Министерство образования и науки Российской Федерации
Российский государственный университет нефти и газа
(национальный исследовательский университет)
имени И.М. Губкина

Факультет Автоматики и вычислительной техники
Кафедра Автоматизированных систем управления

Отчёт по лабораторной работе №2
«УГЛУБЛЕННАЯ РАБОТА С SQL»

по дисциплине *Базы данных*

Выполнил:
студент группы АС-23-04
Ханевский Ярослав

Проверили:
доцент кафедры АСУ, к.т.н. Волков Д.А.
ст. преп. кафедры АСУ Мухина А.Г.

Москва, 2025 г.

Ход работы:

1. Для своей таблицы выполнить запросы, содержащие блок WHERE с использованием различных операторов условий.

Создадим таблицу, включающую в себя информацию о сдаче экзаменов:

```
postgres=# CREATE TABLE exam(
postgres(# student_name varchar(50),
postgres(# subject varchar(50),
postgres(# teacher varchar(50),
postgres(# estimation int);
CREATE TABLE
```

Рисунок 1. Создание таблицы

После заполнения таблицы получаем следующее:

```
postgres=# SELECT * FROM exam;
 student_name |   subject   | teacher | estimation
--------------+-------------+---------+------------
 Ivanov       | Math        | Petrov  |          4
 Kochkin      | Informatika | Sidorov |          3
 Govrin       | Math        | Petrov  |          5
 Golikov      | Informatika | Sidorov |          4
```

Рисунок 2. Заполненная таблица

Выполним запрос с оператором условия OR, выбирающий тех учеников, кто получил оценку «4» или чья фамилия – «Kochkin»:

```
postgres=# SELECT * FROM exam WHERE (estimation = 4 OR student_name = 'Kochkin');
 student_name |   subject   | teacher | estimation
--------------+-------------+---------+------------
 Ivanov       | Math        | Petrov  |          4
 Kochkin      | Informatika | Sidorov |          3
 Golikov      | Informatika | Sidorov |          4
(3 rows)
```

Рисунок 3. Оператор OR

Запрос, выбирающий данные по условию «учитель «Sidorov» и оценка выше 3», используя оператор AND:

```
postgres=# SELECT * FROM exam WHERE (teacher = 'Sidorov' AND estimation > 3);
 student_name |   subject   | teacher | estimation
--------------+-------------+---------+------------
 Golikov      | Informatika | Sidorov |          4
(1 row)
```

Рисунок 4. Оператор AND

Реализуем запрос с помощью оператора LIKE, ищущий среди всех имен учеников те, что заканчиваются на буквы «ov»:

```
postgres=# SELECT * FROM exam WHERE student_name LIKE '%ov';
 student_name |   subject   | teacher | estimation
--------------+-------------+---------+------------
 Ivanov       | Math        | Petrov  |          4
 Golikov      | Informatika | Sidorov |          4
(2 rows)
```

Рисунок 5. Оператор LIKE

Найдем тех учеников, которые сдали экзамен на 4 или 5 (оператор IN):

```
postgres=# SELECT * FROM exam WHERE estimation IN(4, 5);
 student_name |   subject   | teacher | estimation
--------------+-------------+---------+------------
 Ivanov       | Math        | Petrov  |          4
 Govrin       | Math        | Petrov  |          5
 Golikov      | Informatika | Sidorov |          4
(3 rows)
```

Рисунок 6. Оператор IN

Найдем тех учеников, которые сдали экзамен на оценку между 2 и 3 (оператор BETWEEN):

```
postgres=# SELECT * FROM exam WHERE estimation BETWEEN 2 AND 3;
 student_name |   subject   | teacher | estimation
--------------+-------------+---------+------------
 Kochkin      | Informatika | Sidorov |          3
(1 row)
```

Рисунок 7. Оператор BETWEEN

2. Создать колонку с допустимым значением NULL. Выполнить запросы к таблице с использованием NULL. Обратить внимание на работу этого значения.

Создадим новую колонку с допустимым значением NULL (в PostgreSQL по умолчанию NULL является допустимым значением):

```
postgres=# ALTER TABLE exam
postgres-# ADD COLUMN description varchar(255) NULL;
ALTER TABLE
postgres=# \d exam
                    Table "public.exam"
    Column    |          Type          | Collation | Nullable | Default
--------------+------------------------+-----------+----------+---------
 student_name | character varying(50)  |           |          |
 subject      | character varying(50)  |           |          |
 teacher      | character varying(50)  |           |          |
 estimation   | integer                |           |          |
 description  | character varying(255) |           |          |
```

Рисунок 8. Новая колонка с допустимым значением NULL

Сделаем несколько запросов к таблице с использованием NULL; для начала добавим несколько строк:

```
postgres=# INSERT INTO exam VALUES
postgres-# ('Kulikova', 'Physics', 'Serebryakov', 3, NULL),
postgres-# ('Tihonova', 'English', 'Forkova', 5, 'some description');
INSERT 0 2
postgres=# SELECT * FROM exam;
 student_name |   subject   |   teacher   | estimation |   description
--------------+-------------+-------------+------------+------------------
 Ivanov       | Math        | Petrov      |          4 |
 Kochkin      | Informatika | Sidorov     |          3 |
 Govrin       | Math        | Petrov      |          5 |
 Golikov      | Informatika | Sidorov     |          4 |
 Kulikova     | Physics     | Serebryakov |          3 |
 Tihonova     | English     | Forkova     |          5 |
 Kulikova     | Physics     | Serebryakov |          3 |
 Tihonova     | English     | Forkova     |          5 | some description
(8 rows)
```

Рисунок 9. Добавление строки с пустым и с заполненным значением столбца

Выберем все строки из таблицы, в которых есть описание (IS NOT NULL):

```
postgres=# SELECT * FROM exam WHERE description IS NOT NULL;
 student_name | subject | teacher | estimation |   description
--------------+---------+---------+------------+------------------
 Tihonova     | English | Forkova |          5 | some description
(1 row)
```

Рисунок 10. Использование SELECT с NOT NULL

3. Для своей таблицы выполнить запросы, содержащие блок GROUP BY с использованием различных функций агрегирования и фильтрации (HAVING):

Создадим и заполним новую таблицу, в которой указаны данные о проданных товарах:

```
postgres=# CREATE TABLE sales(
postgres(# id SERIAL PRIMARY KEY,
postgres(# product VARCHAR(100) NOT NULL,
postgres(# category VARCHAR(100),
postgres(# quantity INT NOT NULL,
postgres(# price DECIMAL(10, 2) NOT NULL);
CREATE TABLE
postgres=# INSERT INTO sales (product, category, quantity, price) VALUES
postgres-# ('Laptop', 'Electronics', 2, 1000.00),
postgres-# ('Phone', 'Electronics', 5, 800.00),
postgres-# ('Tablet', 'Electronics', 3, 600.00),
postgres-# ('Shirt', 'Clothing', 10, 50.00),
postgres-# ('Pants', 'Clothing', 7, 70.00),
postgres-# ('Laptop', 'Electronics', 1, 1000.00);
INSERT 0 6
postgres=# SELECT * FROM sales;
 id | product |  category   | quantity |  price
----+---------+-------------+----------+---------
  1 | Laptop  | Electronics |        2 | 1000.00
  2 | Phone   | Electronics |        5 |  800.00
  3 | Tablet  | Electronics |        3 |  600.00
  4 | Shirt   | Clothing    |       10 |   50.00
  5 | Pants   | Clothing    |        7 |   70.00
  6 | Laptop  | Electronics |        1 | 1000.00
(6 rows)
```

Рисунок 11. Новая таблица с проданными товарами

Используем функцию GROUP BY для подсчёта общего количества проданных товаров в каждой категории:

```
postgres=# SELECT category, SUM(quantity) FROM sales GROUP BY category;
  category   | sum
-------------+-----
 Electronics |  11
 Clothing    |  17
(2 rows)
```

Применим функции агрегирования MAX и MIN для поиска максимальной и минимальной цены соответственно в каждой из групп категорий продуктов:

```
postgres=# SELECT category, MIN(price), MAX(price) FROM sales GROUP BY category;
  category   |  min   |   max
-------------+--------+---------
 Electronics | 600.00 | 1000.00
 Clothing    |  50.00 |   70.00
(2 rows)
```

Рисунок 12. Поиск максимальной и минимальной цены в каждой группе

Теперь добавим фильтрацию (HAVING) таким образом, чтобы отображались те категории, общее количество проданных товаров в которых больше 10:

```
postgres=# SELECT category, SUM(quantity) FROM sales GROUP BY category HAVING SUM(quantity) > 10;
  category   | sum
-------------+-----
 Electronics |  11
 Clothing    |  17
(2 rows)
```

Рисунок 13. Использование фильтрации для поиска групп, сумма проданных товаров которых больше 10

Найдем среднее значение цены в каждой группе с помощью функции AVG:

```
postgres=# SELECT category, AVG(price) FROM sales GROUP BY category;
  category   |         avg
-------------+----------------------
 Electronics | 850.0000000000000000
 Clothing    |  60.0000000000000000
(2 rows)
```

Рисунок 14. Среднее значение цены среди групп

4. Для своей таблицы выполнить запросы, содержащие блок ORDER BY.

Выполним сортировку (ORDER BY) для предыдущей таблицы по цене от большей к меньшей (DECS):

```
postgres=# SELECT * FROM sales ORDER BY price DESC;
 id | product |  category   | quantity |  price
----+---------+-------------+----------+---------
  1 | Laptop  | Electronics |        2 | 1000.00
  6 | Laptop  | Electronics |        1 | 1000.00
  2 | Phone   | Electronics |        5 |  800.00
  3 | Tablet  | Electronics |        3 |  600.00
  5 | Pants   | Clothing    |        7 |   70.00
  4 | Shirt   | Clothing    |       10 |   50.00
(6 rows)
```

Рисунок 15. Сортировка данных по убыванию

Проведем сортировку по нескольким столбцам, теперь от меньших значений к большим:

```
postgres=# SELECT * FROM sales ORDER BY quantity DESC, price;
 id | product |  category   | quantity |  price
----+---------+-------------+----------+---------
  4 | Shirt   | Clothing    |       10 |   50.00
  5 | Pants   | Clothing    |        7 |   70.00
  2 | Phone   | Electronics |        5 |  800.00
  3 | Tablet  | Electronics |        3 |  600.00
  1 | Laptop  | Electronics |        2 | 1000.00
  6 | Laptop  | Electronics |        1 | 1000.00
(6 rows)
```

Рисунок 16. Сортировка нескольких столбцов по возрастанию

5. Для своей таблицы выполнить запросы, содержащие блок LIMIT.

Осуществим запрос, выбирающий первые 3 строки таблицы с помощью блока LIMIT:

```
postgres=# SELECT * FROM sales LIMIT 3;
 id | product |  category   | quantity |  price
----+---------+-------------+----------+---------
  1 | Laptop  | Electronics |        2 | 1000.00
  2 | Phone   | Electronics |        5 |  800.00
  3 | Tablet  | Electronics |        3 |  600.00
(3 rows)
```

Рисунок 17. Использование блока LIMIT

Теперь пропустим первые 2 строки и выведем из таблицы 3 последующие строки:

```
postgres=# SELECT * FROM sales LIMIT 3 OFFSET 2;
 id | product |  category   | quantity | price
----+---------+-------------+----------+--------
  3 | Tablet  | Electronics |        3 | 600.00
  4 | Shirt   | Clothing    |       10 |  50.00
  5 | Pants   | Clothing    |        7 |  70.00
(3 rows)
```

Рисунок 18. Блок LIMIT, пропускающий указанное количество строк

6. Добавить первичный ключ для таблицы. Добавить авто инкремент, удалить первичный ключ.

С помощью ADD CONSTRAINT добавляем первичный ключ (PRIMARY KEY), которым будут столбцы с названием и количеством товаров:

```
postgres=# ALTER TABLE sales ADD CONSTRAINT sales_pk PRIMARY KEY (product, quantity);
ALTER TABLE
postgres=# \d sales
                    Table "public.sales"
  Column  |          Type          | Collation | Nullable | Default
----------+------------------------+-----------+----------+---------
 product  | character varying(100) |           | not null |
 category | character varying(100) |           | not null |
 quantity | integer                |           | not null |
 price    | numeric(10,2)          |           | not null |
Indexes:
    "sales_pk" PRIMARY KEY, btree (product, quantity)
```

Рисунок 19. Добавление первичного ключа (PRIMARY KEY)

Добавляем авто инкремент (SERIAL), который обеспечивает автоматическую идентификацию новых строк:

```
postgres=# ALTER TABLE sales ADD COLUMN id SERIAL PRIMARY KEY NOT NULL;
ALTER TABLE
postgres=# \d sales
                                Table "public.sales"
  Column  |          Type          | Collation | Nullable |              Default
----------+------------------------+-----------+----------+-----------------------------------
 product  | character varying(100) |           | not null |
 category | character varying(100) |           | not null |
 quantity | integer                |           | not null |
 price    | numeric(10,2)          |           | not null |
 id       | integer                |           | not null | nextval('sales_id_seq'::regclass)
Indexes:
    "sales_pkey" PRIMARY KEY, btree (id)
```

Рисунок 20. Добавление авто инкремента (SERIAL)

Теперь удаляем первичный ключ с помощью DROP CONSTRAINT:

```
postgres=# ALTER TABLE sales ALTER COLUMN id DROP DEFAULT;
ALTER TABLE
postgres=# \d sales;
                      Table "public.sales"
  Column  |          Type          | Collation | Nullable | Default
----------+------------------------+-----------+----------+---------
 product  | character varying(100) |           | not null |
 category | character varying(100) |           | not null |
 quantity | integer                |           | not null |
 price    | numeric(10,2)          |           | not null |
 id       | integer                |           | not null |
Indexes:
    "sales_pkey" PRIMARY KEY, btree (id)


postgres=# ALTER TABLE sales DROP CONSTRAINT sales_pkey;
ALTER TABLE
postgres=# \d sales
                      Table "public.sales"
  Column  |          Type          | Collation | Nullable | Default
----------+------------------------+-----------+----------+---------
 product  | character varying(100) |           | not null |
 category | character varying(100) |           | not null |
 quantity | integer                |           | not null |
 price    | numeric(10,2)          |           | not null |
 id       | integer                |           | not null |
```

Рисунок 21. Удаление первичного ключа

7. Самостоятельно разобрать типы данных DATETIME, DATE и TIMESTAMP. Привести примеры.

Тип DATE используется для хранения только даты (год, месяц, день) без времени. Формат: YYYY-MM-DD (год-месяц-день).

```
postgres=# CREATE TABLE events(
postgres(# event_id SERIAL PRIMARY KEY,
postgres(# event_name VARCHAR(100) NOT NULL,
postgres(# event_date DATE NOT NULL);
CREATE TABLE
postgres=# INSERT INTO events (event_name, event_date) VALUES
postgres-# ('Day of Gubkin', '2025-08-12'),
postgres-# ('New Year', '2025-12-31');
INSERT 0 2
postgres=# SELECT * FROM events;
 event_id |  event_name   | event_date
----------+---------------+------------
        1 | Day of Gubkin | 2025-08-12
        2 | New Year      | 2025-12-31
(2 rows)
```

Рисунок 22. Тип DATE

Тип TIME используется для хранения только времени (часы, минуты, секунды) без даты. Формат: HH:MI:SS (часы:минуты:секунды).

```
postgres=# ALTER TABLE events ADD COLUMN time TIME;
ALTER TABLE
postgres=# INSERT INTO events(event_name, event_date, time) VALUES
postgres-# ('Morning Meeting', '2025-03-20', '09:00:00');
INSERT 0 1
postgres=# SELECT * FROM events;
 event_id |   event_name    | event_date |   time
----------+-----------------+------------+----------
        1 | Day of Gubkin   | 2025-08-12 |
        2 | New Year        | 2025-12-31 |
        3 | Morning Meeting | 2025-03-20 | 09:00:00
(3 rows)
```

Рисунок 23. Тип Time

Тип TIMESTAMP используется для хранения даты и времени вместе. Формат: YYYY-MM-DD HH:MI:SS (год-месяц-день часы:минуты:секунды).

```
postgres=# ALTER TABLE events ADD COLUMN timestamp TIMESTAMP;
ALTER TABLE
postgres=# INSERT INTO events(event_name, event_date, time, timestamp) VALUES
postgres-# ('Morning Meeting', '2025-03-20', '09:00:00', '2025-03-20 09:00:00');
INSERT 0 1
postgres=# SELECT * FROM events;
 event_id |   event_name    | event_date |   time   |      timestamp
----------+-----------------+------------+----------+---------------------
        1 | Day of Gubkin   | 2025-08-12 |          |
        2 | New Year        | 2025-12-31 |          |
        3 | Morning Meeting | 2025-03-20 | 09:00:00 |
        4 | Morning Meeting | 2025-03-20 | 09:00:00 | 2025-03-20 09:00:00
(4 rows)
```

Рисунок 24. Тип TIMESTAMP

8. Обратить внимание на приложение зарезервированных слов.
9. Для своих таблиц создать несколько внешних ключей, создать индекс, воспользоваться возможностями каскадного удаления и обновления.

Для начала создадим две таблицы и настроим внешний ключ:

```
postgres=# CREATE TABLE orders (
postgres(# order_id SERIAL PRIMARY KEY,
postgres(# customer_name VARCHAR(100) NOT NULL,
postgres(# order_date DATE NOT NULL);
CREATE TABLE
postgres=# CREATE TABLE order_items (
postgres(# item_id SERIAL PRIMARY KEY,
postgres(# order_id INT NOT NULL,
postgres(# product_name VARCHAR(100) NOT NULL,
postgres(# quantity INT NOT NULL,
postgres(# price INT NOT NULL,
postgres(# CONSTRAINT fk_order FOREIGN KEY (order_id) REFERENCES orders(order_id));
CREATE TABLE
```

Рисунок 25. Создание двух таблиц и настройка внешнего ключа

Добавим данные в таблицы:

```
postgres=# INSERT INTO orders (customer_name, order_date) VALUES
postgres-# ('Alice', '2023-10-01'),
postgres-# ('Bob', '2023-10-02');
INSERT 0 2
postgres=# INSERT INTO order_items (order_id, product_name, quantity, price) VALUES
postgres-# (1, 'Laptop', 1, 1000.00),
postgres-# (1, 'Mouse', 2, 25.00),
postgres-# (2, 'Keyboard', 1, 50.00),
postgres-# (2, 'Monitor', 1, 300.00);
INSERT 0 4
postgres=# SELECT * FROM orders;
 order_id | customer_name | order_date
----------+---------------+------------
        1 | Alice         | 2023-10-01
        2 | Bob           | 2023-10-02
(2 rows)


postgres=# SELECT * FROM order_items;
 item_id | order_id | product_name | quantity | price
---------+----------+--------------+----------+-------
       1 |        1 | Laptop       |        1 |  1000
       2 |        1 | Mouse        |        2 |    25
       3 |        2 | Keyboard     |        1 |    50
       4 |        2 | Monitor      |        1 |   300
(4 rows)
```

Рисунок 26. Добавление данных в таблицы

Попытка удалить данные из таблицы с заказами вызывает ошибку базы данных:

```
postgres=# DELETE FROM orders WHERE customer_name = 'Bob';
ERROR:  update or delete on table "orders" violates foreign key constraint "fk_order" on table "order_items"
PџPћP”P PћP‘PќPћPЎPўP◙:  Key (order_id)=(2) is still referenced from table "order_items".
```

Рисунок 27. Ошибка при попытке удаления данных

Рассмотрим возможность каскадного удаления, то есть удаления всех связанных данных при удалении строки из таблицы.
Сначала удалим прошлый внешний ключ:

```
postgres=# ALTER TABLE order_items DROP CONSTRAINT fk_order;
ALTER TABLE
postgres=# \d order_items
                                    Table "public.order_items"
    Column    |          Type          | Collation | Nullable |                  Default
--------------+------------------------+-----------+----------+-------------------------------------------
 item_id      | integer                |           | not null | nextval('order_items_item_id_seq'::regclass)
 order_id     | integer                |           | not null |
 product_name | character varying(100) |           | not null |
 quantity     | integer                |           | not null |
 price        | integer                |           | not null |
Indexes:
    "order_items_pkey" PRIMARY KEY, btree (item_id)
```

Рисунок 28. Удаление внешнего ключа

Добавляем возможность каскадного удаления ON DELETE и ON UPDATE:

```
postgres=# ALTER TABLE order_items
postgres-# ADD CONSTRAINT fk_order
postgres-# FOREIGN KEY (order_id)
postgres-# REFERENCES orders(order_id)
postgres-# ON DELETE CASCADE ON UPDATE CASCADE;
ALTER TABLE
postgres=# \d order_items
                                    Table "public.order_items"
    Column    |          Type          | Collation | Nullable |                  Default
--------------+------------------------+-----------+----------+-------------------------------------------
 item_id      | integer                |           | not null | nextval('order_items_item_id_seq'::regclass)
 order_id     | integer                |           | not null |
 product_name | character varying(100) |           | not null |
 quantity     | integer                |           | not null |
 price        | integer                |           | not null |
Indexes:
    "order_items_pkey" PRIMARY KEY, btree (item_id)
Foreign-key constraints:
    "fk_order" FOREIGN KEY (order_id) REFERENCES orders(order_id) ON UPDATE CASCADE ON DELETE CASCADE
```

Рисунок 29. Создание внешнего ключа с возможностью каскадного удаления

Теперь удалим одну из строк таблицы «orders» – например, с порядковым номером 2:

```
postgres=# DELETE FROM orders WHERE order_id = 2;
DELETE 1
postgres=# SELECT * FROM orders;
 order_id | customer_name | order_date
----------+---------------+------------
        1 | Alice         | 2023-10-01
(1 row)
```

Рисунок 30. Удаление одной из строк «orders»

При этом удалятся все записи с order_id = 2 в связанной дочерней таблице:

```
postgres=# SELECT * FROM order_items;
 item_id | order_id | product_name | quantity | price
---------+----------+--------------+----------+-------
       2 |        1 | Mouse        |        2 |    25
(1 row)
```

Рисунок 31. Проверка дочерней таблицы после удаления строки

Обновим порядковый номер строки:

```
postgres=# UPDATE orders SET order_id = 5 WHERE order_id = 1;
UPDATE 1
postgres=# SELECT * FROM orders;
 order_id | customer_name | order_date
----------+---------------+------------
        5 | Alice         | 2023-10-01
(1 row)
```

Рисунок 32. Обновление номера строки

Посмотрим изменения в дочерней таблице:

```
postgres=# SELECT * FROM order_items;
 item_id | order_id | product_name | quantity | price
---------+----------+--------------+----------+-------
       2 |        5 | Mouse        |        2 |    25
(1 row)
```

Рисунок 33. Изменения в дочерней таблице после обновления номера строки

Все секции с порядковым номером 1 изменились на 5.

Для ON DELETE и ON UPDATE при создании внешних ключей кроме CASCADE также доступны действия:

- RESTRICT – при попытке удаления или обновления родительской записи, содержащей связанные дочерние записи, вызовется ошибка, операция удаления или обновления не будет выполнена;
- SET NULL – при удалении или обновлении родительской записи все связанные дочерние записи будут обновлены так, чтобы столбец, содержащий внешний ключ в дочерней таблице, принял значение NULL;
- NO ACTION – равнозначно RESTRICT;
- SET DEFAULT – при удалении или обновлении родительской записи все связанные дочерние записи будут обновлены так, чтобы столбец, содержащий внешний ключ в дочерней таблице, принял значение, указанное по умолчанию.

Для ускорения поиска по столбцу «product_name» в таблице «order_items» создадим индекс:

```
postgres=# CREATE INDEX idx_product_name ON order_items (product_name);
CREATE INDEX
postgres=# \d order_items
                                    Table "public.order_items"
    Column    |          Type          | Collation | Nullable |                  Default
--------------+------------------------+-----------+----------+-------------------------------------------
 item_id      | integer                |           | not null | nextval('order_items_item_id_seq'::regclass)
 order_id     | integer                |           | not null |
 product_name | character varying(100) |           | not null |
 quantity     | integer                |           | not null |
 price        | integer                |           | not null |
Indexes:
    "order_items_pkey" PRIMARY KEY, btree (item_id)
    "idx_product_name" btree (product_name)
Foreign-key constraints:
    "fk_order" FOREIGN KEY (order_id) REFERENCES orders(order_id) ON UPDATE CASCADE ON DELETE CASCADE
```

Рисунок 34. Создание и проверка индекса

Индексы могут использоваться для поиска записей по уникальному идентификатору, для фильтрации записей по критериям, для сортировки данных, операций соединения и уникальных ограничений.

10. Для своих таблиц выполнить 10 разных запросов к нескольким таблицам, используя различные типы соединений.

Для демонстрации различных типов соединений создадим две таблицы: students (студенты) и courses (курсы):

```
postgres=# CREATE TABLE students(
postgres(# student_id SERIAL PRIMARY KEY,
postgres(# name VARCHAR(100) NOT NULL,
postgres(# age INT NOT NULL);
CREATE TABLE
postgres=# CREATE TABLE courses (
postgres(# course_id SERIAL PRIMARY KEY,
postgres(# course_name VARCHAR(100) NOT NULL,
postgres(# student_id INT,
postgres(# CONSTRAINT fk_student
postgres(# FOREIGN KEY (student_id)
postgres(# REFERENCES students(student_id)
postgres(# ON DELETE SET NULL);
CREATE TABLE
postgres=# \d students
                                    Table "public.students"
   Column   |          Type          | Collation | Nullable |                   Default
------------+------------------------+-----------+----------+----------------------------------------------
 student_id | integer                |           | not null | nextval('students_student_id_seq'::regclass)
 name       | character varying(100) |           | not null |
 age        | integer                |           | not null |
Indexes:
    "students_pkey" PRIMARY KEY, btree (student_id)
Referenced by:
    TABLE "courses" CONSTRAINT "fk_student" FOREIGN KEY (student_id) REFERENCES students(student_id) ON DELETE SET NULL


postgres=# \d courses
                                     Table "public.courses"
   Column    |          Type          | Collation | Nullable |                   Default
-------------+------------------------+-----------+----------+-----------------------------------------------
 course_id   | integer                |           | not null | nextval('courses_course_id_seq'::regclass)
 course_name | character varying(100) |           | not null |
 student_id  | integer                |           |          |
Indexes:
    "courses_pkey" PRIMARY KEY, btree (course_id)
Foreign-key constraints:
    "fk_student" FOREIGN KEY (student_id) REFERENCES students(student_id) ON DELETE SET NULL
```

Рисунок 35. Создание таблиц для запросов с разными типами соединения

Добавим данные в таблицы:

```
postgres=# INSERT INTO students (name, age) VALUES
postgres-# ('Alice', 20),
postgres-# ('Bob', 22),
postgres-# ('Charlie', 21);
INSERT 0 3
postgres=# INSERT INTO courses (course_name, student_id) VALUES
postgres-# ('Mathematics', 1),
postgres-# ('Physics', 1),
postgres-# ('Chemistry', 2),
postgres-# ('Biology', NULL);
INSERT 0 4
postgres=# SELECT * FROM students;
 student_id |  name   | age
------------+---------+-----
          1 | Alice   |  20
          2 | Bob     |  22
          3 | Charlie |  21
(3 rows)


postgres=# SELECT * FROM courses;
 course_id | course_name | student_id
-----------+-------------+------------
         1 | Mathematics |          1
         2 | Physics     |          1
         3 | Chemistry   |          2
         4 | Biology     |
(4 rows)
```

Рисунок 36. Добавление данных в таблицы

Осуществим декартово произведение, то есть все возможные перестановки строк из таблиц со студентами и предметами (CROSS JOIN):

```
postgres=# SELECT s.name, c.course_name
postgres-# FROM students s
postgres-# CROSS JOIN courses c;
  name   | course_name
---------+-------------
 Alice   | Mathematics
 Alice   | Physics
 Alice   | Chemistry
 Alice   | Biology
 Bob     | Mathematics
 Bob     | Physics
 Bob     | Chemistry
 Bob     | Biology
 Charlie | Mathematics
 Charlie | Physics
 Charlie | Chemistry
 Charlie | Biology
(12 rows)
```

Рисунок 37. Декартово произведение таблиц

Объединим таблицы так, чтобы отображались только строки, где есть совпадения в обеих таблицах (INNER JOIN):

```
postgres=# SELECT s.name, c.course_name
postgres-# FROM students s
postgres-# INNER JOIN courses c ON s.student_id = c.student_id;
 name  | course_name
-------+-------------
 Alice | Mathematics
 Alice | Physics
 Bob   | Chemistry
(3 rows)
```

Рисунок 38. Внутреннее объединение

Объединим таблицы так, чтобы отображались все строки из левой таблицы и соответствующие строки из правой таблицы. Если совпадений нет, отобразится NULL для правой таблицы. (LEFT JOIN):

```
postgres=# SELECT s.name, c.course_name
postgres-# FROM students s
postgres-# LEFT JOIN courses c ON s.student_id = c.student_id;
  name   | course_name
---------+-------------
 Alice   | Mathematics
 Alice   | Physics
 Bob     | Chemistry
 Charlie |
(4 rows)
```

Рисунок 39. Левое объединение

Объединим таблицы так, чтобы отображались все строки из правой таблицы и соответствующие строки из левой таблицы. Если совпадений нет, отобразится NULL для левой таблицы. (RIGHT JOIN):

```
postgres=# SELECT s.name, c.course_name
postgres-# FROM students s
postgres-# RIGHT JOIN courses c ON s.student_id = c.student_id;
 name  | course_name
-------+-------------
 Alice | Mathematics
 Alice | Physics
 Bob   | Chemistry
       | Biology
(4 rows)
```

Рисунок 40. Правое объединение

Объединим таблицы так, чтобы отображались все строки из обеих таблиц. Если совпадений нет, отобразится NULL для недостающих данных. (FULL JOIN):

```
postgres=# SELECT s.name, c.course_name
postgres-# FROM students s
postgres-# FULL JOIN courses c ON s.student_id = c.student_id;
  name   | course_name
---------+-------------
 Alice   | Mathematics
 Alice   | Physics
 Bob     | Chemistry
         | Biology
 Charlie |
(5 rows)
```

Рисунок 41. Полное объединение

11. Для своих таблиц выполнить 2 разных запроса к нескольким таблицам, используя оператор объединения множеств.

С помощью UNION и UNION ALL объединим таблицы:

```
postgres=# SELECT name AS data FROM students
postgres-# UNION
postgres-# SELECT course_name AS data FROM courses;
    data
-------------
 Mathematics
 Physics
 Bob
 Chemistry
 Charlie
 Biology
 Alice
(7 rows)
```

Рисунок 42. Объединение множеств с помощью UNION

```
postgres=# SELECT name AS data FROM students
postgres-# UNION ALL
postgres-# SELECT course_name AS data FROM courses;
    data
-------------
 Alice
 Bob
 Charlie
 Mathematics
 Physics
 Chemistry
 Biology
(7 rows)
```

Рисунок 43. Объединение множеств с помощью UNION ALL

12. Используя официальную документацию PostgreSQL, выполнить произвольные запросы, содержащие функции для работы с данными:

Используем функции SQRT() (квадратный корень) и ROUND() (округление):

```
postgres=# SELECT price, SQRT(price), ROUND(SQRT(price), 2) FROM sales;
  price  |        sqrt        | round
---------+--------------------+-------
 1000.00 | 31.622776601683793 | 31.62
  800.00 | 28.284271247461901 | 28.28
  600.00 | 24.494897427831781 | 24.49
   50.00 |  7.071067811865475 |  7.07
   70.00 |  8.366600265340755 |  8.37
 1000.00 | 31.622776601683793 | 31.62
(6 rows)
```

Рисунок 44. Вычисление и округления корня

Используем функции CONCAT() (объединение строк) и UPPER() (преобразование в верхний регистр):

```
postgres=# SELECT name, course_name, CONCAT(name, ' - ', course_name), UPPER(name)
postgres-# FROM students
postgres-# JOIN courses ON students.student_id = courses.student_id;
 name  | course_name |       concat        | upper
-------+-------------+---------------------+-------
 Alice | Mathematics | Alice - Mathematics | ALICE
 Alice | Physics     | Alice - Physics     | ALICE
 Bob   | Chemistry   | Bob - Chemistry     | BOB
(3 rows)
```

Рисунок 45. Объединение строк и перевод в верхний регистр