ПРАКТИЧЕСКАЯ РАБОТА №1. ОСНОВЫ DDL И ЗАПРОСЫ НА ВЫБОРКУ ДАННЫХ В POSTGRES PRO

Цель работы:

Целью данной практической работы является формирование и закрепление у студентов фундаментальных навыков работы с реляционными базами данных на примере СУБД Postgres Pro. По завершении работы студент должен уметь:

- Сформировать практический навык определения структуры базы данных с использованием языка определения данных DDL (Data Definition Language).
- Научиться применять ограничения целостности (PRIMARY KEY, FOREIGN KEY, CHECK, UNIQUE, NOT NULL) для реализации бизнес-правил и обеспечения консистентности (согласованности) данных, основываясь на теоретических положениях реляционной модели.
- Освоить составление SQL-запросов на выборку данных с использованием расширенного синтаксиса инструкции SELECT, включая выражения в списке выборки, псевдонимы и фильтрацию дубликатов с помощью DISTINCT.
- Развить умение применять разнообразные условия фильтрации записей в предложении WHERE, охватывая логические операции, проверку принадлежности диапазону и множеству, сравнение с шаблоном и корректную проверку на NULL.
- Получить базовые навыки агрегации данных с использованием GROUP BY и агрегатных функций, а также научиться корректно фильтровать агрегированные результаты с помощью предложения HAVING.

Постановка задачи:

Для выполнения практической работы необходимо последовательно выполнить следующие шаги, **основываясь на логической модели данных**, которая была спроектирована в рамках курса «Проектирование баз данных» **в предыдущем семестре**:

1. На основе логической модели данных, созданной в прошлом семестре, письменно описать не менее **5 различных бизнес-правил** и не менее **3 ограничений целостности** для таблиц. Выбор бизнес-правил и ограничений целостности производится на усмотрение студента. Результаты представить в виде таблицы.

2. С использованием DDL-оператора CREATE TABLE создать **все необходимые таблицы** *(согласно созданной в прошлом семестре логической модели данных)* в СУБД Postgres Pro, корректно реализовав все описанные **ограничения целостности**.

3. Заполнить созданные таблицы согласованными тестовыми данными (**не менее 5-7 записей на таблицу**, где это применимо) с помощью оператора INSERT INTO.

4. Составить и выполнить не менее **6 SQL-запросов** к таблицам, иллюстрирующих использование различных операторов SELECT и WHERE, согласно перечню, указанному в задании *(см. Ход выполнения работы)*. В составленных запросах должны быть **использованы все приведённые примеры –** .

5. Составить и выполнить **по два SQL-запроса** к таблицам для демонстрации работы предложений ORDER BY, GROUP BY и HAVING.

6. Каждый SQL-запрос **сопроводить комментарием**, объясняющим его назначение и логику работы.

ХОД ВЫПОЛНЕНИЯ РАБОТЫ

1. Анализ и описание ограничений целостности

Первый этап работы заключается в формализации бизнес-правил вашей предметной области в виде ограничений целостности данных. Ограничения — это правила, которые СУБД автоматически применяет к данным, чтобы гарантировать их точность, надежность и консистентность. Этот механизм является практической реализацией целостной части реляционной модели данных.

Для каждой таблицы из вашей логической модели необходимо составить описание ее атрибутов и применяемых к ним ограничений. Результат следует оформить в виде таблицы.

Таблица 1. Пример описания ограничений для таблицы medicines (лекарства)

Название столбца	**Тип данных**	**Ограничение**	**Обоснование (Бизнес-правило)**
id	SERIAL	PRIMARY KEY	Уникальный идентификатор лекарства, генерируется автоматически.
name	VARCHAR(255)	NOT NULL	Название лекарства является обязательным и не может быть пустым.
quantity_in_stock	INT	NOT NULL, CHECK (quantity_in_stock >= 0), DEFAULT 0	Количество на складе обязательно, не может быть отрицательным. По умолчанию равно 0.
price	DECIMAL(10, 2)	NOT NULL, CHECK (price > 0)	Цена является обязательной и должна быть положительным числом.
production_date	DATE	NOT NULL, CHECK (production_date <= CURRENT_DATE)	Дата производства обязательна и не может быть в будущем.
expiration_date	DATE	CHECK (expiration_date > CURRENT_DATE)	Срок годности, если он указан, должен быть в будущем.
manufacturer_id	INTEGER	FOREIGN KEY (manufacturers)	Ссылка на производителя. Лекарство не может существовать без связи с компанией-производителем.

2. Создание структуры данных

На этом этапе вы преобразуете спроектированные на Шаге 1 правила в SQL-код. Для создания таблиц используется оператор *CREATE TABLE*.

При создании таблиц, связанных внешними ключами, **порядок их создания** имеет **критическое значение**. В первую очередь необходимо создать «родительские» таблицы (в данном случае – таблицу manufacturers), а затем «дочерние» (в данном случае – таблицу medicines).

Если вы попытаетесь создать таблицу medicines первой, СУБД (Postgres Pro, MySQL, и т.д.) выдаст ошибку. Она увидит, что вы пытаетесь создать ограничение (CONSTRAINT), которое ссылается на несуществующую таблицу manufacturers. Операция будет прервана.

Листинг 1. Пример создания таблицы manufacturers и medicines

```
-- В начале создаем таблицу производителей (manufacturers), на которую будем ссылаться
CREATE TABLE manufacturers (
-- SERIAL автоматически создает последовательность и делает столбец NOT NULL
    manufacturer_id SERIAL PRIMARY KEY,
    manufacturer_name VARCHAR(255) NOT NULL UNIQUE,
    country VARCHAR(100)
);
-- Создаем таблицу лекарств (medicines) с различными типами ограничений
CREATE TABLE medicines (
    -- Ограничения на уровне столбца
    -- PRIMARY KEY = UNIQUE + NOT NULL. Главный идентификатор записи.
    id SERIAL PRIMARY KEY,
     -- NOT NULL: столбец не может быть пустым.
    name VARCHAR(255) NOT NULL,
     -- DEFAULT: значение по умолчанию, если не указано при вставке.
    quantity_in_stock INT NOT NULL DEFAULT 0,
    price DECIMAL(10, 2) NOT NULL,
    production_date DATE NOT NULL,
     -- Этот столбец может быть NULL, если у лекарства нет срока годности.
    expiration_date DATE,
    manufacturer_id INTEGER NOT NULL,
    -- Ограничения на уровне таблицы. Они используются, когда ограничение затрагивает несколько столбцов
    -- или для улучшения читаемости кода.
    -- CHECK: проверяет, что значение удовлетворяет логическому выражению.
```

```
    CONSTRAINT check_quantity_positive CHECK (quantity_in_stock >= 0),
    CONSTRAINT check_price_positive CHECK (price > 0),
    CONSTRAINT check_production_date CHECK (production_date <= CURRENT_DATE),
    CONSTRAINT check_expiration_date CHECK (expiration_date > production_date),
    -- FOREIGN KEY: обеспечивает ссылочную целостность.
    -- Значение в manufacturer_id должно существовать в столбце manufacturer_id таблицы manufacturers.
    CONSTRAINT fk_manufacturer
        FOREIGN KEY(manufacturer_id)
        REFERENCES manufacturers(manufacturer_id)
    -- RESTRICT (или NO ACTION) запрещает удалять производителя, если на него ссылаются лекарства.
    ON DELETE RESTRICT
);
```

3. Заполнение таблиц данными (DML – Data Manipulation Language)

После создания структуры, базу данных необходимо наполнить тестовыми данными с помощью оператора INSERT INTO.

При работе с таблицами, связанными внешними ключами, **порядок вставки данных** также имеет **критическое значение**. В первую очередь заполняются родительские таблицы (manufacturers), а затем дочерние (medicines).

Листинг 2. Пример корректного порядка вставки

```
-- 1. Сначала вставляем данные в родительскую таблицу manufacturers
INSERT INTO
    manufacturers (manufacturer_name, country)
VALUES ('ООО "Фармстандарт"', 'Россия');
INSERT INTO
    manufacturers (manufacturer_name, country)
VALUES ('Bayer AG', 'Германия');
-- 2. Теперь можно вставлять данные в дочернюю таблицу medicines, ссылаясь на существующие
manufacturer_id
-- manufacturer_id=1 соответствует 'ООО "Фармстандарт"'
INSERT INTO
    medicines (name, quantity_in_stock, price, production_date, expiration_date, manufacturer_id)
VALUES ('Парацетамол', 200, 50.50, '2025-07-10', '2028-07-10', 1);
```

Проверим, как поведёт себя Postgres Pro, если ввести данные, нарушающие ограничения, которые мы задали при создании таблиц (пункт 2). Попытаемся добавить лекарство с отрицательным количеством, что нарушает ограничение CHECK (quantity_in_stock >= 0).

Листинг 3. Пример вставки, нарушающей ограничение

```
INSERT INTO
  medicines (name, quantity_in_stock, price, production_date, expiration_date, manufacturer_id)
VALUES ('Аспирин', -10, 120.00, '2025-07-12', '2027-07-12', 2);
```

Postgres Pro прервет выполнение этого запроса и вернет ошибку, подобную следующей:

```
ERROR: new row for relation "medicines" violates check constraint "check_quantity_positive"
```

4. Составление запросов на выборку (часть 1)

Этот раздел посвящен выполнению четвертого пункта задания, в рамках которого необходимо составить не менее шести запросов, демонстрируя различные возможности SELECT и WHERE.

4.1 Элементы списка выборки – SELECT

4.1.1 Выбор всех столбцов (*)

Оператор * является сокращением, которое позволяет выбрать все столбцы из таблицы без их явного перечисления. Это удобно для быстрой проверки содержимого таблицы, однако при разработке крайне рекомендуется перечислять все нужные столбцы.

Листинг 4. Выбор всех столбцов

```
-- Вывести всю информацию обо всех лекарствах
SELECT * FROM medicines;
```

4.1.2 Выбор конкретных полей и использование псевдонима (AS)

Для повышения читаемости отчетов и результатов запроса можно выбирать только необходимые столбцы и переименовывать их с помощью ключевого слова AS. Псевдонимы, заключенные в двойные кавычки (например, "Цена, руб."), могут содержать пробелы и становятся регистро-зависимыми.

Листинг 5. Использование псевдонима

```
-- Вывести название и цену каждого лекарства, переименовав столбцы для отчета
SELECT
    name AS "Название лекарства",
    price AS "Цена, руб."
FROM
    medicines;
```

4.1.3 Выражение в списке выборки

Прямо в списке полей в SELECT можно выполнять различные операции над данными. Это позволяет переложить часть логики обработки данных на СУБД, что часто бывает эффективнее, чем выполнять те же вычисления в коде приложения. В данном примере для каждой строки вычисляется общая стоимость запаса лекарства (total_value).

Листинг 6. Использование выражения

```
-- Посчитать общую стоимость каждого вида лекарств на складе (цена * количество)
SELECT
    name,
    price,
    quantity_in_stock,
    price * quantity_in_stock AS total_value
FROM
    medicines;
```

4.1.4 Удаление дубликатов (DISTINCT)

Ключевое слово DISTINCT анализирует весь результирующий набор и исключает из него полностью идентичные строки. Этот запрос вернет только уникальные названия стран. Следует помнить, что операция DISTINCT требует от СУБД дополнительных ресурсов на сортировку или группировку данных, что может замедлить выполнение запроса на очень больших объемах данных.

Листинг 7. Удаление дубликатов

```
-- Вывести список уникальных стран-производителей
SELECT DISTINCT
    country
FROM
    manufacturers;
```

4.2 Условия фильтрации – WHERE

4.2.1 Простое условие и логические связки (AND, OR)

Предложение WHERE фильтрует строки до того, как они попадут в результирующий набор.

Учтите, что логический оператор AND имеет более высокий приоритет, чем OR, поэтому при сложных условиях рекомендуется использовать скобки () для явного указания порядка вычислений. В данном случае в результат попадут только те лекарства, которые удовлетворяют обоим условиям одновременно.

Листинг 8. Простое условие и логические связки (AND, OR)

```
-- Найти все лекарства из России, цена которых меньше 50 рублей
SELECT
    med.name,
    med.price
FROM
    medicines AS med
JOIN
    manufacturers AS man
ON
    med.manufacturer_id = man.manufacturer_id
WHERE
    man.country = 'Россия' AND med.price < 50;
```

4.2.2 Проверка принадлежности диапазону (BETWEEN)

Оператор BETWEEN – это удобный и более читаемый способ проверить, попадает ли значение в диапазон [], т.е. включающий крайние элементы (например, price >= 30 AND price <= 100). Он работает не только с числами, но и с датами, и со строками (в лексикографическом порядке).

Листинг 9. Проверка принадлежности диапазону (BETWEEN)

```
-- Найти лекарства, цена которых находится в диапазоне от 30 до 100 рублей включительно
SELECT
    name,
    price
FROM
    medicines
WHERE
    price BETWEEN 30 AND 100;
```

4.2.3 Проверка вхождения в множество (IN)

Оператор IN является синтаксически более удобной и часто более производительной альтернативой нескольким условиям OR. Он проверяет, совпадает ли значение столбца с любым из значений в предоставленном списке. Для обратной операции существует оператор NOT IN.

Листинг 10. Проверка вхождения в множество (IN)

```
-- Найти лекарства с названиями 'Аспирин', 'Цитрамон' или 'Валидол'
SELECT
  name,
  price
FROM
  medicines
WHERE
  name IN ('Аспирин', 'Цитрамон', 'Валидол');
```

4.2.4 Сравнение с шаблоном (LIKE)

Оператор LIKE используется для гибкого поиска по строкам. Кроме символа % (любое количество любых символов), существует также символ _ (ровно один любой символ). Важно помнить, что поиск по шаблону, который начинается с % (например, «%ол»), как правило, не может использовать индекс по этому столбцу, что приводит к полному сканированию таблицы и может быть очень медленным.

Листинг 11. Сравнение с шаблоном (LIKE)

```
-- Найти все лекарства, в названии которых есть 'ол'
SELECT
  name
FROM
  medicines
WHERE
  name LIKE '%ол%';
```

4.2.5 Проверка на NULL

В SQL действует трехзначная логика (TRUE, FALSE, UNKNOWN). Любая операция сравнения со значением NULL (например, = NULL или <> NULL) дает результат UNKNOWN.

Предложение WHERE отбирает только те строки, для которых условие истинно (TRUE), поэтому строки с результатом UNKNOWN отбрасываются. Именно поэтому для корректной проверки на отсутствие значения всегда следует использовать конструкции IS NULL или IS NOT NULL.

Листинг 12. Проверка на NULL

```
-- Найти все лекарства, у которых не указан срок годности
SELECT
  name
FROM
  medicines
WHERE
  expiration_date IS NULL;
```

5. Составление запросов на выборку (часть 2)

Этот раздел посвящен выполнению пятого пункта задания и охватывает более сложные концепции: сортировку, агрегацию и фильтрацию групп.

5.1. Сортировка результатов (ORDER BY)

Предложение ORDER BY применяется к финальному набору данных, полученному после всех фильтраций и группировок, и упорядочивает его строки. Это одна из последних операций в логическом конвейере выполнения запроса. Сортировка может быть ресурсоемкой операцией, особенно на больших объемах данных, если для столбцов сортировки отсутствуют подходящие индексы.

Есть два способа сортировки данных:

- ASC - по возрастанию (значение по умолчанию)
- DESC - по убыванию.

Листинг 13.

```
-- Вывести все лекарства, отсортированные по цене в порядке убывания
SELECT
  name,
  price
FROM
```

```
    medicines
ORDER BY
    price DESC;
```

Листинг 14.

```
-- Вывести всех производителей, отсортированных сначала по стране, а затем по названию
SELECT
    country,
    manufacturer_name
FROM
    manufacturers
ORDER BY
    country ASC,
    manufacturer_name ASC;
```

5.2 Группировка и агрегатные функции (GROUP BY)

Предложение GROUP BY «схлопывает» несколько строк с одинаковыми значениями в указанных столбцах в одну сводную строку. Оно практически всегда используется в паре с агрегатными функциями (COUNT, SUM, AVG, MIN, MAX), которые выполняют вычисления для каждой такой группы.

«Золотое правило» GROUP BY: любой столбец, указанный в SELECT, должен либо быть частью GROUP BY, либо использоваться внутри агрегатной функции. Это логическая необходимость: если сгруппировать лекарства по производителю и попытаться выбрать name (название лекарства), СУБД не будет знать, название какого из десятков лекарств этого производителя следует отобразить. Агрегатные функции решают эту неоднозначность, сводя множество значений к одному (например, к их количеству COUNT(name)).

Листинг 15.

```
-- Посчитать количество наименований лекарств для каждого производителя
SELECT
    manufacturer_id,
    COUNT(*) AS number_of_medicines
FROM
    medicines
GROUP BY
    manufacturer_id;
```

Листинг 16.

```
-- Найти среднюю, минимальную и максимальную цену лекарств для каждого производителя
SELECT
  manufacturer_id,
  AVG(price) AS average_price,
  MIN(price) AS min_price,
  MAX(price) AS max_price
FROM
  medicines
GROUP BY
  manufacturer_id;
```

5.3 Фильтрация групп (HAVING)

Предложение HAVING очень похоже на WHERE, но с одним ключевым отличием в логике работы:

- WHERE фильтрует **отдельные строки** **до** того, как они будут сгруппированы.
- HAVING фильтрует **целые группы** **после** того, как они были сформированы и для них были вычислены агрегатные значения.

Именно поэтому в HAVING можно использовать агрегатные функции (такие как COUNT(*) или AVG(price)), а в WHERE — нельзя, так как на момент работы WHERE никаких групп еще не существует.

Листинг 17.

```
-- Пример 1: Найти производителей, у которых в ассортименте более 3 наименований лекарств
SELECT
  manufacturer_id,
  COUNT(*) AS number_of_medicines
FROM
  medicines
GROUP BY
  manufacturer_id
HAVING
  COUNT(*) > 3;
```

Листинг 18.

```
-- Пример 2: Вывести только тех производителей, средняя цена лекарств которых превышает 100 рублей
SELECT
    manufacturer_id,
    AVG(price) AS average_price
FROM
    medicines
GROUP BY
    manufacturer_id
HAVING
    AVG(price) > 100;
```

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. В чем заключается разница между ограничениями PRIMARY KEY и UNIQUE? Оба ли они допускают значения NULL?
2. Опишите, что произойдет при попытке удалить запись из главной («родительской») таблицы, если для внешнего ключа в дочерней таблице установлено правило ON DELETE CASCADE.
3. Объясните логический порядок выполнения SQL-запроса с предложениями FROM, WHERE, GROUP BY, HAVING, SELECT, ORDER BY.
4. Почему, исходя из этого порядка, нельзя использовать псевдоним столбца, заданный в SELECT, в предложении WHERE?
5. Сформулируйте «золотое правило» GROUP BY. Объясните, почему это правило является логической необходимостью.
6. Приведите пример запроса для вашей базы данных, в котором необходимо использовать и WHERE, и HAVING. Поясните, какую роль выполняет каждое из этих предложений в вашем примере.

КРАТКИЙ СПРАВОЧНЫЙ МАТЕРИАЛ

1. Определение структуры данных (DDL) и ограничения целостности

Краткое описание: команды языка определения данных (DDL), в частности CREATE TABLE, используются для создания структуры базы данных.

1.1 Ограничения целостности

Ограничения целостности – это правила, которые применяются к столбцам для обеспечения точности, надёжности и согласованности данных.

- **PRIMARY KEY** *(первичный ключ)*: однозначно идентифицирует каждую запись в таблице. Неявно включает ограничения **UNIQUE** *(уникальность)* и **NOT NULL** *(непустое значение)*.
- **UNIQUE** *(уникальность)*: гарантирует, что все значения в столбце являются уникальными. В отличие от PRIMARY KEY, допускает значения NULL (любое их количество, т.к. каждое NULL считается уникальным).
- **NOT NULL** *(непустое значение)*: гарантирует, что столбец не может содержать NULL-значения.
- **FOREIGN KEY** *(внешний ключ)*: связывает две таблицы, обеспечивая ссылочную целостность. Значение в столбце дочерней таблицы должно существовать в первичном ключе родительской таблицы. Позволяет определить поведение при удалении связанной записи (подробнее ниже).
- **CHECK** *(проверка)*: гарантирует, что все значения в столбце удовлетворяют определённому логическому условию (например, price > 0).
- **DEFAULT** *(значение по умолчанию)*: задаёт значение, которое будет вставлено в столбец, если оно не указано явно в команде INSERT.

При определении внешнего ключа (**FOREIGN KEY**), правило **ON DELETE** указывает, что должно произойти **с зависимыми** (дочерними) записями при попытке **удалить основную** (родительскую) запись, на которую они ссылаются. Это механизм для поддержания ссылочной целостности базы данных.

- **ON DELETE RESTRICT** *(ограничить)* и **NO ACTION** *(без действия)* – это самое безопасное и используемое по умолчанию поведение. СУБД запрещает удаление строки из родительской таблицы, если на неё ссылается хотя бы одна строка в дочерней таблице. Операция удаления прерывается, и база данных возвращает ошибку.
  В Postgres Pro RESTRICT и NO ACTION работают практически идентично (разница возникнет только при работе с транзакциями).

- **ON DELETE CASCADE** *(каскадное удаление)* – это самый мощный, но и самый опасный вариант. При удалении строки из родительской таблицы, СУБД автоматически удаляет все строки из дочерней таблицы, которые на неё ссылались. Это запускает "цепную реакцию", и можно вызвать эффект домино.
  Применять следует **крайне** осторожно и только если точно понимаете, что делаете.

- **ON DELETE SET NULL** *(установить NULL)* – при удалении родительской записи, в дочерних записях в столбце внешнего ключа значение заменяется на NULL. Важное условие: столбец внешнего ключа в дочерней таблице **не должен** иметь ограничение NOT NULL.

1.2 Синтаксис DDL-операторов

Работа с таблицами:

Критически важен порядок создания таблиц: в начале создаются «родительские» таблицы (те, на которые будут ссылаться), а затем – «дочерние».

*Листинг 19 – **создание** таблицы.*

```
CREATE TABLE [IF NOT EXISTS] tableName (
  columnName dataType [PRIMARY KEY | UNIQUE | NOT NULL] [DEFAULT defaultExpression],
  ...
  -- Определение ограничений на уровне таблицы
  [CONSTRAINT pk_name] PRIMARY KEY (column1 [, column2, ...]),
  [CONSTRAINT uq_name] UNIQUE (column1 [, column2, ...]),
  [CONSTRAINT fk_name] FOREIGN KEY (column_fk) REFERENCES parentTable (parent_pk_column)
    [ON UPDATE action] [ON DELETE action],
  [CONSTRAINT chk_constraint_name] CHECK (expression)
);
```

Опция **IF NOT EXISTS** предотвращает ошибку, если таблица с таким именем уже существует.

*Листинг 20 – **переименование** таблицы.*

```
ALTER TABLE currentTableName RENAME TO newTableName;
```

*Листинг 21 – **удаление** таблицы.*

```
-- IF EXISTS (опционально) – предотвращает ошибку, если таблица не существует.
-- CASCADE – автоматически удаляет все зависимые объекты (например,
представления).
-- RESTRICT – запрещает удаление, если от таблицы зависят другие объекты
(поведение по умолчанию).
DROP TABLE [IF EXISTS] tableName [CASCADE | RESTRICT];
```

*Листинг 22 – **очистка** таблицы.*

```
-- Очистка таблицы: удаляет все строки из таблицы.
-- Эта операция работает быстрее удаления и сбрасывает автоинкрементные счетчики
(при использовании RESTART IDENTITY).
-- RESTART IDENTITY (опционально) – сбрасывает последовательности (например,
SERIAL), связанные со столбцами таблицы.
-- CASCADE (опционально) – автоматически очищает таблицы, которые ссылаются на
данную через внешние ключи.
TRUNCATE TABLE tableName [RESTART IDENTITY] [CASCADE];
```

Для изменения счётчиков используется оператор **ALTER SEQUENCE**.

*Листинг 23 – **сброс** счётчика.*

```
-- Сброс (обнуление) счётчика. Следующее значение будет 1.
ALTER SEQUENCE tableName_columnName_seq RESTART WITH 1;
```

*Листинг 24 – **установка значения** счётчика.*

```
-- Установка счётчику конкретного значения. Следующее значение будет 100
ALTER SEQUENCE tableName_columnName_seq RESTART WITH 100;
```

*Листинг 25 – **установка актуального значения** счётчика.*

```
-- Установка значения счётчика так, чтобы следующее значение было на 1 больше
максимального текущего id в таблице.
-- Полезно после ручной вставки данных с явным указанием id.
-- Функция pg_get_serial_sequence('tableName', 'columnName') автоматически
получает имя счётчика.
SELECT setval(pg_get_serial_sequence('tableName', 'id'), (SELECT MAX(id) FROM
tableName));
```

Для изменения существующих таблиц используется оператор **ALTER TABLE**.

Работа со столбцами:

*Листинг 26 – **добавление** столбца*

```
ALTER TABLE tableName
    ADD columnName dataType [PRIMARY KEY | UNIQUE | NOT NULL]
                            [DEFAULT defaultExpression];
```

*Листинг 27 – **изменение** столбца*

```
ALTER TABLE tableName
    ALTER COLUMN columnName
    [TYPE newDataType | SET DEFAULT value | DROP DEFAULT |
     SET NOT NULL | DROP NOT NULL];
```

*Листинг 28 – **переименование** столбца*

```
ALTER TABLE tableName
    RENAME COLUMN oldColumnName TO newColumnName;
```

*Листинг 29 – **удаление** столбца*

```
ALTER TABLE tableName
    DROP COLUMN columnName;
```

Работа с ограничениями:

*Листинг 30 – **добавление** первичного ключа*

```
ALTER TABLE tableName
    ADD CONSTRAINT constrName PRIMARY KEY (column1 [, column2, ...]);
```

*Листинг 31 – **добавление** ограничения на уникальность*

```
ALTER TABLE tableName
    ADD CONSTRAINT constrName UNIQUE (column1 [, column2, ...]);
```

*Листинг 31 – добавление **внешнего ключа***

```
ALTER TABLE tableName
    ADD CONSTRAINT constrName FOREIGN KEY (column_fk)
    REFERENCES other_table (other_column);
```

*Листинг 32 – добавление **проверки***

```
ALTER TABLE tableName
    ADD CONSTRAINT constrName CHECK (expression);
```

*Листинг 33 – **удаление ограничения***

```
ALTER TABLE tableName
    DROP CONSTRAINT [IF EXISTS] constrName [CASCADE | RESTRICT];
```

2. Выборка и фильтрация данных (SELECT и WHERE)

Краткое описание: оператор SELECT используется для извлечения данных из таблиц. Оператор WHERE позволяет фильтровать строки, оставляя в результирующем наборе только те, которые удовлетворяют заданным условиям.

Ключевые элементы SELECT

- **SELECT ***: выбрать все столбцы из таблицы. Удобно для анализа, но в рабочем коде рекомендуется избегать этого в пользу явного перечисления столбцов для повышения читаемости и производительности.
- **AS alias**: присваивает столбцу или выражению временное имя (псевдоним), что делает результат более читаемым.
- **DISTINCT**: удаляет дубликаты из результирующего набора, возвращая только уникальные значения.
- **Выражения в списке выборки**: позволяют выполнять вычисления прямо в запросе (например, price * quantity_in_stock).

Основные операторы WHERE

- **AND**, **OR**: логические операторы для комбинирования нескольких условий фильтрации.

- **BETWEEN ... AND ...**: проверяет принадлежность значения заданному диапазону, включая его границы.
- **IN (...)**: проверяет, входит ли значение в указанный набор (список) значений.
- **LIKE**: сравнивает строку с шаблоном. При этом знак % заменяет любое количество символов, а знак _ — ровно один символ.
- **IS NULL / IS NOT NULL**: корректный способ проверки на наличие или отсутствие значения. Простое сравнение (= NULL) никогда не вернёт истину.

3. Сортировка, группировка и агрегация

Описание: эти предложения позволяют упорядочивать данные, объединять их в группы для вычисления итоговых значений и фильтровать эти группы.

- **ORDER BY**: используется для сортировки результирующего набора данных по одному или нескольким столбцам. Направление сортировки задаётся ключевыми словами
  - ASC (по возрастанию, значение по умолчанию)
  - DESC (по убыванию).
- **GROUP BY** и агрегатные функции: предложение GROUP BY объединяет строки с одинаковыми значениями в указанных столбцах в одну сводную строку. Оно почти всегда используется вместе с агрегатными функциями (COUNT, SUM, AVG, MIN, MAX), которые выполняют вычисления для каждой группы.
- **HAVING**: используется для фильтрации результатов, уже сгруппированных с помощью GROUP BY.

Ключевое отличие от WHERE: WHERE фильтрует отдельные строки до группировки, а HAVING — целые группы после того, как агрегатные функции были вычислены. Поэтому в HAVING можно использовать агрегатные функции, а в WHERE — нельзя.

4. Логический порядок выполнения SELECT-запроса

Описание: SQL-запросы выполняются не в том порядке, в котором они написаны. Понимание этого логического "конвейера" является ключом к написанию сложных запросов.

Последовательность выполнения:

- **FROM**: определение и соединение исходных таблиц.
- **WHERE**: фильтрация отдельных строк.
- **GROUP BY**: группировка строк в наборы.
- **HAVING**: фильтрация целых групп.
- **SELECT**: вычисление и возврат запрошенных столбцов.
- **ORDER BY**: сортировка финального набора строк.

Из этой последовательности следует, например, что псевдоним столбца, заданный на шаге 5 (SELECT), ещё не существует на шаге 2 (WHERE), и поэтому не может быть там использован.