МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования

Санкт-Петербургский национальный исследовательский университет ИТМО

Мегафакультет трансляционных информационных технологий

Факультет информационных технологий и программирования

Лабораторная работа №1

По дисциплине «Технологии управления данными»

Выполнили студент группы №M3306

Бочагова Екатерина

Проверил

Повышев Владислав Вячеславович

Санкт-Петербург

2025

Условие лабораторной работы:

Лабораторная работа №1. Разработка базы данных «Филиал» как источника данных.
В данной лабораторной работе требуется разработать прототип базы данных филиала торгового предприятия. База данных будет использована в дальнейшем как источник данных для наполнения хранилища данных.
Разрабатываемый прототип должен обеспечить хранение информации о следующих основных сущностях:
1 Покупатель;
2 Товар;
3 Сделка.
Информацию о покупателях можно ограничить уникальным идентификатором, а так же названием покупателя для обеспечения удобства работы с информацией.
Сущность Товар должна содержать следующие минимальные поля: уникальный номер товара, название товара, цена товара по каталогу. Следует отметить что каждый товар принадлежит к одной или нескольким категориям. Таким образом следует создать дополнительную сущность Категория, и обеспечить связь многие ко многим с сущностью Товар.
Сущность Сделка должна отражать информацию о факте покупки товара покупателем. Требуется отметить, что покупатель за одну покупку может купить несколько наименований товара в разном количестве, а также информацию о цене, за которую был куплен товар (она может отличатся от заявленной цены, например с учетом скидки), а также общую сумму сделки.
Целесообразно сделать связь между сущностями Сделка и Товар через промежуточную сущность содержащую информацию о количестве купленного товара и цена. Сущность Сделка будет содержать итоговую сумму покупки, а также дополнительную информацию, например дату совершения сделки.
На основании имеющейся информации необходимо создать реляционную модель данных, и получить утвердить модель у преподавателя.
На основании имеющейся реляционной модели создать даталогическую модель, в соответствии с требованиями типов данных, используемых на конкретной версии SQL Server, и так же утвердить у преподавателя.
На основании данных моделей создать скрипт создания базы данных, с учетом следующих требований:
1 Каждая таблица должна иметь суррогатный целочисленный первичный ключ;
2 Каждая таблица, помимо атрибутов необходимых для хранения данных в соответствии с заданиями, должны содержать атрибут rowguid – уникальный идентификатор в рамках всей системы, и ModifiedDate - дата/время добавления или последней модификации строки;
3 Все связи должны быть реализованы как дополнительные изменения таблицы после их создания;
4 База данных должна содержать минимально необходимое количество ограничений и триггеров необходимых для стабильной работы;
5 Рекомендовано использовать английский язык для названий сущностей, атрибутов и других объектов БД.
Проверить работоспособность созданного скрипта и убедится в стабильности его работы.
Подготовить два комплекта операций вставки в базу данных, обеспечивающих наполнение базы необходимыми для тестирования данными. Объем данных должен быть минимум двадцать пять строк для базовых сущностей, и минимум пятьдесят строк для сущностей, реализующих связь.
Создать две базы данных по разработанному скрипту, и наполнить соответствующим набором данных. Созданные базы данных обозначить как «Филиал Запад» и «Филиал Восток». При создании скрипта наполнения данными рекомендуется использовать ИИ.
Выбор реляционной СУБД остается за студентом. СУБД, и полученный набор скриптов необходимо согласовать с преподавателем.

Теоретическое описание решения:

План действий:

1) Напишем docker-compose.yaml для того, чтобы поднять бд в контейнере
2) Напишем скрипты для создания схемы
3) Напишем скрипты для заполнения тестовыми данными

Рассмотрим сущности в каждой бд, пропишем необходимые поля и выберем подходящие типы данных

Customer

- id (primary key)
- customer_name
- customer_uid (уникальный идентификатор)
- modified_date

Product

- id (primary key)
- sku (артикул)
- product_name
- price
- product_uid (уникальный идентификатор)
- modified_date

Category

- id (primary key)
- category_name
- category_uid (уникальный идентификатор)
- modified_date

Product_Category (реализуем связь многие-ко-многим между товаром и категорией)

- id (primary key)
- product_id (foreign key с таблицей product)
- category_id ((foreign key с таблицей category)
- uid_product_category (уникальный идентификатор для этой связи)
- modified_date

Sale (покупка)

- id (primary key)
- customer_id (foreign key с таблицей customer)
- sale_date (дата покупки)
- in_total (итоговая сумма сделки)
- sale_uid

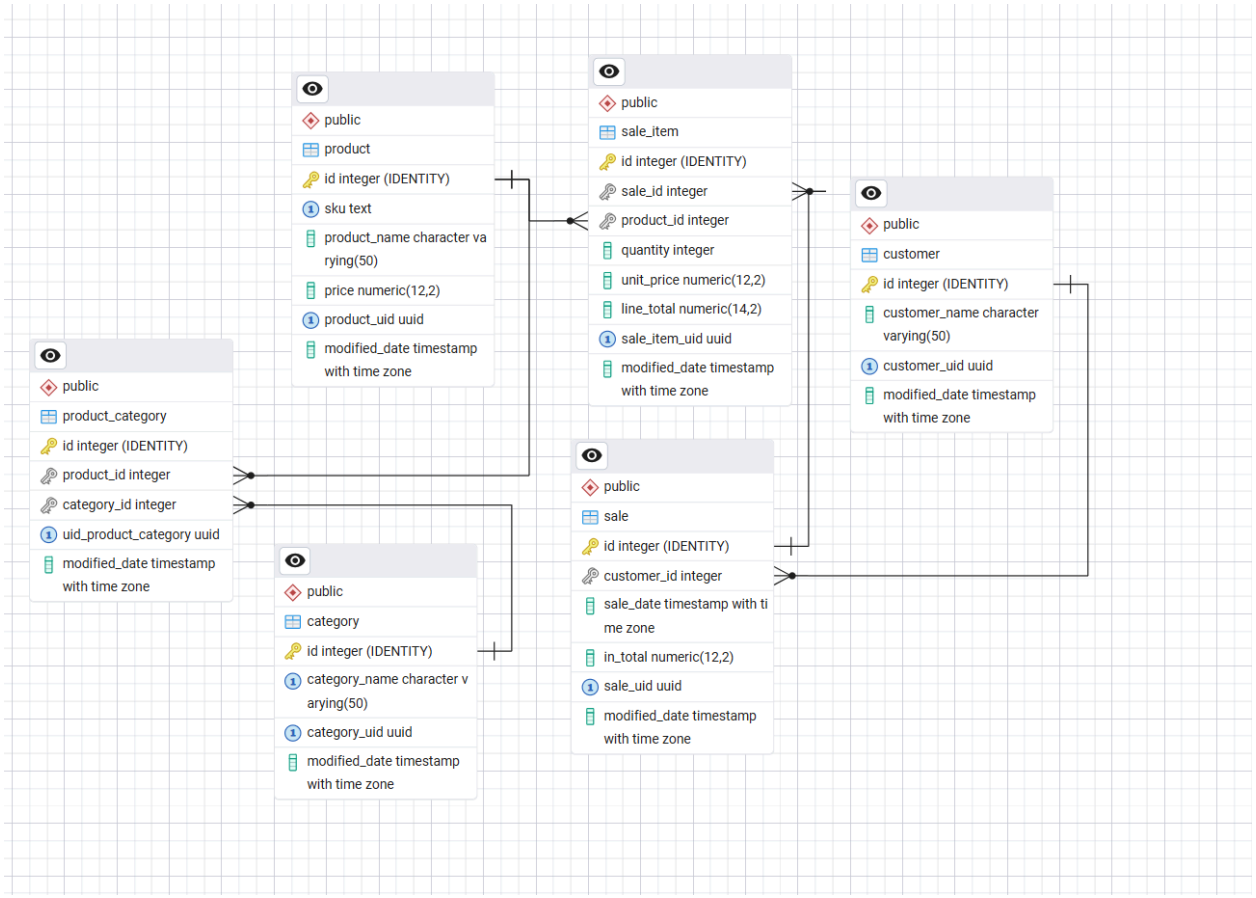
- modified_date

Sale_Item (позиция покупки - будем реализовывать связь многие ко многим)
- id (primary key)
- sale_id ((foreign key с таблицей sale)
- product_id ((foreign key с таблицей product)
- quantity
- unit_price
- line_total (сумма по строке = quantity * unit_price)
- sale_item_uid
- modified_date

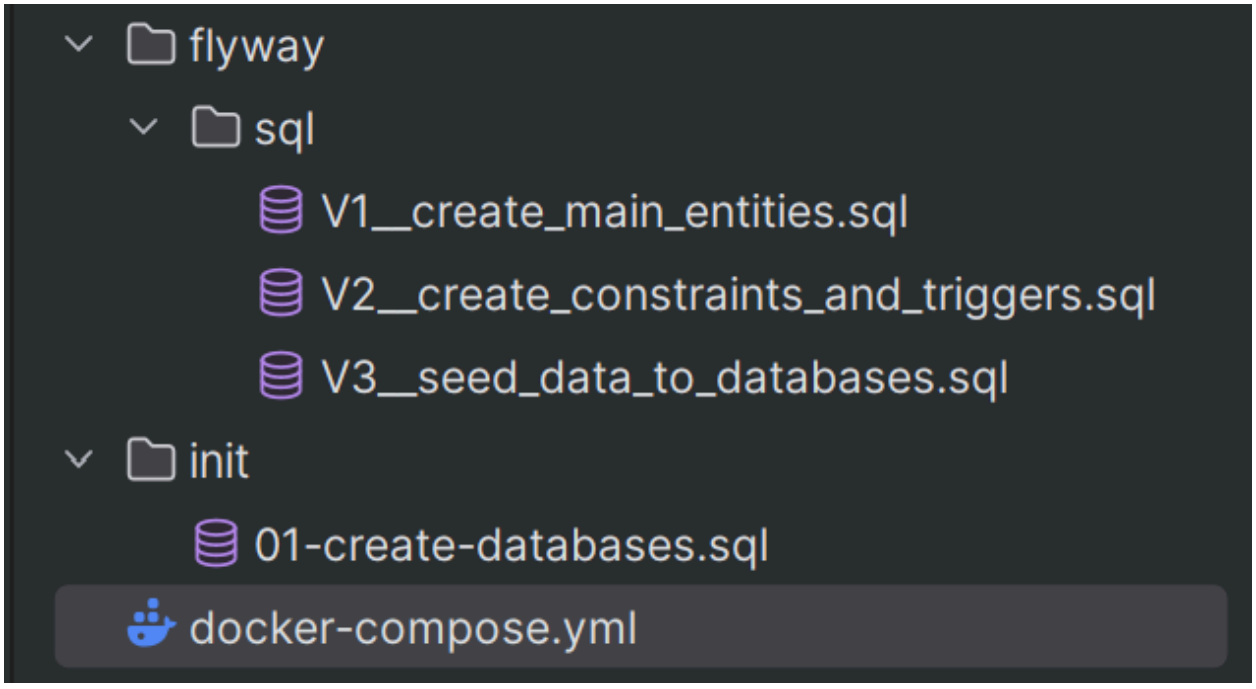
Какие типы данных для чего и почему я буду использовать?

Для id - int (просто как счетчик в в таблице)
Для названия категории/продукта/покупателя VARCHAR(50) - строку ограниченнной длины
Для артикула text - тоже строка, но неограниченной длины
Для уникальных идентификаторов uuid - через gen_random_uuid()
Для стоимости - numeric(12,2) - 12 цифр, 2 полсе запятой
Для подсчета total_line возьмем побольше - numeric(14,2)
Для хранения даты обновления - timestamptz (будет выглядеть вот так: дата + время + часовой пояс)
Для quantity - int, а не numeric(12,2), потому что количество всегда целое

Erd-диаграмма



Структура проекта



Строго обязательно слежу, чтобы после названия версии миграции было двойное нижнее подчеркивание - иначе миграции не пройдут

Docker-compose.yml

```yaml
version: '3.9'

services:
 postgres:
   image: postgres:16
   restart: unless-stopped
   environment:
     POSTGRES_USER: postgres
     POSTGRES_PASSWORD: password
     POSTGRES_DB: postgres
   ports:
     - "5434:5432"
   volumes:
     - pgdata:/var/lib/postgresql/data
     - ./init:/docker-entrypoint-initdb.d
   networks:
     - filialnet
   healthcheck:
     test: ["CMD-SHELL", "pg_isready -U postgres"]
     interval: 5s
     timeout: 5s
     retries: 5
     start_period: 10s

 flyway_west:
   image: flyway/flyway:9.22.3
   depends_on:
     postgres:
       condition: service_healthy
   environment:
     FLYWAY_URL: jdbc:postgresql://postgres:5432/filial_west
     FLYWAY_USER: postgres
     FLYWAY_PASSWORD: password
     FLYWAY_LOCATIONS: filesystem:/flyway/sql
     FLYWAY_BASELINE_ON_MIGRATE: true
     FLYWAY_SQL_MIGRATION_PREFIX: V
     FLYWAY_SQL_MIGRATION_SEPARATOR: __
     FLYWAY_SQL_MIGRATION_SUFFIXES: .sql
   volumes:
     - ./flyway/sql:/flyway/sql
   networks:
     - filialnet
```

```
    command: migrate

  flyway_east:
    image: flyway/flyway:9.22.3
    depends_on:
      postgres:
        condition: service_healthy
    environment:
      FLYWAY_URL: jdbc:postgresql://postgres:5432/filial_east
      FLYWAY_USER: postgres
      FLYWAY_PASSWORD: password
      FLYWAY_LOCATIONS: filesystem:/flyway/sql
      FLYWAY_BASELINE_ON_MIGRATE: true
      FLYWAY_SQL_MIGRATION_PREFIX: V
      FLYWAY_SQL_MIGRATION_SEPARATOR: __
      FLYWAY_SQL_MIGRATION_SUFFIXES: .sql
    volumes:
      - ./flyway/sql:/flyway/sql
    networks:
      - filialnet
    command: migrate

volumes:
  pgdata:

networks:
  filialnet:
```

Что самое важное:

- Написала один сервис для postgress (чтобы можно было подключиться к бд при помощи одних и тех же кред) - прописана только основная бд (та, которая просто инициализируется при поднятии контейнера), другие две бд мы пропишем в отдельном файле - его автоматический запуск в строках

```
volumes:
  - pgdata:/var/lib/postgresql/data
  - ./init:/docker-entrypoint-initdb.d
```

- Два сервиса для миграций - каждый для каждой из бд
- ИИ добавил забытый мной сервис volumes: pgdata и настроил networks (использовала chatgpt) - в качестве референса использовала похожий docker-compose, который написал в рамках курса проектирования бд на втором курсе - моя задача была разобраться, как мне поднять в одном контейнере несколько бд и для каждой провести миграции

01-create-database.sql

```
CREATE DATABASE filial_west;
CREATE DATABASE filial_east;
```

Просто создаем две базы данных согласно заданию
Миграции:

V1__create_entities.sql

Мне очень понравилась эта идея с подключением специальной библиотеки, в которой есть функция для генерации уникального id, поэтому ее и использовала отредактировала ключ и добавила проверки при помощи ии (использовала так же chatgpt)

```
CREATE EXTENSION IF NOT EXISTS pgcrypto;

CREATE TABLE IF NOT EXISTS customer (
 id int GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
 customer_name VARCHAR(50) NOT NULL,
 customer_uid uuid NOT NULL DEFAULT gen_random_uuid(),
 modified_date timestamptz NOT NULL DEFAULT now()
);

CREATE TABLE IF NOT EXISTS product (
 id int GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
 sku text UNIQUE NOT NULL,
 product_name VARCHAR(50) NOT NULL,
 price numeric(12,2) NOT NULL CHECK (price >= 0),
 product_uid uuid NOT NULL DEFAULT gen_random_uuid(),
 modified_date timestamptz NOT NULL DEFAULT now()
);

CREATE TABLE IF NOT EXISTS category (
 id int GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
 category_name VARCHAR(50) UNIQUE NOT NULL,
 category_uid uuid NOT NULL DEFAULT gen_random_uuid(),
 modified_date timestamptz NOT NULL DEFAULT now()
);

CREATE TABLE IF NOT EXISTS product_category (
 id int GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
 product_id int NOT NULL,
 category_id int NOT NULL,
 uid_product_category uuid NOT NULL DEFAULT gen_random_uuid(),
 modified_date timestamptz NOT NULL DEFAULT now(),
 CONSTRAINT product_category_uniq UNIQUE (product_id, category_id)
);

CREATE TABLE IF NOT EXISTS sale (
 id int GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
 customer_id int NOT NULL,
 sale_date timestamptz NOT NULL DEFAULT now(),
 in_total numeric(12,2) NOT NULL DEFAULT 0,
```

```
 sale_uid uuid NOT NULL DEFAULT gen_random_uuid(),
 modified_date timestamptz NOT NULL DEFAULT now()
);

CREATE TABLE IF NOT EXISTS sale_item (
 id int GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
 sale_id int NOT NULL,
 product_id int NOT NULL,
 quantity int NOT NULL CHECK (quantity > 0),
 unit_price numeric(12,2) NOT NULL CHECK (unit_price >= 0),
 line_total numeric(14,2) GENERATED ALWAYS AS (quantity * unit_price)
STORED,
 sale_item_uid uuid NOT NULL DEFAULT gen_random_uuid(),
 modified_date timestamptz NOT NULL DEFAULT now()
);
```

V2__create_constraints_and_triggers.sql

```
ALTER TABLE customer ADD CONSTRAINT customer_uid_uniq UNIQUE
(customer_uid);
ALTER TABLE product ADD CONSTRAINT product_uid_uniq UNIQUE
(product_uid);
ALTER TABLE category ADD CONSTRAINT category_uid_uniq UNIQUE
(category_uid);
ALTER TABLE product_category ADD CONSTRAINT product_category_uid_uniq
UNIQUE (uid_product_category);
ALTER TABLE sale ADD CONSTRAINT sale_uid_uniq UNIQUE (sale_uid);
ALTER TABLE sale_item ADD CONSTRAINT sale_item_uid_uniq UNIQUE
(sale_item_uid);

ALTER TABLE product_category
 ADD CONSTRAINT fk_pc_product FOREIGN KEY (product_id) REFERENCES
product(id) ON DELETE CASCADE;

ALTER TABLE product_category
 ADD CONSTRAINT fk_pc_category FOREIGN KEY (category_id) REFERENCES
category(id) ON DELETE RESTRICT;

ALTER TABLE sale
 ADD CONSTRAINT fk_sale_customer FOREIGN KEY (customer_id) REFERENCES
customer(id) ON DELETE RESTRICT;

ALTER TABLE sale_item
 ADD CONSTRAINT fk_si_sale FOREIGN KEY (sale_id) REFERENCES sale(id)
ON DELETE CASCADE;
```

```
ALTER TABLE sale_item
 ADD CONSTRAINT fk_si_product FOREIGN KEY (product_id) REFERENCES
product(id) ON DELETE RESTRICT;

CREATE INDEX IF NOT EXISTS idx_sale_customer_id ON sale(customer_id);
CREATE INDEX IF NOT EXISTS idx_si_sale_id ON sale_item(sale_id);
CREATE INDEX IF NOT EXISTS idx_si_product_id ON
sale_item(product_id);
CREATE INDEX IF NOT EXISTS idx_pc_product_id ON
product_category(product_id);
CREATE INDEX IF NOT EXISTS idx_pc_category_id ON
product_category(category_id);

CREATE OR REPLACE FUNCTION set_modified_date()
RETURNS trigger AS $$
BEGIN
 NEW.modified_date := now();
 RETURN NEW;
END;
$$ LANGUAGE plpgsql;

DROP TRIGGER IF EXISTS trg_customer_moddate ON customer;
CREATE TRIGGER trg_customer_moddate BEFORE INSERT OR UPDATE ON
customer
 FOR EACH ROW EXECUTE FUNCTION set_modified_date();

DROP TRIGGER IF EXISTS trg_product_moddate ON product;
CREATE TRIGGER trg_product_moddate BEFORE INSERT OR UPDATE ON product
 FOR EACH ROW EXECUTE FUNCTION set_modified_date();

DROP TRIGGER IF EXISTS trg_category_moddate ON category;
CREATE TRIGGER trg_category_moddate BEFORE INSERT OR UPDATE ON
category
 FOR EACH ROW EXECUTE FUNCTION set_modified_date();

DROP TRIGGER IF EXISTS trg_product_category_moddate ON
product_category;
CREATE TRIGGER trg_product_category_moddate BEFORE INSERT OR UPDATE
ON product_category
 FOR EACH ROW EXECUTE FUNCTION set_modified_date();

DROP TRIGGER IF EXISTS trg_sale_moddate ON sale;
CREATE TRIGGER trg_sale_moddate BEFORE INSERT OR UPDATE ON sale
 FOR EACH ROW EXECUTE FUNCTION set_modified_date();
```

```
DROP TRIGGER IF EXISTS trg_sale_item_moddate ON sale_item;
CREATE TRIGGER trg_sale_item_moddate BEFORE INSERT OR UPDATE ON
sale_item
 FOR EACH ROW EXECUTE FUNCTION set_modified_date();
```

V3__seed_data_to_dataases.sql

Сгенерировала при помощи ии и просто провалидировала выполнение

```
INSERT INTO category (category_name)
SELECT 'Category ' || g
FROM generate_series(1,8) g;

INSERT INTO customer (customer_name)
SELECT 'Customer ' || g
FROM generate_series(1,25) g;

INSERT INTO product (sku, product_name, price)
SELECT LPAD(g::text,3,'0') AS sku_full,
       'Product ' || g,
       round((10 + random()*990)::numeric, 2)
FROM generate_series(1,25) g;

INSERT INTO product_category (product_id, category_id)
SELECT p.id, ((p.id - 1) % 8) + 1
FROM product p
UNION ALL
SELECT p.id, ((p.id) % 8) + 1
FROM product p;

INSERT INTO sale (customer_id, sale_date)
SELECT (SELECT id FROM customer ORDER BY random() LIMIT 1),
       now() - ((floor(random()*365)::int) || ' days')::interval
FROM generate_series(1,25);

INSERT INTO sale_item (sale_id, product_id, quantity, unit_price)
SELECT s.id, p.id, (floor(random()*5)+1)::int,
       round((p.price * (0.6 + random()*0.8))::numeric, 2)
FROM generate_series(1,50)
 CROSS JOIN LATERAL (SELECT id, price FROM product ORDER BY random()
LIMIT 1) p
 CROSS JOIN LATERAL (SELECT id FROM sale ORDER BY random() LIMIT 1)
s;

UPDATE sale
SET in_total = sub.sum
```

```
FROM (
  SELECT sale_id, SUM(line_total) AS sum
  FROM sale_item
  GROUP BY sale_id
) sub
WHERE sale.id = sub.sale_id;
```