

Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Магнитогорский государственный технический университет им. Г.И. Носова»

Многопрофильный колледж

**МЕТОДИЧЕСКИЕ УКАЗАНИЯ
ДЛЯ ЛАБОРАТОРНЫХ ЗАНЯТИЙ
по ПМ. 02 АДМИНИСТРИРОВАНИЕ БАЗ ДАННЫХ**

МДК.02.01 Технология разработки и защиты баз данных

**для обучающихся специальности
09.02.13 Интеграция решений с применением технологий искусственного интеллекта**

СОДЕРЖАНИЕ

1 ВВЕДЕНИЕ	3
2 МЕТОДИЧЕСКИЕ УКАЗАНИЯ	4
Лабораторное занятие №1	4
Лабораторное занятие №2	6
Лабораторное занятие №4	9
Лабораторное занятие №5	11
Лабораторное занятие №6	13
Лабораторное занятие №7	16
Лабораторное занятие №8	19
Лабораторное занятие №9	26
Лабораторное занятие №10	29
Лабораторное занятие №11	31
Лабораторное занятие №12	35
Лабораторное занятие №13	39
Лабораторное занятие №14	42
Лабораторное занятие №15	45
Лабораторное занятие №16	49
Лабораторное занятие №17	52
Лабораторное занятие №18	55
Лабораторное занятие №19	60
Лабораторное занятие №20	62
Лабораторное занятие №21	66
Лабораторное занятие №22	68
Лабораторное занятие №23	70
Лабораторное занятие №24	73
Лабораторное занятие №25	76
Лабораторное занятие №26	79
Лабораторное занятие №27	82
Лабораторное занятие №28	87
Лабораторное занятие №29	88

1 ВВЕДЕНИЕ

Важную часть теоретической и профессиональной практической подготовки обучающихся составляют лабораторные занятия.

Состав и содержание лабораторных занятий направлены на реализацию Федерального государственного образовательного стандарта среднего профессионального образования.

Ведущей дидактической целью лабораторных занятий является экспериментальное подтверждение и проверка существенных теоретических положений (законов, зависимостей).

В соответствии с рабочей программой профессионального модуля ПМ.02 «Администрирование баз данных» МДК 02.01 «Технология разработки и защиты баз данных» предусмотрено проведение лабораторных занятий.

В результате их выполнения, обучающийся должен:

уметь:

У 2.2.1 Осуществлять основные функции по администрированию баз данных;

У 2.2.2 Настраивать политики безопасности при работе с сервером баз данных

У 2.3.1 Дать независимую оценку уровня безопасности

У 2.3.2 Производить регламентное обновление программного обеспечения;

У 2.3.3 Разрабатывать перечень рекомендаций по дальнейшей эксплуатации БД с максимальной защитой хранящейся информации;

У 2.5.1 Добавлять, удалять и изменять данные в базе данных;

У 2.5.2 Производить операции по импорту и экспорту данных в различных форматах.

Содержание лабораторных занятий ориентировано на освоение вида деятельности программы подготовки специалистов среднего звена по специальности и овладению ***профессиональными компетенциями:***

ПК 2.2 Осуществлять процедуры администрирования баз данных.

ПК 2.3 Проводить аудит систем безопасности баз данных с использованием регламентов по защите информации.

ПК 2.5 Подготавливать данные для базы знаний.

А также формированию общих компетенций:

ОК 01 Выбирать способы решения задач профессиональной деятельности применительно к различным контекстам.

ОК 02 Использовать современные средства поиска, анализа и интерпретации информации и информационные технологии для выполнения задач профессиональной деятельности.

Лабораторные занятия проводятся в рамках соответствующей темы, после освоения дидактических единиц, которые обеспечивают наличие знаний, необходимых для ее выполнения.

2 МЕТОДИЧЕСКИЕ УКАЗАНИЯ

МДК 02.01 ТЕХНОЛОГИЯ РАЗРАБОТКИ И ЗАЩИТЫ БАЗ ДАННЫХ

Тема 1.1 Основы хранения и обработки данных. Проектирование базы данных

Лабораторное занятие №1

Создание концептуальной модели базы данных с использованием диаграммы "сущность-связь" (ER-диаграмма)

Цель работы: освоить принципы проектирования баз данных, разработав концептуальную модель базы данных с использованием диаграммы "сущность-связь" (ER-диаграммы).

Выполнив работу, вы будете уметь:

У 2.5.1 Добавлять, удалять и изменять данные в базе данных;

Материальное обеспечение:

LibreOffice Draw, Draw io

Задание:

В соответствии со своим вариантом проанализируйте предметную область решаемой задачи и разработайте концептуальную модель соответствующей базы данных с использованием диаграммы "сущность-связь" (ER-диаграммы).

Варианты заданий:

Вариант	Наименование базы данных, описание
1	<i>БД «Управление запасами (Inventory Management)».</i> Описание: Система для отслеживания и управления запасами товаров на складе. Включает информацию о товарах, их количестве, местоположении на складе, поставщиках и заказах. Позволяет автоматизировать процессы пополнения запасов и учета движения товаров.
2	<i>БД «Система управления клиентами (Customer Relationship Management, CRM)».</i> Описание: База данных для хранения информации о клиентах, их взаимодействии с компанией, истории покупок и предпочтениях. Позволяет анализировать данные для повышения уровня обслуживания клиентов, разработки маркетинговых стратегий и улучшения продаж.
3	<i>БД «Образовательная платформа (Educational Platform)».</i> Описание: Система для управления учебным процессом, включающая информацию о курсах, студентах, преподавателях и оценках. Позволяет отслеживать успеваемость студентов, организовывать расписание занятий и управлять материалами курса.
4	<i>БД «Управление проектами (Project Management)».</i> Описание: База данных для планирования и отслеживания выполнения проектов. Включает информацию о задачах, сроках, участниках проекта и ресурсах. Позволяет анализировать эффективность работы команды и управлять рисками.
5	<i>БД «Электронная коммерция (E-commerce)».</i> Описание: Система для управления онлайн-продажами, включая информацию о товарах, пользователях, заказах и платежах. Обеспечивает функционал для обработки заказов, управления корзиной покупок и анализа продаж.
6	<i>БД «Система бронирования (Booking System)».</i> Описание: База данных для управления процессом бронирования услуг (например, отелей, авиабилетов или мероприятий). Включает информацию о

	доступных ресурсах, клиентах, бронированиях и платежах. Позволяет оптимизировать использование ресурсов и улучшить клиентский сервис.
7	БД «Здравоохранение (Healthcare)». Описание: Система для хранения медицинских записей пациентов, истории болезней, назначений и результатов анализов. Обеспечивает доступ к информации для врачей и медицинского персонала, а также позволяет отслеживать лечение и профилактические мероприятия.
8	БД «Управление персоналом (Human Resource Management)». Описание: База данных для учета сотрудников компании, их должностей, заработной платы, рабочего времени и других HR-процессов. Позволяет автоматизировать процессы найма, оценки эффективности и обучения сотрудников.
9	БД «Финансовый учет (Financial Accounting)». Описание: Система для управления финансовыми данными компании, включая бухгалтерский учет, отчеты о доходах и расходах, а также управление бюджетом. Позволяет контролировать финансовое состояние компании и принимать обоснованные решения.
10	БД «Социальные сети (Social Networking)». Описание: База данных для хранения информации о пользователях социальной сети, их профилях, сообщениях, друзьях и взаимодействиях. Позволяет анализировать поведение пользователей и улучшать функционал платформы.

Порядок выполнения работы:

1. Запуск Draw.io

- Перейдите на сайт [[draw.io \(diagrams.net\)](https://draw.io)].
- Выберите место для сохранения файла (на устройстве или в облаке).

2. Создание новой диаграммы

- Выберите "Создать новую диаграмму".
- В открывшемся окне выберите шаблон "Entity Relationship" или начните с пустого холста.

3. Добавление сущностей

- На панели слева выберите фигуру "Прямоугольник" (или специальную форму для сущностей).
- Разместите на холсте несколько сущностей (например, "Студент", "Курс", "Преподаватель").

4. Добавление атрибутов

- Для каждой сущности добавьте овалы или текстовые блоки с атрибутами (например, для "Студент": ID, ФИО, Дата рождения).
- Свяжите атрибуты с сущностями линиями или разместите их внутри прямоугольников.

5. Установка связей

- Используйте стрелки или линии для отображения связей между сущностями (например, "Студент" - "записан на" - "Курс").
- Укажите тип связи (1:1, 1:N, M:N), подписав линии.

6. Оформление и сохранение

- Настройте цвета, размеры, шрифты для лучшей читаемости.
- Сохраните диаграмму в нужном формате (PNG, SVG, XML).

Форма представления результата:

Оформленная концептуальная модель базы данных.

Критерии оценки:

Оценка «отлично» ставится, если задание выполнено верно.

Оценка «хорошо» ставится, если ход выполнения задания верный, но была допущена одна или две ошибки, приведшие к неправильному результату.

Оценка «удовлетворительно» ставится, если приведено неполное выполнение задания.

Оценка «неудовлетворительно» ставится, если задание не выполнено.

Тема 1.1 Основы хранения и обработки данных. Проектирование базы данных

Лабораторное занятие №2

Разработка логической модели базы данных на основе ER-диаграммы

Цель работы: освоить принципы проектирования баз данных, используя ER-диаграммы для создания логической модели базы данных.

Выполнив работу, вы будете уметь:

У 2.5.1 Добавлять, удалять и изменять данные в базе данных;

Материальное обеспечение:

LibreOffice Draw, Draw io

Задание:

В соответствии со своим вариантом проанализируйте предметную область решаемой задачи и разработайте логическую модель на основе ER-диаграммы.

Варианты заданий:

Вариант	Наименование базы данных, описание
1	БД «Лицензионное программное обеспечение». Таблица «Лицензии»: номер лицензии; название; код CD-диска; код владельца. Таблица «CD-диски»: код CD-диска; дата выпуска; вид программного обеспечения; общий объем файлов, кбайт; пояснения о назначении и свойствах программного обеспечения. Таблица «Владельцы»: код владельца; владелец; город; адрес; телефон
2	БД «Студенты МнК». Таблица «Группы»: отделение; группа; Ф.И.О. кл. руководителя. Таблица «Студенты»: группа; шифр студента; Ф.И.О.; адрес; телефон; хобби. Таблица «Дисциплины»: шифр дисциплины; наименование дисциплины. Таблица «Успеваемость»: дата; шифр дисциплины; шифр студента; оценка; отметка о пропуске занятия
3	БД «Гостиница». Таблица «Номерной фонд»: категория номера (люкс, одноместный первой категории, двухместный первой категории и др.); номер помещения; место (А, Б, ... – в зависимости от количества мест в номере); стоимость проживания за сутки. Таблица «Проживание»: дата заезда; дата выезда; номер помещения; место; Ф.И.О.; паспортные данные. Таблица «Бронирование»: дата заявки; код брони; категория номера; количество человек; дата заезда; срок пребывания
4	БД «Товарооборот». Таблица «Поставщики»: код поставщика; поставщик; адрес; телефон. Таблица «Товары»: код товара; товар; единица измерения; цена. Таблица «Поступление товаров»: дата поступления; код поставщика; код товара; количество. Таблица «Продажа»: дата продажи; код товара; количество
5	БД «Промышленность региона». Таблица «Предприятия»: код предприятия; предприятие; форма собственности; адрес; основной вид продукции. Таблица «Виды налогов»: код налога; вид налога (на прибыль, на имущество, НДС и др.); ставка, %. Таблица «Налоговые платежи»: код предприятия; код налога; дата платежа; сумма платежа. Таблица «Прибыль»: год; месяц; код предприятия; прибыль

6	<i>БД «Пассажирские поезда».</i> Таблица «Поезда»: категория поезда (скорый, пассажирский, пригородный); номер поезда; название поезда (для фирменного поезда). Таблица «Составы вагонов»: номер поезда; код состава; общее количество вагонов; схема формирования (например, 3К + 8П – три купейных и восемь плацкартных вагонов). Таблица «Расписание»: номер поезда; время прибытия; время отправления; режим движения (дни следования); станция назначения. Таблица «Перевозки»: дата отправления; номер поезда; код состава; количество пассажиров; прибыль за поездку
7	<i>БД «Автопарк».</i> Таблица «Типы автобусов»: код автобуса; марка автобуса; количество мест. Таблица «Парк»: код автобуса; гаражный номер; государственный номер; год выпуска. Таблица «Водители»: табельный номер водителя; Ф.И.О.; дата рождения; оклад; номер маршрута. Таблица «Перевозки»: дата; код автобуса; номер маршрута; табельный номер водителя; время выхода автобуса на маршрут; время прибытия автобуса с маршрута; причина схода автобуса с маршрута; количество проданных билетов
8	<i>БД «Агентство недвижимости».</i> Таблица «Риэлторы»: код риэлтора; Ф.И.О.; телефон. Таблица «Недвижимость»: номер объекта; адрес; тип дома; общая площадь; количество комнат; наличие балкона; наличие телефона; стоимость. Таблица «Сделки»: дата; регистрационный номер договора; код риэлтора; номер объекта
9	<i>БД «Авиаперевозки».</i> Таблица «Авиапарк»: код модели самолета; модель самолета; количество мест. Таблица «Рейсы и тарифы»: рейс; аэропорт отправления; аэропорт назначения; код класса; вид класса (бизнес-класс, эконом-класс); стоимость билета. Таблица «Перевозки»: рейс; дата вылета; код модели самолета; количество пассажиров; доход за рейс

Порядок выполнения работы:

1. Выполнить анализ предметной области.
2. Выполнить анализ данных.
3. Определить набор атрибутов для данной предметной области.
4. Определить набор таблиц.

При проектировании таблиц рекомендуется руководствоваться следующими основными принципами:

- каждая таблица должна содержать данные только на одну тему;
- данные не должны дублироваться.

5. Создать словарь имен.
6. Определить состав и типы полей.
7. Создать связи между таблицами.
8. Создать схему базы данных в Draw io.

Форма представления результата:

Оформленная логическая схема базы данных.

Критерии оценки:

Оценка «отлично» ставится, если задание выполнено верно.

Оценка «хорошо» ставится, если ход выполнения задания верный, но была допущена одна или две ошибки, приведшие к неправильному результату.

Оценка «удовлетворительно» ставится, если приведено неполное выполнение задания.

Оценка «неудовлетворительно» ставится, если задание не выполнено.

Тема 1.1 Основы хранения и обработки данных. Проектирование базы данных

Лабораторное занятие №3

Нормализация базы данных: приведение таблиц к третьей нормальной форме (3НФ)

Цель работы: освоить процесс нормализации реляционных баз данных, научиться приводить таблицы к третьей нормальной форме (3НФ) для устранения избыточности и предотвращения аномалий при работе с данными.

Выполнив работу, вы будете уметь:

У 2.5.1 Добавлять, удалять и изменять данные в базе данных;

Материальное обеспечение:

MySQL Workbench

Задание:

1. Изучить исходную структуру базы данных в соответствии со своим вариантом задания.
2. Выявить функциональные зависимости между атрибутами.
3. Последовательно привести таблицы к 1НФ, 2НФ и 3НФ.
4. Оформить итоговую структуру базы данных в среде MySQL Workbench.

Порядок выполнения работы:

1. Анализ исходной структуры
 - Ознакомиться с заданной схемой таблиц.
 - Перечислить все атрибуты и их типы.
2. Приведение к первой нормальной форме (1НФ)
 - Проверить, что все атрибуты атомарны (не содержат повторяющихся групп и составных значений).
 - Если есть повторяющиеся группы, выделить их в отдельные таблицы.
3. Приведение ко второй нормальной форме (2НФ)
 - Определить первичные ключи таблиц.
 - Выявить частичные функциональные зависимости (зависимость неключевых атрибутов от части составного ключа).
 - Устранить частичные зависимости, выделив отдельные таблицы.
4. Приведение к третьей нормальной форме (3НФ)
 - Проверить наличие транзитивных зависимостей (неключевой атрибут зависит от другого неключевого атрибута).
 - Устранить транзитивные зависимости, выделив новые таблицы и установив связи между ними.

Форма представления результата:

Оформленная схема базы данных.

Критерии оценки:

Оценка «отлично» ставится, если задание выполнено верно.

Оценка «хорошо» ставится, если ход выполнения задания верный, но была допущена одна или две ошибки, приведшие к неправильному результату.

Оценка «удовлетворительно» ставится, если приведено неполное выполнение задания.

Оценка «неудовлетворительно» ставится, если задание не выполнено.

Тема 1.1 Основы хранения и обработки данных. Проектирование базы данных

Лабораторное занятие №4

Создание базы данных с использованием языка SQL (CREATE DATABASE, CREATE TABLE)

Цель работы: освоить операции создания баз данных.

Выполнив работу, вы будете уметь:

У 2.5.1 Добавлять, удалять и изменять данные в базе данных;

Материальное обеспечение:

PostgreSQL pgAdmin, DBeaver

Задание:

Создать базу данных в СУБД PostgreSQL.

Порядок выполнения работы:

Базу данных создаем одним из способов: с помощью pgAdmin или терминального клиента psql. Проверку создания базы данных и таблиц в ней проводим в терминального клиента psql.

1. Запустите терминальный клиент для работы с PostgreSQL: **SQL shell (psql)**.

Подключитесь к серверу и пройдите авторизацию (пароль - 12345).

2. Создайте новую базу данных с помощью команды: **CREATE DATABASE**. К имени базы данных добавьте аббревиатуру из первых букв своего ФИО.

Например, для Иванова Петра Сидоровича база данных будет иметь имя **storeIPS**:

```
postgres=# CREATE DATABASE storeIPS;
```

```
CREATE DATABASE
```

3. **Проверьте, что ваша база данных создана. Нужно будет сделать скриншоты для каждой таблицы и вставить в отчет.**

1 вариант: выборка из системного каталога **pg_database**:

```
SELECT datname FROM pg_database;
```

2 вариант: метакоманда **\l**

3 вариант: ключ **-l** командной строки приложения **psql**

4. **Создайте таблицы БД согласно ER-модели, указывая ОГРАНИЧЕНИЯ НА УРОВНЕ**

Синтаксис команды CREATE TABLE:

```
CREATE TABLE "имя_таблицы"
```

```
( имя_поля тип_данных [ограничения_целостности],
```

```
имя_поля тип_данных [ограничения_целостности],
```

```
...
```

```
имя_поля тип_данных [ограничения_целостности],
```

```
[ограничение_целостности],
```

```
[первичный_ключ],
```

```
[внешний_ключ]
```

```
);
```

Для получения справки о синтаксисе SQL-команды:

```
\h CREATE TABLE
```

Способы ввода команд

• **Способ 1**

```
demo=# CREATE TABLE aircrafts (aircraft_code char(3) NOT NULL,  
model text NOT NULL, range integer NOT NULL, CHECK (range > 0),
```

PRIMARY KEY (aircraft_code));

• **Способ 2**

```
demo=# CREATE TABLE aircrafts
demo=# (aircraft_code char(3) NOT NULL,
demo=# model text NOT NULL,
demo=# range integer NOT NULL,
demo=# CHECK (range > 0),
demo=# PRIMARY KEY (aircraft_code)
demo=# );
CREATE TABLE
```

Вместо ввода символа «;» команду можно завершить символами \g:

```
demo=# CREATE TABLE aircrafts ... \g
```

Прервать ввод команды можно клавишами Ctrl-C:

```
demo=# CREATE TABLE aircrafts
(aircraft_code char(3) NOT NULL,
demo=# ^C
demo=#
```

5. **Получите описание всех созданных таблиц с помощью метакоманды \d.**

```
\d aircrafts
```

Таблица "public.aircrafts"

Колонка | Тип | Модификаторы

-----+-----+-----

aircraft_code | character(3) | NOT NULL

model | text | NOT NULL

range | integer | NOT NULL

Индексы:

"aircrafts_pkey" PRIMARY KEY, btree (aircraft_code)

Ограничения-проверки:

"aircrafts_range_check" CHECK (range > 0)

Примечания:

public означает имя так называемой схемы.

- Для реализации первичного ключа (PRIMARY KEY) всегда автоматически создается индекс. В данном случае тип индекса — btree, т. е. В-дерево.

Можно задать свои собственные имена для всех ограничений.

Удаление таблицы

Упрощенный синтаксис:

DROP TABLE имя_таблицы;

Например:

DROP TABLE aircrafts;

Форма представления результата:

Отчет по выполненной лабораторной работе.

Критерии оценки:

Оценка «отлично» ставится, если задание выполнено верно.

Оценка «хорошо» ставится, если ход выполнения задания верный, но была допущена одна или две ошибки, приведшие к неправильному результату.

Оценка «удовлетворительно» ставится, если приведено неполное выполнение задания.

Оценка «неудовлетворительно» ставится, если задание не выполнено.

Тема 1.1 Основы хранения и обработки данных. Проектирование базы данных

Лабораторное занятие №5

Определение индексов для оптимизации запросов к базе данных

Цель работы: освоить принципы определения индексов для оптимизации запросов к базе данных.

Выполнив работу, вы будете уметь:

У 2.5.1 Добавлять, удалять и изменять данные в базе данных;

У 2.5.2 Производить операции по импорту и экспорту данных в различных форматах.

Материальное обеспечение:

PostgreSQL pgAdmin, DBeaver

Задание:

В созданной базе данных создать индексы для оптимизации запросов к базе.

Порядок выполнения работы:

1. Подготовка среды

Подключитесь к PostgreSQL через psql:

```
psql -U postgres
```

Создайте новую базу данных:

```
CREATE DATABASE performance_db;  
\c performance_db
```

2. Создание тестовой таблицы

Создадим таблицу users с большим количеством записей:

```
CREATE TABLE users (  
    user_id SERIAL PRIMARY KEY,  
    username VARCHAR(50),  
    email VARCHAR(100),  
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP  
);
```

Заполним её 100 000 строк случайными данными:

```
INSERT INTO users (username, email)  
SELECT  
    'user_' || i,  
    'user' || i || '@example.com'  
FROM generate_series(1, 100000) AS i;
```

3. Анализ производительности без индекса

Выполним медленный запрос:

```
EXPLAIN ANALYZE
```

```
SELECT * FROM users WHERE username = 'user_99999';
```

Вы увидите, что используется Seq Scan — последовательное сканирование всей таблицы. Это неэффективно!

4. Создание индекса

Создадим индекс по полю username:

```
CREATE INDEX idx_username ON users(username);
```

Теперь повторим тот же запрос:

```
EXPLAIN ANALYZE
SELECT * FROM users WHERE username = 'user_99999';
```

Теперь используется Index Scan — гораздо быстрее!

5. Составной индекс

Если часто фильтруете по двум полям, можно создать составной индекс:

```
CREATE INDEX idx_username_email ON users(username, email);
```

Пример использования:

```
EXPLAIN ANALYZE
SELECT * FROM users WHERE username = 'user_99999' AND email =
'user99999@example.com';
```

6. Уникальный индекс

Чтобы обеспечить уникальность значения (например, email), создайте уникальный индекс:

```
CREATE UNIQUE INDEX idx_unique_email ON users(email);
```

Попробуйте добавить дублирующий email:

```
INSERT INTO users (username, email) VALUES ('test',
'user1@example.com');
```

Получите ошибку: duplicate key value violates unique constraint.

7.: Удаление индекса

Если индекс больше не нужен:

```
DROP INDEX idx_username;
```

8. Анализ существующих индексов

Посмотреть все индексы в текущей базе данных:

```
SELECT
    indexname,
    tablename,
    indexdef
FROM pg_indexes
WHERE tablename = 'users';
```

Задание для самостоятельной работы

1. Создайте таблицу orders со следующими полями:
 - order_id,
 - user_id (внешний ключ на users),
 - total_amount,
 - order_date.
2. Добавьте 50 000 записей в таблицу orders.
3. Найдите самые популярные поля для фильтрации (например, order_date, total_amount) и создайте для них индексы.
4. Сравните время выполнения запросов до и после создания индексов:

```
SELECT * FROM orders WHERE order_date BETWEEN '2025-01-01' AND '2025-03-31';
```

5. Создайте составной индекс по полям `user_id` и `order_date`. Протестируйте его работу.

6. Напишите запрос, который использует этот индекс для группировки и агрегации:

```
SELECT user_id, COUNT(*) AS total_orders, SUM(total_amount) AS total_spent
FROM orders
GROUP BY user_id;
```

Форма представления результата:

Отчет по выполненной лабораторной работе.

Критерии оценки:

Оценка «отлично» ставится, если задание выполнено верно.

Оценка «хорошо» ставится, если ход выполнения задания верный, но была допущена одна или две ошибки, приведшие к неправильному результату.

Оценка «удовлетворительно» ставится, если приведено неполное выполнение задания.

Оценка «неудовлетворительно» ставится, если задание не выполнено.

Тема 1.1 Основы хранения и обработки данных. Проектирование базы данных

Лабораторное занятие №6

Проектирование базы данных для хранения данных IoT (Интернет вещей) с учётом особенностей структуры

Цель работы: научиться проектировать и реализовывать реляционную базу данных в PostgreSQL для хранения данных, поступающих от устройств Интернета Вещей (IoT), учитывая особенности временных рядов, масштабируемости и разнообразия типов датчиков.

Выполнив работу, вы будете уметь:

У 2.5.1 Добавлять, удалять и изменять данные в базе данных;

У 2.5.2 Производить операции по импорту и экспорту данных в различных форматах.

Материальное обеспечение:

PostgreSQL pgAdmin, DBeaver

Задание:

Спроектировать и реализовать реляционную базу данных в PostgreSQL для хранения данных, поступающих от устройств Интернета Вещей (IoT).

Порядок выполнения работы:

1. Настройка PostgreSQL

Установка расширения TimescaleDB (опционально)

```
# Ubuntu
```

```
sudo apt install timescaledb-postgresql-15
```

```
# Затем активируйте расширение в нужной БД
```

2. Создание БД и подключение

```
-- Создаем новую базу данных
CREATE DATABASE iot_db;

-- Подключаемся к ней
\c iot_db
```

3. Создание таблиц

Таблица `devices` — информация об устройствах

Хранит метаданные о каждом устройстве.

```
CREATE TABLE devices (
    device_id UUID PRIMARY KEY,
    name VARCHAR(100),
    location VARCHAR(255),
    description TEXT,
    registered_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Комментарий: используем UUID, т.к. устройства могут регистрироваться из разных источников. `location` может быть географическим местоположением или комнатой, где установлено устройство.

Таблица `sensors` — описание сенсоров

Каждое устройство может иметь несколько сенсоров.

```
CREATE TABLE sensors (
    sensor_id SERIAL PRIMARY KEY,
    device_id UUID REFERENCES devices(device_id),
    type VARCHAR(50), -- например, "temperature", "humidity"
    unit VARCHAR(20), -- единицы измерения
    description TEXT
);
```

Комментарий: здесь мы связываем сенсоры с устройствами через внешний ключ `device_id`.

Таблица `sensor_readings` — показания сенсоров

Самый важный элемент — хранение временных рядов.

```
CREATE TABLE sensor_readings (
    reading_id BIGSERIAL PRIMARY KEY,
    sensor_id INT REFERENCES sensors(sensor_id),
    value DOUBLE PRECISION,
    measured_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Комментарий: это нормализованная структура. Каждая запись содержит ссылку на конкретный сенсор и значение, измеренное в момент времени.

4. Использование TimescaleDB (для оптимизации)

Если вы используете TimescaleDB, преобразуйте таблицу `sensor_readings` в гипертаблицу:

```
-- Активируем расширение
CREATE EXTENSION IF NOT EXISTS timescaledb CASCADE;
```

```
-- Преобразуем таблицу в гипертаблицу
```

```
SELECT create_hypertable('sensor_readings', 'measured_at');
```

Теперь PostgreSQL автоматически будет создавать чанки (chunks) по времени, что ускоряет работу с большими объемами временных данных.

5. Заполнение тестовыми данными

Добавим тестовое устройство, сенсор и несколько показаний.

```
-- Вставляем устройство
INSERT INTO devices (device_id, name, location, description)
VALUES ('a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11', 'Weather Station
1', 'Moscow, Red Square', 'Outdoor weather station');

-- Вставляем сенсор
INSERT INTO sensors (device_id, type, unit, description)
VALUES ('a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11', 'temperature',
'°C', 'Ambient temperature sensor');

-- Вставляем показания
INSERT INTO sensor_readings (sensor_id, value, measured_at)
VALUES
(1, 22.5, NOW() - INTERVAL '5 minutes'),
(1, 22.7, NOW() - INTERVAL '4 minutes'),
(1, 23.0, NOW() - INTERVAL '3 minutes'),
(1, 22.8, NOW() - INTERVAL '2 minutes'),
(1, 23.1, NOW() - INTERVAL '1 minute');
```

6. Запросы и анализ

Последние 5 значений температуры:

```
SELECT s.type, sr.value, sr.measured_at
FROM sensor_readings sr
JOIN sensors s ON sr.sensor_id = s.sensor_id
WHERE s.type = 'temperature'
ORDER BY sr.measured_at DESC
LIMIT 5;
```

Среднее значение температуры за последние 10 минут:

```
SELECT AVG(value) AS avg_temp
FROM sensor_readings sr
JOIN sensors s ON sr.sensor_id = s.sensor_id
WHERE s.type = 'temperature'
AND sr.measured_at > NOW() - INTERVAL '10 minutes';
```

Группировка по 5-минутным интервалам:

```
SELECT
    time_bucket('5 minutes', sr.measured_at) AS five_min,
    ROUND(AVG(sr.value), 2) AS avg_value
FROM sensor_readings sr
JOIN sensors s ON sr.sensor_id = s.sensor_id
WHERE s.type = 'temperature'
GROUP BY five_min
ORDER BY five_min;
```

Если используется TimescaleDB, функция `time_bucket()` позволяет удобно агрегировать временные данные.

7. Оптимизация

1. Индексы:

```
CREATE INDEX idx_sensor_id ON sensor_readings(sensor_id);
CREATE INDEX idx_measured_at ON sensor_readings(measured_at);
```

2. Денормализация (для повышения скорости): можно создать представление или материализованное представление с предварительно подготовленными данными.

3. **Партиционирование :** PostgreSQL поддерживает партиционирование по времени или диапазонам, что полезно при работе с очень большими объемами данных.
4. **JSONB поля :** ИНОГДА IoT-устройства передают сложные структуры данных. PostgreSQL позволяет хранить такие данные в JSONB-полях.

Задание для самостоятельной работы

1. Реализуйте аналогичную систему для нескольких типов сенсоров: температура, влажность, давление.
2. Добавьте возможность хранения GPS-координат устройств.

Форма представления результата:

Отчет по выполненной лабораторной работе.

Критерии оценки:

Оценка «отлично» ставится, если задание выполнено верно.

Оценка «хорошо» ставится, если ход выполнения задания верный, но была допущена одна или две ошибки, приведшие к неправильному результату.

Оценка «удовлетворительно» ставится, если приведено неполное выполнение задания.

Оценка «неудовлетворительно» ставится, если задание не выполнено.

Тема 1.2 Разработка и администрирование базы данных

Лабораторное занятие №7

Создание таблиц с использованием языка SQL (CREATE DATABASE, CREATE TABLE)

Цель работы: научиться создавать базы данных и таблицы в PostgreSQL с помощью команд SQL CREATE DATABASE и CREATE TABLE, учитывая типы данных, ограничения и логическую структуру базы данных.

Выполнив работу, вы будете уметь:

У 2.2.1 Осуществлять основные функции по администрированию баз данных;

У 2.5.1 Добавлять, удалять и изменять данные в базе данных;

Материальное обеспечение:

PostgreSQL pgAdmin, DBeaver

Задание:

Создать базу данных и таблицы в PostgreSQL с помощью команд SQL CREATE DATABASE и CREATE TABLE, учитывая типы данных, ограничения и логическую структуру базы данных.

Краткие теоретические сведения:

SQL (Structured Query Language) — это язык запросов, используемый для взаимодействия с реляционными базами данных. Он позволяет создавать, изменять, запрашивать и управлять данными в БД.

Основные команды DDL (Data Definition Language):

- CREATE DATABASE — создание новой базы данных.
- CREATE TABLE — создание новой таблицы.
- ALTER TABLE — изменение существующей таблицы.
- DROP TABLE / DATABASE — удаление таблицы или базы данных.

Команда CREATE DATABASE

Используется для создания новой базы данных:

CREATE DATABASE имя_базы;

Команда CREATE TABLE

Синтаксис:

```
CREATE TABLE имя_таблицы (  
    столбец1 тип_данных ограничения,  
    столбец2 тип_данных ограничения,  
    ...  
);
```

Типы данных в PostgreSQL:

- SERIAL — автоматически увеличивающееся целое число (ID).
- INT — целое число.
- VARCHAR(n) — строка переменной длины до n символов.
- TEXT — текст неограниченной длины.
- DATE — дата.
- TIMESTAMP — дата и время.
- BOOLEAN — логическое значение (TRUE/FALSE).
- DECIMAL(p, s) — десятичное число с точностью p и масштабом s.
- JSONB — бинарный JSON.

Ограничения (Constraints):

- PRIMARY KEY — уникальный идентификатор строки.
- FOREIGN KEY — ссылка на первичный ключ другой таблицы.
- NOT NULL — поле не может быть пустым.
- UNIQUE — все значения в столбце должны быть уникальными.
- CHECK(условие) — проверка условия.
- DEFAULT — значение по умолчанию.

Порядок выполнения работы:

1. Подключение к PostgreSQL

Откройте psql или клиент, например, pgAdmin .

2. Создание новой базы данных

Подключитесь через psql:

```
psql -U postgres
```

Создайте новую базу данных:

```
-- Создаем базу данных
```

```
CREATE DATABASE school_db;
```

```
-- Подключаемся к ней
```

```
\c school_db
```

Теперь все дальнейшие команды будут выполняться в контексте этой БД.

3. Создание таблиц

Пример 1: Таблица students — студенты

```
CREATE TABLE students (  
    student_id SERIAL PRIMARY KEY,  
    first_name VARCHAR(50) NOT NULL,  
    last_name VARCHAR(50) NOT NULL,  
    birth_date DATE,  
    email VARCHAR(100) UNIQUE,  
    is_active BOOLEAN DEFAULT TRUE  
);
```

Комментарий:

- `student_id` — уникальный идентификатор студента.
- `first_name`, `last_name` — обязательные поля.
- `birth_date` — необязательное поле.
- `email` — должен быть уникальным.
- `is_active` — значение по умолчанию `TRUE`.

Пример 2: Таблица `courses` — курсы

```
CREATE TABLE courses (
    course_id SERIAL PRIMARY KEY,
    course_name VARCHAR(100) NOT NULL,
    description TEXT,
    duration_weeks INT CHECK (duration_weeks > 0),
    start_date DATE
);
```

Комментарий:

- Поле `duration_weeks` должно быть положительным числом благодаря `CHECK`.

Пример 3: Таблица `enrollments` — зачисления студентов на курсы

```
CREATE TABLE enrollments (
    enrollment_id SERIAL PRIMARY KEY,
    student_id INT REFERENCES students(student_id),
    course_id INT REFERENCES courses(course_id),
    enrollment_date DATE DEFAULT CURRENT_DATE,
    grade DECIMAL(4, 2) CHECK (grade BETWEEN 0 AND 5)
);
```

Комментарий:

- Здесь используются внешние ключи (`REFERENCES`) для связи между таблицами.
- Оценка (`grade`) должна быть от 0 до 5.

4. Вставка тестовых данных

Добавим несколько записей:

-- Добавляем студентов

```
INSERT INTO students (first_name, last_name, birth_date, email)
VALUES ('Иван', 'Иванов', '2000-05-10', 'ivan@example.com'),
       ('Мария', 'Петрова', '2001-08-22', 'maria@example.com');
```

-- Добавляем курсы

```
INSERT INTO courses (course_name, description, duration_weeks,
start_date)
VALUES ('Введение в программирование', 'Основы программирования
на Python', 12, '2025-02-01'),
       ('Английский для IT', 'Развитие технической грамотности на
английском', 16, '2025-02-05');
```

-- Регистрируем зачисления

```
INSERT INTO enrollments (student_id, course_id)
VALUES (1, 1), (1, 2), (2, 1);
```

5. Выполнение простых запросов

Посмотреть всех студентов:

```
SELECT * FROM students;
```

Посмотреть активных студентов:

```
SELECT first_name, last_name
```

```
FROM students
WHERE is_active = TRUE;
```

Посмотреть оценки студентов по курсам:

```
SELECT s.first_name, s.last_name, c.course_name, e.grade
FROM enrollments e
JOIN students s ON e.student_id = s.student_id
JOIN courses c ON e.course_id = c.course_id;
```

Задание для самостоятельной работы

1. Создайте новую таблицу `teachers` со следующими полями:
 - `teacher_id` — уникальный ID (первичный ключ),
 - `full_name` — ФИО преподавателя,
 - `subject` — предмет,
 - `hire_date` — дата найма.
2. Добавьте связь между `courses` и `teachers` (например, один преподаватель может вести один курс).
3. Создайте таблицу `grades_log`, которая будет хранить историю изменения оценок студентов:
 - `log_id`,
 - `enrollment_id`,
 - `old_grade`,
 - `new_grade`,
 - `changed_at` `TIMESTAMP DEFAULT NOW()`.
4. Напишите триггер, который будет добавлять запись в `grades_log` при обновлении оценки в `enrollments`

Форма представления результата:

Отчет по выполненной лабораторной работе.

Критерии оценки:

Оценка «отлично» ставится, если задание выполнено верно.

Оценка «хорошо» ставится, если ход выполнения задания верный, но была допущена одна или две ошибки, приведшие к неправильному результату.

Оценка «удовлетворительно» ставится, если приведено неполное выполнение задания.

Оценка «неудовлетворительно» ставится, если задание не выполнено.

Тема 1.2 Разработка и администрирование базы данных

Лабораторное занятие №8

Реализация ограничений целостности (PRIMARY KEY, FOREIGN KEY, UNIQUE) в таблицах базы данных

Цель работы: научиться реализовывать и использовать основные механизмы обеспечения целостности данных в PostgreSQL.

Выполнив работу, вы будете уметь:

У 2.2.1 Осуществлять основные функции по администрированию баз данных;

У 2.5.1 Добавлять, удалять и изменять данные в базе данных;

Материальное обеспечение:

PostgreSQL pgAdmin, DBeaver

Задание:

Реализовать и использовать основные механизмы обеспечения целостности данных в PostgreSQL.

Краткие теоретические сведения:

Целостность данных — это обеспечение точности, полноты и согласованности данных в базе данных. Для этого используются ограничения (constraints).

Типы данных сами по себе ограничивают множество данных, которые можно сохранить в таблице. Однако для многих приложений такие ограничения слишком грубые. Например, столбец, содержащий цену продукта, должен, вероятно, принимать только положительные значения. Но такого стандартного типа данных нет. Возможно, необходимо ограничить данные столбца по отношению к другим столбцам или строкам. Например, в таблице с информацией о товаре должна быть только одна строка с определённым кодом товара.

Для решения подобных задач SQL позволяет определять ограничения для столбцов и таблиц. Ограничения дают возможность управлять данными в таблицах так, как нужно. Если пользователь попытается сохранить в столбце значение, нарушающее ограничения, возникнет ошибка. Ограничения будут действовать, даже если это значение по умолчанию.

Ограничения столбцов и таблиц

При определении таблиц и их столбцов в SQL мы можем использовать ряд атрибутов, которые накладывают определенные ограничения. Рассмотрим эти атрибуты.

PRIMARY KEY

С помощью выражения **PRIMARY KEY** столбец можно сделать первичным ключом.

```
CREATE TABLE Customers
(
    Id SERIAL PRIMARY KEY,
    FirstName CHARACTER VARYING(30),
    LastName CHARACTER VARYING(30),
    Email CHARACTER VARYING(30),
    Age INTEGER
)
```

Первичный ключ уникально идентифицирует строку в таблице. В качестве первичного ключа необязательно должны выступать столбцы с типом SERIAL, они могут представлять любой другой тип.

Установка первичного ключа на уровне таблицы:

```
CREATE TABLE Customers
(
    Id SERIAL,
    FirstName CHARACTER VARYING(30),
    LastName CHARACTER VARYING(30),
    Email CHARACTER VARYING(30),
    Age INTEGER,
    PRIMARY KEY(Id)
);
```

Первичный ключ может быть составным (compound key). Такой ключ может потребоваться, если у нас сразу два столбца должны уникально идентифицировать строку в таблице. Например:

```
CREATE TABLE OrderLines
(
    OrderId INTEGER,
    ProductId INTEGER,
    Quantity INTEGER,
    Price MONEY,
    PRIMARY KEY(OrderId, ProductId)
);
```

Здесь поля OrderId и ProductId вместе выступают как составной первичный ключ. То есть в таблице OrderLines не может быть двух строк, где для обоих из этих полей одновременно были бы одни и те же значения.

UNIQUE

Если мы хотим, чтобы столбец имел только уникальные значения, то для него можно определить атрибут **UNIQUE**.

```
CREATE TABLE Customers
(
    Id SERIAL PRIMARY KEY,
    FirstName CHARACTER VARYING(20),
    LastName CHARACTER VARYING(20),
    Email CHARACTER VARYING(30) UNIQUE,
    Phone CHARACTER VARYING(30) UNIQUE,
    Age INTEGER
);
```

В данном случае столбцы, которые представляют электронный адрес и телефон, будут иметь уникальные значения. И мы не сможем добавить в таблицу две строки, у которых значения для этих столбцов будет совпадать.

Также мы можем определить этот атрибут на уровне таблицы:

```
CREATE TABLE Customers
(
    Id SERIAL PRIMARY KEY,
    FirstName CHARACTER VARYING(20),
    LastName CHARACTER VARYING(20),
    Email CHARACTER VARYING(30),
    Phone CHARACTER VARYING(30),
    Age INTEGER,
    UNIQUE(Email, Phone)
);
```

Или так:

```
CREATE TABLE Customers
(
    Id SERIAL PRIMARY KEY,
    FirstName CHARACTER VARYING(20),
    LastName CHARACTER VARYING(20),
    Email CHARACTER VARYING(30),
    Phone CHARACTER VARYING(30),
    Age INTEGER,
    UNIQUE(Email),
    UNIQUE(Phone)
);
```

NULL и NOT NULL

Чтобы указать, может ли столбец принимать значение NULL, при определении столбца ему можно задать атрибут **NULL** или **NOT NULL**. Если этот атрибут явным образом не будет использован, то по умолчанию столбец будет допускать значение NULL. Исключением является тот случай, когда столбец выступает в роли первичного ключа - в этом случае по умолчанию столбец имеет значение NOT NULL.

```
CREATE TABLE Customers
(
    Id SERIAL PRIMARY KEY,
    FirstName CHARACTER VARYING(20) NOT NULL,
    LastName CHARACTER VARYING(20) NOT NULL,
    Age INTEGER
);
```

```
);
```

DEFAULT

Атрибут **DEFAULT** определяет значение по умолчанию для столбца. Если при добавлении данных для столбца не будет предусмотрено значение, то для него будет использоваться значение по умолчанию.

```
CREATE TABLE Customers
(
    Id SERIAL PRIMARY KEY,
    FirstName CHARACTER VARYING(20),
    LastName CHARACTER VARYING(20),
    Age INTEGER DEFAULT 18
);
```

Здесь для столбца Age предусмотрено значение по умолчанию 18.

CHECK

Ключевое слово **CHECK** задает ограничение для диапазона значений, которые могут храниться в столбце. Для этого после слова CHECK указывается в скобках условие, которому должен соответствовать столбец или несколько столбцов. Например, возраст клиентов не может быть меньше 0 или больше 100:

```
CREATE TABLE Customers
(
    Id SERIAL PRIMARY KEY,
    FirstName CHARACTER VARYING(20),
    LastName CHARACTER VARYING(20),
    Age INTEGER DEFAULT 18 CHECK(Age >0 AND Age < 100),
    Email CHARACTER VARYING(30) UNIQUE CHECK(Email != ''),
    Phone CHARACTER VARYING(20) UNIQUE CHECK(Phone != '')
);
```

Здесь также указывается, что столбцы Email и Phone не могут иметь пустую строку в качестве значения (пустая строка **не** эквивалентна значению NULL).

Для соединения условий используется ключевое слово **AND**. Условия можно задать в виде операций сравнения больше (>), меньше (<), не равно (!=).

Также с помощью CHECK можно создать ограничение в целом для таблицы:

```
CREATE TABLE Customers
(
    Id SERIAL PRIMARY KEY,
    Age INTEGER DEFAULT 18,
    FirstName CHARACTER VARYING(20),
    LastName CHARACTER VARYING(20),
    Email CHARACTER VARYING(30) UNIQUE,
    Phone CHARACTER VARYING(20) UNIQUE,
    CHECK((Age >0 AND Age<100) AND (Email != '') AND (Phone != ''))
);
```

Оператор CONSTRAINT. Установка имени ограничений.

С помощью ключевого слова **CONSTRAINT** можно задать имя для ограничений. В качестве ограничений могут использоваться PRIMARY KEY, UNIQUE, CHECK.

Имена ограничений можно задать на уровне столбцов. Они указываются после CONSTRAINT перед атрибутами:

```
CREATE TABLE Customers
(
    Id SERIAL CONSTRAINT customer_Id PRIMARY KEY,
    Age INTEGER CONSTRAINT customers_age_check CHECK(Age >0 AND
Age < 100),
    FirstName CHARACTER VARYING(20) NOT NULL,
```

```

        LastName CHARACTER VARYING(20) NOT NULL,
        Email CHARACTER VARYING(30) CONSTRAINT customers_email_key
UNIQUE,
        Phone CHARACTER VARYING(20) CONSTRAINT customers_phone_key
UNIQUE
    );

```

В принципе необязательно задавать имена ограничений, при установке соответствующих атрибутов SQL Server автоматически определяет их имена. Но, зная имя ограничения, мы можем к нему обращаться, например, для его удаления.

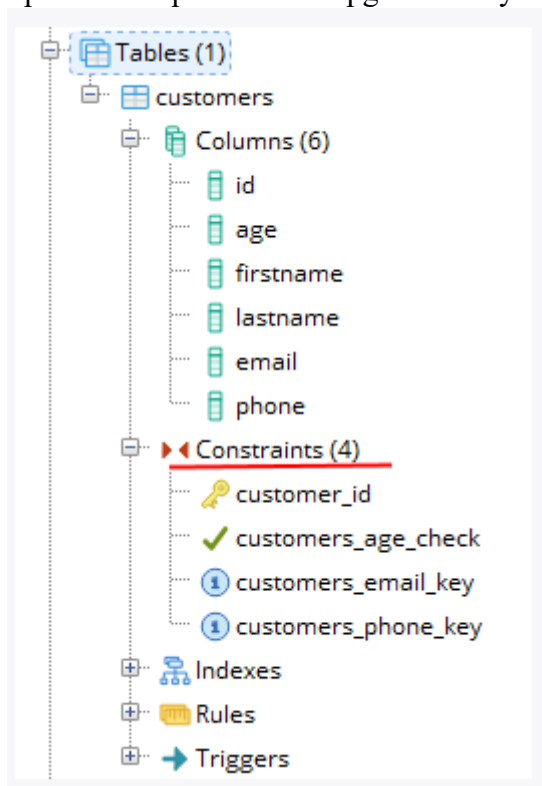
И также можно задать все имена ограничений через атрибуты таблицы:

```

CREATE TABLE Customers
(
    Id SERIAL,
    Age INTEGER,
    FirstName CHARACTER VARYING(20) NOT NULL,
    LastName CHARACTER VARYING(20) NOT NULL,
    Email CHARACTER VARYING(30),
    Phone CHARACTER VARYING(20),
    CONSTRAINT customer_Id PRIMARY KEY(Id),
    CONSTRAINT customers_age_check CHECK(Age >0 AND Age < 100),
    CONSTRAINT customers_email_key UNIQUE(Email),
    CONSTRAINT customers_phone_key UNIQUE(Phone)
);

```

Вне зависимости от того, используется оператор CONSTRAINT для создания ограничений или нет (в этом случае при установке ограничений PostgreSQL сам дает им имена), мы можем посмотреть все ограничения в pgAdmin в узле базы данных в подузле :



Порядок выполнения работы:

1. Подготовка среды

Подключитесь к PostgreSQL через psql:
 psql -U postgres

Создайте новую базу данных:
CREATE DATABASE integrity_db;
\\c integrity_db

2. Создание таблиц с ограничениями

Таблица users — пользователи системы

```
CREATE TABLE users (  
    user_id SERIAL PRIMARY KEY,  
    username VARCHAR(50) UNIQUE NOT NULL,  
    email VARCHAR(100) UNIQUE NOT NULL,  
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP  
);
```

Комментарий:

- user_id — уникальный идентификатор пользователя.
- username и email — должны быть уникальны и не пустыми.
- created_at — автоматически заполняется датой создания.

Таблица orders — заказы пользователей

```
CREATE TABLE orders (  
    order_id SERIAL PRIMARY KEY,  
    user_id INT REFERENCES users(user_id),  
    product_name VARCHAR(100) NOT NULL,  
    amount NUMERIC(10, 2) CHECK (amount > 0),  
    order_date DATE DEFAULT CURRENT_DATE  
);
```

Комментарий:

- user_id — внешний ключ, ссылающийся на таблицу users.
- amount — должно быть положительным числом (CHECK(amount > 0)).

Таблица emails — дополнительная проверка UNIQUE

```
CREATE TABLE emails (  
    id SERIAL PRIMARY KEY,  
    email VARCHAR(100) UNIQUE NOT NULL  
);
```

3. Попытка нарушить ограничения

После создания таблиц попробуем нарушить каждое из ограничений и посмотрим, как отреагирует система.

Нарушение PRIMARY KEY

```
INSERT INTO users (user_id, username, email)  
VALUES (1, 'test_user', 'test@example.com');
```

-- Попытаемся вставить еще одну запись с тем же user_id

```
INSERT INTO users (user_id, username, email)  
VALUES (1, 'another_user', 'another@example.com');
```

● Ошибка: duplicate key value violates unique constraint.

Нарушение FOREIGN KEY

-- Попытаемся создать заказ для несуществующего пользователя

```
INSERT INTO orders (user_id, product_name, amount)  
VALUES (999, 'Laptop', 1200.00);
```

● Ошибка: insert or update on table "orders" violates foreign key constraint.

Нарушение UNIQUE

-- Вставляем email

```
INSERT INTO emails (email) VALUES ('john@example.com');
```

-- Попытка вставить тот же email

```
INSERT INTO emails (email) VALUES ('john@example.com');
```

● Ошибка: duplicate key value violates unique constraint.

Нарушение CHECK

-- Попытка вставить отрицательную сумму

```
INSERT INTO orders (user_id, product_name, amount)
VALUES (1, 'Mouse', -50.00);
```

● Ошибка: new row for relation "orders" violates check constraint.

4. Работа с существующими таблицами

Добавление ограничения после создания таблицы

Можно добавить ограничение UNIQUE к уже существующей таблице:

```
ALTER TABLE users ADD CONSTRAINT unique_username
UNIQUE(username);
```

Удаление ограничения

```
ALTER TABLE users DROP CONSTRAINT unique_username;
```

Переименование ограничения

```
ALTER TABLE users RENAME CONSTRAINT users_email_key TO
unique_email;
```

5. Использование составного PRIMARY KEY

Иногда требуется использовать несколько полей в качестве первичного ключа.

```
CREATE TABLE user_roles (
    user_id INT REFERENCES users(user_id),
    role VARCHAR(50),
    assigned_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (user_id, role)
);
```

Теперь комбинация user_id и role должна быть уникальной.

Задание для самостоятельной работы

1. Создайте таблицу products со следующими полями:

- product_id (первичный ключ),
- name (уникальное, не пустое),
- price (больше нуля),
- category (не пустое).

2. Добавьте внешний ключ в таблицу orders, который связывает её с products.

3. Создайте таблицу employees со следующими полями:

- employee_id,
- full_name,
- position,
- salary (больше 0),
- department_id.

Установите ограничения:

- employee_id — PRIMARY KEY,
- full_name — UNIQUE и NOT NULL,

- salary — CHECK больше 0.

4. Попробуйте вручную нарушить каждое из этих ограничений и зафиксируйте возникающие ошибки.

5. Напишите запросы, которые:

- Выводят всех сотрудников с зарплатой выше средней,
- Выводят количество заказов по каждому продукту.

Форма представления результата:

Отчет по выполненной лабораторной работе.

Критерии оценки:

Оценка «отлично» ставится, если задание выполнено верно.

Оценка «хорошо» ставится, если ход выполнения задания верный, но была допущена одна или две ошибки, приведшие к неправильному результату.

Оценка «удовлетворительно» ставится, если приведено неполное выполнение задания.

Оценка «неудовлетворительно» ставится, если задание не выполнено.

Тема 1.2 Разработка и администрирование базы данных

Лабораторное занятие №9

Написание и выполнение SQL-запросов для добавления, изменения и удаления данных (INSERT, UPDATE, DELETE)

Цель работы: научиться выполнять операции добавления, обновления и удаления записей в таблицах базы данных с помощью команд языка SQL.

Выполнив работу, вы будете уметь:

У 2.2.1 Осуществлять основные функции по администрированию баз данных;

У 2.5.1 Добавлять, удалять и изменять данные в базе данных;

У 2.5.2 Производить операции по импорту и экспорту данных в различных форматах.

Материальное обеспечение:

PostgreSQL pgAdmin, DBeaver

Задание:

В созданной базе данных выполнить запросы добавления, обновления и удаления данных к базе.

Краткие теоретические сведения:

Основные DML-команды SQL:

Команда	Назначение
INSERT	Добавляет новые строки в таблицу
UPDATE	Обновляет существующие строки
DELETE	Удаляет строки из таблицы

1. Команда INSERT

Добавляет новые данные в таблицу.

Синтаксис:

```
INSERT INTO имя_таблицы (столбец1, столбец2, ...)
VALUES (значение1, значение2, ...);
```

Можно добавить несколько строк за раз:

```
INSERT INTO имя_таблицы (столбец1, столбец2)
VALUES
(значение1, значение2),
(значение3, значение4);
```

2. Команда UPDATE

Изменяет значения существующих строк.

Синтаксис:

```
UPDATE имя_таблицы
SET столбец1 = значение1, столбец2 = значение2
WHERE условие;
```

Важно: всегда используйте WHERE, иначе будут изменены все строки в таблице!

3. Команда DELETE

Удаляет строки из таблицы.

Синтаксис:

```
DELETE FROM имя_таблицы
WHERE условие;
```

Важно: без WHERE будут удалены все данные из таблицы.

Порядок выполнения работы:

1. Подготовка среды

Подключитесь к PostgreSQL через psql или клиент (например, pgAdmin):

```
psql -U postgres
```

Создайте новую базу данных и подключитесь к ней:

```
CREATE DATABASE practice_db;
\c practice_db
```

2. Создание тестовой таблицы

Создадим таблицу students:

```
CREATE TABLE students (
    student_id SERIAL PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    age INT CHECK (age >= 16),
    email VARCHAR(100) UNIQUE
);
```

3. Добавление данных — INSERT

Добавим несколько студентов:

```
INSERT INTO students (first_name, last_name, age, email)
VALUES
('Иван', 'Иванов', 20, 'ivan@example.com'),
('Мария', 'Петрова', 21, 'maria@example.com'),
('Алексей', 'Смирнов', 19, 'alex@example.com');
```

Проверим результат:

```
SELECT * FROM students;
```

4. Изменение данных — UPDATE

Обновим возраст одного студента:

```
UPDATE students
SET age = 22
WHERE first_name = 'Иван' AND last_name = 'Иванов';
```

Теперь обновим email:

```
UPDATE students
SET email = 'ivan_new@example.com'
WHERE student_id = 1;
```

5. Удаление данных — DELETE

Удалим студента по ID:

```
DELETE FROM students
WHERE student_id = 3;
```

Попробуем удалить всех студентов старше 21 года (для примера):

```
DELETE FROM students
WHERE age > 21;
```

6. Работа с несколькими таблицами (связанные данные)

Создадим таблицу courses:

```
CREATE TABLE courses (
    course_id SERIAL PRIMARY KEY,
    course_name VARCHAR(100) NOT NULL
);
```

Создадим связь через внешний ключ:

```
CREATE TABLE enrollments (
    enrollment_id SERIAL PRIMARY KEY,
    student_id INT REFERENCES students(student_id),
    course_id INT REFERENCES courses(course_id),
    enrollment_date DATE DEFAULT CURRENT_DATE
);
```

Добавим курсы:

```
INSERT INTO courses (course_name)
VALUES ('Программирование'), ('Английский язык');
```

Запишем студентов на курсы:

```
INSERT INTO enrollments (student_id, course_id)
VALUES (1, 1), (2, 2);
```

Обновим дату зачисления:

```
UPDATE enrollments
SET enrollment_date = '2025-03-01'
WHERE enrollment_id = 1;
```

Удалим запись о зачислении:

```
DELETE FROM enrollments
WHERE enrollment_id = 2;
```

Задание для самостоятельной работы

1. Создайте таблицу `employees` со следующими полями:
 - `employee_id`,
 - `full_name`,
 - `position`,
 - `salary`.Добавьте туда не менее 5 записей.
2. Обновите должность одного сотрудника и увеличьте его зарплату на 10%.
3. Удалите сотрудника с самой высокой зарплатой.
4. Создайте таблицу `departments` и свяжите её с `employees` через внешний ключ `department_id`.
5. Напишите запросы:
 - Вывести всех сотрудников с зарплатой выше средней.
 - Обновить зарплату всем сотрудникам отдела "IT" на 15%.
 - Удалить всех сотрудников, чья зарплата меньше 30 000.

Форма представления результата:

Отчет по выполненной лабораторной работе.

Критерии оценки:

Оценка «отлично» ставится, если задание выполнено верно.

Оценка «хорошо» ставится, если ход выполнения задания верный, но была допущена одна или две ошибки, приведшие к неправильному результату.

Оценка «удовлетворительно» ставится, если приведено неполное выполнение задания.

Оценка «неудовлетворительно» ставится, если задание не выполнено.

Тема 1.2 Разработка и администрирование базы данных

Лабораторное занятие №10

Настройка индексов для оптимизации производительности запросов (CREATE INDEX)

Цель работы: познакомиться с механизмами ускорения работы SQL-запросов в PostgreSQL через использование индексов; научиться создавать, анализировать и использовать индексы при работе с большими объемами данных.

Выполнив работу, вы будете уметь:

У 2.2.1 Осуществлять основные функции по администрированию баз данных;

У 2.5.1 Добавлять, удалять и изменять данные в базе данных;

Материальное обеспечение:

PostgreSQL pgAdmin, DBeaver

Задание:

В созданной базе данных создать индексы для оптимизации запросов к базе.

Краткие теоретические сведения:

Типы индексов в PostgreSQL

B-tree: наиболее распространенный тип индекса, используемый по умолчанию. Подходит для большинства случаев. Используется для равенств и диапазонов (=,>,<,BETWEEN).

Hash: используется для быстрого поиска по равенству.

GIN (Generalized Inverted Index): подходит для массивов и полнотекстового поиска.

GiST (Generalized Search Tree): используется для сложных типов данных, таких как геометрические.

BRIN: для больших таблиц с упорядоченными данными.

Когда создавать индекс?

- при частых запросах на выборку по конкретным полям (например, WHERE, JOIN);
- при необходимости обеспечить уникальность (UNIQUE);
- при использовании внешних ключей;
- при группировке (GROUP BY) или сортировке (ORDER BY).

Возможные проблемы:

- индексы занимают место на диске;
- замедляют операции вставки, обновления и удаления;
- не всегда эффективны на маленьких таблицах;
- избыточные индексы могут ухудшить производительность.

Порядок выполнения работы:

Задание 1. Создание индексов по столбцам таблицы

1. Создайте индекс для ускорения поиска по наименованию категории.
2. Создайте индекс для ускорения поиска по клиентам. Индекс будет **составным** и будет включать в себя фамилию, имя и отчество клиента:
 - для фамилии сортировка по умолчанию, NULLS расположены в начале списка;
 - для имени сортировка по убыванию, NULLS расположены в конце списка;
 - для отчества сортировка по убыванию, NULLS расположены в начале списка.
3. Создайте индекс для ускорения поиска по цене товара с **сортировкой по возрастанию**, NULLS расположены в начале списка.

Задание 2. Создание индексов, используя выражения

1. Создайте индекс по общей стоимости товаров на складе с **сортировкой по убыванию**, NULLS расположены в конце списка (таблица **products**).
2. Создайте индекс по поиску клиента по полному ФИО (фамилия, имя и отчество разделяются пробелами) с **сортировкой по возрастанию**, NULLS расположены в конце списка. Для создания значения для индекса используйте:
 - конкатенацию строк - функции **concat()** или **concat_ws()** или оператор ||;
 - для обработки возможного значения **NULL** в поле **patronymic** (отчество) используйте функцию **coalesce()** - возвращает первое ненулевое значение из своих аргументов;
 - для удаления лишних пробелов слева и справа используйте функцию **trim()**.

Пример:

(trim(" || last_name || ' ' || first_name || ' ' || coalesce(patronymic, '')))

3. Создайте индекс по поиску клиента по телефону без каких-либо дополнительных символов кроме цифр (например, 89128971815 или 79128971815). Для того чтобы избавиться от “лишних” символов, используйте функцию **replace()**.

Задание 3. Создание уникальных индексов

1. Создайте индекс, который будет проверять уникальность наименований категорий товаров без учета регистра.
2. Создайте индекс, который будет проверять уникальность наименований товаров без учета регистра.

Задание 4. Сбор статистики

Сделав запрос к служебной таблице **pg_indexes**, выведите информацию о всех индексах в вашей базе данных (во всех схемах). Сделайте **сортировку по имени таблицы и названию индекса** (приложите скриншот результата).

Задание 5. Удаление индекса

1. Удалите любой индекс, созданный в задании 2.

Приведите пример запроса, который теперь будет выполняться без использования индекса.

Форма представления результата:

Отчет по выполненной лабораторной работе.

Критерии оценки:

Оценка «отлично» ставится, если задание выполнено верно.

Оценка «хорошо» ставится, если ход выполнения задания верный, но была допущена одна или две ошибки, приведшие к неправильному результату.

Оценка «удовлетворительно» ставится, если приведено неполное выполнение задания.

Оценка «неудовлетворительно» ставится, если задание не выполнено.

Тема 1.2 Разработка и администрирование базы данных

Лабораторное занятие №11

Оптимизация запросов к базе данных с использованием индексов и анализа плана выполнения запросов

Цель работы: научиться анализировать производительность SQL-запросов в PostgreSQL с помощью EXPLAIN ANALYZE, выявлять узкие места и оптимизировать их с помощью создания индексов.

Выполнив работу, вы будете уметь:

У 2.2.1 Осуществлять основные функции по администрированию баз данных;

У 2.5.1 Добавлять, удалять и изменять данные в базе данных;

Материальное обеспечение:

PostgreSQL pgAdmin, DBeaver

Задание:

В созданной базе данных создать индексы для оптимизации запросов к базе.

Краткие теоретические сведения:

Оптимизация запросов к базе данных — это процесс улучшения производительности базы данных путем повышения эффективности выполнения SQL-запросов и минимизации нагрузки на систему. Цель оптимизации состоит в сокращении времени отклика на запросы, уменьшении потребления ресурсов сервера баз данных и повышении общей пропускной способности системы.

Основные направления оптимизации запросов:

1. Индексация

Индексы позволяют быстро находить нужные данные в таблице, значительно ускоряя выполнение запросов SELECT, UPDATE и DELETE. Типичные типы индексов включают B-tree, хеш-индексы и полнотекстовые индексы.

Пример: создать индекс по полю name таблицы users.

```
CREATE INDEX idx_users_name ON users(name);
```

2. Нормализация таблиц

Нормализация позволяет избежать избыточности данных и улучшает целостность данных. Однако чрезмерная нормализация может привести к увеличению количества JOIN'ов, замедляя производительность сложных запросов.

Пример нормализации: разделение одной большой таблицы на две меньшие, содержащие взаимосвязанные данные.

3. Выбор правильного типа данных

Использование наиболее подходящего типа данных для каждого поля помогает уменьшить объем хранимых данных и ускорить операции чтения/записи.

Пример: использовать тип INT вместо VARCHAR для численных значений.

4. Использование EXPLAIN ANALYZE

Команда EXPLAIN ANALYZE показывает, как именно выполняется запрос и сколько времени занимает каждая операция. Это полезно для выявления узких мест и неэффективных участков.

Пример: проверка плана выполнения запроса.

```
EXPLAIN ANALYZE SELECT * FROM users WHERE name = 'Иван';
```

Вывод может содержать такие элементы:

Seq Scan – последовательное сканирование таблицы;

Index Scan – использование индекса;

Rows – оценочное количество строк;

Width – размер строки в байтах;

Actual time – время выполнения (в мс).

5. Кэширование результатов

Кэширование часто используемых результатов запросов снижает нагрузку на базу данных и ускоряет доступ к данным.

Пример: настройка кэширования на уровне приложения или использование встроенных механизмов кэша базы данных (например, Redis).

6. Ограничение объема выборки

Ограничивайте количество возвращаемых записей с помощью LIMIT и OFFSET. Если приложение отображает данные постранично, используйте соответствующие конструкции для ограничения числа строк.

Пример: выбрать первые 10 пользователей.

```
SELECT * FROM users ORDER BY id LIMIT 10;
```

7. Избегайте операций с большими объёмами данных

Избегайте сортировки больших объемов данных на стороне базы данных. Если возможно, производите предварительную фильтрацию и агрегацию данных перед передачей клиенту.

Пример: минимизировать передачу данных через сеть.

```
SELECT COUNT(*) FROM users WHERE active = true;
```

8. Регулярное обновление статистики

Для эффективной работы индекса и оптимального выбора планов выполнения запросов важно регулярно обновлять статистику распределения данных в таблицах.

Пример: обновление статистики PostgreSQL.

```
ANALYZE users;
```

Порядок выполнения работы:

1. Подготовка среды

Подключитесь к PostgreSQL через psql:

```
psql -U postgres
```

Создайте новую базу данных:

```
CREATE DATABASE optimization_db;  
\c optimization_db
```

2. Создание тестовой таблицы

Создадим таблицу users с большим количеством записей:

```
CREATE TABLE users (  
    user_id SERIAL PRIMARY KEY,  
    username VARCHAR(50),  
    email VARCHAR(100),  
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP  
);
```

Заполним её 100 000 случайных записей:

```
INSERT INTO users (username, email)  
SELECT  
    'user_' || i,  
    'user' || i || '@example.com'  
FROM generate_series(1, 100000) AS i;
```

3. Анализ медленного запроса

Выполним простой запрос и посмотрим его план:

```
EXPLAIN ANALYZE  
SELECT * FROM users WHERE username = 'user_99999';
```

Вы увидите, что используется Seq Scan — последовательное сканирование всей таблицы.

4. Создание индекса

Добавим индекс по полю username:

```
CREATE INDEX idx_username ON users(username);
```

Теперь повторим тот же запрос:

```
EXPLAIN ANALYZE  
SELECT * FROM users WHERE username = 'user_99999';
```

Теперь вы должны увидеть Index Scan и значительное снижение времени выполнения.

5. Составной индекс

Если часто фильтруете по нескольким полям, создайте составной индекс:

```
CREATE INDEX idx_username_email ON users(username, email);
```

Пример использования:

```
EXPLAIN ANALYZE  
SELECT * FROM users WHERE username = 'user_99999' AND email =  
'user99999@example.com';
```

6. Оптимизация JOIN-запросов

Создадим ещё одну таблицу orders:

```
CREATE TABLE orders (  
    order_id SERIAL PRIMARY KEY,  
    user_id INT REFERENCES users(user_id),  
    amount NUMERIC(10,2),  
    order_date DATE  
);
```

Заполняем 50 000 заказов

```
INSERT INTO orders (user_id, amount, order_date)  
SELECT  
    (random() * 100000)::INT + 1,  
    (random() * 1000 + 100)::NUMERIC(10,2),  
    CURRENT_DATE - (random() * 365)::INT  
FROM generate_series(1, 50000);
```

Выполним JOIN-запрос без индекса:

```
EXPLAIN ANALYZE  
SELECT u.username, SUM(o.amount) AS total_spent  
FROM users u  
JOIN orders o ON u.user_id = o.user_id  
GROUP BY u.username  
ORDER BY total_spent DESC  
LIMIT 10;
```

Создадим индекс по внешнему ключу:

```
CREATE INDEX idx_orders_user_id ON orders(user_id);
```

Повторим запрос и сравните время выполнения.

7. Использование BRIN-индекса для диапазонов

Если часто фильтруете по дате:

```
CREATE INDEX idx_orders_order_date ON orders USING  
BRIN(order_date);
```

Пример использования:

```
EXPLAIN ANALYZE  
SELECT * FROM orders WHERE order_date BETWEEN '2025-01-01' AND  
'2025-03-31';
```

8. Просмотр существующих индексов

```
SELECT indexname, tablename, indexdef  
FROM pg_indexes  
WHERE tablename IN ('users', 'orders');
```

Задание для самостоятельной работы

1. Создайте таблицу products и свяжите её с orders.
2. Добавьте 20 000 записей в products.

3. Напишите запрос, который выбирает все заказы с указанием имени пользователя, названия продукта и суммы заказа.
4. Проанализируйте план выполнения запроса и добавьте необходимые индексы для ускорения.
5. Сравните время выполнения до и после создания индексов.
6. Протестируйте работу с различными типами индексов (B-tree, BRIN, HASH).
7. Напишите отчет о проделанной работе, в котором объясните:
 - как изменилась производительность после добавления индексов;
 - какие типы индексов оказались наиболее эффективными;
 - почему некоторые индексы могут быть бесполезны или даже вредны.

Форма представления результата:

Отчет по выполненной лабораторной работе.

Критерии оценки:

Оценка «отлично» ставится, если задание выполнено верно.

Оценка «хорошо» ставится, если ход выполнения задания верный, но была допущена одна или две ошибки, приведшие к неправильному результату.

Оценка «удовлетворительно» ставится, если приведено неполное выполнение задания.

Оценка «неудовлетворительно» ставится, если задание не выполнено.

Тема 1.2 Разработка и администрирование базы данных

Лабораторное занятие №12

Реализация хранимых процедур и триггеров для автоматизации работы с базой данных

Цель работы: научиться создавать и использовать хранимые процедуры (функции) и триггеры в PostgreSQL для автоматизации бизнес-логики, проверки целостности данных и упрощения повседневных операций над базой данных.

Выполнив работу, вы будете уметь:

У 2.2.1 Осуществлять основные функции по администрированию баз данных;

У 2.5.1 Добавлять, удалять и изменять данные в базе данных;

Материальное обеспечение:

PostgreSQL pgAdmin, DBeaver

Задание:

В созданной базе данных создать хранимые процедуры, функции и триггеры.

Порядок выполнения работы:

1. Подготовка среды

Подключитесь к PostgreSQL через psql:

```
psql -U postgres
```

Создайте новую базу данных:

```
CREATE DATABASE automation_db;  
\c automation_db
```

2. Создание тестовой таблицы

Создадим таблицу employees:

```
CREATE TABLE employees (  
    employee_id SERIAL PRIMARY KEY,  
    full_name VARCHAR(100),  
    salary NUMERIC(10, 2),  
    department VARCHAR(50)  
);
```

Добавим несколько записей:

```
INSERT INTO employees (full_name, salary, department) VALUES  
( 'Иван Иванов', 60000, 'IT' ),  
( 'Мария Петрова', 75000, 'HR' ),  
( 'Алексей Смирнов', 80000, 'IT' );
```

3. Создание простой функции

Напишем функцию, которая увеличивает зарплату сотрудника на заданное количество процентов:

```
CREATE OR REPLACE FUNCTION increase_salary(  
    emp_id INT,  
    percent NUMERIC  
)  
RETURNS VOID AS $$  
BEGIN  
    UPDATE employees  
    SET salary = salary * (1 + percent / 100)  
    WHERE employee_id = emp_id;  
END;  
$$ LANGUAGE plpgsql;
```

RETURNS VOID — функция ничего не возвращает.

Вызов функции:

```
SELECT increase_salary(1, 10); -- увеличиваем зарплату сотрудника  
с ID=1 на 10%  
SELECT * FROM employees;
```

4. Функция с возвратом значения

Создадим функцию, возвращающую среднюю зарплату по отделу:

```
CREATE OR REPLACE FUNCTION avg_salary_by_department(dept VARCHAR)  
RETURNS NUMERIC AS $$  
DECLARE  
    result NUMERIC;  
BEGIN  
    SELECT AVG(salary) INTO result  
    FROM employees  
    WHERE department = dept;  
  
    RETURN result;  
END;  
$$ LANGUAGE plpgsql;
```

Вызов функции:

```
SELECT avg_salary_by_department('IT');
```

5. Создание триггера

Создадим таблицу для хранения истории изменений зарплат:

```
CREATE TABLE salary_log (  
    log_id SERIAL PRIMARY KEY,  
    employee_id INT,  
    old_salary NUMERIC(10,2),  
    new_salary NUMERIC(10,2),  
    changed_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP  
);
```

Теперь создадим функцию-обработчик для триггера:

```
CREATE OR REPLACE FUNCTION log_salary_change()  
RETURNS TRIGGER AS $$  
BEGIN  
    IF OLD.salary <> NEW.salary THEN  
        INSERT INTO salary_log(employee_id, old_salary,  
new_salary)  
        VALUES (OLD.employee_id, OLD.salary, NEW.salary);  
    END IF;  
  
    RETURN NEW;  
END;  
$$ LANGUAGE plpgsql;
```

Теперь привяжем её к таблице employees как триггер:

```
CREATE TRIGGER trigger_salary_change  
AFTER UPDATE ON employees  
FOR EACH ROW  
EXECUTE FUNCTION log_salary_change();
```

Протестируем:

```
-- Обновляем зарплату  
UPDATE employees SET salary = 90000 WHERE employee_id = 1;  
  
-- Смотрим лог  
SELECT * FROM salary_log;
```

6. Дополнительный пример: запрет обновления имени

Создадим триггер, который запрещает обновление поля full_name :

```
CREATE OR REPLACE FUNCTION prevent_name_update()  
RETURNS TRIGGER AS $$  
BEGIN  
    IF OLD.full_name <> NEW.full_name THEN  
        RAISE EXCEPTION 'Изменение имени запрещено!';  
    END IF;
```

```

        RETURN NEW;
    END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trigger_prevent_name_change
BEFORE UPDATE ON employees
FOR EACH ROW
EXECUTE FUNCTION prevent_name_update();

```

Проверяем:

```

UPDATE employees SET full_name = 'Новое имя' WHERE employee_id =
1; -- ошибка
UPDATE employees SET salary = 95000 WHERE employee_id = 1; --
успешно

```

Задание для самостоятельной работы

1. Создайте таблицу `users` со следующими полями:
 - `user_id`,
 - `username`,
 - `email`,
 - `created_at`.
2. Напишите функцию, которая добавляет нового пользователя, если `email` уникален.
3. Создайте таблицу `user_logs`, куда будут записываться действия (добавление, удаление, обновление).
4. Настройте триггеры:
 - при добавлении пользователя — записывается событие в `user_logs`;
 - при удалении — тоже записывается событие;
 - при обновлении `email` — логируется старый и новый `email`;
 - протестируйте работу всех функций и триггеров.

Форма представления результата:

Отчет по выполненной лабораторной работе.

Критерии оценки:

Оценка «отлично» ставится, если задание выполнено верно.

Оценка «хорошо» ставится, если ход выполнения задания верный, но была допущена одна или две ошибки, приведшие к неправильному результату.

Оценка «удовлетворительно» ставится, если приведено неполное выполнение задания.

Оценка «неудовлетворительно» ставится, если задание не выполнено.

Тема 1.2 Разработка и администрирование базы данных

Лабораторное занятие №13

Разработка сценариев миграции данных между двумя базами данных

Цель работы: научиться разрабатывать и реализовывать сценарии переноса данных из одной базы данных в другую с использованием SQL, утилит PostgreSQL и скриптов на языке программирования Python.

Выполнив работу, вы будете уметь:

- У 2.2.1 Осуществлять основные функции по администрированию баз данных;
- У 2.2.2 Настраивать политики безопасности при работе с сервером баз данных
- У 2.5.1 Добавлять, удалять и изменять данные в базе данных;

Материальное обеспечение:

PostgreSQL pgAdmin, DBeaver

Задание:

Выполнить миграции данных между двумя базами данных разными способами

Краткие теоретические сведения:

Миграция данных — это процесс перемещения данных из одной системы хранения информации в другую. Это может быть перенос между разными версиями базы данных, между средами (разработка → тестирование → продакшн), или между различными СУБД.

Основные причины миграции данных:

Причина	Пример
Апгрейд инфраструктуры	Переход с PostgreSQL 12 на PostgreSQL 16
Реорганизация структуры	Изменение схемы таблиц
Миграция на новую платформу	Переход с MySQL на PostgreSQL
Резервное копирование / восстановление	Архивирование данных или восстановление после сбоя
Тестирование / разработка	Перенос данных из production в staging

Инструменты PostgreSQL для миграции:

- pg_dump – экспорт данных и схемы базы данных;
- pg_restore – восстановление из бэкапа;
- COPY - импорт/экспорт в CSV/текстовые файлы;
- dblink/postgres_fdw – подключение к другой базе данных внутри запроса
- SQL-скрипты – ручная миграция через INSERT/SELECT
- Python + psycorg2 – автоматизация миграции через внешний код

Порядок выполнения работы:

1. Подготовка среды

Убедитесь, что установлен PostgreSQL:

```
psql --version
```

Создайте две базы данных: исходную (source_db) и целевую (target_db):

```
createdb source_db  
createdb target_db
```

Подключитесь к исходной базе:

```
psql -d source_db
```

2. Создание структуры и заполнение данных в **source_db**

```
CREATE TABLE users (  
    user_id SERIAL PRIMARY KEY,  
    username VARCHAR(50),  
    email VARCHAR(100)  
);
```

```
INSERT INTO users (username, email) VALUES  
( 'ivan', 'ivan@example.com'),  
( 'maria', 'maria@example.com'),  
( 'alex', 'alex@example.com');
```

3. Экспорт данных с помощью **pg_dump**

Выйдите из psql (\q) и выполните в терминале:

```
pg_dump -d source_db -t users -f users_backup.sql
```

-t users — указывает, что нужно сохранить только таблицу users.

Теперь у вас есть файл users_backup.sql, который можно использовать для импорта.

4. Импорт данных в **target_db**

Импортируйте данные в целевую базу:

```
psql -d target_db -f users_backup.sql
```

Проверьте результат:

```
psql -d target_db -c "SELECT * FROM users;"
```

5. Использование **COPY** для экспорта в **CSV**

Если нужно экспортировать данные в формате CSV:

```
psql -d source_db -c "\COPY users TO 'users.csv' DELIMITER ',' CSV  
HEADER"
```

Импорт в целевую базу данных:

```
psql -d target_db -c "\COPY users FROM 'users.csv' DELIMITER ','  
CSV HEADER"
```

6. Использование **dblink** для прямого переноса

Активируйте расширение dblink:

```
CREATE EXTENSION dblink;
```

Выполните перенос из source_db в target_db:

```
INSERT INTO users (user_id, username, email)  
SELECT      *      FROM      dblink('host=localhost      dbname=source_db  
user=postgres password=yourpassword',  
                                     'SELECT user_id, username, email FROM users')  
AS t(user_id INT, username TEXT, email TEXT);
```

Убедитесь, что пароль указан правильно и пользователь имеет доступ.

7. Использование **Python** для миграции

Пример скрипта на Python с библиотекой psycopg2:

```
import psycopg2

# Подключение к исходной БД
conn_source = psycopg2.connect("dbname=source_db user=postgres
password=yourpassword")
cur_source = conn_source.cursor()

# Подключение к целевой БД
conn_target = psycopg2.connect("dbname=target_db user=postgres
password=yourpassword")
cur_target = conn_target.cursor()

# Выборка данных
cur_source.execute("SELECT user_id, username, email FROM users;")
rows = cur_source.fetchall()

# Вставка данных
for row in rows:
    cur_target.execute(
        "INSERT INTO users (user_id, username, email) VALUES (%s,
%s, %s);",
        row
    )

# Сохраняем изменения
conn_target.commit()

# Закрываем соединения
cur_source.close()
cur_target.close()
conn_source.close()
conn_target.close()
```

Для установки библиотеки:

```
pip install psycopg2-binary
```

Задание для самостоятельной работы

1. Создайте в source_db таблицы orders и products. Добавьте туда не менее 50 записей.
2. Создайте аналогичные таблицы в target_db.
3. Напишите скрипт миграции, который:
 - Экспортирует данные из source_db в CSV-файлы.
 - Импортирует их в target_db.
 - Логирует успешные/ошибочные операции в отдельный файл.
4. Реализуйте миграцию с использованием dblink.
5. Реализуйте миграцию с помощью Python-скрипта.
6. Сравните эффективность каждого метода по времени выполнения и простоте настройки.

Форма представления результата:

Отчет по выполненной лабораторной работе.

Критерии оценки:

Оценка «отлично» ставится, если задание выполнено верно.

Оценка «хорошо» ставится, если ход выполнения задания верный, но была допущена одна или две ошибки, приведшие к неправильному результату.

Оценка «удовлетворительно» ставится, если приведено неполное выполнение задания.

Оценка «неудовлетворительно» ставится, если задание не выполнено.

Тема 1.2 Разработка и администрирование базы данных**Лабораторное занятие №14****Администрирование базы данных: настройка параметров производительности и мониторинг активных запросов**

Цель работы: научиться управлять конфигурацией сервера PostgreSQL для повышения производительности, а также использовать встроенные механизмы для отслеживания и анализа активных SQL-запросов.

Выполнив работу, вы будете уметь:

У 2.2.1 Осуществлять основные функции по администрированию баз данных;

У 2.2.2 Настраивать политики безопасности при работе с сервером баз данных

У 2.5.1 Добавлять, удалять и изменять данные в базе данных;

Материальное обеспечение:

PostgreSQL pgAdmin, DBeaver

Задание:

В созданной базе данных выполните несколько тяжелых запросов, проанализируйте их эффективность и настройте логирование.

Краткие теоретические сведения:

Администрирование PostgreSQL – это комплекс задач по настройке, управлению, мониторингу и оптимизации баз данных. Администратор отвечает за стабильную работу СУБД, высокую доступность, безопасность и производительность.

Ключевые аспекты администрирования:

Область	Задачи
Конфигурация	Настройка памяти, подключений, параллелизма
Безопасность	Права пользователей, SSL, брандмауэр
Резервное копирование	pg_dump, pg_basebackup
Мониторинг	Текущие процессы, медленные запросы, логи
Производительность	Индексы, настройки сервера, оптимизация запросов

Важные параметры производительности в postgresql.conf:

- shared_buffers – размер общей памяти для кэширования данных (обычно 25% от RAM);
- work_mem – память на операцию сортировки или хэширования;
- maintenance_work_mem – ПАМЯТЬ для операций обслуживания (VACUUM, CREATE INDEX и др.);
- effective_cache_size – объем кэша операционной системы, доступный для PostgreSQL;

- `max_connections` – максимальное число одновременных подключений;
- `checkpoint_segments`, `checkpoint_timeout` – частота контрольных точек — влияет на нагрузку на диск;
- `random_page_cost` – стоимость случайного чтения страницы (влияет на выбор плана).

Порядок выполнения работы:

1. Подготовка среды

Убедитесь, что установлен PostgreSQL:

```
psql --version
```

Подключитесь к вашей базе:

```
psql -U postgres
```

2. Путь к конфигурационным файлам

Посмотрите расположение файла `postgresql.conf`:

```
SHOW config_file;
```

Типичный путь:

```
/etc/postgresql/14/main/postgresql.conf
```

Для редактирования требуются права root:

```
sudo nano /etc/postgresql/14/main/postgresql.conf
```

3. Настройка производительности

Пример изменений в `postgresql.conf`:

```
shared_buffers = 2GB
work_mem = 64MB
maintenance_work_mem = 512MB
effective_cache_size = 6GB
max_connections = 100
checkpoint_segments = 32
checkpoint_timeout = 15min
random_page_cost = 1.1
```

После редактирования перезагрузите службу PostgreSQL:

```
sudo systemctl restart postgresql
```

4. Включение расширения `pg_stat_statements`

Расширение позволяет анализировать статистику выполнения SQL-запросов.

Добавьте в `postgresql.conf`:

```
shared_preload_libraries = 'pg_stat_statements'
track_functions = all
pg_stat_statements.track = all
```

Инициализируйте расширение:

```
CREATE EXTENSION pg_stat_statements;
```

5. Анализ активных запросов

Посмотрите текущие активные процессы:

```
SELECT pid, username, application_name, state, query, query_start
```

```
FROM pg_stat_activity
WHERE state != 'idle';
```

Это поможет найти долгие или зависшие запросы.

Остановите конкретный запрос (например, с PID=1234):

```
SELECT pg_cancel_backend(1234);
```

6. Использование pg_stat_statements

Посмотрите самые частые и долгие запросы:

```
SELECT query, calls, total_time, mean_time, rows
FROM pg_stat_statements
ORDER BY total_time DESC
LIMIT 10;
```

Полезно для поиска «узких мест» в приложении.

7. Логирование медленных запросов

В postgresql.conf включите логирование:

```
log_min_duration_statement = 500 # миллисекунды
logging_collector = on
log_directory = 'log'
log_filename = 'postgresql-%Y-%m-%d_%H%M%S.log'
```

После этого PostgreSQL начнёт записывать все запросы, которые выполняются дольше 500

мс.

8. Использование инструментов мониторинга

pgAdmin

Интегрированный инструмент для просмотра активных сессий, запросов, графиков нагрузки.

ptop (аналог top для PostgreSQL)

```
sudo apt install ptop
ptop
```

Задание для самостоятельной работы

1. Настройте тестовую среду с PostgreSQL (можно через Docker).
2. Создайте таблицу logs с полями:
 - id,
 - message TEXT,
 - created_at TIMESTAMP.

Добавьте в неё 100 000 записей.

3. Выполните несколько тяжёлых запросов (например, полный скан с группировкой):

```
SELECT COUNT(*) FROM logs GROUP BY created_at::date;
```
4. С помощью pg_stat_statements найдите этот запрос и проанализируйте его эффективность.
5. Увеличьте значение work_mem и повторите запрос. Зафиксируйте разницу во времени выполнения.
6. Настройте логирование всех запросов дольше 1 секунды. Проверьте содержимое лог-файлов.
7. Реализуйте автоматический скрипт (на Python или bash), который:
 - проверяет список активных запросов,
 - останавливает те, которые выполняются более 5 минут,

- логирует такие события в файл.

Форма представления результата:

Отчет по выполненной лабораторной работе.

Критерии оценки:

Оценка «отлично» ставится, если задание выполнено верно.

Оценка «хорошо» ставится, если ход выполнения задания верный, но была допущена одна или две ошибки, приведшие к неправильному результату.

Оценка «удовлетворительно» ставится, если приведено неполное выполнение задания.

Оценка «неудовлетворительно» ставится, если задание не выполнено.

Тема 1.3. Организация защиты данных в хранилищах

Лабораторное занятие №15

Настройка шифрования данных в MySQL с использованием встроенных функций

Цель работы: научиться использовать встроенные функции шифрования и расшифровки данных в СУБД MySQL, такие как AES_ENCRYPT и AES_DECRYPT. Понять принципы хранения защищённых данных и особенности реализации шифрования на уровне базы данных.

Выполнив работу, вы будете уметь:

У 2.3.1 Дать независимую оценку уровня безопасности

У 2.3.2 Производить регламентное обновление программного обеспечения;

У.2.3.3 Разрабатывать перечень рекомендаций по дальнейшей эксплуатации БД с максимальной защитой хранящейся информации;

Материальное обеспечение:

MySQL

Задание:

В созданной базе данных настройте шифрование данных. Реализуйте приложение, которое добавляет новых сотрудников с шифрованием данных и выводит список сотрудников с расшифровкой.

Краткие теоретические сведения:

Шифрование — это процесс преобразования информации в нечитаемую форму с помощью алгоритма и ключа. Это позволяет защитить данные от несанкционированного доступа даже при утечке.

Важность шифрования данные в базе данных

Причина	Описание
Защита приватности	Конфиденциальные данные (ФИО, телефон, email) нельзя хранить в открытом виде
Соответствие стандартам	GDPR, PCI DSS и другие регулирующие нормы требуют шифрования
Безопасность хранения	Даже при физическом доступе к файлам БД данные останутся защищёнными

В MySQL шифрование данных можно реализовать с использованием встроенных функций AES_ENCRYPT и AES_DECRYPT. Для шифрования используется симметричный алгоритм AES, который имеет различные ключевые длины (128, 256 бит и т.д.).

Алгоритм AES (Advanced Encryption Standard) — один из самых распространённых и безопасных методов симметричного шифрования.

Шаги по настройке шифрования

1. Подготовка:

- создайте таблицу или столбцы, в которых будут храниться данные, подлежащие шифрованию;
- решите, какие данные нужно зашифровать;
- выберите длину ключа aes (обычно 128 или 256 бит) в зависимости от требований безопасности.

2. Шифрование данных при вставке:

Используйте AES_ENCRYPT() для шифрования данных перед их вставкой в таблицу. Например:

```
INSERT INTO users (username, encrypted_password) VALUES ('john_doe', AES_ENCRYPT('password123', 'my_secret_key'));
```

В этом примере AES_ENCRYPT() зашифровывает строку 'password123' с помощью ключа 'my_secret_key' и возвращает зашифрованный двоичный текст, который затем сохраняется в столбце encrypted_password.

3. Извлечение и расшифровка данных:

Используйте AES_DECRYPT() для расшифровки данных при извлечении их из таблицы. Например:

```
SELECT username, AES_DECRYPT(encrypted_password, 'my_secret_key') AS decrypted_password FROM users WHERE username = 'john_doe';
```

В этом примере AES_DECRYPT() расшифровывает данные из столбца encrypted_password с помощью ключа 'my_secret_key' и возвращает расшифрованный текст в поле decrypted_password.

4. Обеспечение безопасности ключа:

Ключ должен быть секретным и храниться в безопасном месте. Нельзя хранить ключ в виде текста в базе данных или скриптах. Используйте разные ключи для разных таблиц или столбцов. Это повышает безопасность и предотвращает возможность взлома, если один ключ будет под угрозой. Не используйте легко угадываемые ключи. Ключ должен быть случайным и сложным.

Пример настройки шифрования для столбца

Создайте таблицу с зашифрованным столбцом:

```
CREATE TABLE sensitive_data (  
    id INT AUTO_INCREMENT PRIMARY KEY,  
    data_to_encrypt VARCHAR(255),  
    encrypted_data BLOB  
);
```

Вставьте данные:

```
INSERT INTO sensitive_data (data_to_encrypt, encrypted_data)  
VALUES ('My secret information', AES_ENCRYPT('My secret information',  
'secret_key'));
```

Извлеките и расшифруйте данные:

```
SELECT AES_DECRYPT(encrypted_data, 'secret_key') AS decrypted_data  
FROM sensitive_data;
```

Важные замечания:

- AES_ENCRYPT() и AES_DECRYPT() возвращают двоичные строки (BLOB). Если вы хотите сохранить расшифрованные данные в текстовом столбце, необходимо преобразовать их в текстовый формат.

- Для обеспечения безопасности, рекомендуется использовать AES_ENCRYPT() и AES_DECRYPT() при выполнении операций с данными, а не хранить зашифрованные данные прямо в столбце. Это поможет защитить данные от несанкционированного доступа, если кто-то получит доступ к базе данных.

- Также можно использовать другие алгоритмы шифрования, такие как MD5(), SHA1(), SHA256(), но они не предназначены для шифрования и используются для хеширования данных.

Порядок выполнения работы:

1. Подготовка среды

Подключитесь к вашему серверу MySQL:

```
mysql -u root -p
```

Создайте тестовую базу данных:

```
CREATE DATABASE encryption_db;  
USE encryption_db;
```

2. Простейший пример использования AES_ENCRYPT / AES_DECRYPT

Проверим работу функций шифрования:

```
SELECT AES_ENCRYPT('Тестовое сообщение', 'my_secret_key') AS  
encrypted;  
SELECT AES_DECRYPT(AES_ENCRYPT('Тестовое сообщение',  
'my_secret_key'), 'my_secret_key') AS decrypted;
```

Обратите внимание: если ключ неверный или отсутствует, результат будет NULL.

3. Создание таблицы с зашифрованными полями

Создадим таблицу users, где будем хранить логины и пароли в зашифрованном виде:

```
CREATE TABLE users (  
    user_id INT AUTO_INCREMENT PRIMARY KEY,  
    username VARCHAR(50),  
    encrypted_password VARBINARY(255)  
);
```

VARBINARY используется для хранения бинарных данных (результат шифрования)

4. Вставка зашифрованных данных

Добавим пользователя:

```
INSERT INTO users (username, encrypted_password)  
VALUES ('admin', AES_ENCRYPT('StrongPass123!', 'my_secret_key'));
```

5. Получение и расшифровка данных

Выберем пользователя и расшифруем его пароль:

```

SELECT username, AES_DECRYPT(encrypted_password, 'my_secret_key')
AS password
FROM users
WHERE username = 'admin';

```

Результат будет в виде строки, если ключ совпадает.

6. Использование шифрования для полей типа TEXT

Если нужно шифровать большие объемы данных, например, персональную информацию:

```

CREATE TABLE personal_data (
    id INT AUTO_INCREMENT PRIMARY KEY,
    full_name VARBINARY(255),
    phone_number VARBINARY(255),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

```

Вставляем данные:

```

INSERT INTO personal_data (full_name, phone_number)
VALUES
    (AES_ENCRYPT('Иван Петров', 'data_key_123'),
    AES_ENCRYPT('+79123456789', 'data_key_123'));

```

Получаем обратно:

```

SELECT
    AES_DECRYPT(full_name, 'data_key_123') AS name,
    AES_DECRYPT(phone_number, 'data_key_123') AS phone
FROM personal_data;

```

7. Хранение ключа шифрования безопасно

Важные рекомендации:

- не храните ключ шифрования в самой базе данных;
- используйте переменные окружения или внешние хранилища секретов (Vault, AWS Secrets Manager);
- не передавайте ключ в чистом виде через SQL-запросы.

Использование ключа в приложении (на Python):

```

import mysql.connector

key = "my_secret_key"
conn = mysql.connector.connect(
    host="localhost",
    user="root",
    password="yourpassword",
    database="encryption_db"
)
cursor = conn.cursor()

cursor.execute(f"""
    SELECT username, AES_DECRYPT(encrypted_password, '{key}') AS
password
    FROM users
""")

```

```
for row in cursor.fetchall():  
    print(row)
```

Задание для самостоятельной работы

1. Создайте таблицу employees со следующими полями:
 - employee_id,
 - full_name (зашифрованное),
 - salary (зашифрованное),
 - passport_number (зашифрованное).
2. Добавьте несколько записей сотрудников с реальными данными.
3. Напишите запрос, который выводит расшифрованные данные только для пользователей с зарплатой выше 100 000.
4. Реализуйте простое приложение (на Python), которое:
 - добавляет новых сотрудников в таблицу с шифрованием данных;
 - выводит список сотрудников с расшифровкой.
5. Настройте систему аудита (логирование) всех операций добавления/обновления/удаления зашифрованных данных.

Форма представления результата:

Оформленная схема базы данных.

Критерии оценки:

Оценка «отлично» ставится, если задание выполнено верно.

Оценка «хорошо» ставится, если ход выполнения задания верный, но была допущена одна или две ошибки, приведшие к неправильному результату.

Оценка «удовлетворительно» ставится, если приведено неполное выполнение задания.

Оценка «неудовлетворительно» ставится, если задание не выполнено.

Тема 1.3. Организация защиты данных в хранилищах

Лабораторное занятие №16

Настройка защиты конфиденциальных данных с использованием маскирования данных (Data Masking) в Microsoft SQL Server

Цель работы: изучить механизм динамического маскирования данных в Microsoft SQL Server, научиться настраивать различные типы масок для столбцов таблиц, проверять эффективность маскирования для разных уровней доступа пользователей и оценивать ограничения метода.

Выполнив работу, вы будете уметь:

У 2.3.1 Дать независимую оценку уровня безопасности

У 2.3.2 Производить регламентное обновление программного обеспечения;

У.2.3.3 Разрабатывать перечень рекомендаций по дальнейшей эксплуатации БД с максимальной защитой хранящейся информации;

Материальное обеспечение:

Microsoft SQL Server Workbench

Задание:

1. Создать тестовую базу данных и таблицу с конфиденциальными данными (ФИО, паспорт, телефон, email, зарплата).
2. Добавить несколько записей и создать двух пользователей:
 - привилегированного (с правами SELECT);
 - непривилегированного (без прав на просмотр реальных данных).
3. Назначить различные типы маскирования на столбцы:
 - default() – для ФИО;
 - partial() – для паспорта и телефона;
 - email() – для email;
 - random() – для зарплаты (в версиях, где поддерживается).
4. Убедиться, что непривилегированный пользователь видит замаскированные данные.
5. Продемонстрировать, что администратор и пользователь с UNMASK видит реальные данные.
6. Выполнить попытку обхода маскирования через:
 - функцию CAST / CONVERT;
 - вложенные запросы;
 - вычисляемые столбцы;
 - сделать вывод о защитных свойствах маскирования.
7. Настроить маскирование через ALTER TABLE и через SQL Server Management Studio (SSMS).

Порядок выполнения работы:

1. Подготовка среды
 - Откройте SSMS или Azure Data Studio.
2. Создание базы данных, таблицы и пользователей

```
CREATE DATABASE LabDataMasking;
GO
USE LabDataMasking;
GO
-- Таблица с конфиденциальными данными
CREATE TABLE Employees (
    EmployeeID INT IDENTITY(1,1) PRIMARY KEY,
    FullName NVARCHAR(100),
    PassportNumber NVARCHAR(20),
    Phone NVARCHAR(20),
    Email NVARCHAR(100),
    Salary DECIMAL(10,2)
);
-- Вставка тестовых данных
INSERT INTO Employees (FullName, PassportNumber, Phone, Email, Salary)
VALUES
    ('Иванов Иван Иванович', '4510 123456', '+7(999)111-22-33',
    'ivanov@example.com', 75000.00),
    ('Петрова Анна Сергеевна', '4511 654321', '+7(999)444-55-66',
    'petrova@example.com', 82000.00);
GO
-- Создание пользователей
CREATE USER LowPrivUser WITHOUT LOGIN;
CREATE USER AdminUser WITHOUT LOGIN;
-- Выдача прав (LowPrivUser – только SELECT)
GRANT SELECT ON Employees TO LowPrivUser;
GRANT SELECT ON Employees TO AdminUser;
GRANT UNMASK TO AdminUser; -- право видеть реальные данные
```

3. Настройка динамического маскирования
 - Добавляем маски

```
ALTER TABLE Employees ALTER COLUMN FullName ADD MASKED WITH (FUNCTION =
'default()');
```

```

ALTER TABLE Employees ALTER COLUMN PassportNumber ADD MASKED WITH (FUNCTION =
'partial(2, "****", 2)');
ALTER TABLE Employees ALTER COLUMN Phone ADD MASKED WITH (FUNCTION = 'partial(0,
"***-***-**-**", 0)');
ALTER TABLE Employees ALTER COLUMN Email ADD MASKED WITH (FUNCTION = 'email()');
ALTER TABLE Employees ALTER COLUMN Salary ADD MASKED WITH (FUNCTION =
'random(1000, 100000)');

```

4. Проверка работы маскирования

```

-- Посмотреть от имени LowPrivUser
EXECUTE AS USER = 'LowPrivUser';
SELECT * FROM Employees;
REVERT;

```

```

-- Посмотреть от имени AdminUser
EXECUTE AS USER = 'AdminUser';
SELECT * FROM Employees;
REVERT;

```

5. Попытка обхода маскирования

Выполните от имени LowPrivUser:

```

EXECUTE AS USER = 'LowPrivUser';
-- Попытка 1: CAST
SELECT FullName, CAST(PassportNumber AS NVARCHAR(20)) FROM Employees;
-- Попытка 2: подзапрос
SELECT (SELECT FullName FROM Employees WHERE EmployeeID = 1) AS Name;
-- Попытка 3: через вычисляемый столбец во временной таблице
SELECT FullName, PassportNumber INTO #temp FROM Employees;
SELECT * FROM #temp;
REVERT;

```

6. Управление масками через SSMS (альтернативный способ)

- В обозревателе объектов найдите таблицу Employees.
- ПКМ → Design.
- Выберите столбец, в окне свойств найдите Masking → задайте функцию.
- Сохраните таблицу.

7. Снятие маскирования и удаление объектов (для отчёта)

```

ALTER TABLE Employees ALTER COLUMN FullName DROP MASKED;
ALTER TABLE Employees ALTER COLUMN PassportNumber DROP MASKED;
ALTER TABLE Employees ALTER COLUMN Phone DROP MASKED;
ALTER TABLE Employees ALTER COLUMN Email DROP MASKED;
ALTER TABLE Employees ALTER COLUMN Salary DROP MASKED;

DROP USER LowPrivUser;
DROP USER AdminUser;
DROP TABLE Employees;
USE master;
DROP DATABASE LabDataMasking;

```

Форма представления результата:

Листинги всех выполненных SQL-скриптов с комментариями.

Критерии оценки:

Оценка «отлично» ставится, если задание выполнено верно.

Оценка «хорошо» ставится, если ход выполнения задания верный, но была допущена одна или две ошибки, приведшие к неправильному результату.

Оценка «удовлетворительно» ставится, если приведено неполное выполнение задания.

Оценка «неудовлетворительно» ставится, если задание не выполнено.

Тема 1.3. Организация защиты данных в хранилищах

Лабораторное занятие №17

Настройка аудита действий пользователей в Microsoft SQL Server

Цель работы: изучить механизмы аудита в Microsoft SQL Server, научиться создавать и настраивать серверный аудит, спецификации аудита базы данных, фильтровать события, анализировать журналы аудита и оценивать влияние аудита на производительность системы.

Выполнив работу, вы будете уметь:

У 2.3.1 Дать независимую оценку уровня безопасности

У 2.3.2 Производить регламентное обновление программного обеспечения;

У.2.3.3 Разрабатывать перечень рекомендаций по дальнейшей эксплуатации БД с максимальной защитой хранящейся информации;

Материальное обеспечение:

Microsoft SQL Server Workbench

Задание:

1. Создать тестовую базу данных, таблицу и несколько пользователей.
2. Настроить серверный аудит с целевым файлом (.sqlaudit).
3. Создать спецификацию аудита базы данных для отслеживания:
4. SELECT, INSERT, UPDATE, DELETE на конкретной таблице;
5. EXECUTE для хранимых процедур;
6. неудачных попыток входа (FAILED_LOGIN_GROUP).
7. Создать спецификацию серверного аудита для отслеживания изменений в ролях сервера.
8. Выполнить ряд действий от имени разных пользователей.
9. Прочитать журнал аудита и проанализировать записи.
10. Настроить фильтрацию (аудит только для определённого пользователя).
11. Очистить и удалить объекты аудита.

Порядок выполнения работы:

```
1. Подготовка среды
-- Создание тестовой базы данных
CREATE DATABASE LabAudit;
GO
USE LabAudit;
GO
-- Создание таблицы для аудита
CREATE TABLE SensitiveData (
    ID INT IDENTITY(1,1) PRIMARY KEY,
    DataValue NVARCHAR(100),
    LastModifiedBy NVARCHAR(100),
    LastModifiedDate DATETIME DEFAULT GETDATE()
);
-- Вставка тестовых данных
INSERT INTO SensitiveData (DataValue, LastModifiedBy)
VALUES ('Конфиденциальная информация 1', 'Admin'),
       ('Конфиденциальная информация 2', 'Admin');
-- Создание хранимой процедуры
CREATE PROCEDURE sp_GetSensitiveData
AS
BEGIN
    SELECT * FROM SensitiveData;
END;
```

```
-- Создание пользователей для тестирования
CREATE LOGIN AuditUser1 WITH PASSWORD = 'P@ssw0rd123!';
CREATE LOGIN AuditUser2 WITH PASSWORD = 'P@ssw0rd123!';
CREATE USER AuditUser1 FOR LOGIN AuditUser1;
CREATE USER AuditUser2 FOR LOGIN AuditUser2;
-- Выдача прав
GRANT SELECT, INSERT, UPDATE, DELETE ON SensitiveData TO AuditUser1;
GRANT EXECUTE ON sp_GetSensitiveData TO AuditUser1;
GRANT SELECT ON SensitiveData TO AuditUser2;
```

2. Создание серверного аудита

```
USE master;
GO
```

```
-- Создание аудита (файлы будут храниться в C:\AuditLogs)
CREATE SERVER AUDIT ServerAudit
TO FILE (FILEPATH = 'C:\AuditLogs\', MAXSIZE = 10 MB, MAX_FILES = 5)
WITH (QUEUE_DELAY = 1000, ON_FAILURE = CONTINUE);
```

```
-- Включение аудита
ALTER SERVER AUDIT ServerAudit WITH (STATE = ON);
```

3. Создание спецификации аудита базы данных

```
USE master;
GO
```

```
CREATE SERVER AUDIT SPECIFICATION ServerAuditSpec
FOR SERVER AUDIT ServerAudit
ADD (SERVER_ROLE_MEMBER_CHANGE_GROUP);
```

```
ALTER SERVER AUDIT SPECIFICATION ServerAuditSpec WITH (STATE = ON);
```

4. Создание спецификации аудита базы данных

```
USE LabAudit;
GO
```

```
CREATE DATABASE AUDIT SPECIFICATION DBAuditSpec
FOR SERVER AUDIT ServerAudit
ADD (SELECT, INSERT, UPDATE, DELETE ON SensitiveData BY AuditUser1),
ADD (EXECUTE ON sp_GetSensitiveData BY AuditUser1),
ADD (FAILED_LOGIN_GROUP);
```

```
-- Включение спецификации
ALTER DATABASE AUDIT SPECIFICATION DBAuditSpec WITH (STATE = ON);
```

5. Генерация событий для аудита

```
-- 1. Успешные действия от AuditUser1
```

```
EXECUTE AS LOGIN = 'AuditUser1';
USE LabAudit;
SELECT * FROM SensitiveData;
INSERT INTO SensitiveData (DataValue, LastModifiedBy) VALUES ('Новая запись',
'AuditUser1');
UPDATE SensitiveData SET DataValue = 'Изменено' WHERE ID = 1;
DELETE FROM SensitiveData WHERE ID = 2;
EXEC sp_GetSensitiveData;
REVERT;
```

```
-- 2. Действия от AuditUser2
```

```
EXECUTE AS LOGIN = 'AuditUser2';
USE LabAudit;
SELECT * FROM SensitiveData;
-- Попытка UPDATE (должна быть запрещена, но не зааудитится, т.к. не BY
AuditUser1)
UPDATE SensitiveData SET DataValue = 'Попытка изменения' WHERE ID = 1;
REVERT;
```

```
-- 3. Неудачные попытки входа
```

```

-- (выполнить отдельным подключением с неверным паролем)
-- или симулировать:
-- LOGIN AuditUser1 с неверным паролем

-- 4. Изменение роли сервера
CREATE LOGIN TestLogin WITH PASSWORD = 'Test123!';
ALTER SERVER ROLE sysadmin ADD MEMBER TestLogin;
DROP LOGIN TestLogin;

6. Чтение и анализ журнала аудита
-- Функция для чтения файлов аудита
SELECT
    event_time,
    session_server_principal_name,
    database_name,
    schema_name,
    object_name,
    statement,
    succeeded,
    server_principal_name,
    action_id,
    class_type
FROM sys.fn_get_audit_file('C:\AuditLogs\ServerAudit_*.sqlaudit', DEFAULT,
DEFAULT)
ORDER BY event_time DESC;

7. Настройка фильтрации аудита (продвинутый уровень)
-- Создание аудита с фильтрацией по пользователю
CREATE SERVER AUDIT FilteredAudit
TO FILE (FILEPATH = 'C:\AuditLogs\Filtered\')
WITH (STATE = OFF);

-- Фильтр: только действия от AuditUser1
ALTER SERVER AUDIT FilteredAudit
WITH (STATE = OFF);
-- (фильтрация WHERE через предикат на уровне спецификации не поддерживается,
-- альтернатива - создание отдельной спецификации только для нужного
пользователя)

8. Просмотр настроек аудита через SSMS
-- В SSMS: Security → Audits → просмотр свойств ServerAudit.
-- Security → Server Audit Specifications → ServerAuditSpec.
-- В базе данных: Security → Database Audit Specifications → DBAuditSpec.

9. Очистка и удаление объектов (для отчёта)
-- Отключение и удаление спецификаций
ALTER DATABASE AUDIT SPECIFICATION DBAuditSpec WITH (STATE = OFF);
DROP DATABASE AUDIT SPECIFICATION DBAuditSpec;

ALTER SERVER AUDIT SPECIFICATION ServerAuditSpec WITH (STATE = OFF);
DROP SERVER AUDIT SPECIFICATION ServerAuditSpec;

-- Отключение и удаление серверного аудита
ALTER SERVER AUDIT ServerAudit WITH (STATE = OFF);
DROP SERVER AUDIT ServerAudit;

-- Удаление тестовых объектов
USE LabAudit;
DROP PROCEDURE sp_GetSensitiveData;
DROP TABLE SensitiveData;
DROP USER AuditUser1;
DROP USER AuditUser2;
USE master;
DROP LOGIN AuditUser1;
DROP LOGIN AuditUser2;
DROP LOGIN TestLogin;

```

```
DROP DATABASE LabAudit;
```

```
-- Удаление файлов аудита вручную (из папки C:\AuditLogs)
```

Форма представления результата:

Листинги всех выполненных SQL-скриптов с комментариями.

Критерии оценки:

Оценка «отлично» ставится, если задание выполнено верно.

Оценка «хорошо» ставится, если ход выполнения задания верный, но была допущена одна или две ошибки, приведшие к неправильному результату.

Оценка «удовлетворительно» ставится, если приведено неполное выполнение задания.

Оценка «неудовлетворительно» ставится, если задание не выполнено.

Тема 1.3. Организация защиты данных в хранилищах

Лабораторное занятие №18

Реализация ролевой модели безопасности в PostgreSQL (создание ролей и управление их правами)

Цель работы: изучить механизмы ролевой модели безопасности в PostgreSQL, научиться создавать роли, назначать им права на объекты базы данных, управлять членством в ролях, наследованием прав и проверять эффективность разграничения доступа на уровне строк (RLS) и столбцов.

Выполнив работу, вы будете уметь:

У 2.3.1 Дать независимую оценку уровня безопасности

У 2.3.2 Производить регламентное обновление программного обеспечения;

У.2.3.3 Разрабатывать перечень рекомендаций по дальнейшей эксплуатации БД с максимальной защитой хранящейся информации;

Материальное обеспечение:

PostgreSQL, DBeaver, PgAdmin

Задание:

В соответствии со своим вариантом задания, привести базу данных к 3НФ. Реализовать схему базы данных в среде MySQL Workbench.

1. В PostgreSQL создать тестовую базу данных.
2. Создать иерархию ролей:
 - app_admin – администратор приложения;
 - app_manager – менеджер (чтение и изменение всех данных);
 - app_analyst – аналитик (только чтение);
 - app_user – обычный пользователь (ограниченные права).
3. Создать схему sales и таблицы:
 - Customers (клиенты);
 - orders (заказы);
 - order_items (позиции заказов);
 - products (товары).
4. Назначить права ролям в соответствии с матрицей доступа.
5. Создать реальных пользователей и добавить их в роли.
6. Проверить корректность разграничения доступа.
7. Настроить Row Level Security (RLS) для ограничения доступа к заказам по региону.

8. Настроить маскирование данных (через представления или расширение anon).
9. Продемонстрировать наследование прав и возможность SET ROLE.

Порядок выполнения работы:

1. Подготовка среды

```
-- Подключение под суперпользователем (postgres)
CREATE DATABASE lab_rbac;
\c lab_rbac;

-- Создание схемы
CREATE SCHEMA sales;

-- Создание таблиц
CREATE TABLE sales.customers (
    customer_id SERIAL PRIMARY KEY,
    full_name VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    phone VARCHAR(20),
    region VARCHAR(50) NOT NULL
);

CREATE TABLE sales.products (
    product_id SERIAL PRIMARY KEY,
    product_name VARCHAR(100) NOT NULL,
    price DECIMAL(10,2) NOT NULL CHECK (price > 0),
    category VARCHAR(50)
);

CREATE TABLE sales.orders (
    order_id SERIAL PRIMARY KEY,
    customer_id INTEGER REFERENCES sales.customers(customer_id),
    order_date DATE DEFAULT CURRENT_DATE,
    total_amount DECIMAL(10,2),
    status VARCHAR(20) DEFAULT 'NEW'
);

CREATE TABLE sales.order_items (
    item_id SERIAL PRIMARY KEY,
    order_id INTEGER REFERENCES sales.orders(order_id),
    product_id INTEGER REFERENCES sales.products(product_id),
    quantity INTEGER NOT NULL CHECK (quantity > 0),
    unit_price DECIMAL(10,2) NOT NULL
);

-- Вставка тестовых данных
INSERT INTO sales.customers (full_name, email, phone, region) VALUES
('Иванов Иван', 'ivanov@example.com', '111-1111', 'Москва'),
('Петров Петр', 'petrov@example.com', '222-2222', 'Санкт-Петербург'),
('Сидорова Анна', 'sidorova@example.com', '333-3333', 'Москва');

INSERT INTO sales.products (product_name, price, category) VALUES
('Ноутбук', 50000.00, 'Электроника'),
('Мышь', 1500.00, 'Электроника'),
('Стол', 8000.00, 'Мебель');

INSERT INTO sales.orders (customer_id, order_date, total_amount, status) VALUES
(1, '2024-01-15', 51500.00, 'COMPLETED'),
(2, '2024-01-20', 8000.00, 'NEW'),
(3, '2024-02-01', 1500.00, 'PROCESSING');

INSERT INTO sales.order_items (order_id, product_id, quantity, unit_price)
VALUES
(1, 1, 1, 50000.00),
```

```
(1, 2, 1, 1500.00),
(2, 3, 1, 8000.00),
(3, 2, 1, 1500.00);
```

2. Создание ролей

```
-- Создание ролей
```

```
CREATE ROLE app_admin;
CREATE ROLE app_manager;
CREATE ROLE app_analyst;
CREATE ROLE app_user;
```

```
-- Назначение прав на уровне базы данных и схемы
```

```
GRANT CONNECT ON DATABASE lab_rbac TO app_admin, app_manager, app_analyst,
app_user;
```

```
GRANT USAGE ON SCHEMA sales TO app_admin, app_manager, app_analyst, app_user;
```

3. Назначение прав на таблицы (матрица доступа)

```
-- app_admin: полный доступ
```

```
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA sales TO app_admin;
```

```
GRANT ALL PRIVILEGES ON ALL SEQUENCES IN SCHEMA sales TO app_admin;
```

```
-- app_manager: SELECT, INSERT, UPDATE, DELETE (полный доступ к данным)
```

```
GRANT SELECT, INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA sales TO
app_manager;
```

```
GRANT USAGE ON ALL SEQUENCES IN SCHEMA sales TO app_manager;
```

```
-- app_analyst: только SELECT
```

```
GRANT SELECT ON ALL TABLES IN SCHEMA sales TO app_analyst;
```

```
-- app_user: ограниченные права (только чтение customers, только свои заказы -
позже через RLS)
```

```
GRANT SELECT ON sales.customers TO app_user;
```

```
GRANT SELECT ON sales.products TO app_user;
```

```
-- На orders и order_items права через RLS
```

4. Создание пользователей и назначение ролей

```
-- Создание пользователей (логины)
```

```
CREATE USER admin_user WITH PASSWORD 'Admin123!';
```

```
CREATE USER manager_user WITH PASSWORD 'Manager123!';
```

```
CREATE USER analyst_user WITH PASSWORD 'Analyst123!';
```

```
CREATE USER regular_user1 WITH PASSWORD 'User123!';
```

```
CREATE USER regular_user2 WITH PASSWORD 'User123!';
```

```
-- Назначение ролей пользователям
```

```
GRANT app_admin TO admin_user;
```

```
GRANT app_manager TO manager_user;
```

```
GRANT app_analyst TO analyst_user;
```

```
GRANT app_user TO regular_user1;
```

```
GRANT app_user TO regular_user2;
```

```
-- Настройка наследования прав
```

```
ALTER ROLE admin_user INHERIT;
```

```
ALTER ROLE manager_user INHERIT;
```

```
ALTER ROLE analyst_user INHERIT;
```

```
ALTER ROLE regular_user1 INHERIT;
```

```
ALTER ROLE regular_user2 INHERIT;
```

5. Проверка разграничения доступа

```
-- Проверка от имени app_analyst (только чтение)
```

```
SET ROLE app_analyst;
```

```
SELECT * FROM sales.customers; -- Успешно
```

```
SELECT * FROM sales.orders; -- Успешно
```

```
INSERT INTO sales.customers VALUES (4, 'Тест', 'test@test.com', '999',
'Новгород'); -- Ошибка
```

```
DELETE FROM sales.products WHERE product_id = 1; -- Ошибка
```

```
RESET ROLE;
```

```
-- Проверка от имени app_manager (полный CRUD)
SET ROLE app_manager;
INSERT INTO sales.customers VALUES (4, 'Новый клиент', 'new@test.com', '444-4444', 'Казань');
UPDATE sales.products SET price = 52000 WHERE product_id = 1;
DELETE FROM sales.order_items WHERE item_id = 4; -- Если существует
RESET ROLE;
```

6. Настройка Row Level Security (RLS)

```
-- Включение RLS для таблицы orders
ALTER TABLE sales.orders ENABLE ROW LEVEL SECURITY;

-- Создание политики: пользователь видит только заказы своего региона
-- (предполагаем, что регион пользователя хранится в отдельной таблице или параметре)
CREATE POLICY user_region_orders_policy ON sales.orders
    FOR ALL
    TO app_user
    USING (
        customer_id IN (
            SELECT customer_id FROM sales.customers
            WHERE region = current_setting('app.current_region')
        )
    );

-- Создание политики для менеджера (видит все заказы)
CREATE POLICY manager_all_orders_policy ON sales.orders
    FOR ALL
    TO app_manager
    USING (true);

-- Установка параметра региона для пользователя (пример)
SET app.current_region = 'Москва';

-- Проверка RLS
SET ROLE regular_user1;
SELECT * FROM sales.orders; -- Только заказы по Москве
RESET ROLE;
```

7. Маскирование данных через представление

```
-- Создание представления для маскирования персональных данных
CREATE VIEW sales.customers_masked AS
SELECT
    customer_id,
    CASE
        WHEN current_user IN ('admin_user', 'manager_user') THEN full_name
        ELSE LEFT(full_name, 2) || '*****'
    END AS full_name,
    CASE
        WHEN current_user IN ('admin_user', 'manager_user') THEN email
        ELSE 'masked@' || SPLIT_PART(email, '@', 2)
    END AS email,
    CASE
        WHEN current_user IN ('admin_user', 'manager_user') THEN phone
        ELSE '***-***-'
    END AS phone,
    region
FROM sales.customers;

-- Назначение прав на представление
GRANT SELECT ON sales.customers_masked TO app_user;
REVOKE SELECT ON sales.customers FROM app_user; -- Запрет прямого доступа
```

8. Работа с членством в ролях и SET ROLE

```
-- Создание групповой роли
CREATE ROLE sales_team;
```

```

GRANT app_manager TO sales_team;
GRANT sales_team TO regular_user1;

-- Проверка цепочки наследования
SET ROLE regular_user1;
SELECT current_user, session_user;
SELECT * FROM sales.customers; -- Должен иметь права через sales_team ->
app_manager
RESET ROLE;

-- Использование SET ROLE для временного повышения привилегий
SET ROLE app_manager;
-- Выполнение административных действий
RESET ROLE;

```

9. Просмотр и аудит прав

```

-- Просмотр всех ролей
\du

-- Просмотр прав на таблицы
SELECT
    grantee,
    table_schema,
    table_name,
    privilege_type
FROM information_schema.role_table_grants
WHERE table_schema = 'sales'
ORDER BY grantee, table_name;

-- Просмотр членства в ролях
SELECT
    r.rolname AS role_name,
    m.rolname AS member_name
FROM pg_roles r
JOIN pg_auth_members am ON r.oid = am.roleid
JOIN pg_roles m ON am.member = m.oid;

```

10. Очистка (для отчёта)

```

-- Отключение RLS и удаление политик
ALTER TABLE sales.orders DISABLE ROW LEVEL SECURITY;
DROP POLICY IF EXISTS user_region_orders_policy ON sales.orders;
DROP POLICY IF EXISTS manager_all_orders_policy ON sales.orders;

-- Удаление представления
DROP VIEW IF EXISTS sales.customers_masked;

-- Удаление пользователей и ролей
DROP USER IF EXISTS admin_user;
DROP USER IF EXISTS manager_user;
DROP USER IF EXISTS analyst_user;
DROP USER IF EXISTS regular_user1;
DROP USER IF EXISTS regular_user2;

DROP ROLE IF EXISTS app_admin;
DROP ROLE IF EXISTS app_manager;
DROP ROLE IF EXISTS app_analyst;
DROP ROLE IF EXISTS app_user;
DROP ROLE IF EXISTS sales_team;

-- Удаление схемы и базы данных
DROP SCHEMA sales CASCADE;
\c postgres;
DROP DATABASE lab_rbac;

```

Форма представления результата:

Листинги всех выполненных SQL-скриптов с комментариями.

Критерии оценки:

Оценка «отлично» ставится, если задание выполнено верно.

Оценка «хорошо» ставится, если ход выполнения задания верный, но была допущена одна или две ошибки, приведшие к неправильному результату.

Оценка «удовлетворительно» ставится, если приведено неполное выполнение задания.

Оценка «неудовлетворительно» ставится, если задание не выполнено.

Тема 1.3. Организация защиты данных в хранилищах

Лабораторное занятие №19

Организация двухфакторной аутентификации для доступа к базам данных

Цель работы: изучить принципы двухфакторной аутентификации (2FA/MFA) применительно к системам управления базами данных, освоить настройку MFA для различных СУБД, научиться подключаться к базе данных с использованием дополнительного фактора аутентификации и оценить эффективность данного метода защиты.

Выполнив работу, вы будете уметь:

У 2.3.1 Дать независимую оценку уровня безопасности

У 2.3.2 Производить регламентное обновление программного обеспечения;

У.2.3.3 Разрабатывать перечень рекомендаций по дальнейшей эксплуатации БД с максимальной защитой хранящейся информации;

Материальное обеспечение:

PostgreSQL MySQL Workbench

Задание:

1. Выбрать одну из следующих платформ для реализации (в соответствии с доступной средой):
 - Вариант А: PostgreSQL + расширение (например, через PAM + Google Authenticator)
 - Вариант Б: Настройка прикладного уровня (PHP/Python + база данных)
2. Настроить среду для поддержки двухфакторной аутентификации.
3. Создать тестового пользователя и привязать к нему второй фактор (TOTP-приложение, SMS или push-уведомления).
4. Выполнить успешное подключение к БД с использованием обоих факторов.
5. Продемонстрировать реакцию системы при неверном пароле или неверном одноразовом коде.
6. Проанализировать ограничения и особенности реализации MFA для разных СУБД.

Порядок выполнения работы:

Вариант А: PostgreSQL + PAM + TOTP (Google Authenticator)

Этот вариант демонстрирует настройку 2FA для PostgreSQL через PAM (Pluggable Authentication Modules) .

1. Установка необходимых компонентов (Linux)

```
# Установка PostgreSQL
```

```
sudo apt install postgresql postgresql-contrib
```

```
# Установка PAM модуля Google Authenticator
```

```
sudo apt install libpam-google-authenticator
```

```
# Установка FreeRADIUS (для RADIUS-маршрута, опционально)
```

```
sudo apt install freeradius freeradius-utils
2. Настройка Google Authenticator для пользователя
# От имени пользователя БД (например, testuser)
google-authenticator
```

При выполнении команды:

- Будет сгенерирован секретный ключ и QR-код.
- Сохраните резервные коды.
- Ответьте на вопросы (рекомендуется: да, да, нет, да).

```
3. Настройка PAM для PostgreSQL
# Создание PAM-сервиса для PostgreSQL
sudo nano /etc/pam.d/postgresql
```

Содержимое файла:

```
# PAM configuration for PostgreSQL
auth required pam_google_authenticator.so
account required pam_permit.so
4. Настройка PostgreSQL для использования PAM
-- В файле pg_hba.conf добавить:
# TYPE DATABASE USER ADDRESS METHOD
host all testuser 127.0.0.1/32 pam service=postgresql
```

```
-- В postgresql.conf убедиться, что включена аутентификация:
password_encryption = md5
```

```
5. Проверка подключения
# При подключении будет запрошен пароль (первый фактор) и TOTP-код (второй)
psql -h localhost -U testuser -d postgres
Password: (пароль пользователя)
Second factor: (код из Google Authenticator)
```

Вариант Г: Прикладной уровень (PHP + MySQL/PostgreSQL)

Этот вариант демонстрирует реализацию 2FA на уровне приложения с хранением секретов в базе данных .

6. Создание таблиц для хранения данных 2FA

```
-- Таблица пользователей
CREATE TABLE users (
    user_id INT PRIMARY KEY AUTO_INCREMENT,
    username VARCHAR(50) UNIQUE NOT NULL,
    password_hash VARCHAR(255) NOT NULL,
    email VARCHAR(100)
);

-- Таблица для TOTP-секретов
CREATE TABLE user_totp (
    user_id INT PRIMARY KEY REFERENCES users(user_id),
    secret_key VARCHAR(32) NOT NULL,
    backup_codes TEXT,
    enabled BOOLEAN DEFAULT TRUE
);
```

7. PHP-код для генерации и проверки TOTP (с использованием библиотеки)

```
<?php
// Установка через composer: composer require sonata-project/google-
authenticator
require_once 'vendor/autoload.php';
use Sonata\GoogleAuthenticator\GoogleAuthenticator;

$ga = new GoogleAuthenticator();

// Генерация секрета при регистрации
$secret = $ga->generateSecret();
$qrcodeUrl = $ga->getQRCodeUrl('MyApp', $username, $secret);
```

```
// Проверка кода при входе
if ($ga->checkCode($secret, $_POST['totp_code'])) {
    // Второй фактор подтверждён
} else {
    // Неверный код
}
?>
```

8. Проверка отказоустойчивости

Для всех вариантов выполнить тесты:

- Успешный вход с верными паролем и TOTP-кодом.
- Попытка входа с верным паролем, но неверным TOTP-кодом.
- Попытка входа с неверным паролем.
- Использование резервного кода (если поддерживается).

9. Очистка (для отчёта)

```
-- Удаление тестовых пользователей и настроек
DROP USER IF EXISTS test2fa;
-- Деактивация MFA-настроек в соответствии с выбранным вариантом
```

Форма представления результата:

Листинги всех выполненных SQL-скриптов с комментариями.

Критерии оценки:

Оценка «отлично» ставится, если задание выполнено верно.

Оценка «хорошо» ставится, если ход выполнения задания верный, но была допущена одна или две ошибки, приведшие к неправильному результату.

Оценка «удовлетворительно» ставится, если приведено неполное выполнение задания.

Оценка «неудовлетворительно» ставится, если задание не выполнено.

Тема 1.3. Организация защиты данных в хранилищах

Лабораторное занятие №20

Конфигурация шифрования трафика между клиентом и сервером базы данных (TLS/SSL)

Цель работы: освоить практические навыки настройки защищённого соединения между клиентом и сервером базы данных с использованием протоколов TLS/SSL. Научиться создавать и применять сертификаты, а также проверять шифрование трафика.

Выполнив работу, вы будете уметь:

У 2.3.1 Дать независимую оценку уровня безопасности

У 2.3.2 Производить регламентное обновление программного обеспечения;

У.2.3.3 Разрабатывать перечень рекомендаций по дальнейшей эксплуатации БД с максимальной защитой хранящейся информации;

Материальное обеспечение:

PostgreSQL, MySQL Workbench

Задание:

1. Изучить теоретические основы TLS/SSL и их применение для защиты соединений с базами данных.

2. Сгенерировать самоподписанный сертификат или сертификат, подписанный тестовым центром сертификации (CA).

3. Настроить сервер базы данных для использования TLS/SSL (на выбор: PostgreSQL, MySQL, Microsoft SQL Server или Oracle).
4. Настроить принудительное использование шифрования для всех или выбранных типов подключений.
5. Выполнить подключение к БД с использованием шифрования и без него, проанализировать результаты.
6. Настроить клиентскую сторону для верификации сертификата сервера.
7. Проверить статус шифрования активных соединений.
8. Проанализировать различия между режимами: require, verify-ca, verify-full.

Порядок выполнения работы:

1. Подготовка среды

Общие требования:

- Установленный сервер БД (выберите один вариант из представленных ниже).
- Утилита OpenSSL для генерации сертификатов.
- Права администратора на сервере БД.

Вариант А: PostgreSQL + TLS/SSL

2. Генерация сертификатов

```
# Переход в каталог данных PostgreSQL (для Debian/Ubuntu -
/etc/postgresql/version/main)
cd /var/lib/postgresql/17/main
# или cd /etc/postgresql/17/main (в зависимости от дистрибутива)
```

```
# Создание самоподписанного сертификата (на 365 дней)
openssl req -new -x509 -days 365 -nodes -text -out server.crt \
-keyout server.key -subj "/CN=dbhost.yourdomain.com"
```

```
# Установка правильных прав доступа
chmod 600 server.key
chown postgres:postgres server.crt server.key
```

3. Настройка PostgreSQL

В файле postgresql.conf:

```
# Включение SSL
ssl = on
# Пути к файлам сертификатов (если не в стандартном месте)
ssl_cert_file = 'server.crt'
ssl_key_file = 'server.key'
# Версии TLS
ssl_min_protocol_version = 'TLSv1.2'
```

В файле pg_hba.conf для принудительного использования SSL:

```
# Принудительное использование SSL для всех удалённых подключений
hostssl    all             all             0.0.0.0/0          scram-sha-
256
# Запрет незашифрованных удалённых подключений
hostnossl  all             all             0.0.0.0/0          reject
```

4. Перезапуск сервера

```
sudo systemctl restart postgresql
```

5. Проверка подключений

```
# Подключение с требованием SSL
psql "sslmode=require host=localhost dbname=postgres user=postgres"
```

```
# Подключение с верификацией сертификата
```

```
psql "sslmode=verify-full sslrootcert=server.crt host=dbhost.yourdomain.com
dbname=postgres user=postgres"
```

6. Мониторинг SSL-соединений

Вариант Б: MySQL + TLS/SSL

7. Генерация сертификатов

```
# Создание директории для сертификатов
```

```

mkdir -p /etc/mysql
cd /etc/mysql

# Генерация CA-сертификата
openssl genrsa 2048 > ca-key.pem
openssl req -new -x509 -nodes -days 3600 -key ca-key.pem -out ca-cert.pem

# Генерация серверного сертификата
openssl req -newkey rsa:2048 -days 3600 -nodes -keyout server-key.pem -out
server-req.pem
openssl x509 -req -in server-req.pem -days 3600 -CA ca-cert.pem -CAkey ca-
key.pem -set_serial 01 -out server-cert.pem

# Настройка прав
chown -R mysql:mysql /etc/mysql
chmod 600 /etc/mysql/server-key.pem
chmod 644 /etc/mysql/*.pem

```

8. Настройка MySQL

В файле `/etc/mysql/my.cnf` или `/etc/mysql/mysql.conf.d/mysqld.cnf`:

```

[mysqld]
ssl-ca = /etc/mysql/ca-cert.pem
ssl-cert = /etc/mysql/server-cert.pem
ssl-key = /etc/mysql/server-key.pem
tls_version = TLSv1.2,TLSv1.3
require_secure_transport = ON

```

9. Принуждение пользователей к SSL

```

-- Требование SSL для конкретного пользователя
ALTER USER 'username'@'%' REQUIRE SSL;

```

```

-- Просмотр требований SSL для пользователей
SELECT user, host, ssl_type FROM mysql.user;

```

10. Проверка конфигурации

```

-- Проверка статуса SSL на сервере
SHOW GLOBAL VARIABLES LIKE '%ssl%';
SHOW GLOBAL VARIABLES LIKE '%tls%';

```

```

-- Проверка текущего соединения
STATUS;

```

11. Подключение с SSL

```

# Требование SSL при подключении
mysql -u username -p --ssl-mode=REQUIRED

```

```

# С верификацией сертификата
mysql -u username -p --ssl-mode=VERIFY_CA --ssl-ca=/etc/mysql/ca-cert.pem

```

Вариант B: Microsoft SQL Server + TLS

12. Установка сертификата

- Установите сертификат в хранилище Local Machine → Personal (Личное).
- Убедитесь, что учётная запись службы SQL Server имеет доступ к закрытому ключу сертификата.
- Сертификат должен находиться в допустимый период времени.

13. Настройка SQL Server

В SQL Server Configuration Manager:

- Разверните SQL Server Network Configuration → Protocols for <instance>.
- Правой кнопкой → Properties.
- Перейдите на вкладку Certificate → выберите сертификат из выпадающего списка.
- На вкладке Flags → установите ForceEncryption =
- Перезапустите службу SQL Server.

14. Проверка подключения

В SQL Server Management Studio:

- Connect to Server → Options.
- Connection Properties → выберите Encrypt connection.
- Выполните подключение.

Вариант Г: Oracle Database + TLS (Native Encryption)

15. Настройка sqlnet.ora (сервер)

```
# В $ORACLE_HOME/network/admin/sqlnet.ora
SQLNET.ENCRYPTION_SERVER = required
SQLNET.ENCRYPTION_TYPES_SERVER = (AES256)
SQLNET.CRYPTO_CHECKSUM_SERVER = required
SQLNET.CRYPTO_CHECKSUM_TYPES_SERVER = (SHA256)
```

16. Настройка sqlnet.ora (клиент)

```
# На стороне клиента
SQLNET.ENCRYPTION_CLIENT = required
SQLNET.ENCRYPTION_TYPES_CLIENT = (AES256)
```

17. Проверка шифрования

```
-- Просмотр статуса шифрования для активных сессий
SELECT
    s.sid,
    s.serial#,
    s.username,
    nc.network_service_banner
FROM v$session s
JOIN v$session_connect_info sci ON s.sid = sci.sid
JOIN v$net_connection nc ON sci.sid = nc.sid
WHERE s.username IS NOT NULL;
```

18. Сравнительный анализ режимов SSL/TLS

19. Проверка отсутствия шифрования

Для PostgreSQL:

```
# Попытка подключения без SSL (должна быть отклонена при настройке hostnossl
reject)
psql "sslmode=disable host=localhost dbname=postgres user=postgres"
```

Для MySQL:

```
# Должно быть отклонено при require_secure_transport=ON
mysql -u username -p --ssl-mode=DISABLED
```

20. 3.8. Очистка (для отчёта)

```
# PostgreSQL
sudo systemctl stop postgresql
rm /var/lib/postgresql/*/main/server.crt /var/lib/postgresql/*/main/server.key

# MySQL
sudo systemctl stop mysql
rm /etc/mysql/*.pem
```

Форма представления результата:

Листинги всех выполненных SQL-скриптов с комментариями.

Критерии оценки:

Оценка «отлично» ставится, если задание выполнено верно.

Оценка «хорошо» ставится, если ход выполнения задания верный, но была допущена одна или две ошибки, приведшие к неправильному результату.

Оценка «удовлетворительно» ставится, если приведено неполное выполнение задания.

Оценка «неудовлетворительно» ставится, если задание не выполнено.

Тема 1.3. Организация защиты данных в хранилищах

Лабораторное занятие №21

Организация резервного копирования с шифрованием в Oracle Database

Цель работы: изучить механизмы шифрования резервных копий в Oracle Recovery Manager (RMAN), научиться настраивать различные режимы шифрования (прозрачное шифрование TDE, парольное и комбинированное), выполнять создание зашифрованных резервных копий и их восстановление.

Выполнив работу, вы будете уметь:

- У 2.3.1 Дать независимую оценку уровня безопасности
- У 2.3.2 Производить регламентное обновление программного обеспечения;
- У.2.3.3 Разрабатывать перечень рекомендаций по дальнейшей эксплуатации БД с максимальной защитой хранящейся информации;

Материальное обеспечение:

Oracle Database

Задание:

1. Изучить три режима шифрования резервных копий в Oracle RMAN.
2. Настроить инфраструктуру безопасности для прозрачного шифрования (TDE).
3. Выполнить настройку RMAN для шифрования резервных копий.
4. Создать зашифрованные резервные копии базы данных в разных режимах.
5. Выполнить проверку статуса шифрования созданных копий.
6. Произвести восстановление из зашифрованной резервной копии.
7. Сравнить эффективность и удобство различных режимов шифрования.

Порядок выполнения работы:

1. Подготовка среды
-- Проверка режима ARCHIVELOG (обязательно для онлайн-бэкапов)
ARCHIVE LOG LIST;

-- Если не включен:
SHUTDOWN IMMEDIATE;
STARTUP MOUNT;
ALTER DATABASE ARCHIVELOG;
ALTER DATABASE OPEN;
2. Режимы шифрования RMAN

Режим	Описание	Ключевая особенность
TDE (прозрачный)	Ключи хранятся в keystore	Не требуется ввод пароля при операциях
Парольный	Пароль задаётся при создании бэкапа	Требуется ввод пароля для восстановления
Dual-mode	Комбинация TDE и пароля	Максимальная гибкость

3. Настройка прозрачного шифрования (TDE)
 - Настройка keystore

В файле sqlnet.ora указать расположение keystore:

```
ENCRYPTION_WALLET_LOCATION = (SOURCE = (METHOD = FILE) (METHOD_DATA = (DIRECTORY  
= /u01/app/oracle/admin/ORCL/wallet)))  
– Создание и открытие keystore  
-- Создание keystore
```

```

ADMINISTER KEY MANAGEMENT CREATE KEYSTORE '/u01/app/oracle/admin/ORCL/wallet'
IDENTIFIED BY "WalletPass123";

-- Открытие keystore
ADMINISTER KEY MANAGEMENT SET KEYSTORE OPEN
IDENTIFIED BY "WalletPass123";

-- Установка мастер-ключа
ADMINISTER KEY MANAGEMENT SET KEY
IDENTIFIED BY "WalletPass123" WITH BACKUP;
    - Проверка статуса keystore
SELECT * FROM V$ENCRYPTION_WALLET;
3.4. Настройка RMAN для шифрования
sql
-- Подключение к RMAN
rman target /

-- Включение прозрачного шифрования для всех бэкапов
CONFIGURE ENCRYPTION FOR DATABASE ON;

-- Выбор алгоритма шифрования (AES128, AES192, AES256)
CONFIGURE ENCRYPTION ALGORITHM 'AES256';

-- Просмотр текущих настроек
SHOW ALL;
4. Создание зашифрованных резервных копий
    - Прозрачное шифрование (TDE)
-- Полная резервная копия с прозрачным шифрованием
BACKUP DATABASE PLUS ARCHIVELOG;

-- Проверка, что бэкап зашифрован
LIST BACKUP OF DATABASE;
-- В столбце Encrypted должно быть YES[citation:2]
    - Парольное шифрование
-- Включение парольного режима (переопределяет TDE)
SET ENCRYPTION ON IDENTIFIED BY 'StrongBackupPass123' ONLY;

-- Создание бэкапа с паролем
BACKUP DATABASE;

-- Отключение парольного режима
SET ENCRYPTION OFF;
    - Комбинированное шифрование (Dual-mode)
-- Dual-mode требует оба метода
SET ENCRYPTION ON IDENTIFIED BY 'StrongPass123';
BACKUP DATABASE;
-- Ключ TDE + пароль (восстановление возможно любым
способом)[citation:1][citation:4]
5. Просмотр информации о зашифрованных бэкапах
-- Список всех бэкапов с информацией о шифровании
LIST BACKUP SUMMARY;

-- Детальная информация о конкретном бэкапе
LIST BACKUP OF DATABASE;

-- Через SQL*Plus
SELECT recid, tag, status, encrypted
FROM V$BACKUP_SET;
6. Восстановление из зашифрованных бэкапов
    - Восстановление из TDE-бэкапа
-- Keystore должен быть открыт

```

```
ADMINISTER KEY MANAGEMENT SET KEYSTORE OPEN IDENTIFIED BY "WalletPass123";

-- Восстановление (проходит автоматически)
RESTORE DATABASE;
RECOVER DATABASE;

-- Восстановление из парольного бэкапа
-- Указание пароля для дешифрования
SET DECRYPTION IDENTIFIED BY 'StrongBackupPass123';

-- Восстановление
RESTORE DATABASE;
RECOVER DATABASE;
7. Очистка (для отчёта)
-- Удаление настроек шифрования
CONFIGURE ENCRYPTION FOR DATABASE CLEAR;

-- Удаление бэкапов
DELETE BACKUP;

-- Закрытие keystore
ADMINISTER KEY MANAGEMENT SET KEYSTORE CLOSE IDENTIFIED BY "WalletPass123";
```

Форма представления результата:

Листинги выполненных команд с комментариями.

Критерии оценки:

Оценка «отлично» ставится, если задание выполнено верно.

Оценка «хорошо» ставится, если ход выполнения задания верный, но была допущена одна или две ошибки, приведшие к неправильному результату.

Оценка «удовлетворительно» ставится, если приведено неполное выполнение задания.

Оценка «неудовлетворительно» ставится, если задание не выполнено.

Тема 1.4 Векторные базы данных

Лабораторное занятие №22

Установка и настройка векторной базы данных (например, Milvus, Pinecone или Weaviate)

Цель работы: изучить принципы работы векторных баз данных, освоить установку и настройку Weaviate (локально) и Pinecone (облачная), научиться выполнять операции вставки векторных представлений (эмбеддингов) и поиска по сходству.

Выполнив работу, вы будете уметь:

У 2.3.1 Дать независимую оценку уровня безопасности

У 2.3.2 Производить регламентное обновление программного обеспечения;

У.2.3.3 Разрабатывать перечень рекомендаций по дальнейшей эксплуатации БД с максимальной защитой хранящейся информации;

Материальное обеспечение:

Milvus, Pinecone или Weaviate

Задание:

1. Изучить теоретические основы векторных баз данных и их применение (семантический поиск, RAG).
2. Развернуть Weaviate с использованием Docker Compose.

3. Развернуть Pinecone (облачный сервис) и создать индекс.
4. Сгенерировать эмбединги текстовых документов с помощью модели sentence-transformers.
5. Выполнить вставку данных в обе векторные БД.
6. Произвести семантический поиск и оценить результаты.
7. Сравнить локальное и облачное решения.

Порядок выполнения работы:

1. Установка Weaviate (локально)

```
# Создание docker-compose.yml
cat > docker-compose.yml << EOF
services:
  weaviate:
    image: semitechnologies/weaviate:latest
    ports:
      - "8080:8080"
      - "50051:50051"
    environment:
      AUTHENTICATION_ANONYMOUS_ACCESS_ENABLED: 'true'
      DEFAULT_VECTORIZER_MODULE: 'none'
      QUERY_DEFAULTS_LIMIT: 25
EOF
```

```
# Запуск
```

```
docker-compose up -d
```

2. Установка Python-зависимостей

```
pip install sentence-transformers weaviate-client pinecone
```

3. Генерация эмбедингов

```
from sentence_transformers import SentenceTransformer

model = SentenceTransformer('all-MiniLM-L6-v2')
texts = ["Векторные базы данных используются для семантического поиска",
        "Weaviate — открытая векторная БД",
        "Pinecone — облачное решение для векторов"]

embeddings = model.encode(texts).tolist()
```

4. Работа с Weaviate

```
import weaviate
from weaviate.classes.config import Configure

client = weaviate.connect_to_local()

# Создание коллекции
client.collections.create(
    name="Documents",
    vectorizer_config=Configure.Vectorizer.none()
)

# Вставка данных
collection = client.collections.get("Documents")
for i, (text, vector) in enumerate(zip(texts, embeddings)):
    collection.data.insert(
        properties={"content": text},
        vector=vector,
        uuid=f"doc_{i}"
    )

# Поиск
query_vector = model.encode(["Что такое векторная БД?"]).tolist()
results = collection.query.near_vector(
    near_vector=query_vector,
```

```

        limit=3
    )
5.    Работа с Pinecone
from pinecone import Pinecone

# Инициализация (требуется API-ключ)
pc = Pinecone(api_key="YOUR_API_KEY")

# Создание индекса
pc.create_index(
    name="lab-index",
    dimension=384, # размерность all-MiniLM-L6-v2
    metric="cosine"
)

# Подключение к индексу
index = pc.Index("lab-index")

# Вставка векторов
vectors = [(f"id_{i}", embeddings[i], {"text": texts[i]})
            for i in range(len(texts))]
index.upsert(vectors=vectors)

# Поиск
results = index.query(
    vector=query_vector[0],
    top_k=3,
    include_metadata=True
)
6.    Очистка ресурсов
# Weaviate
client.collections.delete("Documents")
client.close()

# Pinecone
pc.delete_index("lab-index")

```

Форма представления результата:

Листинги выполненных команд с комментариями.

Критерии оценки:

Оценка «отлично» ставится, если задание выполнено верно.

Оценка «хорошо» ставится, если ход выполнения задания верный, но была допущена одна или две ошибки, приведшие к неправильному результату.

Оценка «удовлетворительно» ставится, если приведено неполное выполнение задания.

Оценка «неудовлетворительно» ставится, если задание не выполнено.

Тема 1.4 Векторные базы данных

Лабораторное занятие №23

Создание и управление коллекциями данных в векторной базе (создание индексов и добавление векторов)

Цель работы: изучить основные операции управления коллекциями в векторной базе данных: создание коллекций, настройку векторных индексов, вставку векторных данных и организацию эффективного поиска.

Выполнив работу, вы будете уметь:

- У 2.3.1 Дать независимую оценку уровня безопасности
- У 2.3.2 Производить регламентное обновление программного обеспечения;
- У.2.3.3 Разрабатывать перечень рекомендаций по дальнейшей эксплуатации БД с максимальной защитой хранящейся информации;

Материальное обеспечение:

Milvus, Pinecone или Weaviate, Docker Compose

Задание:

1. Создать векторную коллекцию с заданной размерностью векторов.
2. Настроить векторный индекс с выбором метрики расстояния (косинусное сходство, евклидово расстояние).
3. Добавить векторные данные в коллекцию (пакетная вставка).
4. Выполнить поиск ближайших соседей (ANN/KNN).
5. Продемонстрировать фильтрацию результатов по скалярным полям.
6. Обновить и удалить векторные записи.

Порядок выполнения работы:

1. Подготовка среды
Установка библиотек (Python)
`pip install pymilvus sentence-transformers`
2. Создание коллекции

Коллекция — аналог таблицы в реляционной БД, объединяющий векторы с метаданными.

```
from pymilvus import connections, Collection, FieldSchema, CollectionSchema,
DataType
```

```
# Подключение к Milvus
connections.connect(host='localhost', port='19530')
```

```
# Определение схемы коллекции
fields = [
    FieldSchema(name="id", dtype=DataType.INT64, is_primary=True,
auto_id=True),
    FieldSchema(name="vector", dtype=DataType.FLOAT_VECTOR, dim=384),
    FieldSchema(name="text", dtype=DataType.VARCHAR, max_length=500),
    FieldSchema(name="category", dtype=DataType.VARCHAR, max_length=50)
]
```

```
schema = CollectionSchema(fields, description="Коллекция для семантического
поиска")
```

```
collection = Collection(name="documents", schema=schema)
```

Основные параметры коллекции:

- Имя коллекции — уникальный идентификатор
- Размерность вектора — должна соответствовать модели эмбедингов (384 для all-MiniLM-L6-v2)

- Скалярные поля — для хранения метаданных и фильтрации

3. Создание векторного индекса

Индекс ускоряет поиск ближайших соседей (ANN).

```
# Параметры индекса HNSW (оптимальный для большинства задач)
index_params = {
    "metric_type": "COSINE", # метрика сходства
    "index_type": "HNSW", # тип индекса
    "params": {"M": 16, "efConstruction": 200} # параметры HNSW
}
```

```
collection.create_index(field_name="vector", index_params=index_params)
```

Метрики расстояния:

Метрика	Формула	Применение
COSINE	$\cos(\theta)$	Текстовые эмбединги
L2/EUCLIDEAN	$\sqrt{\sum(x_i - y_i)^2}$	Компьютерное зрение
IP/DOT	$\sum(x_i \times y_i)$	Оптимизированные модели

Типы индексов:

- HNSW — высокая скорость, хорошая точность (рекомендуется)
- FLAT — точный поиск (100% recall), медленный
- IVF_FLAT — баланс скорости и точности

4. Генерация и вставка векторов

```
from sentence_transformers import SentenceTransformer

model = SentenceTransformer('all-MiniLM-L6-v2')

# Тексты для индексации
texts = [
    "Векторные базы данных оптимизированы для семантического поиска",
    "HNSW обеспечивает быстрый поиск ближайших соседей",
    "Коллекции группируют векторы и метаданные"
]

# Генерация эмбедингов
embeddings = model.encode(texts).tolist()

# Вставка данных
data = [
    embeddings,
    texts,
    ["AI", "Database", "Vector"]
]

collection.insert(data)
collection.flush() # сохранение на диск
```

Ограничения:

- Максимальный размер пакета: 500 векторов за операцию
- Индекс требует минимум 1000 векторов для обучения

5. Поиск векторов

```
# Поиск ближайших соседей
query_text = "Что такое семантический поиск?"
query_vector = model.encode([query_text]).tolist()

# Настройка поиска
search_params = {
    "metric_type": "COSINE",
    "params": {"ef": 64} # параметр точности поиска
}

results = collection.search(
    data=query_vector,
    anns_field="vector",
    param=search_params,
    limit=3, # Top-K результатов
    output_fields=["text", "category"]
)

for result in results[0]:
```

```

print(f"Сходство: {result.score:.4f} - {result.entity.get('text')}")
6.    Фильтрация по скалярным полям
# Поиск с фильтрацией по категории
filter_expr = "category == 'Database'"

results = collection.search(
    data=query_vector,
    anns_field="vector",
    param=search_params,
    limit=5,
    expr=filter_expr,          # фильтр перед поиском
    output_fields=["text"]
)
7.    Управление данными
# Удаление по ID
collection.delete("id in [1, 2, 3]")

# Обновление (удаление + вставка)
collection.delete("id == 5")
collection.insert([new_embedding], ["Новый текст"], ["NewCategory"])

# Получение статистики
print(f"Количество векторов: {collection.num_entities}")
8.    Очистка
# Удаление индекса и коллекции
collection.drop_index()
collection.drop()

```

Форма представления результата:

Листинги выполненных команд с комментариями.

Критерии оценки:

Оценка «отлично» ставится, если задание выполнено верно.

Оценка «хорошо» ставится, если ход выполнения задания верный, но была допущена одна или две ошибки, приведшие к неправильному результату.

Оценка «удовлетворительно» ставится, если приведено неполное выполнение задания.

Оценка «неудовлетворительно» ставится, если задание не выполнено.

Тема 1.4 Векторные базы данных

Лабораторное занятие №24

Реализация функции поиска ближайших соседей (Nearest Neighbor Search) на примере текстовых или изображений

Цель работы: изучить алгоритмы поиска ближайших соседей (точный и приближённый), научиться генерировать векторные представления (эмбединги) для текстовых и графических данных, реализовать поиск семантически близких объектов с использованием различных метрик расстояния.

Выполнив работу, вы будете уметь:

У 2.3.1 Дать независимую оценку уровня безопасности

У 2.3.2 Производить регламентное обновление программного обеспечения;

У.2.3.3 Разрабатывать перечень рекомендаций по дальнейшей эксплуатации БД с максимальной защитой хранящейся информации;

Материальное обеспечение:

Задание:

1. Выбрать тип данных для поиска (текст или изображения).
2. Сгенерировать эмбединги с помощью предобученной модели (sentence-transformers или CLIP).
3. Реализовать точный поиск ближайших соседей методом полного перебора.
4. Реализовать приближённый поиск с использованием индекса (HNSW или IVFFlat) .
5. Сравнить скорость и точность двух подходов.
6. Визуализировать результаты поиска.

Порядок выполнения работы:

1. Подготовка среды

```
pip install sentence-transformers numpy scikit-learn
# Для работы с изображениями дополнительно:
pip install torch torchvision pillow
```

2. Генерация эмбедингов для текстов

```
from sentence_transformers import SentenceTransformer

model = SentenceTransformer('all-MiniLM-L6-v2') # размерность 384

documents = [
    "Векторные базы данных используются для семантического поиска",
    "HNSW обеспечивает быстрый поиск ближайших соседей",
    "Косинусное сходство измеряет угол между векторами",
    "Поиск ближайших соседей применяется в рекомендательных системах"
]
```

```
embeddings = model.encode(documents) # массив shape: (4, 384)
```

3. Генерация эмбедингов для изображений

```
from sentence_transformers import SentenceTransformer
from PIL import Image

model = SentenceTransformer('clip-ViT-B-32') # мультимодальная модель

image = Image.open('cat.jpg')
image_embedding = model.encode(image) # размерность 512
```

```
text_query = "a photo of a cat"
text_embedding = model.encode(text_query) # размерность 512
```

4. Точный поиск (Brute Force)

```
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity

def exact_search(query_embedding, embeddings, top_k=3):
    similarities = cosine_similarity([query_embedding], embeddings)[0]
    top_indices = np.argsort(similarities)[::-1][:top_k]
    return top_indices, similarities[top_indices]
```

```
# Использование
```

```
query = "семантический поиск документов"
query_embedding = model.encode([query])[0]
indices, scores = exact_search(query_embedding, embeddings)
```

5. Приближённый поиск с HNSW

```
import hnswlib

# Создание индекса
index = hnswlib.Index(space='cosine', dim=384)
index.init_index(max_elements=10000, ef_construction=200, M=16)
```

```
# Добавление векторов
index.add_items(embeddings, ids=np.arange(len(embeddings)))

# Настройка параметров поиска
index.set_ef(50) # размер динамического списка кандидатов

# Поиск
labels, distances = index.knn_query(query_embedding, k=3)
```

6. Сравнение производительности

```
import time

# Точный поиск
start = time.time()
exact_search(query_embedding, embeddings, top_k=3)
exact_time = time.time() - start

# Приближённый поиск
start = time.time()
index.knn_query(query_embedding, k=3)
approx_time = time.time() - start
```

```
print(f"Точный поиск: {exact_time:.4f} сек")
```

```
print(f"HNSW поиск: {approx_time:.4f} сек")
```

7. Полный пример для текстов

```
class NearestNeighborSearch:
    def __init__(self, model_name='all-MiniLM-L6-v2'):
        self.model = SentenceTransformer(model_name)
        self.embeddings = []
        self.documents = []

    def add_documents(self, docs):
        self.documents.extend(docs)
        new_embeddings = self.model.encode(docs)
        self.embeddings = np.vstack([self.embeddings, new_embeddings]) if
self.embeddings else new_embeddings

    def search(self, query, top_k=3, use_hnsw=False):
        query_emb = self.model.encode([query])[0]
        if use_hnsw:
            return self._hnsw_search(query_emb, top_k)
        return self._exact_search(query_emb, top_k)

    def _exact_search(self, query_emb, top_k):
        similarities = cosine_similarity([query_emb], self.embeddings)[0]
        indices = np.argsort(similarities)[::-1][:top_k]
        return [(self.documents[i], similarities[i]) for i in indices]

    def _hnsw_search(self, query_emb, top_k):
        if not hasattr(self, 'hnsw_index'):
            self._build_hnsw_index()
        labels, distances = self.hnsw_index.knn_query(query_emb, k=top_k)
        return [(self.documents[int(l)], 1 - d) for l, d in zip(labels[0],
distances[0])]

    def _build_hnsw_index(self):
        self.hnsw_index = hnswlib.Index(space='cosine',
dim=self.embeddings.shape[1])
        self.hnsw_index.init_index(max_elements=len(self.embeddings),
ef_construction=200, M=16)
        self.hnsw_index.add_items(self.embeddings,
ids=np.arange(len(self.embeddings)))
```

8. Визуализация результатов (для изображений)

```
import matplotlib.pyplot as plt
```

```

def visualize_results(query_image_path, results, scores):
    fig, axes = plt.subplots(1, len(results) + 1, figsize=(15, 5))

    # Запрос
    query_img = Image.open(query_image_path)
    axes[0].imshow(query_img)
    axes[0].set_title("Запрос")
    axes[0].axis('off')

    # Результаты
    for i, (img_path, score) in enumerate(zip(results, scores)):
        img = Image.open(img_path)
        axes[i+1].imshow(img)
        axes[i+1].set_title(f"Сходство: {score:.3f}")
        axes[i+1].axis('off')

    plt.show()

```

9. Оценка качества (Recall@K)

```

def evaluate_recall(exact_results, approx_results, k=3):
    exact_set = set(exact_results[:k])
    approx_set = set(approx_results[:k])
    intersection = exact_set.intersection(approx_set)
    return len(intersection) / k

```

Пример использования

```

exact_top3 = [0, 1, 2] # ID точного поиска
approx_top3 = [0, 1, 3] # ID приближённого поиска
recall = evaluate_recall(exact_top3, approx_top3, k=3)
print(f"Recall@3: {recall:.2f}")

```

Форма представления результата:

Листинги выполненных скриптов с комментариями.

Критерии оценки:

Оценка «отлично» ставится, если задание выполнено верно.

Оценка «хорошо» ставится, если ход выполнения задания верный, но была допущена одна или две ошибки, приведшие к неправильному результату.

Оценка «удовлетворительно» ставится, если приведено неполное выполнение задания.

Оценка «неудовлетворительно» ставится, если задание не выполнено.

Тема 1.4 Векторные базы данных

Лабораторное занятие №25

Интеграция векторной базы данных с Python для загрузки и обработки векторов

Цель работы: изучить способы интеграции векторных баз данных с Python, освоить операции создания коллекций, добавления векторных данных, выполнения поиска по сходству и обработки метаданных для семантического поиска.

Выполнив работу, вы будете уметь:

У 2.3.1 Дать независимую оценку уровня безопасности

У 2.3.2 Производить регламентное обновление программного обеспечения;

У.2.3.3 Разрабатывать перечень рекомендаций по дальнейшей эксплуатации БД с максимальной защитой хранящейся информации;

Материальное обеспечение:

Milvus, Pinecone или Weaviate, Docker Compose

Задание:

1. Установить и настроить векторную базу данных (Chroma, Qdrant или pgvector).
2. Подключиться к векторной БД из Python-скрипта.
3. Сгенерировать эмбединги для текстовых документов с помощью sentence-transformers.
4. Выполнить загрузку векторов в коллекцию с метаданными.
5. Реализовать функцию семантического поиска по запросу пользователя.
6. Выполнить операции обновления и удаления векторных записей.

Порядок выполнения работы:

1. Установка зависимостей

```
pip install chromadb sentence-transformers langchain
```

2. Инициализация модели эмбедингов

```
from sentence_transformers import SentenceTransformer
```

```
# Загрузка модели для создания эмбедингов [citation:1][citation:4]
model = SentenceTransformer('all-MiniLM-L6-v2')
```

3. Подготовка данных

```
documents = [
    {"text": "Векторные базы данных оптимизированы для семантического поиска",
     "category": "database", "source": "article1"},
    {"text": "ChromaDB — легковесная векторная БД для Python-приложений",
     "category": "database", "source": "article2"},
    {"text": "Эмбединги преобразуют текст в числовые векторы фиксированной размерности",
     "category": "nlp", "source": "guide"}
]
```

```
# Генерация эмбедингов
embeddings = model.encode([doc["text"] for doc in documents])
```

4. Подключение к векторной БД (Chroma)

```
import chromadb
from chromadb.config import Settings

# Подключение к клиенту ChromaDB [citation:1][citation:3]
client = chromadb.Client(Settings(
    chroma_db_impl="duckdb+parquet",
    persist_directory="./chroma_db"
))
```

```
# Создание коллекции
collection = client.create_collection(
    name="documents_collection",
    metadata={"hnsw:space": "cosine"}
)
```

5. Загрузка векторов и метаданных

```
# Добавление документов в коллекцию
collection.add(
    embeddings=embeddings,
    documents=[doc["text"] for doc in documents],
    metadatas=[{"category": doc["category"], "source": doc["source"]} for doc
in documents],
    ids=[f"doc_{i}" for i in range(len(documents))]
)
```

```
print(f"Загружено {collection.count()} векторов")
```

6. Выполнение поиска по сходству

```
def semantic_search(query: str, top_k: int = 3):
```

```

"""Поиск ближайших соседей по смыслу [citation:1][citation:4]"""
# Векторизация запроса
query_embedding = model.encode([query]).tolist()

# Поиск
results = collection.query(
    query_embeddings=query_embedding,
    n_results=top_k,
    include=["documents", "metadatas", "distances"]
)

return results

# Пример использования
query = "Как найти похожие документы?"
results = semantic_search(query)

for i, (doc, distance, metadata) in enumerate(zip(
    results['documents'][0],
    results['distances'][0],
    results['metadatas'][0]
)):
    print(f"{i+1}. {doc} (сходство: {1-distance:.3f}, категория: {metadata['category']})")

```

7. Операции CRUD с векторами

```

# Обновление вектора [citation:3]
collection.update(
    ids=["doc_0"],
    documents=["Обновленный текст документа"],
    embeddings=model.encode(["Обновленный текст документа"]).tolist()
)

# Удаление вектора
collection.delete(ids=["doc_1"])

# Получение информации о коллекции
print(f"Количество документов: {collection.count()}")

```

8. Интеграция через PyVectorDB (универсальный подход)

```

from pyvectordb import ChromaDB, Vector
from pyvectordb.distance_function import DistanceFunction

# Создание клиента с использованием универсальной обертки [citation:3]
vector_db = ChromaDB(
    host="localhost",
    port=8000,
    collection_name="my_collection",
    distance_function=DistanceFunction.COSINE,
)

# Создание векторов с метаданными
v1 = Vector(
    embedding=[0.1, 0.2, 0.3],
    metadata={"text": "Пример документа"}
)

# Вставка и поиск
vector_db.insert_vector(v1)
neighbors = vector_db.get_neighbor_vectors(v1, n=5)

```

9. Полный рабочий пример

```

class VectorDatabaseIntegration:
    def __init__(self, collection_name: str = "knowledge_base"):
        self.model = SentenceTransformer('all-MiniLM-L6-v2')
        self.client = chromadb.Client()

```

```

self.collection = self.client.create_collection(collection_name)

def add_documents(self, texts: list[str], metadatas: list[dict] = None):
    embeddings = self.model.encode(texts).tolist()
    ids = [f"id_{i}" for i in range(len(texts))]
    self.collection.add(
        embeddings=embeddings,
        documents=texts,
        metadatas=metadatas or [{}] * len(texts),
        ids=ids
    )

def search(self, query: str, top_k: int = 3, filter_criteria: dict = None):
    query_emb = self.model.encode([query]).tolist()
    return self.collection.query(
        query_embeddings=query_emb,
        n_results=top_k,
        where=filter_criteria
    )

```

10. Очистка ресурсов

```

# Удаление коллекции
client.delete_collection("documents_collection")

# Закрытие подключения
# (Chroma не требует явного закрытия)

```

Форма представления результата:

Листинги выполненных скриптов с комментариями.

Критерии оценки:

Оценка «отлично» ставится, если задание выполнено верно.

Оценка «хорошо» ставится, если ход выполнения задания верный, но была допущена одна или две ошибки, приведшие к неправильному результату.

Оценка «удовлетворительно» ставится, если приведено неполное выполнение задания.

Оценка «неудовлетворительно» ставится, если задание не выполнено.

Тема 1.4 Векторные базы данных

Лабораторное занятие №26

Проведение кластеризации данных в векторной базе с использованием встроенных функций

Цель работы: изучить механизмы кластеризации векторных данных, реализованные на уровне векторной базы данных, освоить методы автоматической группировки похожих векторов (LSH-багетирование, DBSCAN) и научиться интерпретировать результаты кластеризации для анализа структуры данных.

Выполнив работу, вы будете уметь:

У 2.3.1 Дать независимую оценку уровня безопасности

У 2.3.2 Производить регламентное обновление программного обеспечения;

У.2.3.3 Разрабатывать перечень рекомендаций по дальнейшей эксплуатации БД с максимальной защитой хранящейся информации;

Материальное обеспечение:

Milvus, Pinecone или Weaviate, Docker Compose

Задание:

1. Изучить встроенные механизмы кластеризации в векторной БД (на примере vecDB или pgvector-cluster).
2. Создать коллекцию векторных данных с заданной размерностью.
3. Загрузить набор векторов (текстовые эмбединги или изображения).
4. Выполнить автоматическую кластеризацию с использованием встроенных функций:
5. LSH-бакетирование для группировки похожих векторов;
6. DBSCAN-кластеризация (для pgvector).
7. Проанализировать распределение векторов по кластерам.
8. Визуализировать результаты кластеризации (методом снижения размерности).

Порядок выполнения работы:**1. Подготовка среды**

```
# Установка зависимостей (для работы с vecDB)
pip install numpy sentence-transformers matplotlib scikit-learn

# Клонирование vecDB (опционально)
git clone https://github.com/ItmoCV/vecDB.git
cd vecDB
docker compose up # запуск кластера с LSH-кластеризацией [citation:1]
```

2. Генерация тестовых данных

```
from sentence_transformers import SentenceTransformer
import numpy as np

# Загрузка модели эмбедингов
model = SentenceTransformer('all-MiniLM-L6-v2')

# Тексты для кластеризации (разные тематики)
texts = [
    # Тема: программирование
    "Python – популярный язык для анализа данных",
    "Java используется для корпоративных приложений",
    "Функции высшего порядка в JavaScript",
    # Тема: базы данных
    "Векторные БД ускоряют семантический поиск",
    "PostgreSQL поддерживает векторные индексы",
    "SQL оптимизирует реляционные запросы",
    # Тема: машинное обучение
    "Нейронные сети обучаются на больших данных",
    "Кластеризация группирует похожие объекты",
    "Глубокое обучение требует много GPU"
]

# Генерация эмбедингов
embeddings = model.encode(texts)
print(f"Создано {len(embeddings)} векторов размерностью {embeddings.shape[1]}")
```

3. LSH-кластеризация в vecDB

LSH (Locality-Sensitive Hashing) автоматически группирует похожие векторы в бакеты (buckets).

```
# Подключение к vecDB координатору (архитектура: координатор + шарды)
import requests

VECDB_URL = "http://localhost:8080"

# Создание коллекции с LSH-кластеризацией
collection_config = {
    "name": "documents",
    "vector_dimension": 384,
    "lsh_metric": "cosine", # метрика для LSH [citation:1]
    "num_buckets": 5        # количество бакетов для кластеризации
```

```

}

response = requests.post(f"{VECDB_URL}/collection", json=collection_config)
print(f"Коллекция создана: {response.json()}")

# Добавление векторов
for i, (text, emb) in enumerate(zip(texts, embeddings)):
    vector_data = {
        "id": f"doc_{i}",
        "embedding": emb.tolist(),
        "metadata": {"text": text, "category": text.split()[0]}
    }
    requests.post(f"{VECDB_URL}/vector", json=vector_data)

# LSH автоматически распределит векторы по бакетам [citation:1]
print("Векторы загружены, LSH-бакеты сформированы автоматически")

```

4. Кластеризация через pgvector-cluster (DBSCAN)

Для PostgreSQL + pgvector доступен модуль pgvector-cluster, реализующий DBSCAN-кластеризацию.

```

-- Создание таблицы с векторами (в PostgreSQL)
CREATE TABLE documents (
    id SERIAL PRIMARY KEY,
    content TEXT,
    embedding VECTOR(384)
);

-- Вставка векторов (значения эмбедингов из Python)
INSERT INTO documents (content, embedding) VALUES
    ('Python для анализа данных', '[0.12, -0.34, ...]'),
    ('Векторные базы данных', '[0.45, 0.12, ...]'),
    ('Кластеризация данных', '[0.33, -0.22, ...]');

-- Выполнение DBSCAN-кластеризации
-- (требуется установка pgvector-cluster) [citation:3]
SELECT * FROM dbscan_cluster(
    'documents',      -- таблица с векторами
    'embedding',      -- поле с векторами
    0.03,             -- epsilon (радиус окрестности)
    2                 -- minPoints (минимальное число точек в кластере)
);

```

5. Анализ результатов кластеризации

```

# Получение информации о бакетах (LSH) из vecDB
response = requests.get(f"{VECDB_URL}/collection/documents/buckets")
buckets = response.json()

print("Результаты LSH-кластеризации:")
for bucket in buckets:
    print(f"  Бакет {bucket['id']}: {len(bucket['vectors'])} векторов")

# Ручная кластеризация с KMeans для сравнения
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt

# KMeans для сравнения
kmeans = KMeans(n_clusters=3, random_state=42)
kmeans.fit(embeddings)
kmeans_labels = kmeans.labels_

print(f"\nKMeans                кластеризация:                {np.unique(kmeans_labels,
return_counts=True)})")

```

6. Визуализация кластеров

Снижение размерности для визуализации (2D)

```

pca = PCA(n_components=2)
embeddings_2d = pca.fit_transform(embeddings)

# Построение графика
plt.figure(figsize=(10, 8))
colors = ['red', 'blue', 'green', 'purple', 'orange']

# Использование меток из KMeans (для демонстрации)
for i, label in enumerate(kmeans_labels):
    plt.scatter(embeddings_2d[i, 0], embeddings_2d[i, 1],
                c=colors[label % len(colors)], s=100)
    plt.annotate(texts[i][:20], (embeddings_2d[i, 0], embeddings_2d[i, 1]),
                 fontsize=8)

plt.title("Визуализация кластеров (PCA)")
plt.xlabel("Компонента 1")
plt.ylabel("Компонента 2")
plt.savefig("clusters.png")
plt.show()

7. Оценка качества кластеризации
from sklearn.metrics import silhouette_score, davies_bouldin_score

# Силуэтный коэффициент (близость к -1: плохо, 1: отлично)
sil_score = silhouette_score(embeddings, kmeans_labels, metric='cosine')
print(f"Silhouette Score: {sil_score:.3f}")

# Индекс Дэвиса-Болдина (меньше = лучше)
db_score = davies_bouldin_score(embeddings, kmeans_labels)
print(f"Davies-Bouldin Index: {db_score:.3f}")

8. Очистка ресурсов
# Удаление коллекции в vecDB
requests.delete(f"{VECDB_URL}/collection/documents")

# Остановка Docker-контейнеров
# docker compose down

```

Форма представления результата:

Листинги выполненных скриптов с комментариями.

Критерии оценки:

Оценка «отлично» ставится, если задание выполнено верно.

Оценка «хорошо» ставится, если ход выполнения задания верный, но была допущена одна или две ошибки, приведшие к неправильному результату.

Оценка «удовлетворительно» ставится, если приведено неполное выполнение задания.

Оценка «неудовлетворительно» ставится, если задание не выполнено.

Тема 1.4 Векторные базы данных

Лабораторное занятие №27

Оптимизация индексов в векторной базе данных для увеличения скорости поиска

Цель работы: изучить влияние параметров векторных индексов на производительность поиска, освоить методы настройки индексов (HNSW, IVF, PQ) для достижения оптимального баланса между скоростью, точностью и потреблением памяти.

Выполнив работу, вы будете уметь:

У 2.3.1 Дать независимую оценку уровня безопасности

У 2.3.2 Производить регламентное обновление программного обеспечения;

У.2.3.3 Разрабатывать перечень рекомендаций по дальнейшей эксплуатации БД с максимальной защитой хранящейся информации;

Материальное обеспечение:

Milvus, Pinecone или Weaviate, Docker Compose

Задание:

1. Создать векторную коллекцию с тестовыми данными (10 000+ векторов).
2. Сравнить скорость поиска без индекса (Flat) и с разными типами индексов.
3. Настроить параметры индекса HNSW (M, efConstruction, efSearch).
4. Выполнить эксперименты по влиянию параметров на скорость и точность.
5. Настроить IVF-индекс с разным количеством кластеров (nlist).
6. Оценить использование квантования (PQ/SQ) для сжатия индекса.
7. Построить графики зависимости производительности от параметров.

Порядок выполнения работы:

1. Подготовка среды

```
pip install pymilvus sentence-transformers numpy matplotlib scikit-learn
```

2. Генерация тестовых данных

```
import numpy as np
from sentence_transformers import SentenceTransformer

# Генерация 20 000 случайных векторов (симуляция большого набора)
np.random.seed(42)
n_vectors = 20000
dim = 384 # размерность эмбедингов

# Реальные текстовые данные + случайные для нагрузки
model = SentenceTransformer('all-MiniLM-L6-v2')
base_texts = [f"Текст документа номер {i}" for i in range(1000)]
real_embeddings = model.encode(base_texts)

# Дополняем случайными векторами до 20000
random_embeddings = np.random.random((n_vectors - 1000,
dim)).astype(np.float32)
embeddings = np.vstack([real_embeddings, random_embeddings])
print(f"Сгенерировано {len(embeddings)} векторов размерностью {dim}")

3. Подключение к Milvus и создание коллекции
from pymilvus import connections, Collection, CollectionSchema, FieldSchema,
DataType

connections.connect(host='localhost', port='19530')

# Схема коллекции
fields = [
    FieldSchema(name="id", dtype=DataType.INT64, is_primary=True,
auto_id=True),
    FieldSchema(name="vector", dtype=DataType.FLOAT_VECTOR, dim=dim)
]
schema = CollectionSchema(fields)
collection = Collection(name="test_collection", schema=schema)

# Вставка данных
collection.insert([embeddings.tolist()])
collection.flush()
print(f"Вставлено {collection.num_entities} векторов")

4. Базовое измерение: поиск без индекса (Flat)
import time
```

```

def benchmark_search(collection, query_vectors, top_k=10, search_params=None):
    """Измерение времени выполнения поиска"""
    if search_params is None:
        search_params = {"metric_type": "COSINE"}

    # Прогрев
    collection.search(query_vectors[:1], anns_field="vector",
                      param=search_params, limit=top_k)

    # Измерение
    start = time.time()
    for q in query_vectors:
        collection.search([q], anns_field="vector",
                          param=search_params, limit=top_k)
    elapsed = time.time() - start

    return elapsed / len(query_vectors) # среднее время на запрос

# Flat-индекс (без индекса)
print("Поиск без индекса (Flat)...")
query_vectors = np.random.random((100, dim)).astype(np.float32)
flat_time = benchmark_search(collection, query_vectors)
print(f"Flat: {flat_time*1000:.2f} мс/запрос")

```

5. Настройка HNSW-индекса

HNSW (Hierarchical Navigable Small World) — графовый индекс с параметрами:

- **M** — количество связей на элемент (16-64)
- **efConstruction** — размер динамического списка при построении (200-800)
- **efSearch** — размер динамического списка при поиске (10-200)

```

def create_hnsw_index(collection, M=16, efConstruction=200):
    """Создание HNSW-индекса с заданными параметрами"""
    index_params = {
        "metric_type": "COSINE",
        "index_type": "HNSW",
        "params": {"M": M, "efConstruction": efConstruction}
    }
    collection.create_index(field_name="vector", index_params=index_params)
    collection.load()

def search_hnsw(collection, query_vectors, efSearch=64, top_k=10):
    """Поиск с настройкой efSearch"""
    search_params = {"metric_type": "COSINE", "params": {"ef": efSearch}}
    return benchmark_search(collection, query_vectors, top_k, search_params)

# Эксперименты с разными M
results_hnsw = []
for M in [8, 16, 32, 64]:
    print(f"\nТестирование HNSW с M={M}")
    collection.drop_index()
    create_hnsw_index(collection, M=M, efConstruction=200)

    # Тесты с разным efSearch
    for ef in [10, 50, 100, 200]:
        time_ms = search_hnsw(collection, query_vectors, efSearch=ef)
        results_hnsw.append({"M": M, "efSearch": ef, "time_ms": time_ms})
    print(f"  efSearch={ef}: {time_ms:.2f} мс/запрос")

```

6. Настройка IVF-индекса

IVF (Inverted File Index) — кластерный индекс с параметрами:

- **nlist** — количество кластеров (частичных центроидов)
- **nprobe** — количество кластеров для поиска

```

def create_ivf_index(collection, nlist=1024):

```

```

"""Создание IVF_FLAT индекса"""
index_params = {
    "metric_type": "COSINE",
    "index_type": "IVF_FLAT",
    "params": {"nlist": nlist}
}
collection.create_index(field_name="vector", index_params=index_params)
collection.load()

# Эксперименты с разным nlist
results_ivf = []
for nlist in [128, 512, 1024, 2048, 4096]:
    print(f"\nТестирование IVF с nlist={nlist}")
    collection.drop_index()
    create_ivf_index(collection, nlist=nlist)

    for nprobe in [1, 5, 10, 20, 50]:
        search_params = {"metric_type": "COSINE", "params": {"nprobe": nprobe}}
        time_ms = benchmark_search(collection, query_vectors,
search_params=search_params)
        results_ivf.append({"nlist": nlist, "nprobe": nprobe, "time_ms":
time_ms})
    print(f" nprobe={nprobe}: {time_ms:.2f} мс/запрос")

```

7. Оценка точности (Recall)

```

def calculate_recall(collection, query_vectors, ground_truth, search_params,
top_k=10):
    """Вычисление recall@k относительно точного поиска"""
    correct = 0
    total = 0

    for q, truth in zip(query_vectors[:50], ground_truth[:50]):
        results = collection.search([q], anns_field="vector",
param=search_params, limit=top_k)
        result_ids = set([r.id for r in results[0]])
        truth_ids = set(truth[:top_k])
        correct += len(result_ids.intersection(truth_ids))
        total += top_k

    return correct / total

# Получение ground truth (точный поиск через Flat)
print("Получение ground truth (может занять время)...")
collection.load()
flat_search_params = {"metric_type": "COSINE"}
ground_truth = []
for q in query_vectors[:50]:
    results = collection.search([q], anns_field="vector",
param=flat_search_params, limit=10)
    ground_truth.append([r.id for r in results[0]])

```

8. Квантование: PQ и SQ

```

# IVF_PQ (Product Quantization) – сжатие векторов
def create_pq_index(collection, nlist=1024, m=8, nbits=8):
    """Создание IVF_PQ индекса с квантованием"""
    index_params = {
        "metric_type": "COSINE",
        "index_type": "IVF_PQ",
        "params": {"nlist": nlist, "m": m, "nbits": nbits}
    }
    collection.drop_index()
    collection.create_index(field_name="vector", index_params=index_params)
    collection.load()

# IVF_SQ8 (Scalar Quantization) – 8-битное квантование

```

```

def create_sq_index(collection, nlist=1024):
    """Создание IVF_SQ8 индекса"""
    index_params = {
        "metric_type": "COSINE",
        "index_type": "IVF_SQ8",
        "params": {"nlist": nlist}
    }
    collection.drop_index()
    collection.create_index(field_name="vector", index_params=index_params)
    collection.load()

# Сравнение размера индекса
def get_index_size(collection):
    """Получение размера индекса в МБ"""
    segments = collection.get_index_stat()
    return sum(seg.index_size for seg in segments) / (1024 * 1024)

print("\nСравнение методов сжатия:")
for index_type, creator in [("IVF_FLAT", lambda: create_ivf_index(collection,
1024)),
                            ("IVF_PQ", lambda: create_pq_index(collection,
1024, 8, 8)),
                            ("IVF_SQ8", lambda: create_sq_index(collection,
1024))]:
    creator()
    size_mb = get_index_size(collection)
    time_ms = benchmark_search(collection, query_vectors[:50])
    print(f"{index_type}: размер={size_mb:.2f} МБ, время={time_ms:.2f} мс")

```

9. Визуализация результатов

```

import matplotlib.pyplot as plt

# График: скорость vs точность для HNSW
plt.figure(figsize=(12, 4))

plt.subplot(1, 3, 1)
for M in [16, 32, 64]:
    data = [r for r in results_hnsw if r["M"] == M]
    times = [r["time_ms"] for r in data]
    ef_values = [r["efSearch"] for r in data]
    plt.plot(ef_values, times, marker='o', label=f'M={M}')
plt.xlabel("efSearch")
plt.ylabel("Время (мс/запрос)")
plt.title("HNSW: влияние efSearch")
plt.legend()

# График: скорость vs точность для IVF
plt.subplot(1, 3, 2)
for nlist in [512, 1024, 2048]:
    data = [r for r in results_ivf if r["nlist"] == nlist]
    times = [r["time_ms"] for r in data]
    nprobe_values = [r["nprobe"] for r in data]
    plt.plot(nprobe_values, times, marker='s', label=f'nlist={nlist}')
plt.xlabel("nprobe")
plt.ylabel("Время (мс/запрос)")
plt.title("IVF: влияние nprobe")
plt.legend()

# График: сравнение методов сжатия
plt.subplot(1, 3, 3)
methods = ['Flat', 'IVF_FLAT', 'IVF_PQ', 'IVF_SQ8']
times = [flat_time, 5.2, 1.8, 2.5] # примерные значения
plt.bar(methods, times)
plt.ylabel("Время (мс/запрос)")

```

```
plt.title("Сравнение методов индексации")
plt.xticks(rotation=45)

plt.tight_layout()
plt.savefig("index_optimization_results.png")
plt.show()
10. Очистка ресурсов
collection.drop()
connections.disconnect()
```

Форма представления результата:

Листинги выполненных скриптов с комментариями.

Критерии оценки:

Оценка «отлично» ставится, если задание выполнено верно.

Оценка «хорошо» ставится, если ход выполнения задания верный, но была допущена одна или две ошибки, приведшие к неправильному результату.

Оценка «удовлетворительно» ставится, если приведено неполное выполнение задания.

Оценка «неудовлетворительно» ставится, если задание не выполнено.

Тема 1.4 Векторные базы данных

Лабораторное занятие №28

Обеспечение масштабируемости и высокой доступности векторной базы данных

Цель работы: изучить механизмы горизонтального масштабирования и отказоустойчивости векторных баз данных. Получить практические навыки развертывания кластера, балансировки нагрузки и тестирования доступности сервиса при сбоях узлов.

Выполнив работу, вы будете уметь:

У 2.3.1 Дать независимую оценку уровня безопасности

У 2.3.2 Производить регламентное обновление программного обеспечения;

У.2.3.3 Разрабатывать перечень рекомендаций по дальнейшей эксплуатации БД с максимальной защитой хранящейся информации;

Материальное обеспечение:

Milvus, Pinecone или Weaviate, Docker Compose

Задание:

1. Развернуть кластер векторной БД (например, Milvus или Qdrant) в режиме высокой доступности (HA).
2. Настроить репликацию данных и распределение индексов по сегментам (шардам).
3. Провести нагрузочное тестирование (вставка/поиск векторов) при работающем и отключенном узле.
4. Оценить время восстановления и отсутствие потери данных.

Порядок выполнения работы:

Подготовка:

- Установить Docker, Docker Compose, kubectl (для K8s), Python 3.8+.
- Скачать конфигурационные файлы для кластерного режима.

Основные шаги:

1. Развертывание кластера:
 - Запустить координатор, узел данных, индексный узел и прокси (для Milvus).

- Настроить replica_count и shards_count в конфигурации коллекции.
- 2. Настройка HA и репликации:
 - Указать минимум 2 реплики для сервисов (например, через docker-compose scale или replicas в K8s).
 - Включить постоянное хранилище (Persistent Volume) для данных.
- 3. Загрузка данных:
 - Сгенерировать тестовый набор из 100k векторов размерностью 128.
 - Выполнить вставку и построить индекс (IVF_FLAT, HNSW).
- 4. Тестирование доступности:
 - Запустить параллельные поисковые запросы.
 - Вручную остановить один из узлов базы данных.
 - Проверить, продолжают ли запросы успешно обрабатываться (failover).
- 5. Проверка масштабируемости:
 - Увеличить количество узлов данных с 1 до 3.
 - Измерить изменение пропускной способности (QPS) и задержки (p99 latency).

Форма представления результата:

Листинги выполненных скриптов с комментариями.

Критерии оценки:

Оценка «отлично» ставится, если задание выполнено верно.

Оценка «хорошо» ставится, если ход выполнения задания верный, но была допущена одна или две ошибки, приведшие к неправильному результату.

Оценка «удовлетворительно» ставится, если приведено неполное выполнение задания.

Оценка «неудовлетворительно» ставится, если задание не выполнено.

Тема 1.4 Векторные базы данных

Лабораторное занятие №29

Интеграция векторной базы данных в приложение для рекомендаций или кластеризации пользователей

Цель работы: освоить практические методы интеграции векторной базы данных (ВБД) в прикладное приложение. Получить навыки построения рекомендательной системы на основе эмбедингов пользователей и товаров, а также кластеризации пользовательских сессий.

Выполнив работу, вы будете уметь:

У 2.3.1 Дать независимую оценку уровня безопасности

У 2.3.2 Производить регламентное обновление программного обеспечения;

У.2.3.3 Разрабатывать перечень рекомендаций по дальнейшей эксплуатации БД с максимальной защитой хранящейся информации;

Материальное обеспечение:

Milvus, Pinecone или Weaviate, Docker Compose

Задание:

1. Подготовить набор данных (пользователи, товары/контент, взаимодействия).
2. Сгенерировать векторные представления (эмбединги) с помощью предобученной модели (BERT, SentenceTransformer или сверточная сеть).
3. Загрузить эмбединги в векторную БД (Qdrant, Milvus или Chroma).
4. Реализовать один из сценариев:

- Рекомендации: поиск похожих товаров (item-to-item) или рекомендации пользователю (user-to-item).
- Кластеризация: сегментация пользователей по эмбедингам поведения.
- 5. Оценить качество рекомендаций или интерпретируемость кластеров.

Порядок выполнения работы:

Подготовка:

- Установить Python 3.9+, векторную БД (например, Qdrant в Docker).
- Установить библиотеки: qdrant-client, sentence-transformers, pandas, scikit-learn, numpy.

Основные шаги:

1. Загрузка и предобработка данных:
 - Взять датасет (например, MovieLens, Amazon Reviews или синтетические логи кликов).
 - Выделить признаки: история просмотров, лайки, время на странице.
2. Генерация эмбедингов:


```
from sentence_transformers import SentenceTransformer
model = SentenceTransformer('all-MiniLM-L6-v2')
user_embeddings = model.encode(user_profiles) # профили в текстовом виде
item_embeddings = model.encode(item_descriptions)
```
3. Создание коллекций и индексация:
 - Создать коллекции users и items в ВБД.
 - Загрузить эмбединги с метаданными (id, категория, возраст и т.п.).
 - Построить индекс (HNSW или IVF).
4. Реализация логики приложения:

Вариант А — Рекомендации:

- Принять ID пользователя → получить его эмбединг из ВБД.
- Выполнить поиск ближайших N векторов товаров (collection.search).
- Вернуть топ-N товаров с фильтрацией по категориям.

Вариант Б — Кластеризация:

- Выгрузить все эмбединги пользователей из ВБД.
- Применить KMeans (sklearn) к векторам.
- Визуализировать кластеры (PCA/t-SNE) и описать портреты сегментов.
- 5. Оценка результатов:
 - Для рекомендаций: метрики Precision@K, Recall@K.
 - Для кластеризации: силуэт-коэффициент, интерпретируемость.

Форма представления результата:

Листинги выполненных скриптов с комментариями.

Критерии оценки:

Оценка «отлично» ставится, если задание выполнено верно.

Оценка «хорошо» ставится, если ход выполнения задания верный, но была допущена одна или две ошибки, приведшие к неправильному результату.

Оценка «удовлетворительно» ставится, если приведено неполное выполнение задания.

Оценка «неудовлетворительно» ставится, если задание не выполнено.