединение таблиц»

Полезные предложения SELECT (по порядку написания с точки зрения синтаксиса):

1. SELECT — какие столбцы/выражения вернуть.
2. FROM — из каких таблиц (и их соединения).
3. WHERE — фильтр строк до группировки.
4. GROUP BY — группировка.
5. HAVING — фильтр групп.
6. ORDER BY — сортировка.
7. LIMIT [OFFSET] — ограничение и смещение.

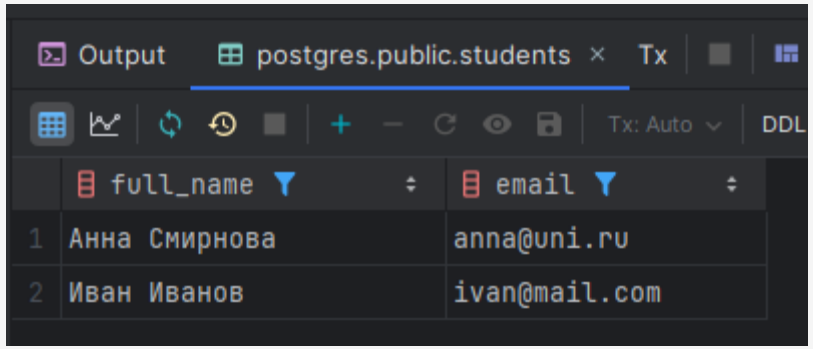
Внутренний (внутри PostgreSQL) порядок выполнения другой: FROM → WHERE → GROUP BY → HAVING → SELECT → ORDER BY → LIMIT.

```
SELECT full_name, email
FROM students
ORDER BY full_name ASC
LIMIT 2 OFFSET 0; -- первые 2
```

Лекция №3

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Оператор ALTER TABLE. Оператор SELECT. Предложения оператора SELECT. Оператор LIKE. Регулярные выражения POSIX. SQL для работы со строками. Соединение таблиц»



Лекция №3

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Оператор ALTER TABLE. Оператор SELECT. Предложения оператора SELECT. Оператор LIKE. Регулярные выражения POSIX. SQL для работы со строками. Соединение таблиц»

WHERE – оператор фильтрации строк

Операторы сравнения: =, <> (или !=), <, >, <=, >=
Логика (логические операторы): AND, OR, NOT

Другие полезные операторы:
BETWEEN a AND b — включительно
IN (список) — принадлежит множеству
IS NULL / IS NOT NULL — проверка на NULL

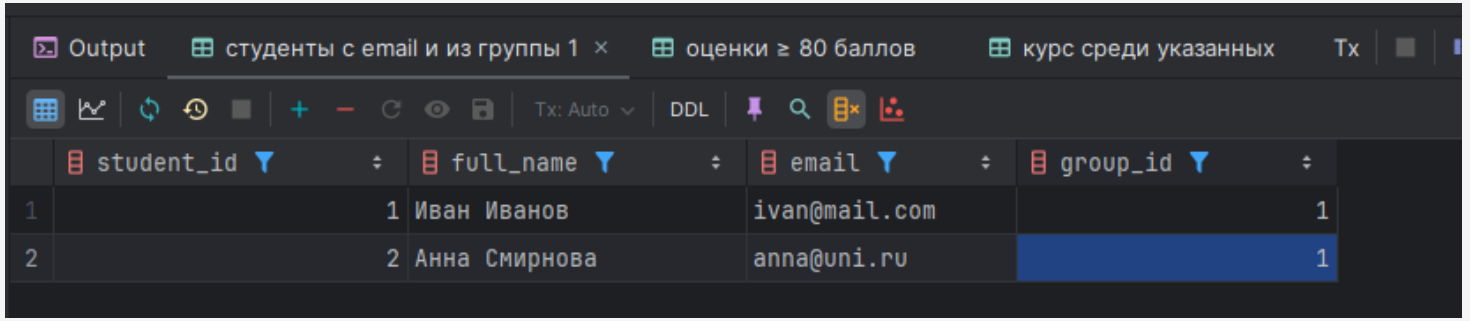
```sql
-- студенты с email и из группы 1
SELECT *
FROM students
WHERE email IS NOT NULL AND group_id = 1;


-- оценки ≥ 80 баллов
SELECT *
FROM enrollments
WHERE grade >= 80;


-- курс среди указанных
SELECT * FROM courses WHERE title IN ('SQL basics', 'Python intro');
```

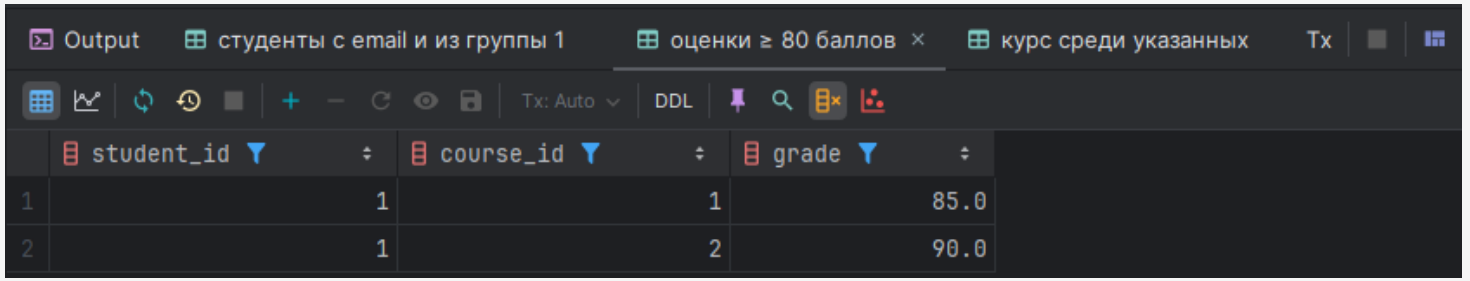
Лекция №3

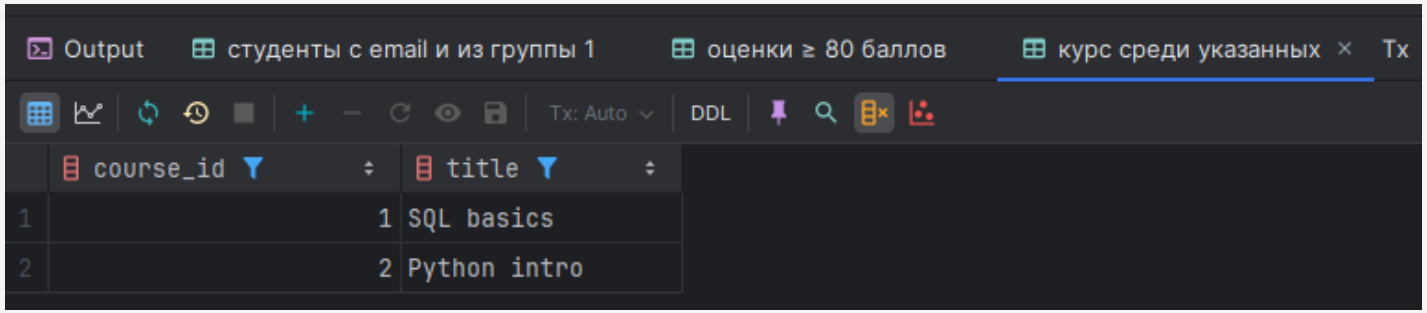
Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Оператор ALTER TABLE. Оператор SELECT. Предложения оператора SELECT. Оператор LIKE. Регулярные выражения POSIX. SQL для работы со строками. Соединение таблиц»

Основы синтаксиса SQL в PostgreSQL – Операторы ALTER и SELECT. Типы соединений (JOIN). WHERE – результаты







Лекция №3

Тема: «Основы синтаксиса SQL в PostgreSQL»

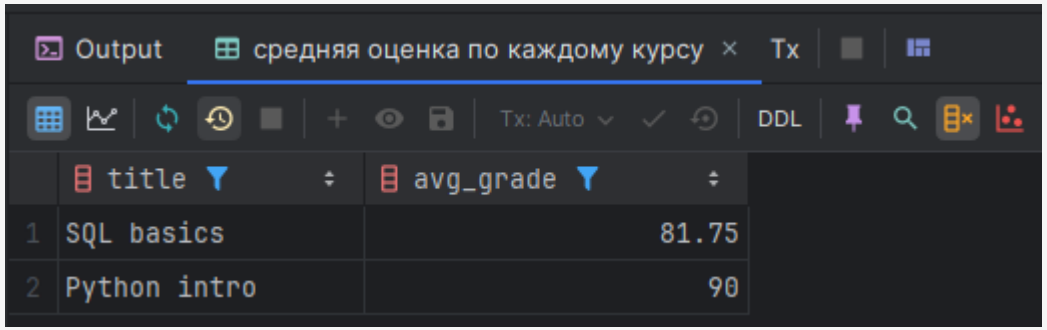
Подтема: «Оператор ALTER TABLE. Оператор SELECT. Предложения оператора SELECT. Оператор LIKE. Регулярные выражения POSIX. SQL для работы со строками. Соединение таблиц»

Основы синтаксиса SQL в PostgreSQL. GROUP BY / HAVING – сортировка (группирование), синтаксис и примеры

Для группировки данных в PostgreSQL применяются операторы **GROUP BY** и **HAVING**, для использования которых применяется следующий формальный синтаксис:

1. **SELECT** столбцы
2. **FROM** таблица
3. [**WHERE** условие_фильтрации_строк]
4. [**GROUP BY** столбцы_для_группировки]
5. [**HAVING** условие_фильтрации_групп]
6. [**ORDER BY** столбцы_для_сортировки]

```
-- средняя оценка по каждому курсу
SELECT c.title, AVG(e.grade) AS avg_grade
FROM enrollments e
JOIN courses c ON c.course_id = e.course_id
GROUP BY c.title
HAVING AVG(e.grade) >= 80; -- фильтр уже по группам (у кого более 80)
```



Помним: **WHERE** фильтрует строки до группировки, **HAVING** — группы после неё.

Лекция №3

Тема: «Основы синтаксиса SQL в PostgreSQL»

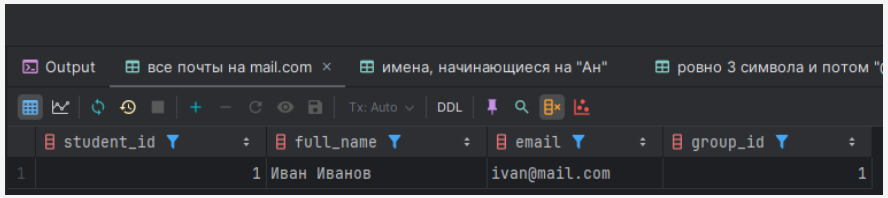
Подтема: «Оператор ALTER TABLE. Оператор SELECT. Предложения оператора SELECT. Оператор LIKE. Регулярные выражения POSIX. SQL для работы со строками. Соединение таблиц»

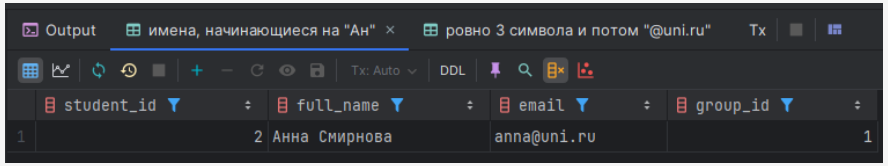
Основы синтаксиса SQL в PostgreSQL. LIKE/ILIKE – операторы, позволяющие возвращать текст по совпадению

Оператор **LIKE** чувствителен к регистру, то есть различает прописные и строчные буквы. Например, шаблон «A%» не соответствует строке «apple», но соответствует строке «Apple».

Оператор **ILIKE** не чувствителен к регистру. Это означает, что не имеет значения, имеют ли строка и шаблон одинаковый регистр или нет. Например, шаблон «A%» совпадает как с «apple», так и с «Apple» при использовании оператора ILIKE.

```
-- все почты на mail.com
SELECT * FROM students WHERE email LIKE '%@mail.com';

-- имена, начинающиеся на "Ан"
SELECT * FROM students WHERE full_name LIKE 'Ан%';

-- ровно 3 символа и потом "@uni.ru", результата не будет, так как нет студентов с такими e-mail
SELECT * FROM students WHERE email LIKE '___@uni.ru';
```





Лекция №3

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Оператор ALTER TABLE. Оператор SELECT. Предложения оператора SELECT. Оператор LIKE. Регулярные выражения POSIX. SQL для работы со строками. Соединение таблиц»

Основы синтаксиса SQL в PostgreSQL. POSIX-регулярные выражения в SQL

Операторы:

~ — соответствует (регистрозависимо);
~* — соответствует (без регистра);
!~ — не соответствует (регистрозависимо);
!~* — не соответствует (без регистра).

Мини-шпаргалка по шаблонам:

^ — начало строки, **$** — конец строки;
. — любой символ ;
[abc] — любой из перечисленных;
[^abc] — любой, кроме перечисленных;
[0-9] — цифра (или [:digit:]);
+ — один или больше, * ноль или больше, ? ноль или один;
{m,n} — от m до n повторов.

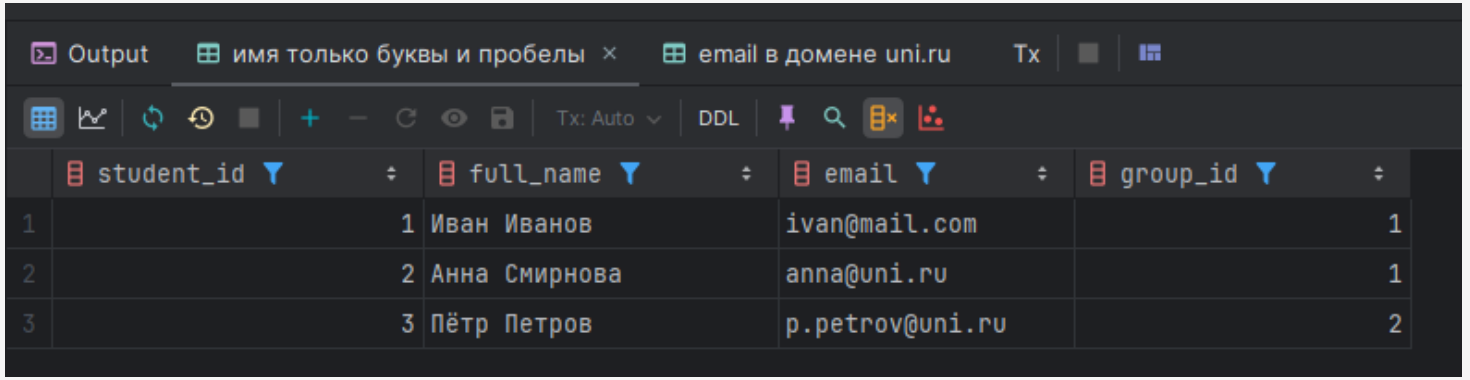
```
-- email в домене uni.ru
SELECT * FROM students
WHERE email ~* '@uni\.ru$'; -- \. — точка буквально


-- имя только буквы и пробелы
SELECT * FROM students
WHERE full_name ~* '^[a-za-яё ]+$';
```

Лекция №3

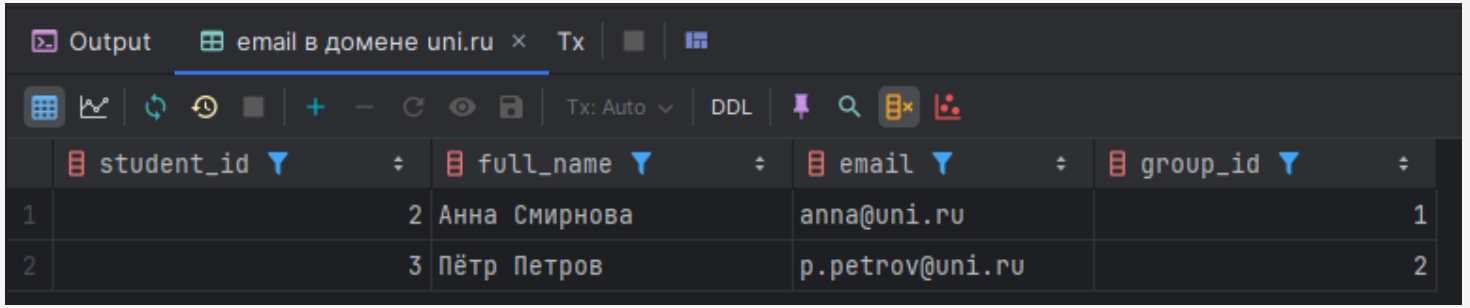
Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Оператор ALTER TABLE. Оператор SELECT. Предложения оператора SELECT. Оператор LIKE. Регулярные выражения POSIX. SQL для работы со строками. Соединение таблиц»

Основы синтаксиса SQL в PostgreSQL. POSIX-регулярные выражения в SQL – результаты выполнения SQL-запроса



	student_id	full_name	email	group_id
1	1	Иван Иванов	ivan@mail.com	1
2	2	Анна Смирнова	anna@uni.ru	1
3	3	Пётр Петров	p.petrov@uni.ru	2



	student_id	full_name	email	group_id
1	2	Анна Смирнова	anna@uni.ru	1
2	3	Пётр Петров	p.petrov@uni.ru	2

Лекция №3

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Оператор ALTER TABLE. Оператор SELECT. Предложения оператора SELECT. Оператор LIKE. Регулярные выражения POSIX. SQL для работы со строками. Соединение таблиц»

Основы синтаксиса SQL в PostgreSQL. Строковые функции – LENGTH, LOWER, TRIM, LTRIM, RTRIM, POSITION, SUBSTRING, REPLACE, LAPD, RPAD

Функция	Что делает (коротко)	Зачем используют	Пример (SQL → результат)
LENGTH(text)	Длина строки **в символах**	Проверка пустоты, валидация, формат	SELECT LENGTH('Привет'); → 6
LOWER(text)	Переводит в **нижний регистр**	Сравнения без учёта регистра, нормализация e-mail	SELECT LOWER('Ivan@Uni.Ru'); → 'ivan@uni.ru'
`TRIM([BOTH	LEADING	TRAILING] [chars] FROM text)/TRIM(text)`	Убирает пробелы по краям (или указанные символы)
LTRIM(text [, chars])	Убирает **слева** пробелы/символы	Снять лишние префиксы/паддинг	SELECT LTRIM('---id','-'); → 'id'
RTRIM(text [, chars])	Убирает **справа** пробелы/символы	Снять «хвосты»	SELECT RTRIM('id###','#'); → 'id'
POSITION(sub IN str)	Номер первого вхождения (с **1**; если нет — **0**)	Найти индекс символа/фрагмента	SELECT POSITION('@' IN 'a@b'); → 2
SUBSTRING(str FROM start FOR count) / SUBSTRING(str FROM 'regex')	Достаёт подстроку по позиции/длине или по **регулярке**	Вырезать часть строки (домен, код и т.п.)	SELECT SUBSTRING('abcdef' FROM 2 FOR 3); → 'bcd' SELECT SUBSTRING('a@uni.ru' FROM '@(.+)$'); → 'uni.ru'
REPLACE(str, from, to)	Заменяет **все** вхождения	Массовые правки текста	SELECT REPLACE('a_b_c','_','-'); → 'a-b-c'
LPAD(str, length [, fill])	Дополняет **слева** до length (если длина больше — **обрежет**)	Форматировать коды/номера, выравнивание	SELECT LPAD('42',5,'0'); → '00042' SELECT LPAD('abcdef',3,'*'); → 'abc'
RPAD(str, length [, fill])	Дополняет **справа** до length (если длина больше — **обрежет**)	Колонковое выравнивание, шаблоны	SELECT RPAD('OK',4,'!'); → 'OK!!' SELECT RPAD('abcdef',3,'.'); → 'abc'

Лекция №3

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Оператор ALTER TABLE. Оператор SELECT. Предложения оператора SELECT. Оператор LIKE. Регулярные выражения POSIX. SQL для работы со строками. Соединение таблиц»

Основы синтаксиса SQL в PostgreSQL. Строковые функции – LENGTH, LOWER, TRIM, LTRIM, RTRIM, POSITION, SUBSTRING, REPLACE, LAPD, RPAD – пример запроса

```sql
-- длина строки
SELECT LENGTH(full_name) FROM students;

-- регистр
SELECT LOWER(email), UPPER(full_name) FROM students;

-- обрезка пробелов
SELECT TRIM(' hello '); -- 'hello'
SELECT LTRIM(' hello'); -- 'hello'
SELECT RTRIM('hello '); -- 'hello'

-- позиция и подстрока
SELECT POSITION('@' IN email) FROM students; -- индекс '@'
SELECT SUBSTRING(email FROM '@(.+)$') -- домен по regex
FROM students;

-- замена и конкатенация
SELECT REPLACE(full_name, 'Иван', 'Иоанн') FROM students;
SELECT full_name || ' <' || COALESCE(email,'—') || '>' AS contact
FROM students;

-- дополнение слева/справа
SELECT LPAD('42', 5, '0'); -- '00042'
SELECT RPAD('OK', 4, '!'); -- 'OK!!'
```

4 и **5** - это целевые длины результата (в символах), а не "сколько добавить".

LPAD('42', 5, '0') → сделать итоговую строку длиной 5. Было 2 символа ('42'), не хватает 3 — добавили слева три '0' → '00042'.

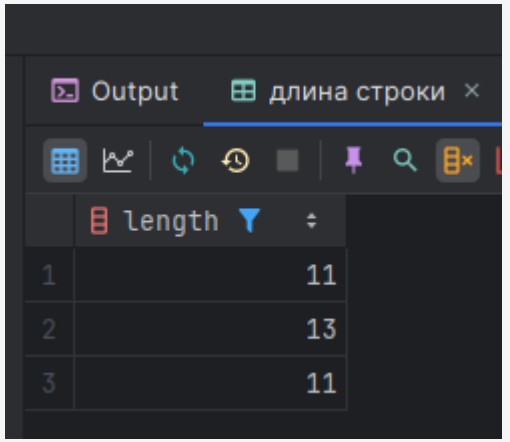
RPAD('OK', 4, '!') → сделать итоговую строку длиной 4. Было 2 символа ('OK'), не хватает 2 — добавили справа два '!' → 'OK!!'.

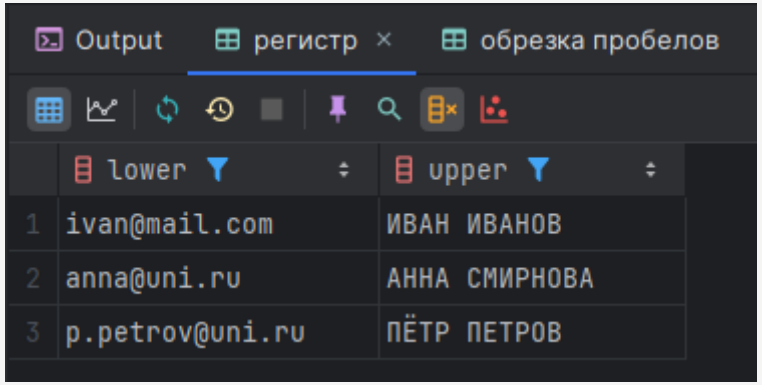
Лекция №3

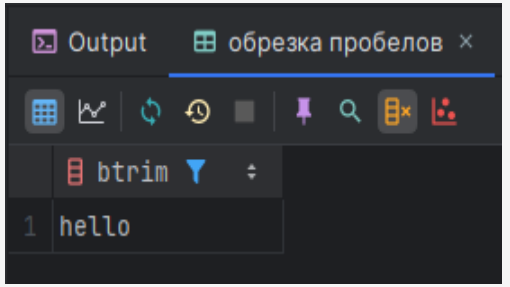
Тема: «Основы синтаксиса SQL в PostgreSQL»

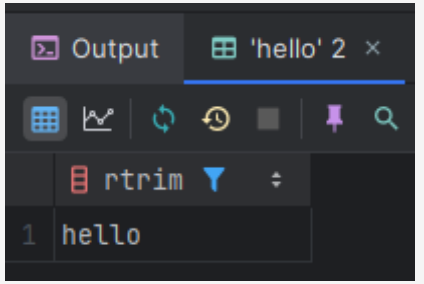
Подтема: «Оператор ALTER TABLE. Оператор SELECT. Предложения оператора SELECT. Оператор LIKE. Регулярные выражения POSIX. SQL для работы со строками. Соединение таблиц»

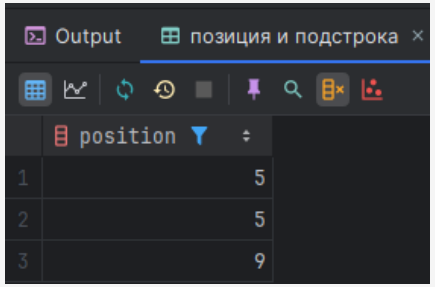
Основы синтаксиса SQL в PostgreSQL. Строковые функции – LENGTH, LOWER, TRIM, LTRIM, RTRIM, POSITION, SUBSTRING, REPLACE, LAPD, RPAD – результаты SQL-запроса

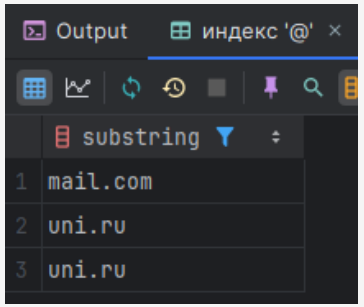


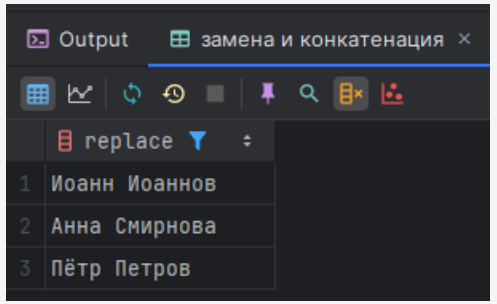


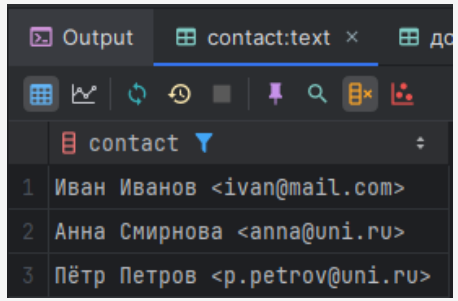












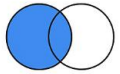
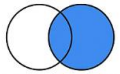
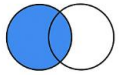
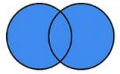
Лекция №3

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Оператор ALTER TABLE. Оператор SELECT. Предложения оператора SELECT. Оператор LIKE. Регулярные выражения POSIX. SQL для работы со строками. Соединение таблиц»

Основы синтаксиса SQL в PostgreSQL. Соединение таблиц: INNER JOIN, LEFT JOIN, RIGHT JOIN, FULL OUTER JOIN, CROSS JOIN

Тип JOIN	Что выбираем	Пример ситуации
INNER JOIN	Берём **только те строки**, которые совпадают в обеих таблицах. (Пересечение)	Список студентов и их группы — будут только те студенты, у которых есть группа.
LEFT JOIN	Берём **все строки из левой таблицы** + совпадения из правой. Где нет пары — ставится **NULL**.	Все студенты + их группы, даже если у кого-то группа не указана (будет NULL).
RIGHT JOIN	Берём **все строки из правой таблицы** + совпадения из левой. Где нет пары — **NULL**.	Все группы + студенты, даже если в группе пока никого нет (студент = NULL).
FULL OUTER JOIN	Берём **все строки из обеих таблиц**. Где нет пары — **NULL**.	Все студенты и все группы, даже если кто-то не связан.
CROSS JOIN	Делаем **каждый с каждым** (декартово произведение).	Если 3 студента и 2 группы → получится 6 комбинаций (каждый студент со всеми группами).

 INNER JOIN	Берём только те строки, которые совпадают в обеих таблицах. (Пересечение)
 LEFT JOIN	Берём все строки из левой таблицы + совпадения из правой. Где нет пары — ставится NULL
 FULL OUTER JOIN	Берём все строки из правой таблицы + совпадения из левой. Где нет пары — NULL
 CROSS JOIN	Берём все строки из обеих таблиц. Где нет пары — NULL
	Делаем каждый с каждым (декартово произведение)

Лекция №3

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Оператор ALTER TABLE. Оператор SELECT. Предложения оператора SELECT. Оператор LIKE. Регулярные выражения POSIX. SQL для работы со строками. Соединение таблиц»

Основы синтаксиса SQL в PostgreSQL. Соединение таблиц: INNER JOIN, LEFT JOIN, RIGHT JOIN, FULL OUTER JOIN, CROSS JOIN – пример SQL-запроса

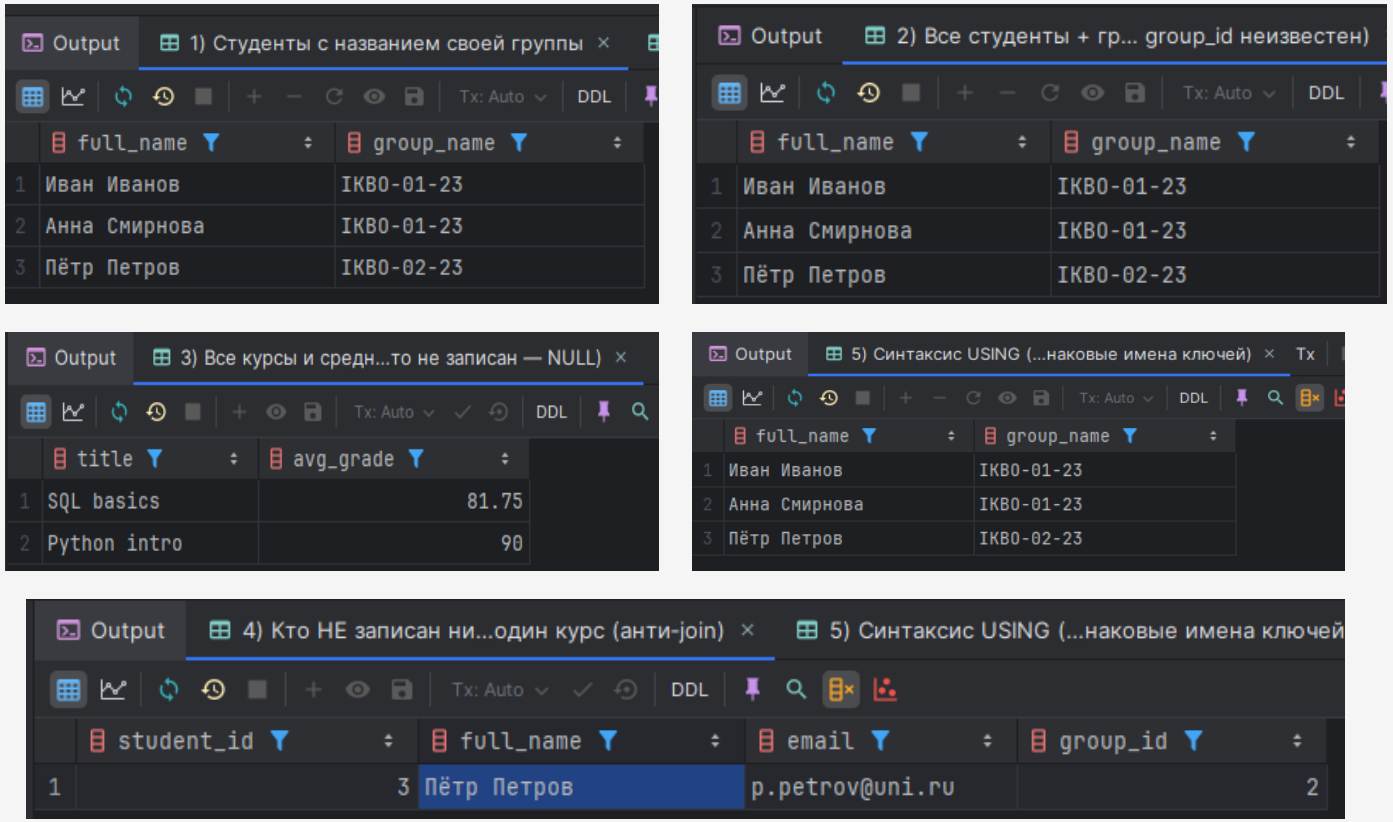
```sql
-- 1) Студенты с названием своей группы
SELECT s.full_name, g.group_name
FROM students s
JOIN groups g ON g.group_id = s.group_id; -- INNER JOIN


-- 2) Все студенты + группы (даже если group_id неизвестен)
SELECT s.full_name, g.group_name
FROM students s
LEFT JOIN groups g ON g.group_id = s.group_id; -- LEFT JOIN


-- 3) Все курсы и средняя оценка по ним (если никто не записан — NULL)
SELECT c.title, AVG(e.grade) AS avg_grade
FROM courses c
LEFT JOIN enrollments e ON e.course_id = c.course_id
GROUP BY c.title;


-- 4) Кто НЕ записан ни на один курс (анти-join)
SELECT s.*
FROM students s
LEFT JOIN enrollments e ON e.student_id = s.student_id
WHERE e.student_id IS NULL; -- нет пары


-- 5) Синтаксис USING (если одинаковые имена ключей)
SELECT s.full_name, g.group_name
FROM students s
JOIN groups g USING (group_id);
```

Лекция №3

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Оператор ALTER TABLE. Оператор SELECT. Предложения оператора SELECT. Оператор LIKE. Регулярные выражения POSIX. SQL для работы со строками. Соединение таблиц»

Основы синтаксиса SQL в PostgreSQL. Соединение таблиц: INNER JOIN, LEFT JOIN, RIGHT JOIN, FULL OUTER JOIN, CROSS JOIN – результаты выполнения SQL-запроса



Лекция №3

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Оператор ALTER TABLE. Оператор SELECT. Предложения оператора SELECT. Оператор LIKE. Регулярные выражения POSIX. SQL для работы со строками. Соединение таблиц»

Основы синтаксиса SQL в PostgreSQL. Изменение структуры таблицы (ALTER TABLE)

Название операции	SQL-пример	Пояснение
Добавить столбец	ALTER TABLE students ADD COLUMN phone TEXT;	В таблице появляется новый столбец phone.
Удалить столбец	ALTER TABLE students DROP COLUMN phone;	Удаляем столбец phone.
Переименовать столбец	ALTER TABLE students RENAME COLUMN full_name TO name;	Меняем имя столбца.
Переименовать таблицу	ALTER TABLE students RENAME TO learners;	Меняем имя таблицы.
Изменить тип данных	ALTER TABLE enrollments ALTER COLUMN grade TYPE NUMERIC(4,1);	Меняем тип на число с одной цифрой после запятой.
Задать значение по умолчанию	ALTER TABLE students ALTER COLUMN email SET DEFAULT '';	Если при вставке не указано — будет пустая строка.
Убрать значение по умолчанию	ALTER TABLE students ALTER COLUMN email DROP DEFAULT;	Убираем дефолтное значение.
Запретить NULL (NOT NULL)	ALTER TABLE students ALTER COLUMN email SET NOT NULL;	Теперь email не может быть пустым.
Сделать уникальным (UNIQUE)	ALTER TABLE students ADD CONSTRAINT students_email_uniq UNIQUE(email);	Значения email не должны повторяться.
Проверка значения (CHECK)	ALTER TABLE enrollments ADD CONSTRAINT enrollments_grade_chk CHECK (grade BETWEEN 0 AND 100);	Балл должен быть от 0 до 100.
Добавить внешний ключ (FOREIGN KEY)	ALTER TABLE students ADD CONSTRAINT students_group_fk FOREIGN KEY (group_id) REFERENCES groups(group_id);	group_id в students должен ссылаться на groups.

Лекция №3

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Оператор ALTER TABLE. Оператор SELECT. Предложения оператора SELECT. Оператор LIKE. Регулярные выражения POSIX. SQL для работы со строками. Соединение таблиц»

Основы синтаксиса SQL в PostgreSQL. Изменение структуры таблицы (ALTER TABLE) – пример SQL-запроса

```
-- добавить/удалить столбец
ALTER TABLE learners ADD COLUMN IF NOT EXISTS phone TEXT;
ALTER TABLE learners DROP COLUMN IF EXISTS phone;

-- переименовать столбец (если ещё не переименовали)
ALTER TABLE enrollments ALTER COLUMN grade TYPE NUMERIC(4,1);

ALTER TABLE learners ALTER COLUMN email SET DEFAULT '';
ALTER TABLE learners ALTER COLUMN email DROP DEFAULT;

-- 1) Запрет NULL для email
ALTER TABLE learners ALTER COLUMN email SET NOT NULL;

-- 2) Уникальность email
ALTER TABLE learners ADD CONSTRAINT learners_email_uniq UNIQUE(email);

-- 3) Внешний ключ на groups(group_id)
ALTER TABLE learners
  ADD CONSTRAINT learners_group_fk
  FOREIGN KEY (group_id) REFERENCES groups(group_id);

-- 4) CHECK на диапазон оценок
ALTER TABLE enrollments
  ADD CONSTRAINT enrollments_grade_chk CHECK (grade BETWEEN 0 AND 100);
```

Лекция №4

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Запросы и подзапросы. Пользовательские типы. Регулярные выражения SIMILAR TO. Агрегатные функции. Подзапросы. ANY. ALL. EXISTS. Группировка данных. Пользовательские типы. CASE. COALESCE и NULLIF»

Основы синтаксиса SQL в PostgreSQL – Запросы и подзапросы. Пользовательские типы. Часть 1. Запросы, соединения, подзапросы, группировка

Элемент (действие)	Пример синтаксиса	Зачем нужно
Выбрать столбцы	SELECT full_name, email	Что показывать в результате.
Из какой таблицы	FROM learners	Откуда брать данные.
Фильтр строк (до группировки)	WHERE grade >= 80	Оставить только нужные строки.
Соединение таблиц	JOIN … ON …	Соединить связанные таблицы по ключам.
Левое соединение	LEFT JOIN … ON …	Все слева + совпадения справа; нет — NULL.
Убрать дубликаты	SELECT DISTINCT course	Только уникальные значения.
Сортировка	ORDER BY avg_grade DESC	Отсортировать вывод.
Порядок NULL	… NULLS LAST	Куда ставить NULL при сортировке.
Группировка	GROUP BY course	Собирать строки по признаку.
Фильтр групп	HAVING AVG(grade) >= 80	Фильтр после подсчётов по группам.
Подзапрос (скалярный)	(SELECT AVG(grade) FROM … WHERE …=l.learner_id)	Вставить одно вычисленное значение.
Подзапрос-список	WHERE learner_id IN (SELECT …)	Сравнить со списком значений.
Подзапрос в FROM	FROM (SELECT …) AS t	Временная «таблица» для следующего шага.
Проверка наличия	EXISTS (SELECT 1 FROM … WHERE …=внешн.)	Узнать, «есть ли такие строки».
Сравнить с «хотя бы одним»	x > ANY(SELECT grade …)	Достаточно совпадения с одним значением.
Сравнить «со всеми»	x > ALL(SELECT grade …)	Должно быть верно для каждого значения.

Лекция №4

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Запросы и подзапросы. Пользовательские типы. Регулярные выражения SIMILAR TO. Агрегатные функции. Подзапросы. ANY. ALL. EXISTS. Группировка данных. Пользовательские типы. CASE. COALESCE и NULLIF»

Элемент (действие)	Пример синтаксиса	Зачем нужно
Принадлежит набору	course IN ('SQL','Math')	Проверить, входит ли значение в список.
Диапазон	grade BETWEEN 60 AND 74	Проверить, попадает ли в отрезок.
Проверка NULL	email IS NULL	Явно проверить «нет значения».
Условие в выводе	CASE WHEN grade>=90 THEN 'отлично' ... END	Превратить условия в метки/категории.
Заменить NULL	COALESCE(email,'нет почты')	Взять первое не-NULL (подставить заглушку).
Сделать NULL, если равны	NULLIF(divider,0)	Избежать деления на ноль и т. п.
Регулярные шаблоны	`SIMILAR TO '(SQL	Math
Подсчёт строк	COUNT(*)	Сколько строк/элементов.
Среднее	AVG(grade)	Среднее значение.
Сумма	SUM(grade)	Сложить значения.
Минимум / максимум	MIN(grade) / MAX(grade)	Наименьшее / наибольшее.
Склейка строк	STRING_AGG(name, ', ')	Объединить значения в одну строку.
Перечисление (ENUM)	CREATE TYPE term AS ENUM (...)	Закрытый список допустимых значений.
Домен (DOMAIN)	CREATE DOMAIN grade_0_100 AS NUMERIC CHECK (VALUE BETWEEN 0 AND 100)	Свой тип + правило проверки.
Составной тип	CREATE TYPE full_name_t AS (last TEXT, first TEXT)	«Структура» из нескольких полей как один тип.

Лекция №4

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Запросы и подзапросы. Пользовательские типы. Регулярные выражения SIMILAR TO. Агрегатные функции. Подзапросы. ANY. ALL. EXISTS. Группировка данных. Пользовательские типы. CASE. COALESCE и NULLIF»

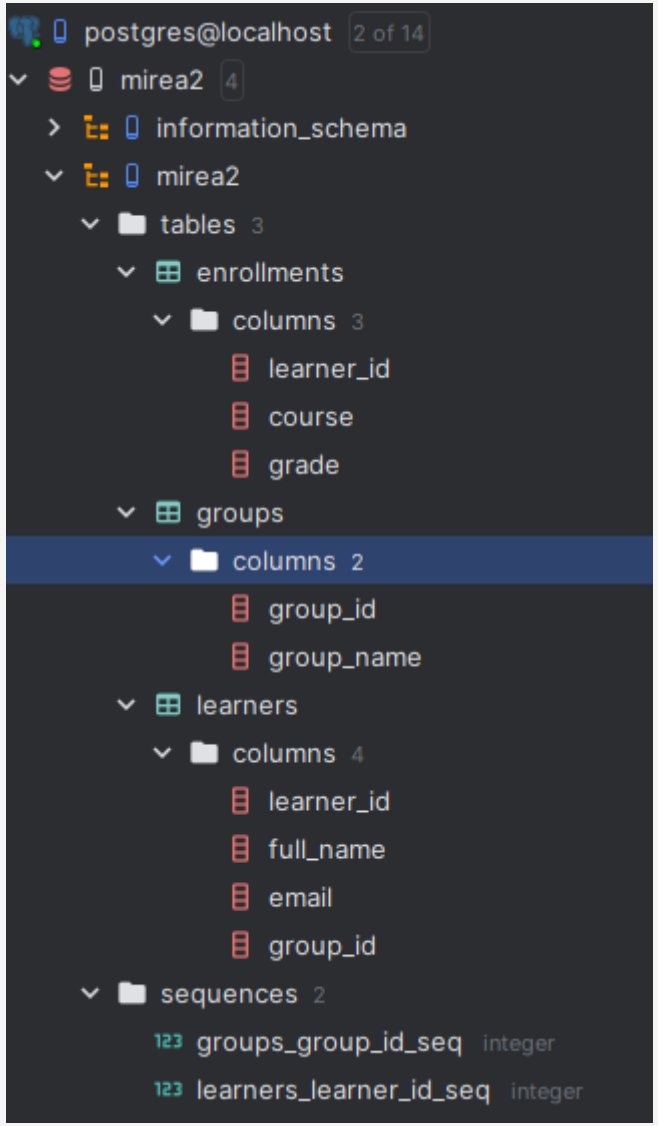
Основы синтаксиса SQL в PostgreSQL – Запросы и подзапросы. Пользовательские типы. Создаем БД и таблицы

```
-- Создаем БД
CREATE SCHEMA IF NOT EXISTS mirea2;

-- чтобы не писать префиксы "mirea." каждый раз
SET search_path = mirea2, public;

-- Группы
CREATE TABLE IF NOT EXISTS groups(
  group_id   SERIAL PRIMARY KEY, -- автоматически создает sequence для каждого такого столбца
  group_name TEXT NOT NULL
);

-- Студенты
CREATE TABLE IF NOT EXISTS learners(
  learner_id SERIAL PRIMARY KEY,
  full_name  TEXT NOT NULL,
  email      TEXT,
  group_id   INT REFERENCES groups(group_id)
);

-- Оценки по курсам
CREATE TABLE IF NOT EXISTS enrollments(
  learner_id INT REFERENCES learners(learner_id),
  course     TEXT NOT NULL,
  grade      NUMERIC(5,2), -- 0..100
  PRIMARY KEY (learner_id, course)
);
```

Лекция №4

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Запросы и подзапросы. Пользовательские типы. Регулярные выражения SIMILAR TO. Агрегатные функции. Подзапросы. ANY. ALL. EXISTS. Группировка данных. Пользовательские типы. CASE. COALESCE и NULLIF»



Лекция №4

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Запросы и подзапросы. Пользовательские типы. Регулярные выражения SIMILAR TO. Агрегатные функции. Подзапросы. ANY. ALL. EXISTS. Группировка данных. Пользовательские типы. CASE. COALESCE и NULLIF»

Основы синтаксиса SQL в PostgreSQL – Запросы и подзапросы. Пользовательские типы. Создаем БД и таблицы. Вносим данные –Часть 1

```
-- ===== 1) ГРУППЫ (5 строк) =====
INSERT INTO groups (group_id, group_name) VALUES
  (1, 'ИВТ-101'),
  (2, 'ИВТ-102'),
  (3, 'ПМИ-201'),
  (4, 'КБ-301'),
  (5, 'ЭВМ-401')
ON CONFLICT (group_id) DO NOTHING;

-- ===== 2) СТУДЕНТЫ (25 строк) =====
INSERT INTO learners (learner_id, full_name, email, group_id) VALUES
  ( 1, 'Иван Петров',       'ivan.petrov@example.com',       1),
  ( 2, 'Мария Иванова',     'm.ivanova@example.com',         1),
  ( 3, 'Сергей Смирнов',    NULL,                            1),
  ( 4, 'Анна Соколова',     'anna.sokolova@example.com',     1),
  ( 5, 'Дмитрий Кузнецов',  NULL,                            1),
  ( 6, 'Елена Попова',      'e.popova@example.com',          2),
  ( 7, 'Алексей Новиков',   'a.novikov@example.com',         2),
  ( 8, 'Виктория Морозова', NULL,                            2),
  ( 9, 'Никита Волков',     'n.volkov@example.com',          2),
  (10, 'Ольга Федорова',    NULL,                            2),
  (11, 'Павел Васильев',    'p.vasiliev@example.com',        3),
  (12, 'Кира Александрова', 'k.aleks@example.com',           3),
  (13, 'Илья Михайлов',     NULL,                            3),
  (14, 'Софья Громова',     's.gromova@example.com',         3),
  (15, 'Тимур Данилов',     NULL,                            3),
  (16, 'Яна Быкова',        'y.bykkova@example.com',         4),
  (17, 'Георгий Никитин',   'g.nikitin@example.com',         4),
  (18, 'Алина Егорова',     NULL,                            4),
  (19, 'Максим Захаров',    'm.zakharov@example.com',        4),
  (20, 'Вероника Павлова',  NULL,                            4),
  (21, 'Лев Тихонов',       'l.tikhonov@example.com',        5),
  (22, 'Полина Орлова',     NULL,                            5),
  (23, 'Арсений Комаров',   'a.komarov@example.com',         5),
  (24, 'Екатерина Белова',  'e.belova@example.com',          5),
  (25, 'Роман Крылов',      NULL,                            5)
ON CONFLICT (learner_id) DO NOTHING;
```

Лекция №4

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Запросы и подзапросы. Пользовательские типы. Регулярные выражения SIMILAR TO. Агрегатные функции. Подзапросы. ANY. ALL. EXISTS. Группировка данных. Пользовательские типы. CASE. COALESCE и NULLIF»


```
-- ===== 3) ОЦЕНКИ (20 строк) =====
-- Курсы: SQL / Math / ML
INSERT INTO enrollments (learner_id, course, grade) VALUES
 ( 1, 'SQL',  95.0),
 ( 2, 'SQL',  88.5),
 ( 3, 'SQL',  72.0),
 ( 4, 'SQL',  NULL),
 ( 5, 'SQL',  64.0),
 ( 6, 'SQL',  81.5),
 ( 7, 'SQL',  90.0),
 ( 8, 'SQL',  53.0),
 ( 9, 'SQL',  77.0),
 (10, 'SQL',  85.0),
 ( 1, 'Math', 78.0),
 ( 2, 'Math', 91.0),
 ( 3, 'Math', 60.0),
 ( 4, 'Math', 82.0),
 ( 5, 'Math', NULL),
 ( 6, 'ML',   89.0),
 ( 7, 'ML',   93.5),
 ( 8, 'ML',   68.0),
 ( 9, 'ML',   NULL),
 (10, 'ML',   71.0)
ON CONFLICT (learner_id, course) DO NOTHING;

-- (необязательно, но полезно) синхронизируем sequence после ручных id
SELECT setval(pg_get_serial_sequence('mirea2.groups','group_id'),
          (SELECT MAX(group_id) FROM mirea2.groups));
SELECT setval(pg_get_serial_sequence('mirea2.learners','learner_id'),
          (SELECT MAX(learner_id) FROM mirea2.learners));
```

Лекция №4

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Запросы и подзапросы. Пользовательские типы. Регулярные выражения SIMILAR TO. Агрегатные функции. Подзапросы. ANY. ALL. EXISTS. Группировка данных. Пользовательские типы. CASE. COALESCE и NULLIF»

Основы синтаксиса SQL в PostgreSQL – Запросы и подзапросы

Обычный SELECT

```
-- Обычный SELECT
SELECT full_name, email
FROM learners
WHERE group_id = 1
ORDER BY full_name;
```

Output | mirea2.mirea2.learners | Tx

	full_name	email
1	Анна Соколова	anna.sokolova@example.com
2	Дмитрий Кузнецов	<null>
3	Иван Петров	ivan.petrov@example.com
4	Мария Иванова	m.ivanova@example.com
5	Сергей Смирнов	<null>

Подзапросы — это SELECT внутри другого запроса. Виды подзапросов: скалярный, список значений, табличный и коррелированный

```
-- Скалярный подзапрос: он возвращает одно значение (средний балл) для каждого студента
SELECT
  full_name,   -- имя студента из таблицы learners
  (
    -- подзапрос: для текущего студента (по learner_id)
    -- считаем среднюю оценку по всем его курсам
    SELECT AVG(grade)
    FROM enrollments e   -- e = псевдоним таблицы enrollments
    WHERE e.learner_id = l.learner_id
    -- здесь l.learner_id: l = псевдоним таблицы learners (см. ниже),
    -- таким образом связываем текущего студента из внешнего запроса
    -- с его оценками из таблицы enrollments
  ) AS avg_grade  -- псевдоним для полученного среднего значения
FROM learners l;   -- learners = таблица студентов, l = её псевдоним
```

Output | Скалярный подзапрос:... для каждого студента

	full_name	avg_grade
1	Иван Петров	86.5
2	Мария Иванова	89.75
3	Сергей Смирнов	66
4	Анна Соколова	82
5	Дмитрий Кузнецов	64
6	Елена Попова	85.25
7	Алексей Новиков	91.75
8	Виктория Морозова	60.5
9	Никита Волков	77
10	Ольга Федорова	78
11	Павел Васильев	<null>
12	Кира Александрова	<null>
13	Илья Михайлов	<null>
14	Софья Громова	<null>
15	Тимур Данилов	<null>

Лекция №4

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Запросы и подзапросы. Пользовательские типы. Регулярные выражения SIMILAR TO. Агрегатные функции. Подзапросы. ANY. ALL. EXISTS. Группировка данных. Пользовательские типы. CASE. COALESCE и NULLIF»

Основы синтаксиса SQL в PostgreSQL – Запросы и подзапросы. Список значений и табличный подзапрос

```sql
-- Пример использования подзапроса списка значений (IN):
-- выбираем студентов, у которых есть записи по курсу "SQL"
SELECT
  full_name       -- имя студента
FROM learners       -- основная таблица студентов
WHERE learner_id IN (   -- условие: id студента должен входить в список
  -- подзапрос: возвращает список learner_id студентов,
  -- которые учатся на курсе "SQL"
  SELECT learner_id
  FROM enrollments     -- таблица с оценками/зачислениями
  WHERE course = 'SQL' -- фильтруем только курс "SQL"
);
```

Output | выбираем студентов, ...записи по курсу "SQL" | Tx

Tx: Auto | DDL

	full_name
1	Иван Петров
2	Мария Иванова
3	Сергей Смирнов
4	Анна Соколова
5	Дмитрий Кузнецов
6	Елена Попова
7	Алексей Новиков
8	Виктория Морозова
9	Никита Волков
10	Ольга Федорова

```sql
-- Табличный подзапрос: используется в секции FROM как "виртуальная таблица"
SELECT
  t.group_id,   -- идентификатор группы
  t.cnt        -- количество студентов в этой группе
FROM (
  -- подзапрос: считаем количество студентов в каждой группе
  SELECT
    group_id,          -- id группы
    COUNT(*) AS cnt     -- число студентов в группе
  FROM learners          -- таблица студентов
  GROUP BY group_id       -- группируем по каждой группе
) AS t                    -- подзапросу присвоен псевдоним t → теперь это виртуальная таблица
WHERE t.cnt >= 3;          -- фильтруем: оставляем только те группы, где студентов ≥ 3
```

Output | Табличный подзапрос:..."виртуальная таблица"

Tx: Auto | DDL

	group_id	cnt
1	3	5
2	5	5
3	4	5
4	2	5
5	1	5

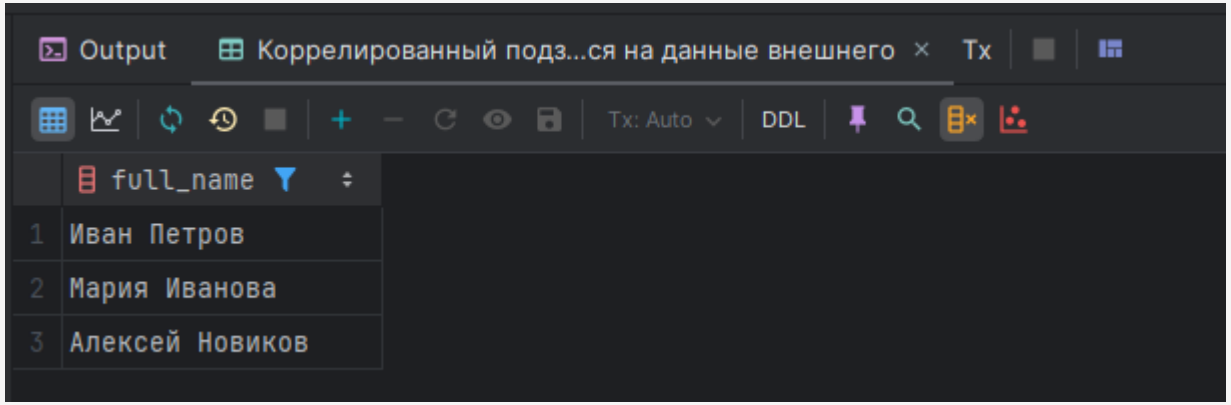
Лекция №4

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Запросы и подзапросы. Пользовательские типы. Регулярные выражения SIMILAR TO. Агрегатные функции. Подзапросы. ANY. ALL. EXISTS. Группировка данных. Пользовательские типы. CASE. COALESCE и NULLIF»

Основы синтаксиса SQL в PostgreSQL – Запросы и подзапросы. Коррелированный подзапрос

```sql
-- Коррелированный подзапрос: внутренний запрос ссылается на данные внешнего
SELECT
  full_name        -- имя студента
FROM learners l      -- основная таблица студентов (псевдоним l)
WHERE EXISTS (       -- проверяем: существует ли хотя бы одна строка в подзапросе
  SELECT 1         -- можно выбирать любое фиксированное значение (например, 1),
                   -- т.к. важно только условие существования строк
  FROM enrollments e -- таблица с оценками студентов (псевдоним e)
  WHERE e.learner_id = l.learner_id
      -- связываем подзапрос с внешней таблицей:
      -- берём только те оценки, что относятся к текущему студенту l
    AND e.grade >= 90
      -- дополнительное условие: оценка должна быть не меньше 90
);
```



Подзапрос коррелирован, потому что он ссылается на l.learner_id из внешнего запроса.
Для каждого студента из learners проверяется, существует ли хотя бы одна оценка ≥ 90 в enrollments.
Если такая запись есть → условие EXISTS возвращает TRUE → студент попадает в результат.

Лекция №4

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Запросы и подзапросы. Пользовательские типы. Регулярные выражения SIMILAR TO. Агрегатные функции. Подзапросы. ANY. ALL. EXISTS. Группировка данных. Пользовательские типы. CASE. COALESCE и NULLIF»

Основы синтаксиса SQL в PostgreSQL – Регулярные выражения SIMILAR TO

Оператор SIMILAR TO в PostgreSQL — это «гибрид»: умеет всё, что делает LIKE (%, _), плюс поддерживает некоторые элементы регулярных выражений:

1. () — группировка.
2. | — альтернатива (ИЛИ).
3. {m,n} — количество повторов.
4. [0-9] — диапазоны символов (как в regex).

Синтаксис:

```
<строка> SIMILAR TO '<шаблон>'
```

```
SELECT *
FROM learners
WHERE full_name SIMILAR TO '(А|Б)%';
-- (А|В)  → имя должно начинаться на А или Б
-- %      → после может быть любая последовательность символов
-- Например: «Анна", «Борис", но не “Константин”
```

Output | mirea2.mirea2.learners | Tx | Tx: Auto | DDL

	learner_id	full_name	email	group_id
1	4	Анна Соколова	anna.sokolova@example.com	1
2	7	Алексей Новиков	a.novikov@example.com	2
3	18	Алина Егорова	<null>	4
4	23	Арсений Комаров	a.komarov@example.com	5

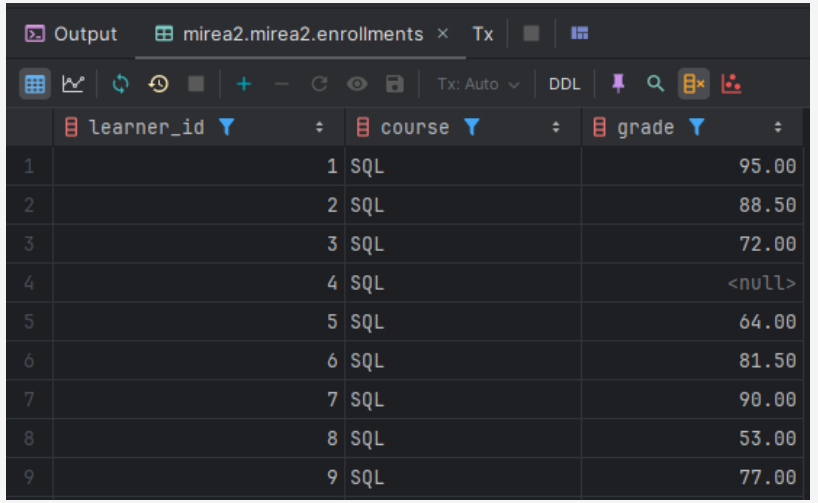
Лекция №4

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Запросы и подзапросы. Пользовательские типы. Регулярные выражения SIMILAR TO. Агрегатные функции. Подзапросы. ANY. ALL. EXISTS. Группировка данных. Пользовательские типы. CASE. COALESCE и NULLIF»

Основы синтаксиса SQL в PostgreSQL – Регулярные выражения SIMILAR TO

```
SELECT *
FROM enrollments
WHERE course SIMILAR TO '(SQL|Math|ML)';
-- (SQL|Math|ML) → курс должен быть ровно одним из трёх вариантов:
-- "SQL", "Math" или "ML"
```

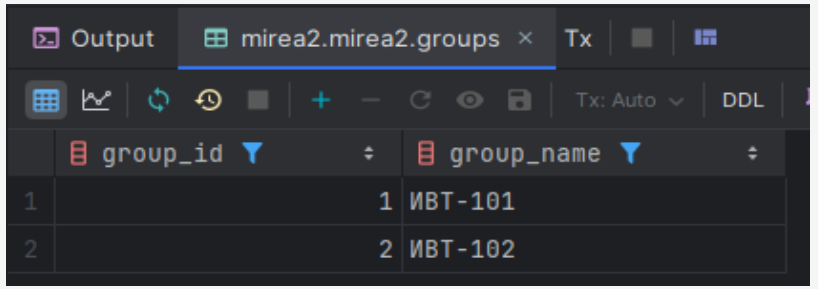


Output | mirea2.mirea2.enrollments | Tx

Tx: Auto | DDL

	learner_id	course	grade
1	1	SQL	95.00
2	2	SQL	88.50
3	3	SQL	72.00
4	4	SQL	<null>
5	5	SQL	64.00
6	6	SQL	81.50
7	7	SQL	90.00
8	8	SQL	53.00
9	9	SQL	77.00

```
SELECT *
FROM groups
WHERE group_name SIMILAR TO 'ИВТ-[0-9]{3}';
-- 'ИВТ-'   → строка должна начинаться с ИВТ-
-- [0-9]{3} → после идёт ровно три цифры
-- Подойдут: "ИВТ-101", "ИВТ-999"
-- Не подойдут: "ИВТ-10", "ИВТ-1000", "АВС-123"
```



Output | mirea2.mirea2.groups | Tx

Tx: Auto | DDL

	group_id	group_name
1	1	ИВТ-101
2	2	ИВТ-102

Лекция №4

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Запросы и подзапросы. Пользовательские типы. Регулярные выражения SIMILAR TO. Агрегатные функции. Подзапросы. ANY. ALL. EXISTS. Группировка данных. Пользовательские типы. CASE. COALESCE и NULLIF»

Основы синтаксиса SQL в PostgreSQL – Регулярные выражения SIMILAR TO

Оператор	Для чего нужен	Синтаксис и спецсимволы	Примеры
LIKE	Простой поиск по шаблону. Работает быстро, но возможностей мало.	% — любая последовательность символов_ — ровно один символ	full_name LIKE 'A%' → имена, начинающиеся с A.group_name LIKE 'ИВТ-___' → ИВТ- и ровно 3 символа.
SIMILAR TO	Гибрид LIKE + упрощённые регулярные выражения. Удобен, когда нужен набор вариантов.	Всё из LIKE + ( ) — группировка`	— альтернатива [ ] — диапазон {m,n}` — количество повторов
~ / !~	Полноценные регулярные выражения (PCRE). Самый гибкий и мощный способ.	Поддерживает весь синтаксис регэкспов (кванторы, классы, границы слов и т.д.)	email ~ '^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$' → проверка email.full_name !~ '^[A-Z]' → имена НЕ с заглавной буквы.

Лекция №4

Тема: «Основы синтаксиса SQL в PostgreSQL»

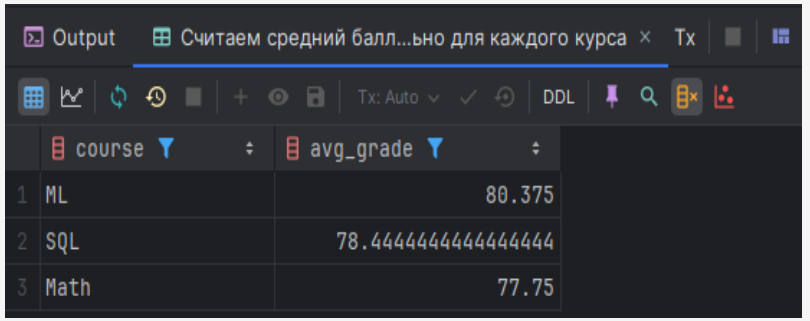
Подтема: «Запросы и подзапросы. Пользовательские типы. Регулярные выражения SIMILAR TO. Агрегатные функции. Подзапросы. ANY. ALL. EXISTS. Группировка данных. Пользовательские типы. CASE. COALESCE и NULLIF»

Основы синтаксиса SQL в PostgreSQL – Агрегатные функции

Агрегатные функции применяются не к отдельным строкам, а к целым наборам строк.
Часто используемые:
1. COUNT(*) — количество строк.
2. SUM(col) — сумма значений столбца.
3. AVG(col) — среднее значение.
4. MIN(col) — минимальное значение.
5. MAX(col) — максимальное значение.
6. STRING_AGG(col, sep) — склеивает строки через разделитель (удобно в PostgreSQL).

Чтобы агрегаты работали по группам, используется GROUP BY.

```
-- Считаем средний балл отдельно для каждого курса
SELECT
    course,              -- название курса
    AVG(grade) AS avg_grade -- средняя оценка по этому курсу
FROM enrollments         -- таблица "зачисления / оценки"
GROUP BY course          -- группируем данные по каждому курсу
ORDER BY avg_grade DESC;    -- сортируем по средней оценке (от большего к меньшему)
```



Лекция №4

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Запросы и подзапросы. Пользовательские типы. Регулярные выражения SIMILAR TO. Агрегатные функции. Подзапросы. ANY. ALL. EXISTS. Группировка данных. Пользовательские типы. CASE. COALESCE и NULLIF»

Основы синтаксиса SQL в PostgreSQL – Агрегатные функции

```
-- Считаем, сколько студентов в каждой учебной группе
SELECT
    g.group_name,          -- название группы
    COUNT(*) AS students   -- количество студентов в этой группе
FROM groups g              -- таблица групп
LEFT JOIN learners l       -- соединяем с таблицей студентов
        ON l.group_id = g.group_id
        -- LEFT JOIN → даже если в группе нет студентов,
        -- группа всё равно попадёт в результат
GROUP BY g.group_id, g.group_name  -- группируем по каждой группе
ORDER BY students DESC;            -- сортируем по числу студентов (сначала больше)
```

Output | Считаем, сколько сту...каждой учебной группе | Tx

	group_name	students
1	ИВТ-102	5
2	ПМИ-201	5
3	ЭВМ-401	5
4	КБ-301	5
5	ИВТ-101	5

```
-- Показываем только те курсы, где средняя оценка >= 80
SELECT
    course,              -- курс
    AVG(grade) AS avg_grade -- средняя оценка по курсу
FROM enrollments
GROUP BY course          -- сначала группируем по курсам
HAVING AVG(grade) >= 80;    -- затем фильтруем группы по условию
```

Output | Показываем только те... средняя оценка >= 80 | Tx

	course	avg_grade
1	ML	80.375

Лекция №4

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Запросы и подзапросы. Пользовательские типы. Регулярные выражения SIMILAR TO. Агрегатные функции. Подзапросы. ANY. ALL. EXISTS. Группировка данных. Пользовательские типы. CASE. COALESCE и NULLIF»

Основы синтаксиса SQL в PostgreSQL – Подзапросы с ANY, ALL, EXISTS

1. ANY — «хотя бы одно совпадение»

Работает так: условие верно, если найдётся хотя бы одно значение в подзапросе, для которого выполняется сравнение.

```sql
-- Студенты, у которых есть оценка выше хотя бы одной оценки по курсу "SQL"
SELECT
   full_name        -- имя студента
FROM learners l      -- таблица студентов
WHERE 85 > ANY (     -- проверяем: есть ли хотя бы одна оценка по SQL, которая меньше 85
   SELECT grade      -- подзапрос возвращает все оценки по курсу SQL
   FROM enrollments e
   WHERE e.course = 'SQL'
);
-- Если хотя бы одна оценка по SQL < 85, то условие TRUE,
-- и студент с оценкой 85 попадёт в результат.
-- По сути, 85 = ANY (подзапрос) ≈ 85 IN (список).
```

Output | Студенты, у которых ...оценки по курсу "SQL" | Tx

Tx: Auto | DDL

	full_name
1	Иван Петров
2	Мария Иванова
3	Сергей Смирнов
4	Анна Соколова
5	Дмитрий Кузнецов
6	Елена Попова
7	Алексей Новиков
8	Виктория Морозова

2. ALL — «условие верно для всех значений»

Условие выполняется только если для всех значений подзапроса оно истинно.

```sql
-- Студенты, у которых оценки по курсу "Math" выше всех оценок по "SQL"
SELECT DISTINCT
   l.full_name        -- имя студента
FROM learners l
JOIN enrollments em
    ON em.learner_id = l.learner_id
   AND em.course = 'Math'  -- берём только оценки студентов по курсу "Math"
WHERE em.grade > ALL (     -- проверяем: оценка студента по Math выше КАЖДОЙ оценки по SQL
   SELECT grade
   FROM enrollments
   WHERE course = 'SQL'
);
-- Если у студента по Math, например, 95,
-- а все оценки по SQL ≤ 90, то условие выполняется.
-- Таковых нет
```

Лекция №4

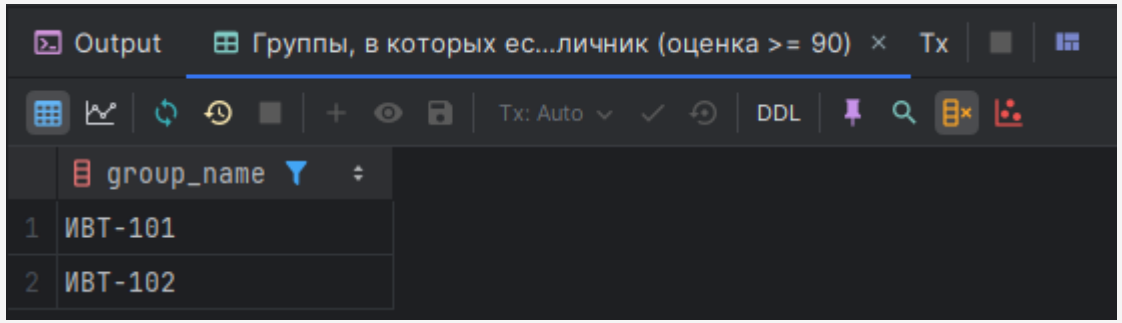
Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Запросы и подзапросы. Пользовательские типы. Регулярные выражения SIMILAR TO. Агрегатные функции. Подзапросы. ANY. ALL. EXISTS. Группировка данных. Пользовательские типы. CASE. COALESCE и NULLIF»

Основы синтаксиса SQL в PostgreSQL – Подзапросы с ANY, ALL, EXISTS

3. EXISTS — «есть ли хотя бы одна строка»
Не важно, что именно возвращает подзапрос (обычно пишут SELECT 1).
Главное — существует ли хоть одна строка.

```
-- Группы, в которых есть хотя бы один отличник (оценка >= 90)
SELECT DISTINCT
   g.group_name       -- название группы
FROM groups g         -- таблица групп
WHERE EXISTS (        -- проверяем: существует ли хотя бы одна строка
   SELECT 1           -- обычно пишут просто 1 (значение не имеет значения)
   FROM learners l
   JOIN enrollments e
      ON e.learner_id = l.learner_id
   WHERE l.group_id = g.group_id  -- связываем группу и студентов
     AND e.grade >= 90            -- условие: у студента есть оценка ≥ 90
);
-- Если хотя бы один студент из группы получил >=90,
-- то эта группа будет показана в результате.
```



EXISTS → «проверяем, есть ли хотя бы одна строка».
IN или = **ANY** → «значение входит в список».
ALL → «условие выполняется для всех значений подзапроса».

Лекция №4

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Запросы и подзапросы. Пользовательские типы. Регулярные выражения SIMILAR TO. Агрегатные функции. Подзапросы. ANY. ALL. EXISTS. Группировка данных. Пользовательские типы. CASE. COALESCE и NULLIF»

Основы синтаксиса SQL в PostgreSQL – Оператор CASE

CASE — условия прямо в SELECT
Здесь мы сравниваем значение одного столбца (course) с разными вариантами.

```
-- Показываем студентов и название курса по-русски
SELECT
   full_name,     -- имя студента
   CASE course    -- проверяем поле course
      WHEN 'SQL'  THEN 'База данных'  -- если курс = SQL
      WHEN 'Math' THEN 'Математика'   -- если курс = Math
      ELSE 'Другое'                   -- во всех остальных случаях
   END AS course_rus  -- псевдоним для нового столбца
FROM enrollments
JOIN learners USING (learner_id);
```

Output | Показываем студентов...вание курса по-русски | Tx

	full_name	course_rus
1	Иван Петров	База данных
2	Мария Иванова	База данных
3	Сергей Смирнов	База данных
4	Анна Соколова	База данных
5	Дмитрий Кузнецов	База данных
6	Елена Попова	База данных
7	Алексей Новиков	База данных
8	Виктория Морозова	База данных

Можно делать гибкое сравнение, т.е. «По выражению»
Здесь мы пишем произвольные условия с WHEN. Это мощнее, чем сравнение с одним полем.

Лекция №4

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Запросы и подзапросы. Пользовательские типы. Регулярные выражения SIMILAR TO. Агрегатные функции. Подзапросы. ANY. ALL. EXISTS. Группировка данных. Пользовательские типы. CASE. COALESCE и NULLIF»

Основы синтаксиса SQL в PostgreSQL – Оператор CASE

```sql
-- Показываем студентов, их оценку и словесный уровень
SELECT
  full_name,   -- имя студента
  grade,       -- числовая оценка
  CASE
    WHEN grade IS NULL        THEN 'нет оценки' -- если оценки нет
    WHEN grade >= 90          THEN 'отлично'    -- 90 и выше
    WHEN grade BETWEEN 75 AND 89 THEN 'хорошо'    -- от 75 до 89
    WHEN grade BETWEEN 60 AND 74 THEN 'удовл.'    -- от 60 до 74
    ELSE 'плохо'                          -- всё, что ниже 60
  END AS level  -- псевдоним для словесной оценки
FROM enrollments
JOIN learners USING (learner_id);
```

Output | Показываем студентов...у и словесный уровень | Tx

Tx: Auto | DDL

	full_name	grade	level
1	Иван Петров	95.00	отлично
2	Мария Иванова	88.50	хорошо
3	Сергей Смирнов	72.00	удовл.
4	Анна Соколова	<null>	нет оценки
5	Дмитрий Кузнецов	64.00	удовл.
6	Елена Попова	81.50	хорошо
7	Алексей Новиков	90.00	отлично
8	Виктория Морозова	53.00	плохо
9	Никита Волков	77.00	хорошо

CASE — это аналог if-else внутри SQL.
Можно проверять одно поле (CASE course WHEN ...) или писать произвольные условия (CASE WHEN ...).
Работает в SELECT, но также можно использовать в WHERE, ORDER BY и даже в GROUP BY.

Лекция №4

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Запросы и подзапросы. Пользовательские типы. Регулярные выражения SIMILAR TO. Агрегатные функции. Подзапросы. ANY. ALL. EXISTS. Группировка данных. Пользовательские типы. CASE. COALESCE и NULLIF»

Основы синтаксиса SQL в PostgreSQL – COALESCE и NULLIF

COALESCE и NULLIF — работа с NULL
1. COALESCE(a, b, c, ...)

Берёт первое не-NULL значение из списка.
Если все значения NULL → вернёт NULL.
Очень удобно для «подстановки значения по умолчанию».

```
-- Показать почту, а если её нет (NULL) — вывести текст "нет почты"
SELECT
  full_name,                          -- имя студента
  COALESCE(email, 'нет почты') AS email_show     -- если email = NULL, показать "нет почты"
FROM learners;
```

Output | Показать почту, а ес...сти текст "нет почты" | Tx | Tx: Auto | DDL

	full_name	email_show
1	Иван Петров	ivan.petrov@example.com
2	Мария Иванова	m.ivanova@example.com
3	Сергей Смирнов	нет почты
4	Анна Соколова	anna.sokolova@example.com
5	Дмитрий Кузнецов	нет почты
6	Елена Попова	e.popova@example.com

Лекция №4

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Запросы и подзапросы. Пользовательские типы. Регулярные выражения SIMILAR TO. Агрегатные функции. Подзапросы. ANY. ALL. EXISTS. Группировка данных. Пользовательские типы. CASE. COALESCE и NULLIF»

Основы синтаксиса SQL в PostgreSQL – COALESCE и NULLIF

2. Использование COALESCE внутри агрегата

Если у студента нет оценки (NULL), можно подставить 0, чтобы не ломался AVG (средний балл).

```sql
-- Считаем среднюю оценку студента, подставляя 0 вместо NULL
SELECT
   l.full_name,
   AVG(COALESCE(e.grade, 0)) AS avg_grade_zero  -- NULL заменяем на 0
FROM learners l
LEFT JOIN enrollments e
     ON e.learner_id = l.learner_id   -- берём все записи студентов, даже без оценок
GROUP BY l.full_name;                 -- считаем средний балл по каждому студенту
```

Output | Считаем среднюю оцен...ставляя 0 вместо NULL | Tx

Tx: Auto | DDL

	full_name	avg_grade_zero
1	Илья Михайлов	0
2	Мария Иванова	89.75
3	Павел Васильев	0
4	Никита Волков	38.5
5	Виктория Морозова	60.5
6	Кира Александрова	0
7	Яна Быкова	0

Лекция №4

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Запросы и подзапросы. Пользовательские типы. Регулярные выражения SIMILAR TO. Агрегатные функции. Подзапросы. ANY. ALL. EXISTS. Группировка данных. Пользовательские типы. CASE. COALESCE и NULLIF»

Основы синтаксиса SQL в PostgreSQL – Пользовательские типы

ENUM — перечисления (фиксированный список значений).
Плюсы:
1. Коротко и безопасно (никаких «левых» строк).
2. Удобно для фиксированных справочников (например, времена года).

Минусы:
1. Добавить новые значения можно (ALTER TYPE ... ADD VALUE),
2. Но удалить или переименовать — проблематично (требует обходных манёвров).

```
-- Создаём ENUM: список допустимых значений для семестра
CREATE TYPE term AS ENUM ('autumn','spring','summer','winter');

-- Добавляем в таблицу enrollments новый столбец типа ENUM
ALTER TABLE enrollments
  ADD COLUMN term_enum term NOT NULL DEFAULT 'autumn';
  -- теперь курс всегда должен иметь одно из четырёх значений
  -- значение по умолчанию — 'autumn'
```

DOMAIN — «тип с правилом» (с ограничением)

```
-- Создаём тип "email_text" — строка с проверкой на формат email
CREATE DOMAIN email_text AS TEXT
CHECK (
  VALUE IS NULL OR VALUE SIMILAR TO '%@%\.%'
  -- допускаем NULL или что-то похожее на email
);

-- Применяем домен к столбцу email
ALTER TABLE learners
  ALTER COLUMN email TYPE email_text;
```

Лекция №4

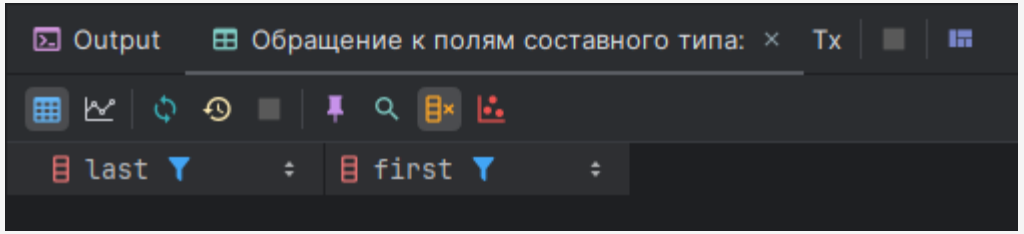
Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Запросы и подзапросы. Пользовательские типы. Регулярные выражения SIMILAR TO. Агрегатные функции. Подзапросы. ANY. ALL. EXISTS. Группировка данных. Пользовательские типы. CASE. COALESCE и NULLIF»

Основы синтаксиса SQL в PostgreSQL – Пользовательские типы

Составные типы (composite).
Особенность: составной тип хранит несколько полей внутри одного столбца.
Похоже на struct в языках программирования.

```
-- Создаём составной тип для полного имени
CREATE TYPE full_name_t AS (
  last  TEXT,  -- фамилия
  first TEXT   -- имя
);

-- Используем этот тип в таблице
CREATE TABLE persons(
  id   SERIAL PRIMARY KEY,
  name full_name_t  -- здесь хранится сразу (фамилия + имя)
);

-- Обращение к полям составного типа:
SELECT (name).last, (name).first
FROM persons;
```



ENUM → фиксированный список значений (удобно для закрытых категорий, например: времена года, статусы заказов).
DOMAIN → «тип + ограничение», удобно для валидации (например, оценки, email).
Составной тип → хранение сразу нескольких полей в одном столбце (например, имя и фамилия вместе).

Лекция №4

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Запросы и подзапросы. Пользовательские типы. Регулярные выражения SIMILAR TO. Агрегатные функции. Подзапросы. ANY. ALL. EXISTS. Группировка данных. Пользовательские типы. CASE. COALESCE и NULLIF»

Основы синтаксиса SQL в PostgreSQL – Группировка данных

Алгоритм группировки данных:

1. Отбираем строки → WHERE (какие записи участвуют в анализе).
2. Группируем → GROUP BY (по какому признаку объединяем строки).
3. Считаем агрегаты → COUNT, AVG, SUM, MIN, MAX, ...
4. Фильтруем группы → HAVING (оставляем только нужные группы).
5. Сортируем и выводим → ORDER BY, SELECT.

```
-- По учебным группам считаем:
-- 1) сколько студентов в каждой
-- 2) средний балл именно по курсу "SQL"
SELECT
   g.group_name,                    -- название группы
   COUNT(DISTINCT l.learner_id) AS students, -- количество студентов (DISTINCT → без дублей)
   AVG(e.grade)              AS avg_sql   -- средний балл по курсу SQL
FROM groups g
-- соединяем с таблицей студентов
LEFT JOIN learners l
     ON l.group_id = g.group_id
-- соединяем с таблицей оценок, но только для курса "SQL"
LEFT JOIN enrollments e
     ON e.learner_id = l.learner_id
    AND e.course = 'SQL'
GROUP BY g.group_id, g.group_name          -- группируем по каждой группе
HAVING COUNT(DISTINCT l.learner_id) > 0     -- отбрасываем пустые группы (без студентов)
ORDER BY avg_sql DESC NULLS LAST;          -- сортируем по среднему баллу (сначала больше),
                                           -- а NULL ставим в конец
```

Output | 2) средний балл именно по курсу "SQL" | Tx

	group_name	students	avg_sql
1	ИВТ-101	5	79.875
2	ИВТ-102	5	77.3
3	ПМИ-201	5	<null>
4	КБ-301	5	<null>
5	ЭВМ-401	5	<null>

Лекция №4

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «Запросы и подзапросы. Пользовательские типы. Регулярные выражения SIMILAR TO. Агрегатные функции. Подзапросы. ANY. ALL. EXISTS. Группировка данных. Пользовательские типы. CASE. COALESCE и NULLIF»

Основы синтаксиса SQL в PostgreSQL – Частые ошибки и как их избежать

1. GROUP BY не совпадает с SELECT: все неагрегатные поля должны быть в GROUP BY.
2. NULL в вычислениях: помни про COALESCE и NULLIF.
3. IN с NULL: x NOT IN (...NULL...) даёт UNKNOWN → условие не проходит. Безопаснее NOT EXISTS.
4. HAVING вместо WHERE: фильтрация агрегатов — только в HAVING.
5. SIMILAR TO/regex: начинаем с простых шаблонов, сложные — через обычные ~.

Лекция №5

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «ROLLUP. GROUPING SETS. CUBE. Создание и использование представлений. Материализованные представления. Common Table Expression»

Основы синтаксиса SQL в PostgreSQL – GROUP BY, GROUPING SETS, ROLLUP, CUBE, VIEW

Термин	Что это (очень просто)	Когда использовать	Короткий синтаксис / пример
GROUP BY	Склеивает строки по столбцам и считает агрегаты (SUM/AVG/COUNT…).	Нужен один «разрез»(агрегация, вывод только определенных данных) данных (без подитогов). Базовая агрегация.	SELECT a, SUM(x) FROM t GROUP BY a;
GROUPING SETS	Несколько разных группировок **в одном** запросе.	Хочется сразу и по (a,b), и по a, и общий итог — без трёх запросов.	GROUP BY GROUPING SETS ((a,b),(a),())
ROLLUP	Подитоги по **иерархии слева направо** + общий итог.	Есть естественная иерархия: Год→Месяц→День; Группа→Курс.	GROUP BY ROLLUP (year, month, day)
CUBE	**Все комбинации** измерений + общий итог.	Нужны все подитоги сразу (по каждому столбцу, по парам и т.д.). Осторожно: много строк.	GROUP BY CUBE (group_name, course)
VIEW (представление)	«Сохранённый SELECT» — виртуальная таблица, считает «на лету».	Спрятать сложные JOIN/фильтры, дать удобный слой чтения.	CREATE VIEW v AS SELECT …; SELECT * FROM v;
MATERIALIZED VIEW	**Снимок** результата запроса, хранится как таблица.	Тяжёлые отчёты читаются быстро, но нужно вручную обновлять.	CREATE MATERIALIZED VIEW mv AS SELECT …; REFRESH MATERIALIZED VIEW mv;
CTE (WITH)	Временная именованная «мини-таблица» из подзапроса.	Делить длинный запрос на шаги, переиспользовать промежуточный результат.	WITH t AS (SELECT …) SELECT … FROM t;
CTE (WITH RECURSIVE)	Рекурсивный CTE для деревьев/иерархий.	Структуры «родитель-потомок», уровни, путь.	WITH RECURSIVE t AS (anchor UNION ALL step FROM t) SELECT …;

Лекция №5

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «ROLLUP. GROUPING SETS. CUBE. Создание и использование представлений. Материализованные представления. Common Table Expression»

Основы синтаксиса SQL в PostgreSQL – GROUP BY, GROUPING SETS, ROLLUP, CUBE, VIEW. Подготовка примера для работы

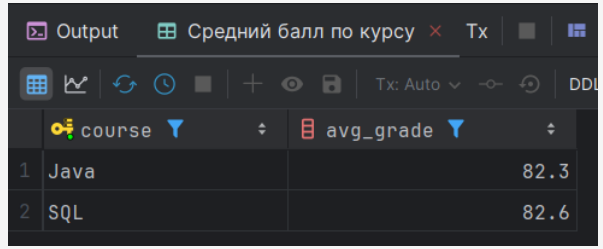
```
CREATE DATABASE test123;

-- Создаем схему
CREATE SCHEMA IF NOT EXISTS demo;
SET search_path = demo;

-- Группы
DROP TABLE IF EXISTS groups CASCADE;
CREATE TABLE groups(
  group_id   SERIAL PRIMARY KEY,
  group_name TEXT NOT NULL
);

-- Студенты
DROP TABLE IF EXISTS learners CASCADE;
CREATE TABLE learners(
  learner_id SERIAL PRIMARY KEY,
  full_name  TEXT NOT NULL,
  email      TEXT,
  group_id   INT REFERENCES groups(group_id)
);

-- Оценки по курсам
DROP TABLE IF EXISTS enrollments CASCADE;
CREATE TABLE enrollments(
  learner_id INT REFERENCES learners(learner_id),
  course     TEXT NOT NULL,
  grade      NUMERIC(5,2),
  PRIMARY KEY (learner_id, course)
);

-- Наполнение
INSERT INTO groups(group_name) VALUES
 ('A-101'), ('B-202'), ('C-303');

INSERT INTO learners(full_name, email, group_id) VALUES
 ('Иван Петров',   'ivan@u', 1),
 ('Мария Сидорова','maria@u',1),
 ('Анна Смирнова', 'anna@u', 2),
 ('Павел Орлов',   'pavel@u',2),
 ('Дмитрий Рой',   'dmitry@u',3),
 ('Елена Рай',     'elena@u',3);

INSERT INTO enrollments(learner_id, course, grade) VALUES
 (1,'SQL', 82),(1,'Java', 75),
 (2,'SQL', 90),
 (3,'SQL', 70),(3,'Java', 88),
 (4,'SQL', 76),
 (5,'SQL', 95),
 (6,'Java', 84);
```

Лекция №5

Тема: «Основы синтаксиса SQL в PostgreSQL»

Подтема: «ROLLUP. GROUPING SETS. CUBE. Создание и использование представлений. Материализованные представления. Common Table Expression»


```
-- Средний балл по курсу
SELECT course, ROUND(AVG(grade), 1) AS avg_grade
FROM enrollments
GROUP BY course
ORDER BY course;
```



Практические задания (контрольные работы)

Контрольная работа №1 (4 балла)

1. Назначение и цель

Разработать настольное приложение с графическим интерфейсом (GUI), взаимодействующее с БД PostgreSQL. В рамках работы студент обязан демонстративно и корректно применить:

1.1. Типы данных в PostgreSQL (текстовые, числовые, булевы, дата/время, перечисления ENUM, массивы — обязательно).

1.2. Операторы SQL: CREATE (схема/типы/таблицы) и INSERT.

1.3. Ограничения: NOT NULL, UNIQUE, CHECK.

1.4. Ключи: PRIMARY KEY, FOREIGN KEY (+ варианты поведения ON DELETE/ON UPDATE).

1.5. Механизмы ссылочной целостности (показать на практике, что БД не позволяет нарушить связи).

Предметная область должна отражать текущие технологические тренды.

Рекомендуемая область: «Эксперименты по внедрению ИИ в обнаружение DDoS-атак» (MLOps-мини-система). Допустимы альтернативы с ИИ (контент-модерация, генерация текстов/изображений, рекомендации и т.п.) — при сохранении требований, обозначенных на следующем слайде.

Практические задания (контрольные работы)

Контрольная работа №1 (4 балла)

2. Функциональные требования

2.1. GUI:

2.2.1. Главное окно с кнопками:

- «Создать схему и таблицы» (запуск скрипта CREATE ...);
- «Внести данные» — открывает **модальное** окно ввода (родительское окно **заблокировано** на время ввода);
- «Показать данные» — открывает **отдельное окно** (можно немодальное) со сводной таблицей (например, EXPERIMENTS/RUNS с JOIN).

2.2. Отдельные **модальные** окна:

- «Добавить данные» (каждое окно — отдельная форма). Родительское окно неподвижно/недоступно до закрытия формы;
- «Вывести данные». Родительское окно неподвижно/недоступно до закрытия формы.

2.2. Работа с БД:

- создание схемы БД, пользовательских типов (ENUM), таблиц с перечисленными ограничениями. Для этого должна быть реализована кнопка «Создать схему в БД»;
- вставка данных через подготовленные выражения (parameterized queries) и обёртку транзакций (кнопка «Добавить данные», обозначенная ранее);
- вывод данных (табличная форма) — по кнопке, в отдельном окне (кнопка «Вывести ранее, как и описывалось ранее). Обязательно необходимо реализовать фильтры (например, по типу атаки, дате, порогу метрик).

2.3. Обработка ошибок/исключений:

- любые ошибки БД должны перехватываться и отображаться человеку-понятными сообщениями: нарушение UNIQUE, NOT NULL, CHECK, нарушение внешнего ключа, несоответствие типов, ошибки подключения;
- предусмотреть откат (ROLLBACK) транзакций при ошибках вставки.

2.4. Логирование:

- лог-файл приложения (уровни INFO/ERROR) с временными метками: попытки подключения, выполненные DDL/DML, ошибки ограничений.

2.5. Кроссплатформенность: обязательна. ОС: Windows/Linux/macOS. Язык — на выбор (см. далее). Если платформа ограничена, указать это в README.

Практические задания (контрольные работы)

Контрольная работа №1 (4 балла)

3. Технологический стек (на выбор)

Вариант А (рекомендуемый): Python 3.12+, **PySide6/PyQt5** (GUI), psycopg2 или asyncpg/sqlalchemy (БД), PostgreSQL 13+.

Допустимые альтернативы (Вариант Б):
- C# + WPF + Npgsql;
- Java + JavaFX + JDBC;
- Kotlin + TornadoFX + JDBC.

Требования к GUI неизменны, т.е. должны соответствовать функциональности, описанной ранее.

Практические задания (контрольные работы)

Контрольная работа №2 (5 баллов)

ТЗ полностью идентичное контрольной работе №1, но должны быть добавлены следующие функции:

1. В графический интерфейс должны быть добавлены средства работы с оператором ALTER TABLE. Пользователь должен иметь возможность изменять структуру базы данных: добавлять или удалять столбцы, переименовывать таблицы и поля, изменять типы данных, а также задавать или убирать ограничения (CHECK, NOT NULL, UNIQUE, FOREIGN KEY). Все изменения должны выполняться транзакционно с выводом понятных сообщений об ошибках.

2. Для работы с оператором SELECT в окне вывода данных должны быть предусмотрены расширенные функции. Пользователь должен иметь возможность выбирать конкретные столбцы для вывода, задавать фильтры через WHERE, сортировать результаты при помощи ORDER BY, выполнять группировку с использованием GROUP BY и агрегатных функций (COUNT, AVG, SUM и др.), а также фильтровать группы через HAVING. Это должно позволять собирать сложные выборки без ручного написания SQL.

3. В интерфейсе должны быть реализованы инструменты поиска по тексту. Пользователь должен иметь возможность использовать оператор LIKE для шаблонного поиска, а также POSIX-регулярные выражения (~, ~*, !~, !~*). Выбор типа поиска должен осуществляться через выпадающий список.

4. Для отработки функций работы со строками должно быть создана отдельная кнопка в окне с выводом данных. Т.е. пользователь должен иметь возможность выполнять преобразования регистра (UPPER, LOWER), извлекать подстроки (SUBSTRING), удалять пробелы (TRIM), дополнять строки (LPAD, RPAD), а также объединять их через CONCAT или оператор ||. Все операции должны отображаться наглядно.

5. В интерфейс должен быть встроен «мастер соединений» для работы с JOIN. Пользователь должен иметь возможность выбирать таблицы, поля для связывания и тип соединения (INNER, LEFT, RIGHT, FULL). Результат должен отображаться в сводной таблице с возможностью дальнейшей сортировки и фильтрации.

Практические задания (контрольные работы)

Контрольная работа №3 (6 баллов)

Продолжение контрольной работы №2:

1. В окне вывода данных, которое открывается отдельно, должны быть реализованы встроенные фильтры на основе подзапросов. Пользователь должен иметь возможность формировать вложенные SELECT-запросы, в том числе коррелированные, и применять их в качестве условий отбора строк. Должна быть предусмотрена поддержка операторов ANY, ALL и EXISTS. Все операции должны выполняться без ручного написания SQL — только через интерфейс в виде кнопок и выпадающих списков.

2. Для работы с пользовательскими типами должен быть предусмотрен отдельный модуль. Пользователь сможет создавать новые типы данных (ENUM, составные типы), использовать их при создании таблиц, а также просматривать и управлять уже существующими пользовательскими типами. Все действия должны выполняться через GUI-кнопки без необходимости вручную писать SQL.

3. В интерфейсе поиска по строкам должна быть добавлена возможность использования регулярных выражений SIMILAR TO. Пользователь сможет проверять соответствие строк шаблонам, а также задавать отрицание условия (NOT SIMILAR TO). Выбор столбца и задание шаблона должны выполняться через интерфейсные элементы, без ручного ввода SQL-запросов.

В окне работы с SELECT-запросами должны быть добавлены инструменты агрегирования и группировки. Пользователь сможет выбирать агрегатные функции (COUNT, SUM, AVG, MIN, MAX), группировать данные по выбранным атрибутам с помощью GROUP BY и фильтровать группы через HAVING. Все операции должны быть реализованы в виде кнопок и выпадающих меню, без ручного написания SQL.

4. Для условной обработки данных должен быть реализован конструктор выражений CASE. Пользователь сможет задавать условия WHEN ... THEN ... ELSE и использовать их при построении выборок для формирования новых вычисляемых столбцов. Конструктор CASE должен быть выполнен в виде формы с кнопками и полями, без ручного ввода SQL.

Дополнительно в интерфейс должны быть встроены средства работы с NULL-значениями через функции COALESCE и NULLIF. Пользователь сможет задавать подстановку значений вместо NULL и замену совпадающих значений на NULL. Все операции должны выполняться через кнопки и формы интерфейса, без ручного написания SQL.

Практические задания (контрольные работы)

Контрольная работа №4 (7 баллов)

Продолжение контрольной работы №3:

1. **В окне работы с SELECT-запросами должны быть реализованы расширенные средства группировки данных.** Пользователь должен иметь возможность использовать операции ROLLUP, CUBE и GROUPING SETS для построения многомерных агрегированных отчетов. Настройка группировок и выбор уровней детализации должны выполняться через интерфейс в виде кнопок и выпадающих списков, без ручного написания SQL.Для работы с представлениями в графический интерфейс должен быть добавлен отдельный модуль.

2. **Пользователь сможет создавать новые представления (VIEW), просматривать их структуру и данные, а также управлять уже созданными представлениями.** Все действия (создание, обновление, удаление) должны выполняться через кнопки и формы интерфейса, без ручного ввода SQL-кода.

3. **В дополнение к обычным представлениям должна быть реализована поддержка материализованных представлений (MATERIALIZED VIEW).** Пользователь сможет создавать такие объекты, обновлять их данные (REFRESH) и просматривать результаты. Управление должно быть доступно через интерфейсные элементы без необходимости ручного написания SQL.

4. **Также должен быть добавлен модуль для работы с Common Table Expressions (CTE).** Пользователь должен иметь возможность формировать временные выборки через WITH-запросы и использовать их для построения более сложных выборок. В интерфейсе должен быть предусмотрен конструктор CTE в виде кнопок, форм и блоков, позволяющий объединять несколько подзапросов, без ручного ввода SQL.

Практические задания (контрольные работы)

Контрольная работа №5 (8 баллов)

Продолжение контрольной работы №4:

1. В окне работы с SELECT-запросами должен быть реализован модуль для оконных функций. Пользователь должен иметь возможность применять функции RANK, LAG и LEAD к выбранным наборам данных. Интерфейс должен позволять задавать разбиение (PARTITION BY) и порядок (ORDER BY) через кнопки и выпадающие списки, без ручного написания SQL.

2. Для функции RANK пользователь должен иметь возможность получать ранжирование строк в зависимости от выбранного столбца или набора столбцов. Все настройки (выбор колонок для сортировки, направление сортировки) должны выполняться через интерфейсные элементы.

3. Для функций LAG и LEAD должна быть предусмотрена возможность сравнения текущей строки с предыдущей или последующей. Пользователь сможет выбирать величину смещения, а также указывать столбцы, для которых функция применяется. Настройка параметров должна происходить через форму, без ручного ввода SQL.

4. Результаты работы оконных функций должны отображаться в выводимой таблице с возможностью сортировки и дальнейшей фильтрации данных. Все операции должны выполняться только через графический интерфейс, без написания SQL-запросов вручную.