МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное
образовательное учреждение высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ПРАКТИЧЕСКИМ РАБОТАМ

«Инжиниринг данных»

Направление подготовки 09.03.04 «Программная инженерия»
Направленность (профиль) «Разработка и сопровождение программного обеспечения»
Квалификация выпускника: бакалавр

Ставрополь
2026

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Настоящий сборник методических указаний предназначен для проведения лабораторных работ по дисциплине «Инжиниринг данных». Дисциплина ориентирована на формирование у студентов комплексных компетенций в области построения конвейеров обработки данных (ETL/ELT), проектирования хранилищ данных и озёр данных, а также обеспечения качества и надёжности данных для последующего анализа и использования в программных продуктах.

Цели дисциплины:

Сформировать у студентов комплексные компетенции в области построения конвейеров обработки данных (ETL/ELT), проектирования хранилищ данных и озёр данных, а также обеспечения качества и надёжности данных для последующего анализа и использования в программных продуктах.

Задачи дисциплины:

1. Аналитические:

– Научиться анализировать источники данных (структурированные, полуструктурированные, неструктурированные).

– Оценивать требования к объёму, скорости и разнообразию данных (Data Variety, Velocity, Volume).

2. Проектные:

– Проектировать схемы хранилищ данных (золотой слой), витрин данных и озёр данных.

– Выбирать архитектуру обработки данных (batch, streaming, lambda, kappa).

3. Технологические:

– Освоить инструменты извлечения, трансформации и загрузки данных (Apache Airflow, dbt, Pandas/SQL).

– Научиться работать с очередями сообщений (Kafka, RabbitMQ) и контейнеризацией (Docker).

4. Контрольно-измерительные:

– Реализовывать проверки качества данных (great expectations, data validation).

– Настраивать мониторинг и логирование пайплайнов.

5. Интеграционные:

– Подготовить данные для передачи в системы анализа (OLAP-кубы, ML-модели) как мост к дисциплине «Алгоритмы обработки и анализа данных».

Лабораторная работа №1.
Знакомство с Docker для ETL

Цель работы: научиться контейнеризовать простой Python-скрипт извлечения данных (Extract) в контексте задач ETL-пайплайна с использованием Docker.

Задачи:

1. Написать скрипт на Python для чтения структурированных данных из CSV-файла.
2. Создать Dockerfile для упаковки скрипта и его зависимостей в образ.
3. Собрать Docker-образ и запустить контейнер с монтированием томов (volume binding).
4. Обеспечить доступ контейнера к входному CSV-файлу и сохранение выходных данных на хостовой машине.
5. Проанализировать разницу между запуском Python-скрипта «на голой машине» и внутри контейнера.

Теоретическая часть

Docker — платформа для контейнеризации приложений. Контейнер изолирует процесс на уровне ОС, но использует ядро хоста, что делает его легче виртуальной машины.

Основные понятия:

- *Образ (image)* — неизменяемый шаблон с кодом, библиотеками, зависимостями.
- *Контейнер (container)* — запущенный экземпляр образа.
- *Том (volume)* — механизм для сохранения данных вне контейнера и обмена файлами с хостом.

Для ETL-задач Docker даёт:

- Воспроизводимость окружения (один и тот же Python/зависимости в dev/prod).
- Изоляцию от хостовой системы (без конфликтов библиотек).
- Лёгкую интеграцию с оркестраторами (Airflow, Kubernetes).

Важное понятие — монтирование томов (bind mount):

bash

docker run -v /путь/на/хосте:/путь/в/контейнере

Позволяет контейнеру читать и писать файлы в указанную директорию хоста.

Типовой сценарий в ETL:

1. Исходные CSV-файлы лежат на хосте.
2. Контейнер с Python-скриптом монтирует эту директорию.
3. Скрипт читает данные, обрабатывает (трансформирует) и записывает результат в смонтированную папку.
4. Итоговый файл оказывается доступным на хосте для следующего этапа пайплайна.

Методика выполнения работы

1. **Подготовка рабочего каталога**

   1.1. Создайте на хосте каталог lab1_etl_docker

   1.2. Внутри создайте подкаталоги: input, output, src

   1.3. Поместите в input тестовый CSV-файл sales.csv с произвольными данными о продажах (минимум 5 строк).

2. **Разработка Python-скрипта извлечения**

   2.1. В каталоге src создайте файл extract_csv.py

   2.2. Реализуйте в нём:

   – Чтение CSV из папки /data/input (это путь внутри контейнера).

- Вывод в консоль количества строк и имён колонок.
- Запись краткой сводки (например, общая сумма по полю amount) в файл /data/output/summary.txt.
- Обработку ошибок (файл не найден, пустой файл).

2.3. Используйте только стандартную библиотеку Python (csv, os, sys) либо добавьте pandas — по желанию.

3. **Создание Docker-образа**

3.1. В корне каталога lab1_etl_docker создайте файл Dockerfile

3.2. Напишите в нём:

```
FROM python:3.11-slim
WORKDIR /app
COPY src/extract_csv.py /app/extract_csv.py
CMD ["python", "/app/extract_csv.py"]
```

3.3. Постройте образ с тегом etl_extract:latest:

```
docker build -t etl_extract .
```

4. **Запуск контейнера с монтированием томов**

4.1. Выполните команду (из корня lab1_etl_docker):

```
docker run --rm \
  -v $(pwd)/input:/data/input \
  -v $(pwd)/output:/data/output \
  etl_extract
```

Пояснение: $(pwd) — текущая директория (на Linux/macOS). Для Windows PowerShell используйте ${PWD} или абсолютный путь.

4.2. Проверьте, что в папке output появился файл summary.txt с правильным содержимым.

5. **Модификация скрипта для проверки разделения ответственности**

5.1. Измените extract_csv.py так, чтобы он принимал пути к входному и выходному каталогам через аргументы командной строки (sys.argv).

5.2. Обновите Dockerfile, убрав фиксированные пути из кода.

5.3. Пересоберите образ.

5.4. Запустите с передачей аргументов:

docker run --rm -v $(pwd)/input:/in -v $(pwd)/output:/out etl_extract python /app/extract_csv.py --input_dir /in --output_dir /out

6. **Проверка качества данных (элементарная)**

   6.1. Добавьте в скрипт проверку: если какой-либо столбец содержит NULL или пустую строку — записать предупреждение в errors.log в выходной каталог.

   6.2. Запустите контейнер снова и убедитесь, что лог создаётся.

Пример выполнения задания

Исходный CSV (input/sales.csv**):**

```
date,product,amount
2025-01-01,Laptop,1200
2025-01-02,Mouse,25
2025-01-03,Keyboard,75
```

Скрипт src/extract_csv.py **(упрощённая версия):**

```
import csv
import os
import sys

def main():
    input_dir = sys.argv[2] if '--input_dir' in sys.argv else '/data/input'
    output_dir = sys.argv[4] if '--output_dir' in sys.argv else '/data/output'

    input_path = os.path.join(input_dir, 'sales.csv')
    output_path = os.path.join(output_dir, 'summary.txt')

    if not os.path.exists(input_path):
        print(f"Error: {input_path} not found")
```

```
        sys.exit(1)

    total = 0
    with open(input_path, 'r') as f:
        reader = csv.DictReader(f)
        rows = list(reader)
        for row in rows:
            total += float(row['amount'])

    with open(output_path, 'w') as f:
        f.write(f"Total rows: {len(rows)}\n")
        f.write(f"Total amount: {total}\n")

    print("Extraction completed successfully")

if __name__ == "__main__":
    main()
```

Запуск и результат:

- После docker run ... в output/summary.txt появляется:

  text

  Total rows: 3

  Total amount: 1300.0

Индивидуальные задания

Каждому студенту выдаётся вариант с модификацией формата входных данных и типа требуемой сводки.

Вариант	Формат входного файла	Необходимая сводка
1	CSV: id,user_id,amount,timestamp	Количество уникальных user_id и максимальная сумма (amount)
2	CSV: city,orders_count,revenue	Город с максимальным revenue и средний чек (revenue / orders_count)
3	CSV: product_code,stock,price	Общая стоимость запасов (stock * price) для каждого товара + итоговая сумма
4	CSV с разделителем ; (точка с запятой): date;category;sales	Итог по каждой категории (группировка) в отдельный CSV-файл category_summary.csv
5	CSV с пустыми значениями: name,value (некоторые value пустые)	Посчитать только строки с непустым value, вывести их количество и среднее арифметическое
6	CSV: transaction_id,client_id,product,quantity,unit_price	Вычислить общую сумму каждой транзакции (quantity * unit_price) и записать JSON-файл transaction_totals.json
7	CSV: date,hour,page_views,unique_visitors	Найти час с максимальным отношением page_views / unique_visitors
8	CSV: region,product_category,month_sales,quarter_sales	Сравнить month_sales и quarter_sales: вывести регионы, где quarter_sales меньше month_sales (аномалия)
9	CSV большой (100+ строк) с полями user_id,action,timestamp	Сформировать файл user_stats.txt для каждого пользователя: количество действий и временной диапазон (min/max timestamp)

Вариант	Формат входного файла	Необходимая сводка
10	CSV с некорректными числовыми полями: id,value (значения могут быть "N/A", "", "-")	Очистить данные: заменить некорректные значения на 0, записать очищенный CSV в cleaned_data.csv и лог замен в replacements.log
11	CSV: order_id,item_name, discount_pct,base_price	Вычислить финальную цену с учётом скидки (base_price * (1 - discount_pct/100)), вывести топ-3 самых дорогих товара после скидки
12	CSV с дубликатами: record_id,field1,field2	Удалить дубликаты по record_id, сохранить только первое вхождение, записать результат в dedup.csv и указать количество удалённых дубликатов в summary.txt
13	CSV: sensor_id,temperature,humidity,datetime	Найти среднюю температуру и влажность для каждого sensor_id, вывести в JSON sensor_avg.json
14	CSV с несколькими листами (имитация: один CSV, но внутри есть маркер [SECTION] между блоками данных)	Разделить файл на два выходных CSV: customers.csv и orders.csv на основе маркера-разделителя (задание на обработку полуструктурированного файла)
15	CSV: product_name,price, currency (валюты: USD, EUR, RUB)	Добавить колонку price_usd, пересчитав все цены в USD по курсам (1 USD = 0.92 EUR, 1 USD = 92 RUB). Записать обогащённый CSV в converted_prices.csv

Примечание для варианта 14: пример входного файла:

id,name

1,Alice

2,Bob

[SECTION]

order_id,product

101,Laptop

102,Mouse

Скрипт должен считать всё до [SECTION] в customers.csv, а после – в orders.csv.

Дополнительное задание (для всех вариантов)

Добавить в контейнер проверку, что выходной каталог доступен на запись:

- Если доступ есть — продолжать выполнение.
- Если нет (например, права только на чтение) — завершить работу с кодом ошибки 2 и записать сообщение об ошибке в error.log внутри выходного каталога (если возможно) или вывести в консоль.

Требования к отчёту

1. Титульный лист с названием работы, ФИО, группой.
2. Цель и задачи (скопировать из методички).
3. Краткая теоретическая справка (своими словами, 0.5–1 страница).
4. Ход выполнения работы:
   - Листинг extract_csv.py с комментариями.
   - Содержимое Dockerfile.
   - Команда сборки и запуска.
   - Скриншот выполнения команды docker run ... с выводом в терминале.
5. Результаты:
   - Скриншот содержимого output после работы контейнера.
   - Пример выходного файла (summary.txt или errors.log).

6. Ответы на контрольные вопросы (письменно).
7. Вывод – что дало использование Docker для ETL-задачи.

Контрольные вопросы

1. Чем отличается Docker-контейнер от виртуальной машины в контексте ETL-скриптов?

2. Зачем в ETL-пайплайнах монтировать том, а не копировать CSV внутрь образа?

3. Что произойдёт с данными в output, если не использовать --rm и запустить контейнер дважды?

4. Как изменить команду docker run, чтобы контейнер получил переменную окружения INPUT_PATH=/data/in.csv?

5. Почему в примере используется python:3.11-slim вместо python:3.11? Назовите два преимущества.

6. Как проверить изнутри контейнера, что том примонтирован корректно?

7. Что такое «разделение ответственности» между скриптом и Dockerfile и зачем оно нужно?

8. Какой код возврата (exit code) будет у контейнера, если скрипт не найдет входной CSV-файл (при условии, что вы корректно обработали ошибку через sys.exit(1))?

Лабораторная работа №2.
PostgreSQL как хранилище слоя staging

Цель работы: закрепить навыки использования продвинутых конструкций SQL (оконные функции, обобщённые табличные выражения CTE, трансформация типов) для первичной очистки и нормализации данных в staging-слое хранилища данных.

Задачи

1. Развернуть PostgreSQL в Docker-контейнере с монтированием тома для сохранности данных.
2. Создать staging-схему и таблицы для приёма «сырых» данных.
3. Реализовать загрузку данных из CSV-файлов в staging-таблицы с использованием утилиты COPY.
4. Написать скрипт трансформации (INSERT INTO ... SELECT) с применением:
   - приведения типов (CAST, ::)
   - обработки NULL и значений по умолчанию
   - оконных функций (ROW_NUMBER, LAG, LEAD)
   - CTE (WITH-блоков) для многоэтапной очистки
5. Обеспечить детектирование дубликатов и выборку только актуальных записей.
6. Подготовить staging-данные для передачи в следующий слой (core / gold).

Теоретическая часть

Staging-слой в хранилище данных

Staging (или слой «песочницы») — это временное хранилище, куда данные попадают непосредственно из источников. Основные характеристики:

- Данные хранятся в максимально близком к исходному виде.
- Допустимы дубликаты и «грязные» данные.

- Задачи слоя: изоляция источника от последующих слоёв, первичная проверка целостности, приведение типов.

Почему PostgreSQL для staging?

- Поддерживает все необходимые типы (включая JSON, массивы).
- Мощный SQL с оконными функциями и CTE.
- Легко разворачивается в Docker.
- Бесплатно и широко распространён.

Ключевые SQL-конструкции для очистки данных

1. **Приведение типов:**

   CAST(some_column AS INTEGER)

   -- *или*

   some_column::NUMERIC(10,2)

2. **Обработка NULL:**

   COALESCE(column, 'default_value')

   NULLIF(column, 'N/A')

3. **Оконные функции для дедупликации:**

   ROW_NUMBER() OVER (PARTITION BY id ORDER BY update_date DESC) AS rn

   Позволяет оставить только самую свежую запись для каждого ключа.

4. **CTE (Common Table Expressions) для пошаговой очистки:**

```
WITH cleaned AS (
   SELECT ... FROM raw WHERE ...
),
deduped AS (
   SELECT ... FROM cleaned WHERE ...
)
INSERT INTO staging SELECT * FROM deduped;
```

Типичный пайплайн загрузки в staging:

CSV → COPY INTO raw_table → очистка/трансформация → INSERT INTO staging_table

Методика выполнения работы

1. **Подготовка окружения**

   1.1. Создайте на хосте каталог lab2_postgres_staging

   1.2. Внутри создайте подкаталоги: data, sql, input

   1.3. Поместите в input тестовый CSV-файл raw_customers.csv с «грязными» данными (пропуски, неверные форматы, дубликаты).

2. **Запуск PostgreSQL в Docker-контейнере**

   2.1. Выполните команду для запуска контейнера с PostgreSQL:

```
docker run -d \
  --name postgres_staging \
  -e POSTGRES_USER=etl_user \
  -e POSTGRES_PASSWORD=etl_pass \
  -e POSTGRES_DB=staging_db \
  -v $(pwd)/data:/var/lib/postgresql/data \
  -p 5432:5432 \
  postgres:15
```

   2.2. Убедитесь, что контейнер запущен: docker ps

   2.3. Подключитесь к БД (через docker exec -it postgres_staging psql -U etl_user -d staging_db или через DBeaver/pgAdmin).

3. **Создание схемы и таблиц**

   3.1. Подключитесь к базе данных и создайте схему staging:

```
CREATE SCHEMA IF NOT EXISTS staging;
```

   3.2. Создайте таблицу для сырых данных (raw_customers):

```
CREATE TABLE staging.raw_customers (
    raw_id SERIAL PRIMARY KEY,
```

raw_data TEXT,

loaded_at TIMESTAMP DEFAULT NOW()

);

Примечание: сначала будем загружать всё в текстовое поле, затем парсить.

3.3. Создайте целевую staging-таблицу с правильными типами:

CREATE TABLE staging.customers_clean (

customer_id INTEGER,

full_name VARCHAR(100),

email VARCHAR(255),

age INTEGER,

registration_date DATE,

is_active BOOLEAN,

source_file VARCHAR(255),

valid_from TIMESTAMP DEFAULT NOW(),

is_current BOOLEAN DEFAULT TRUE

);

4. **Загрузка сырых данных из CSV**

4.1. Скопируйте CSV-файл в контейнер:

docker cp input/raw_customers.csv postgres_staging:/tmp/raw_customers.csv

4.2. В psql выполните загрузку:

COPY staging.raw_customers (raw_data)

FROM '/tmp/raw_customers.csv'

DELIMITER ',' CSV HEADER;

5. **Реализация трансформации с продвинутым SQL**

5.1. Напишите скрипт трансформации с использованием CTE и оконных функций. Сохраните его в файл sql/transform.sql:

-- 1. Парсинг и приведение типов

WITH parsed AS (

```
    SELECT
        SPLIT_PART(raw_data, ',', 1)::INTEGER AS customer_id,
        SPLIT_PART(raw_data, ',', 2) AS full_name,
        SPLIT_PART(raw_data, ',', 3) AS email_raw,
        NULLIF(SPLIT_PART(raw_data, ',', 4), '')::INTEGER AS age,
        NULLIF(SPLIT_PART(raw_data, ',', 5), '')::DATE AS registration_date,
        SPLIT_PART(raw_data, ',', 6) AS active_flag,
        loaded_at
    FROM staging.raw_customers
    WHERE raw_data IS NOT NULL
),
-- 2. Очистка строковых полей
cleaned AS (
    SELECT
        customer_id,
        TRIM(full_name) AS full_name,
        LOWER(TRIM(email_raw)) AS email,
        COALESCE(age, 0) AS age,
        COALESCE(registration_date, '1900-01-01') AS registration_date,
        active_flag = 'Y' OR active_flag = '1' AS is_active,
        loaded_at
    FROM parsed
    WHERE customer_id IS NOT NULL
      AND full_name != ''
),
-- 3. Дедупликация: оставляем последнюю (самую свежую) запись для каждого customer_id
deduped AS (
    SELECT *,
```

ROW_NUMBER() OVER (PARTITION BY customer_id ORDER BY loaded_at DESC) AS rn

FROM cleaned

)

-- 4. Вставка в очищенную таблицу

INSERT INTO staging.customers_clean (

customer_id, full_name, email, age, registration_date, is_active, source_file

)

SELECT

customer_id,

full_name,

email,

age,

registration_date,

is_active,

'raw_customers.csv'

FROM deduped

WHERE rn = 1;

5.2. Выполните скрипт в psql:

sql

\i sql/transform.sql

5.3. Проверьте результат:

sql

SELECT * FROM staging.customers_clean;

6. **Добавление мониторинга качества данных**

6.1. Создайте таблицу для лога загрузок:

CREATE TABLE staging.load_log (

load_id SERIAL PRIMARY KEY,

source_file VARCHAR(255),

rows_loaded INTEGER,

rows_rejected INTEGER,

load_timestamp TIMESTAMP DEFAULT NOW()

);

6.2. Модифицируйте трансформацию, добавив запись в лог (используйте отдельную транзакцию или функцию на PL/pgSQL).

7. **Итоговая проверка**

7.1. Убедитесь, что дубликаты отсутствуют (выполните запрос с GROUP BY customer_id HAVING COUNT(*) > 1).

7.2. Проверьте, что типы полей соответствуют ожидаемым (возраст — число, дата — DATE, email — строки с @).

7.3. Убедитесь, что контейнер с PostgreSQL можно остановить и запустить заново без потери данных (благодаря тому).

Пример выполнения задания

Входной CSV (input/raw_customers.csv**):**

customer_id,name,email,age,reg_date,is_active

101,John Doe,john@example.com,35,2023-01-15,Y

102,Jane Smith,,,2023-02-20,N

101, John Doe ,john.doe@example.com,36,2023-03-10,Y

103,Bob Johnson,bob@test.com,25,2024-01-01,Y

101,John Doe,john@example.com,35,2023-01-15,Y

Результат в staging.customers_clean **после выполнения трансформации:**

customer_id	full_name	email	age	registration_date	is_active	source_file
101	John Doe	john.doe@example.com	36	2023-03-10	true	raw_customers.csv
102	Jane Smith		0	2023-02-20	false	raw_customers.csv
103	Bob Johnson	bob@test.com	25	2024-01-01	true	raw_customers.csv

Обратите внимание: дубликаты для customer_id=101 удалены, оставлена самая свежая запись (2023-03-10). Пропущенный email у Jane Smith остался NULL (пустая строка после очистки). Возраст 0 для пропущенного значения.

Индивидуальные задания

Каждый вариант имеет свою структуру входного файла и специфические правила очистки. Студент должен адаптировать SQL-трансформацию под свой вариант.

Вар.	Входной CSV (поля)	Специфические правила очистки / трансформации
1	order_id,user_id,amount,order_date,status	- Удалить заказы с amount <= 0; - order_date привести к DATE, невалидные заменить на '2024-01-01'; - добавить колонку amount_usd, пересчитав по курсу (1 USD = 1.05 EUR)
2	product_id,name,price,category,stock_quantity	- price может быть отрицательным → взять модуль; - stock_quantity NULL заменить на 0;

Вар.	Входной CSV (поля)	Специфические правила очистки / трансформации
		- с помощью оконной функции LAG добавить колонку prev_price
3	transaction_id,card_number,amount,timestamp,merchant	- Маскировать card_number (оставить последние 4 цифры: ****-####); - удалить транзакции с суммой > 10000 (подозрение на фрод); - добавить колонку transaction_hour из timestamp
4	employee_id,full_name,birth_date,salary,department	- Вычислить возраст на текущую дату (AGE()); - удалить сотрудков младше 18 лет; - с помощью ROW_NUMBER оставить только 3 самых высокооплачиваемых в каждом департаменте
5	log_id,ip_address,user_agent,access_time,response_time_ms	- Из ip_address извлечь подсеть (первые 3 октета); - response_time_ms может быть 'timeout' → заменить на NULL; - добавить колонку slow_request (TRUE, если > 500ms)
6	sensor_id,reading_value,reading_unit,recorded_at,battery_level	- Привести все значения к единой единице (из разных unit в 'Celsius'); - удалить показания, где battery_level < 5; - с помощью оконной функции LEAD добавить next_reading_value
7	book_id,title,author,published_year,isbn,price	- Проверить ISBN на длину (10 или 13 цифр), невалидные пометить в отдельную колонку isbn_valid; - published_year до 1900 → заменить на NULL; - дедуплицировать по title+author
8	user_id,action,action_time,page_url,referrer	- Из page_url извлечь домен (regexp_matches); - удалить действия с пустым action; - добавить

Вар.	Входной CSV (поля)	Специфические правила очистки / трансформации
		колонку session_id (группировка по user_id с разрывом > 30 минут) — задание на оконные функции с LAG по времени
9	student_id,name,grade,subject,exam_date,score	- Оценка score может быть 0-100 или буквенная A,B,C,D,F → преобразовать в числовую (A=90-100, B=80-89 и т.д.); - удалить студентов, у которых >3 пропусков (поле grade = 'ABSENT'); - вычислить ранг студента по каждому предмету (RANK)
10	call_id,phone_number,call_duration_sec,start_time,end_time,disposition	- call_duration_sec NULL вычислить как end_time - start_time; - удалить звонки короче 3 секунд; - добавить колонку call_hour_category (morning/afternoon/evening/night)
11	product_sku,warehouse,stock,reorder_level,last_restock_date	- Объединить product_sku и warehouse в составной ключ для дедупликации; - если last_restock_date старше 1 года → пометить needs_restock = TRUE; - с помощью оконной функции SUM вычислить общий запас по каждому складу
12	ticket_id,issue_text,priority,created_at,resolved_at,satisfaction_score	- satisfaction_score может быть 1-5 или 'N/A' → заменить на NULL; - вычислить time_to_resolve_hours (в часах); - оконная функция: для каждого приоритета найти медианное время решения (использовать PERCENTILE_CONT)
13	api_request_id,endpoint,response_code,response_size_bytes,request_payload	- request_payload — JSON-строка, распарсить в отдельные колонки (использовать jsonb); - удалить запросы с response_code >=

Вар.	Входной CSV (поля)	Специфические правила очистки / трансформации
		500; - добавить колонку size_category ('small' < 1KB, 'medium' 1-100KB, 'large' >100KB)
14	click_id,cookie_id,event_type,page,element_clicked,timestamp	- Выделить element_clicked из полного пути (последняя часть после '/'); - удалить дубликаты по (cookie_id, event_type, timestamp) с разницей менее 1 секунды; - добавить колонку click_sequence (номер клика в сессии)
15	shipment_id,tracking_number,carrier,weight_kg,dimensions,delivery_status	- Из dimensions (формат "30x20x15") вычислить volume_cm3; - weight_kg может быть 'unknown' → заменить на среднее по carrier; - с помощью оконной функции DENSE_RANK пронумеровать shipments по весу в рамках каждого carrier

Дополнительное задание для всех вариантов: Реализовать логирование ошибок: если при трансформации конкретной строки происходит ошибка (например, неверный формат даты), эта строка должна попасть в таблицу staging.rejected_rows с указанием причины ошибки, но трансформация остальных строк должна продолжиться.

Требования к отчёту

1. Титульный лист с названием работы, ФИО, группой, номером варианта.
2. Цель и задачи (из методички).
3. Теоретическая часть (кратко о staging-слое, оконных функциях, CTE).
4. Ход выполнения работы:

- Команды запуска PostgreSQL в Docker (скриншот или листинг).
- SQL-код создания схемы и таблиц.
- SQL-код трансформации (с комментариями, объясняющими логику очистки).
- Пример входных данных (первые 5–10 строк из вашего CSV).

5. Результаты:
   - Вывод запроса SELECT * FROM staging.customers_clean (скриншот).
   - Вывод из таблицы лога загрузок (если реализовали).
   - Статистика: сколько строк загружено, сколько отклонено.
6. Ответы на контрольные вопросы (письменно).
7. Вывод — что дало использование PostgreSQL и продвинутого SQL для staging-слоя, какие сложности возникли при очистке.

Контрольные вопросы

1. Почему staging-слой не рекомендуется использовать как конечное хранилище для отчётов?

2. Чем отличается ROW_NUMBER() от RANK() и DENSE_RANK() при дедупликации?

3. Как в PostgreSQL обработать строку вида '2023-13-40' при приведении к типу DATE, чтобы запрос не упал?

4. Зачем нужны CTE (WITH-блоки) вместо вложенных подзапросов? Приведите пример преимущества.

5. Как проверить, что в колонке email все записи содержат символ '@'? Напишите условие WHERE.

6. В чём разница между NULLIF(column, '') и COALESCE(column, 'default')?

7. Как удалить дубликаты, если нет явного поля loaded_at для определения «свежести» записи?

8. Можно ли выполнить трансформацию непосредственно во время COPY (через промежуточную таблицу и триггер)? В чём минусы такого подхода?

9. Как с помощью оконной функции вычислить скользящее среднее (moving average) за последние 3 записи, упорядоченные по дате?

10. Что произойдёт с транзакцией, если во время INSERT INTO ... SELECT возникнет ошибка преобразования типа в середине выполнения? Как сделать так, чтобы «хорошие» строки всё равно вставились?

11. Как смонтировать том для PostgreSQL так, чтобы даже после docker rm данные не потерялись?

12. Зачем в staging-таблицу добавлять поля valid_from и is_current? Где это используется дальше (SCD Type 2)?

Лабораторная работа №3.

Извлечение данных из REST API

Цель работы: получить данные из публичного REST API и сохранить в структурированном формате JSON с использованием Python-библиотеки requests, реализовав обработку пагинации, соблюдение ограничений частоты запросов (rate limiting) и механизм повторных попыток при сбоях.

Задачи:

1. Изучить документацию публичного REST API и определить структуру запросов.
2. Реализовать на Python функцию для извлечения данных с обработкой HTTP-ошибок.
3. Реализовать механизм пагинации для получения всех страниц данных.
4. Реализовать соблюдение rate limiting (задержки между запросами, обработка заголовков X-RateLimit-*).
5. Реализовать механизм повторных попыток (retry) с экспоненциальной задержкой для временных сбоев.
6. Сохранить полученные данные в JSON-файл(ы) с правильной структурой.
7. Добавить базовое логирование процесса извлечения.

Теоретическая часть

REST API для инжиниринга данных

REST API (Representational State Transfer Application Programming Interface) — один из наиболее распространённых способов получения данных из внешних систем (CRM, маркетинговые платформы, финансовые сервисы, погодные сервисы). Для инженера данных критически важно уметь:

- Аутентифицироваться в API (API key, OAuth, JWT).
- Формировать корректные HTTP-запросы (GET, POST) с параметрами.

- Обрабатывать ответы (JSON, XML, CSV).
- Работать с пагинацией, когда данных больше, чем API отдаёт за один раз.
- Соблюдать rate limiting, чтобы не быть заблокированным.

Пагинация в REST API

Типы пагинации, встречающиеся на практике:

Тип	Механизм	Пример параметров
Offset/limit	?offset=0&limit=100	GitHub API, Reddit
Cursor (токен)	?cursor=abc123	Stripe, Twitter v2
Page number	?page=1&per_page=50	Many CMS APIs
URL следующей страницы	next в теле ответа	REST Framework, Salesforce

Rate limiting (ограничение частоты запросов)

API защищаются от перегрузки через ограничения:

- **Per minute/second** — например, 100 запросов/минуту.
- **Header'ы информирования**: X-RateLimit-Limit, X-RateLimit-Remaining, X-RateLimit-Reset.
- При превышении API возвращает HTTP 429 (Too Many Requests).

Стратегии работы с rate limiting:

1. **Pre-emptive** — рассчитывать задержку самостоятельно.
2. **Reactive** — при получении 429 ждать согласно Retry-After.
3. **Адаптивная** — динамически подстраивать частоту.

Повторные попытки (Retry logic)

Временные сбои (timeout, 5xx ошибки, 429) требуют повторных попыток:

- Экспоненциальная задержка: delay = initial * (backoff_factor ** attempt).
- Максимальное количество попыток (обычно 3–5).

- Джиттер (случайное отклонение) для предотвращения «эффекта стада».

Библиотека requests **с адаптерами:**

```
python
from requests.adapters import HTTPAdapter
from urllib3.util.retry import Retry

retry_strategy = Retry(
    total=3,
    backoff_factor=1,
    status_forcelist=[429, 500, 502, 503, 504]
)
adapter = HTTPAdapter(max_retries=retry_strategy)
session.mount("https://", adapter)
```

Методика выполнения работы

1. **Подготовка рабочего окружения**

   1.1. Создайте каталог lab3_api_extract

   1.2. Внутри создайте подкаталоги: src, output, logs

   1.3. Создайте виртуальное окружение Python и установите зависимости:

```
python -m venv venv
source venv/bin/activate  # Linux/Mac
venv\Scripts\activate     # Windows
pip install requests python-dotenv
```

1.4. Получите бесплатный API-ключ для одного из публичных API (например, OpenWeatherMap, ExchangeRate-API, JSONPlaceholder для тестов).

2. **Изучение документации API**

   2.1. Выберите API согласно вашему индивидуальному варианту.

   2.2. Определите:

- Базовый URL и эндпоинт.
- Необходимые заголовки (API key, Content-Type).
- Параметры запроса (параметры фильтрации, пагинации).
- Структуру ответа.
- Ограничения частоты запросов.

3. **Создание базового скрипта для одного запроса**

   3.1. В каталоге src создайте файл extract_api.py

   3.2. Реализуйте функцию fetch_data(url, params, headers) с обработкой ошибок:

```
import requests
import time
import logging
from typing import Dict, Any, Optional

logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')
logger = logging.getLogger(__name__)

def fetch_single(url: str, params: Optional[Dict] = None, headers: Optional[Dict] = None) -> Optional[Dict]:
    try:
        response = requests.get(url, params=params, headers=headers, timeout=30)
        response.raise_for_status()
        return response.json()
    except requests.exceptions.RequestException as e:
        logger.error(f"Request failed: {e}")
        return None
```

4. **Реализация обработки rate limiting**

   4.1. Модифицируйте функцию для чтения заголовков rate limit:

```
def fetch_with_rate_limit(url, params, headers):
    response = requests.get(url, params=params, headers=headers)

    # Чтение заголовков rate limit (названия зависят от API)
    remaining = int(response.headers.get('X-RateLimit-Remaining', 1))
    reset_time = int(response.headers.get('X-RateLimit-Reset', 0))

    if remaining == 0 and reset_time > 0:
        wait_time = reset_time - time.time() + 1
        if wait_time > 0:
            logger.warning(f"Rate limit reached, waiting {wait_time:.2f} seconds")
            time.sleep(wait_time)

    response.raise_for_status()
    return response.json()
```

5. **Реализация механизма повторных попыток (retry)**

5.1. Создайте функцию с экспоненциальной задержкой:

```
def fetch_with_retry(url, params, headers, max_retries=3, base_delay=1):
    for attempt in range(max_retries):
        try:
            response = requests.get(url, params=params, headers=headers, timeout=30)

            if response.status_code == 429:
                retry_after = int(response.headers.get('Retry-After', base_delay * (2 ** attempt)))
                logger.warning(f"HTTP 429, retrying after {retry_after}s (attempt {attempt+1}/{max_retries})")
                time.sleep(retry_after)
```

```
            continue

        response.raise_for_status()
        return response.json()

    except requests.exceptions.RequestException as e:
        logger.error(f"Attempt {attempt+1} failed: {e}")
        if attempt < max_retries - 1:
            delay = base_delay * (2 ** attempt)
            logger.info(f"Retrying in {delay}s...")
            time.sleep(delay)
        else:
            logger.error(f"Max retries exceeded for {url}")
            return None
    return None
```

6. **Реализация пагинации**

6.1. В зависимости от типа пагинации (по варианту) реализуйте цикл сбора всех страниц:

Вариант 1: offset/limit

```
def paginate_offset_limit(base_url, params, headers, page_size=100):
    all_data = []
    offset = 0
    while True:
        params['offset'] = offset
        params['limit'] = page_size
        data = fetch_with_retry(base_url, params, headers)
        if not data or not data.get('results'):
            break
        all_data.extend(data['results'])
        if len(data['results']) < page_size:
```

```
            break
        offset += page_size
        logger.info(f"Fetched {len(all_data)} records so far...")
        time.sleep(0.5)  # rate limit protection
    return all_data
```

Вариант 2: cursor/next page token

```
def paginate_cursor(base_url, params, headers):
    all_data = []
    cursor = None
    while True:
        if cursor:
            params['cursor'] = cursor
        data = fetch_with_retry(base_url, params, headers)
        if not data:
            break
        all_data.extend(data.get('items', []))
        cursor = data.get('next_cursor')
        if not cursor:
            break
        logger.info(f"Fetched {len(all_data)} records, next cursor: {cursor[:20]}...")
        time.sleep(0.5)
    return all_data
```

7. **Сохранение данных в JSON**

7.1. Добавьте функцию для сохранения с добавлением метаданных:

```
import json
from datetime import datetime

def save_to_json(data, filename, metadata=None):
    output = {
```

```
        "extracted_at": datetime.now().isoformat(),
        "record_count": len(data) if isinstance(data, list) else 1,
        "metadata": metadata or {},
        "data": data
    }
    with open(filename, 'w', encoding='utf-8') as f:
        json.dump(output, f, indent=2, ensure_ascii=False)
    logger.info(f"Saved {output['record_count']} records to {filename}")
```

8. **Добавление логирования**

   8.1. Настройте логирование в файл и консоль:

   python

   fh = logging.FileHandler('logs/extract.log')

   fh.setLevel(logging.INFO)

   formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')

   fh.setFormatter(formatter)

   logger.addHandler(fh)

9. **Итоговый скрипт и запуск**

   9.1. Соберите все компоненты в extract_api.py с функцией main().

   9.2. Запустите скрипт:

   python src/extract_api.py

   9.3. Убедитесь, что в папке output появился JSON-файл с данными.

Пример выполнения задания

Задача: Извлечь данные о постах из JSONPlaceholder API (тестовое API без пагинации и rate limiting — упрощённый пример).

Код src/extract_api.py**:**

python

import requests

```
import json
import logging
import time
from datetime import datetime
from typing import Dict, Any, Optional, List

logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s',
    handlers=[
        logging.FileHandler('logs/extract.log'),
        logging.StreamHandler()
    ]
)
logger = logging.getLogger(__name__)

def fetch_with_retry(url: str, max_retries: int = 3, base_delay: float = 1.0) ->
Optional[List[Dict]]:
    for attempt in range(max_retries):
        try:
            logger.info(f"Fetching {url} (attempt {attempt+1})")
            response = requests.get(url, timeout=30)
            response.raise_for_status()
            return response.json()
        except requests.exceptions.RequestException as e:
            logger.error(f"Attempt {attempt+1} failed: {e}")
            if attempt < max_retries - 1:
                delay = base_delay * (2 ** attempt)
                logger.info(f"Retrying in {delay}s...")
                time.sleep(delay)
```

```
    return None

def save_to_json(data: List, filename: str):
    output = {
        "extracted_at": datetime.now().isoformat(),
        "record_count": len(data),
        "data": data
    }
    with open(filename, 'w', encoding='utf-8') as f:
        json.dump(output, f, indent=2, ensure_ascii=False)
    logger.info(f"Saved {len(data)} records to {filename}")

def main():
    url = "https://jsonplaceholder.typicode.com/posts"
    data = fetch_with_retry(url)

    if data:
        save_to_json(data, "output/posts.json")
        logger.info("Extraction completed successfully")
    else:
        logger.error("Extraction failed")

if __name__ == "__main__":
    main()
```

Результат в output/posts.json **(фрагмент):**

```
json
{
  "extracted_at": "2026-05-04T14:30:25.123456",
  "record_count": 100,
  "data": [
```

```
    {
      "userId": 1,
      "id": 1,
      "title": "sunt aut facere repellat provident",
      "body": "quia et suscipit..."
    }
  ]
}
```


Индивидуальные задания

Каждый студент получает API из таблицы ниже. Для каждого API необходимо реализовать полный цикл извлечения с учётом специфических особенностей.

Вар.	API	Эндпоинт	Особенности	Необходимые параметры
1	OpenWeatherMap	forecast (5 дней)	API key, rate limit 60 запросов/мин	q={город}, appid, units=metric
2	ExchangeRate-API	latest	API key, курс к USD	base=USD, apikey
3	GitHub API	repos/{owner}/{repo}/issues	Пагинация через Link header, rate limit 60 запросов/час без токена	state=open, per_page=30
4	Rick and Morty API	character	Пагинация через info.next	page={page}
5	SpaceX API	launches	Нет пагинации (все данные в одном ответе), но большие объёмы	?limit=1000
6	NewsAPI	everything	API key, пагинация page, rate limit 100 запросов/день	q=technology, pageSize=20, page={page}
7	The Dog API	v1/images/search	Пагинация через limit и page, API key опционален	limit=10, page={page}, mime_types=jpg
8	JSONPlaceholder	posts + comments	Два эндпоинта, связанные через postId	извлечь посты и для каждого поста комментарии (вложенный вызов)

Вар.	API	Эндпоинт	Особенности	Необходимые параметры
9	Pokémon API	pokemon	Пагинация через offset и limit, большой объём	offset=0, limit=20
10	NASA API	neo/rest/v1/feed (Near Earth Objects)	API key, нужна передача даты start_date	api_key, start_date=YYYY-MM-DD
11	CoinGecko	/coins/markets	Пагинация per_page/page, rate limit 30 вызовов/мин	vs_currency=usd, per_page=100, page={page}
12	HackerNews API	v0/topstories.json + v0/item/{id}.json	Сначала получаем список ID (200+), затем для каждого ID отдельный запрос	реализовать паузу между запросами к деталям
13	Random User API	api/	Пагинация через page и results, можно указать nat=us	results=50, page={page}, seed=abc
14	Open Library API	/search.json	Пагинация через limit и offset, поиск по автору	q=author:asimov, limit=20, offset={offset}
15	Rest Countries	all	Нет пагинации, все страны в одном ответе (~250 записей)	добавить фильтрацию по региону (параметр не поддерживается — реализовать post-filter)

Дополнительное задание для всех вариантов:

Реализовать сохранение «сырых» данных в формате NDJSON (каждая запись на отдельной строке) для возможности потоковой обработки. Добавить чекпоинт (checkpoint) в виде файла last_cursor.txt или last_offset.txt, чтобы можно было запускать извлечение инкрементально.

Требования к отчёту

1. Титульный лист с названием работы, ФИО, группой, номером варианта.
2. Цель и задачи (скопировать из методички).
3. Теоретическая часть (пагинация, rate limiting, retry — своими словами).
4. Ход выполнения работы:
    - Ссылка на документацию использованного API.
    - Листинг extract_api.py с подробными комментариями.
    - Обоснование выбранной стратегии пагинации.
    - Обоснование выбранной стратегии rate limiting и повторных попыток.
5. Результаты:
    - Скриншот лога выполнения (файл logs/extract.log или вывод консоли).
    - Фрагмент выходного JSON-файла (первые 3–5 записей).
    - Статистика: количество извлечённых записей, время выполнения, количество повторных попыток (если были).
6. Ответы на контрольные вопросы.
7. Вывод — какие сложности возникли при работе с выбранным API, как их удалось решить.

Контрольные вопросы

1. Чем отличается пагинация по offset/limit от cursor-based пагинации? Почему cursor предпочтительнее для больших наборов данных?

2. Как определить, что API использует токен следующей страницы в теле ответа, а не в заголовках?

3. Что такое экспоненциальная задержка (exponential backoff) и зачем к ней добавляют джиттер (jitter)?

4. Какие HTTP-статусы должны вызывать повторную попытку, а какие — немедленное завершение с ошибкой?

5. Как правильно интерпретировать заголовок Retry-After? В каких единицах он может быть передан?

6. Как можно протестировать обработку rate limiting без реального превышения лимитов?

7. Почему важно логировать не только успешные запросы, но и каждый повторный вызов с причиной?

8. Как бы вы реализовали возобновляемую загрузку (resumable extraction), если API не поддерживает пагинацию от определённого ID?

9. В чём отличие синхронного извлечения (через requests.get) от асинхронного (aiohttp)? Когда стоит переходить на асинхронность?

10. Как обеспечить, чтобы скрипт извлечения не «упал» при изменении структуры JSON в ответе API?

11. Как можно проверить, что JSON, полученный от API, соответствует ожидаемой схеме (без использования сторонних валидаторов)?

12. Что такое API-шлюз (API gateway) с точки зрения инженера данных и как он влияет на rate limiting?

13. Как бы вы реализовали кэширование ответов API, чтобы при повторном запуске не извлекать одни и те же данные?

14. Какие альтернативы библиотеке requests существуют для работы с API в Python? Назовите хотя бы две и их ключевые отличия.

15. Если API возвращает 50 000 записей, разбитых на 500 страниц по 100 записей, то сколько запросов вы выполните при наивной реализации? Как можно оптимизировать?

Лабораторная работа 4.
Трансформация данных с Pandas

Цель работы: очистить и агрегировать данные с помощью библиотеки Pandas: обработать пропущенные значения (NaN), выполнить фильтрацию выбросов, провести группировку и агрегацию, а также сохранить результат в эффективном колоночном формате Parquet.

Задачи:

1. Загрузить исходный CSV-файл с «грязными» данными (пропуски, дубликаты, выбросы, неверные форматы) в DataFrame Pandas.
2. Выполнить первичный анализ данных (информация о типах, описательная статистика, выявление пропусков).
3. Реализовать очистку данных:
   - замену или удаление пропущенных значений (NaN)
   - удаление дубликатов строк
   - фильтрацию аномальных значений (выбросов) на основе статистических критериев
   - приведение типов данных (строки в даты, числа)
   - нормализацию текстовых полей (очистка пробелов, приведение регистра)
4. Выполнить агрегацию данных с использованием groupby() и нескольких агрегатных функций (sum, mean, count, min, max, nunique).
5. Создать итоговый DataFrame с результатами агрегации и сохранить его в формате Parquet.
6. Задокументировать все выполненные шаги трансформации.

Теоретическая часть

Pandas в пайплайнах обработки данных

Pandas является основным инструментом для интерактивной и пакетной обработки структурированных данных в Python. В контексте ETL/ELT-пайплайнов Pandas часто используется для:

- **Трансформации малых и средних объёмов данных** (до нескольких гигабайт)
- **Прототипирования** логики очистки перед переносом в SQL/dbt
- **Работы с файлами** в различных форматах (CSV, Excel, JSON, Parquet)

Ключевые методы очистки данных в Pandas

Операция	Методы Pandas	Пример
Просмотр информации	df.info(), df.describe(), df.isnull().sum()	базовый анализ
Удаление пропусков	df.dropna()	удалить строки с любым NaN
Заполнение пропусков	df.fillna(value)	заполнить нулём или средним
Интерполяция	df.interpolate()	заполнить временные ряды
Удаление дубликатов	df.drop_duplicates()	по всем или выбранным колонкам
Фильтрация по условию	df[df['col'] > 0]	оставить только положительные
Обработка выбросов	df[(df['col'] > lower) & (df['col'] < upper)]	IQR или Z-score
Приведение типов	df['col'].astype(), pd.to_datetime()	строки → даты
Нормализация текста	df['col'].str.strip().str.lower()	очистка от пробелов

Агрегация данных

```
# Группировка с несколькими агрегациями
result = df.groupby('category').agg(
```

```
    total_sales=('amount', 'sum'),
    avg_price=('price', 'mean'),
    count_orders=('order_id', 'count'),
    unique_customers=('customer_id', 'nunique')
).reset_index()
```

Формат Parquet

Parquet — колоночный формат хранения данных, широко используемый в data-инжиниринге:

- **Сжатие** (обычно Snappy, GZIP) — экономит место
- **Схема встроена** в файл — не нужно отдельно хранить типы
- **Колоночная структура** — эффективна для аналитических запросов
- **Поддержка сложных типов** (вложенные структуры, списки)

```
df.to_parquet('output.parquet', index=False)
df_loaded = pd.read_parquet('output.parquet')
```

Типичный пайплайн трансформации:

CSV → read_csv() → очистка (fillna/dropna/drop_duplicates) → фильтрация выбросов → группировка/агрегация → to_parquet()

Методика выполнения работы

1. **Подготовка рабочего окружения**
   1.1. Создайте каталог lab_pandas_transform
   1.2. Внутри создайте подкаталоги: input, output, notebooks (опционально)
   1.3. Установите необходимые библиотеки:

```
pip install pandas numpy pyarrow
```

1.4. Поместите в input/ исходный файл raw_sales.csv в соответствии с вашим вариантом.

2. **Загрузка и первичный анализ данных**

2.1. Создайте файл src/transform_sales.py

2.2. Напишите код для загрузки данных:

```
import pandas as pd
import numpy as np
import logging
from datetime import datetime

logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')
logger = logging.getLogger(__name__)

def load_data(filepath):
    df = pd.read_csv(filepath)
    logger.info(f"Loaded {len(df)} rows from {filepath}")
    return df
```

2.3. Выполните первичный анализ и выведите в лог:

```
python
logger.info(f"Data types:\n{df.dtypes}")
logger.info(f"Missing values:\n{df.isnull().sum()}")
logger.info(f"Descriptive stats:\n{df.describe()}")
```

3. **Очистка данных**

3.1. **Обработка пропущенных значений:**

```
# Проверка доли пропусков
missing_pct = df.isnull().mean() * 100
logger.info(f"Missing percentage:\n{missing_pct}")

# Стратегия 1: удаление строк с критическими пропусками (например, в ID)
df = df.dropna(subset=['transaction_id', 'date'])
```

```
# Стратегия 2: заполнение числовых колонок медианой/средним
df['amount'] = df['amount'].fillna(df['amount'].median())

# Стратегия 3: заполнение категориальных колонок модой или 'unknown'
df['category'] = df['category'].fillna('unknown')

# Стратегия 4: интерполяция для временных рядов (если уместно)
# df['value'] = df['value'].interpolate()
```

3.2. **Удаление дубликатов:**

```
initial_count = len(df)
df = df.drop_duplicates(subset=['transaction_id'])
logger.info(f"Removed {initial_count - len(df)} duplicate rows")
```

3.3. **Обработка выбросов:**

```
# Метод IQR (межквартильный размах)
def remove_outliers_iqr(df, column, multiplier=1.5):
    Q1 = df[column].quantile(0.25)
    Q3 = df[column].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - multiplier * IQR
    upper_bound = Q3 + multiplier * IQR
    before = len(df)
    df = df[(df[column] >= lower_bound) & (df[column] <= upper_bound)]
    logger.info(f"Removed {before - len(df)} outliers from {column}")
    return df

df = remove_outliers_iqr(df, 'amount')
df = remove_outliers_iqr(df, 'quantity', multiplier=2)
```

3.4. **Приведение типов данных:**

```
# Преобразование даты
df['date'] = pd.to_datetime(df['date'], errors='coerce')

# Удаление строк с некорректной датой
df = df.dropna(subset=['date'])

# Преобразование числовых колонок
df['amount'] = pd.to_numeric(df['amount'], errors='coerce')
df['quantity'] = pd.to_numeric(df['quantity'], errors='coerce').fillna(1)

# Категориальный тип для экономии памяти
df['category'] = df['category'].astype('category')
```

3.5. **Нормализация текстовых полей:**

```
python
df['product_name'] = df['product_name'].str.strip().str.lower()
df['customer_email'] = df['customer_email'].str.lower().str.strip()
```

4. **Агрегация данных**

4.1. Выполните группировку согласно вашему индивидуальному заданию:

```
def aggregate_data(df):
    # Пример агрегации по категориям и месяцам
    df['year_month'] = df['date'].dt.to_period('M')

    aggregated = df.groupby(['category', 'year_month']).agg(
        total_amount=('amount', 'sum'),
        avg_amount=('amount', 'mean'),
        total_quantity=('quantity', 'sum'),
```

```
        transaction_count=('transaction_id', 'count'),
        unique_customers=('customer_id', 'nunique')
    ).reset_index()

    # Преобразование Period в строку для сохранения
    aggregated['year_month'] = aggregated['year_month'].astype(str)

    logger.info(f"Aggregation complete: {len(aggregated)} groups")
    return aggregated
```

5. **Сохранение в Parquet**

   5.1. Сохраните очищенный и агрегированный результат:

```
def save_parquet(df, filepath, compression='snappy'):
    df.to_parquet(filepath, index=False, compression=compression)
    logger.info(f"Saved to {filepath} ({compression} compression)")

    # Вывод информации о размере
    import os
    size_mb = os.path.getsize(filepath) / (1024 * 1024)
    logger.info(f"File size: {size_mb:.2f} MB")
```

6. **Дополнительные проверки качества**

   6.1. Добавьте валидацию результатов:

```
def validate_data(df):
    assertions = {
        'no_negative_amount': (df['amount'] >= 0).all(),
        'no_null_dates': df['date'].notna().all(),
        'positive_quantity': (df['quantity'] > 0).all()
    }
```

```
    for check, result in assertions.items():
        if result:
            logger.info(f"✓ {check} passed")
        else:
            logger.error(f"✗ {check} failed")
            raise ValueError(f"Validation failed: {check}")
```

7. **Итоговый скрипт**

   7.1. Объедините все функции в main():

```
def main():
    df = load_data('input/raw_sales.csv')
    df = clean_data(df)
    df = aggregate_data(df)
    validate_data(df)
    save_parquet(df, 'output/aggregated_sales.parquet')

if __name__ == "__main__":
    main()
```

Пример выполнения задания

Исходные данные (input/raw_sales.csv**):**

```
transaction_id,date,category,product_name,amount,quantity,customer_id
1,2025-01-15,Electronics,Laptop,1200.50,1,CUST001
2,2025-01-15,Books,Python Guide,45.00,2,CUST002
3,2025-01-16,Electronics,,,1,CUST001
4,2025-01-16,Books,Data Science Book,9999.99,1,CUST003
5,2025-01-17,Electronics,Mouse,25.50,-1,CUST002
6,2025-01-17,Clothing,Shirt,35.00,3,CUST001
7,2025-01-18,Electronics,Keyboard,85.00,1,
```

8,2025-01-18,Books,Python Guide,45.00,2,CUST002

Выполнение скрипта:

1. **Загрузка:** 8 строк
2. **Очистка:**
   - Удаление строки 3 (пропущены product_name и amount)
   - Заполнение customer_id пустых → 'unknown'
   - Удаление строки 4 (amount 9999.99 как выброс)
   - Удаление строки 5 (quantity = -1)
   - Удаление дубликата строки 2 и 8
3. **Результат после очистки:** 4 строки
4. **Агрегация по category и date:**

Выходной файл (output/aggregated_sales.parquet**) в виде таблицы:**

category	date	total_amount	avg_amount	total_quantity	transaction_count	unique_customers
Books	2025-01-15	45.00	45.00	2	1	1
Clothing	2025-01-17	35.00	35.00	3	1	1
Electronics	2025-01-15	1200.50	1200.50	1	1	1
Electronics	2025-01-17	25.50	25.50	1	1	1

Индивидуальные задания (15 вариантов)

Каждый вариант описывает структуру исходного CSV-файла и требования к очистке и агрегации. Студент самостоятельно определяет пороговые значения для выбросов, стратегии заполнения пропусков и набор агрегатных функций.

Вар.	Исходный CSV (колонки)	Особенности данных	Требуемая агрегация
1	order_id,user_id,amount,order_date,status,payment_method	amount: отрицательные и нулевые; status: 'pending','completed','cancelled'; есть дубликаты order_id	Группировка по status и payment_method: сумма amount, количество заказов, средний amount
2	product_id,name,price,stock_quantity,category,supplier	price: 0 и выбросы > 10000; stock_quantity: отрицательные; name: пробелы, разный регистр	Группировка по category: мин/макс/средняя цена, общий запас, количество уникальных поставщиков
3	employee_id,name,department,salary,hire_date,performance_score	salary: пропуски и выбросы (IQR); hire_date: неверные форматы; performance_score: 0-100, но есть -1	Группировка по department: средняя зарплата, медианный performance_score, мин/макс дата найма
4	sale_id,date,store_id,product,sales_amount,units_sold	sales_amount: NaN; units_sold: отрицательные; date: пропуски	Группировка по store_id и месяцу(date): общая выручка, общее кол-во единиц, средний чек, количество дней с продажами
5	log_id,timestamp,user_id,action,duration_seconds,page_url	duration_seconds: выбросы > 3600 (1 час); user_id: пропуски; action: пустые строки	Группировка по action: средняя длительность, количество уникальных пользователей, общее число действий

Вар.	Исходный CSV (колонки)	Особенности данных	Требуемая агрегация
6	student_id,name,test_score,grade,subject,test_date	test_score: 0-100, но есть -1 и 999; grade: A,B,C,D,F, но есть пропуски; name: дубликаты	Группировка по subject и grade: средний test_score, количество студентов, мин/макс даты
7	transaction_id,customer_id,amount,timestamp,merchant,location	amount: выбросы > 50000; timestamp: строки разного формата; merchant: пробелы	Группировка по merchant: количество транзакций, сумма amount, средний amount, количество уникальных location
8	reading_id,sensor_id,temperature,humidity,recorded_at,battery	temperature: -50 до +50, но есть 999; humidity: 0-100, пропуски; battery: 0-100, отрицательные	Группировка по sensor_id: средние температура/влажность, минимальный battery, количество записей
9	click_id,cookie_id,page,event_type,timestamp,referrer	timestamp: пропуски и неверный формат; event_type: 'click','view','scroll'; cookie_id: NULL	Группировка по event_type и часу(timestamp): количество кликов, количество уникальных cookie_id
10	shipment_id,order_id,carrier,tracking_number,weight_kg,shipping_cost,destination	weight_kg: отрицательные и нулевые; shipping_cost: выбросы; tracking_number: дубликаты	Группировка по carrier и destination: общий вес, общая стоимость доставки, средняя стоимость за кг

Вар.	Исходный CSV (колонки)	Особенности данных	Требуемая агрегация
11	call_id,agent_id,customer_phone,call_duration,start_time,end_time,disposition	call_duration: выбросы > 7200; start_time/end_time: несоответствия; disposition: 'answered','missed'	Группировка по agent_id: средняя длительность, количество звонков, доля отвеченных звонков
12	review_id,product_id,rating,review_text,date,helpful_votes	rating: 1-5, но есть 0 и 6; date: пропуски; helpful_votes: отрицательные	Группировка по product_id: средний рейтинг, количество отзывов, сумма helpful_votes
13	event_id,user_id,event_type,value,event_date,platform	event_type: 'purchase','view','add_to_cart'; value: NaN для view	Группировка по platform и event_type: сумма value (только для purchase), количество событий
14	stock_id,product_code,warehouse,quantity,last_updated,reorder_point	quantity: отрицательные и выбросы; last_updated: пропуски; reorder_point: NaN	Группировка по warehouse: общее quantity, количество товаров ниже reorder_point, средний reorder_point
15	subscription_id,user_id,plan,start_date,end_date,monthly_price,status	start_date/end_date: логические ошибки (end < start); monthly_price: 0 и отрицательные; status: 'active','canceled'	Группировка по plan и status: средняя monthly_price, количество подписок, средняя длительность (end_date - start_date)

Дополнительное задание для всех вариантов:

Добавьте к итоговому отчёту визуализацию (с использованием matplotlib или seaborn), показывающую:

- Распределение числовой переменной до и после удаления выбросов (гистограмма)
- Топ-5 категорий/групп по агрегированному показателю (горизонтальная бар-диаграмма)

Требования к отчёту

1. Титульный лист с названием работы, ФИО, группой, номером варианта.
2. Цель и задачи (из методички).
3. Теоретическая часть (Pandas для очистки, методы обработки пропусков/выбросов, формат Parquet).
4. Ход выполнения работы:
   - Листинг скрипта transform_sales.py с комментариями.
   - Описание каждой стадии очистки с обоснованием выбранных стратегий.
   - Выводы из описательной статистики до и после очистки.
5. Результаты:
   - Таблица с количеством строк на каждом этапе (загружено → после удаления пропусков → после удаления выбросов → после агрегации).
   - Пример строк исходных и очищенных данных (по 3-5 строк).
   - Содержимое итогового агрегированного DataFrame (таблица/скриншот).
   - Информация о размере Parquet-файла vs исходного CSV.
6. Ответы на контрольные вопросы.
7. Вывод — какие методы очистки оказались наиболее полезными, с какими сложностями столкнулись.

Контрольные вопросы

1. В каких ситуациях предпочтительнее удалить строку с пропуском (dropna), а в каких — заполнить пропуск (fillna)?

2. Чем отличается удаление выбросов по IQR от удаления по Z-score? Когда какой метод применять?

3. Почему датасеты с выбросами могут «ломать» среднее арифметическое, но не медиану?

4. Как проверить, что преобразование типов (astype, pd.to_datetime) прошло успешно?

5. Зачем использовать errors='coerce' в pd.to_numeric() и pd.to_datetime()?

6. В чём преимущество формата Parquet перед CSV для хранения очищенных данных?

7. Какой метод groupby().agg() позволяет задать разные агрегатные функции для разных колонок?

8. Как бы вы реализовали обработку выбросов не по одному столбцу, а по комбинации условий (например, слишком высокая цена для данной категории)?

9. Что произойдёт с категориальными колонками при сохранении в Parquet, если они имеют тип category?

10. Как с помощью Pandas найти строки, в которых дата не соответствует ожидаемому диапазону (например, 2100 год)?

11. В чём разница между методами fillna(ffill) и interpolate() для временных рядов?

12. Как сохранить DataFrame в несколько Parquet-файлов (шардирование) с помощью partition_cols?

13. Почему для больших датасетов (сотни GB) не рекомендуется использовать Pandas? Какие альтернативы существуют?

14. Как с помощью Pandas найти и отобразить дубликаты, не удаляя их сразу (только для анализа)?

15. Если после агрегации groupby вы получаете MultiIndex, как его преобразовать в обычные колонки?

Лабораторная работа №5.

Проектирование схемы «звезда»

Цель работы: спроектировать и реализовать схему типа «звезда» (таблица фактов и таблицы измерений) на основе исходной нормализованной OLTP-схемы данных о продажах, заказах, клиентах и товарах.

Задачи:

1. Проанализировать структуру исходной нормализованной OLTP-базы данных (таблицы customers, products, orders, order_items).
2. Выделить бизнес-процессы, подлежащие анализу (продажи, заказы, движение товаров).
3. Определить зерно (granularity) таблицы фактов — уровень детализации каждой строки факта.
4. Спроектировать таблицы измерений (dimensions): дата, клиент, товар, категория.
5. Спроектировать таблицу фактов (fact table) с внешними ключами на измерения и числовыми мерами (количество, сумма, скидка).
6. Написать SQL-скрипты (INSERT INTO ... SELECT) для заполнения измерений и фактов из исходных OLTP-таблиц.
7. Проверить целостность данных (отсутствие «сиротских» записей в фактах).
8. Выполнить аналитический запрос для верификации схемы.

Теоретическая часть

Схема «звезда» (Star Schema) в хранилищах данных

Схема «звезда» — это наиболее распространённая модель организации данных в витринах данных (data marts) и слое золота (gold layer) хранилища. Она представляет собой денормализованную структуру, оптимизированную для аналитических запросов (OLAP), а не для транзакционной обработки (OLTP).

Основные компоненты схемы «звезда»:

Компонент	Описание	Пример
Таблица фактов (Fact Table)	Содержит события/факты бизнес-процесса, внешние ключи на измерения и числовые меры	fact_sales с полями: date_key, customer_key, product_key, quantity, amount
Таблицы измерений (Dimension Tables)	Содержат атрибуты анализируемых объектов (кто? что? где? когда?)	dim_customer (имя, город, сегмент), dim_product (название, категория, цена)
Внешние ключи (Foreign Keys)	Связывают факты с измерениями	customer_key ссылается на dim_customer
Меры (Measures)	Числовые значения, которые агрегируются (SUM, AVG, COUNT)	quantity, revenue, discount_amount

Почему «звезда»?

- Название возникло из-за визуального сходства: таблица фактов в центре, измерения — «лучи» вокруг неё.
- Денормализация уменьшает количество JOIN'ов в аналитических запросах.
- Высокая производительность для агрегаций и группировок.

Сравнение OLTP и DWH-схем:

Характеристика	OLTP (нормализованная)	DWH (схема «звезда»)
Цель	Транзакции (CRUD)	Аналитика (чтение, агрегация)

Характеристика	OLTP (нормализованная)	DWH (схема «звезда»)
Нормализация	Высокая (3NF)	Низкая (денормализация)
Количество JOIN'ов в типовом запросе	5-10	1-3
Избыточность данных	Минимальная	Намеренная (для скорости)
Типичные операции	INSERT/UPDATE/DELETE	SELECT, GROUP BY, агрегации

Типы медленно меняющихся измерений (SCD):

Тип	Описание	Пример
SCD Type 0	Не меняется	ID, дата рождения
SCD Type 1	Перезаписывается (история теряется)	Email, телефон
SCD Type 2	Хранится полная история (версионность)	Адрес, должность
SCD Type 3	Хранится только предыдущее значение	Старый/новый менеджер

Типичный процесс построения схемы «звезда»:

OLTP (нормализованный) → извлечение → трансформация (денормализация) → загрузка → DWH (звезда)

Методика выполнения работы

1. **Подготовка окружения**

   1.1. Создайте каталог lab5_star_schema

   1.2. Внутри создайте подкаталоги: sql, data, reports

   1.3. Запустите PostgreSQL в Docker-контейнере (как в ЛР №2):

```
docker run -d \
  --name postgres_dwh \
  -e POSTGRES_USER=dwh_user \
  -e POSTGRES_PASSWORD=dwh_pass \
  -e POSTGRES_DB=dwh_db \
  -v $(pwd)/data:/var/lib/postgresql/data \
  -p 5432:5432 \
  postgres:15
```

2. **Создание исходной OLTP-схемы**

   2.1. Подключитесь к базе данных и создайте схему oltp

   2.2. Выполните следующий SQL-скрипт для создания и заполнения исходных таблиц:

```
CREATE SCHEMA IF NOT EXISTS oltp;

-- Таблица клиентов
CREATE TABLE oltp.customers (
    customer_id SERIAL PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    city VARCHAR(50),
    registration_date DATE DEFAULT CURRENT_DATE,
    customer_segment VARCHAR(20) DEFAULT 'Standard'
);
```

```
-- Таблица товаров
CREATE TABLE oltp.products (
    product_id SERIAL PRIMARY KEY,
    product_name VARCHAR(100) NOT NULL,
    category VARCHAR(50),
    subcategory VARCHAR(50),
    unit_price DECIMAL(10,2) NOT NULL
);

-- Таблица заказов
CREATE TABLE oltp.orders (
    order_id SERIAL PRIMARY KEY,
    customer_id INTEGER REFERENCES oltp.customers(customer_id),
    order_date DATE NOT NULL,
    status VARCHAR(20) DEFAULT 'Pending'
);

-- Таблица позиций заказа (детали)
CREATE TABLE oltp.order_items (
    order_item_id SERIAL PRIMARY KEY,
    order_id INTEGER REFERENCES oltp.orders(order_id),
    product_id INTEGER REFERENCES oltp.products(product_id),
    quantity INTEGER NOT NULL CHECK (quantity > 0),
    discount DECIMAL(4,2) DEFAULT 0 CHECK (discount >= 0 AND
discount <= 100)
);

-- Заполнение тестовыми данными
```

INSERT INTO oltp.customers (first_name, last_name, email, city, registration_date, customer_segment) VALUES

('Иван', 'Петров', 'ivan@mail.com', 'Москва', '2023-01-15', 'Gold'),

('Мария', 'Сидорова', 'maria@mail.com', 'СПб', '2023-02-20', 'Standard'),

('Алексей', 'Иванов', 'alex@mail.com', 'Москва', '2023-03-10', 'Silver');

INSERT INTO oltp.products (product_name, category, subcategory, unit_price) VALUES

('Ноутбук', 'Электроника', 'Компьютеры', 55000),

('Мышь', 'Электроника', 'Аксессуары', 1500),

('Клавиатура', 'Электроника', 'Аксессуары', 3500),

('Книга Python', 'Книги', 'Программирование', 1200);

INSERT INTO oltp.orders (customer_id, order_date, status) VALUES

(1, '2024-01-10', 'Completed'),

(2, '2024-01-10', 'Completed'),

(1, '2024-02-15', 'Completed'),

(3, '2024-02-20', 'Pending');

INSERT INTO oltp.order_items (order_id, product_id, quantity, discount) VALUES

(1, 1, 1, 0),

(1, 2, 2, 10),

(2, 3, 1, 0),

(3, 1, 1, 5),

(3, 4, 1, 0),

(4, 2, 3, 0);

3. **Проектирование схемы «звезда»**

3.1. Определите зерно таблицы фактов (строка = одна позиция заказа — order_item_id).

3.2. Создайте схему dwh для будущего хранилища:

```
CREATE SCHEMA IF NOT EXISTS dwh;
```

3.3. Спроектируйте таблицу измерения dim_date. Создайте её с уровнями иерархии (год, месяц, день):

```
CREATE TABLE dwh.dim_date (
    date_key INTEGER PRIMARY KEY,
    full_date DATE NOT NULL,
    year INTEGER,
    quarter INTEGER,
    month INTEGER,
    month_name VARCHAR(20),
    day INTEGER,
    day_of_week INTEGER,
    day_name VARCHAR(20),
    is_weekend BOOLEAN
);
```

3.4. Создайте таблицу измерения dim_customer:

```
CREATE TABLE dwh.dim_customer (
    customer_key SERIAL PRIMARY KEY,
    customer_id INTEGER,
    full_name VARCHAR(100),
    email VARCHAR(100),
    city VARCHAR(50),
    customer_segment VARCHAR(20),
    valid_from DATE,
```

```
    valid_to DATE,
    is_current BOOLEAN DEFAULT TRUE
);
```

3.5. Создайте таблицу измерения dim_product:

```
CREATE TABLE dwh.dim_product (
    product_key SERIAL PRIMARY KEY,
    product_id INTEGER,
    product_name VARCHAR(100),
    category VARCHAR(50),
    subcategory VARCHAR(50),
    unit_price DECIMAL(10,2)
);
```

3.6. Создайте таблицу фактов fact_sales:

```
CREATE TABLE dwh.fact_sales (
    sales_key SERIAL PRIMARY KEY,
    date_key INTEGER REFERENCES dwh.dim_date(date_key),
    customer_key INTEGER REFERENCES dwh.dim_customer(customer_key),
    product_key INTEGER REFERENCES dwh.dim_product(product_key),
    quantity INTEGER NOT NULL,
    unit_price DECIMAL(10,2),
    discount_percent DECIMAL(4,2),
    gross_amount DECIMAL(12,2),
    discount_amount DECIMAL(12,2),
    net_amount DECIMAL(12,2)
);
```

4. **Заполнение измерений**

4.1. Заполните dim_date за последние 5 лет с помощью генерации дат:

```
INSERT INTO dwh.dim_date (date_key, full_date, year, quarter, month, month_name, day, day_of_week, day_name, is_weekend)
SELECT
    TO_CHAR(d, 'YYYYMMDD')::INTEGER AS date_key,
    d AS full_date,
    EXTRACT(YEAR FROM d)::INTEGER AS year,
    EXTRACT(QUARTER FROM d)::INTEGER AS quarter,
    EXTRACT(MONTH FROM d)::INTEGER AS month,
    TO_CHAR(d, 'Month') AS month_name,
    EXTRACT(DAY FROM d)::INTEGER AS day,
    EXTRACT(DOW FROM d)::INTEGER AS day_of_week,
    TO_CHAR(d, 'Day') AS day_name,
    EXTRACT(DOW FROM d) IN (0, 6) AS is_weekend
FROM generate_series('2020-01-01'::DATE, '2025-12-31'::DATE, '1 day'::INTERVAL) AS d;
```

4.2. Заполните dim_customer (SCD Type 1 — перезаписываем для простоты):

```
INSERT INTO dwh.dim_customer (customer_id, full_name, email, city, customer_segment, valid_from, valid_to, is_current)
SELECT
    customer_id,
    CONCAT(first_name, ' ', last_name) AS full_name,
    email,
    city,
    customer_segment,
    registration_date AS valid_from,
    '9999-12-31'::DATE AS valid_to,
```

```
    TRUE AS is_current
FROM oltp.customers;
```

4.3. Заполните dim_product:

```
INSERT INTO dwh.dim_product (product_id, product_name, category, subcategory, unit_price)
SELECT
    product_id,
    product_name,
    category,
    subcategory,
    unit_price
FROM oltp.products;
```

5. **Заполнение таблицы фактов**

5.1. Напишите запрос для заполнения fact_sales:

```
INSERT INTO dwh.fact_sales (
    date_key, customer_key, product_key, quantity,
    unit_price, discount_percent, gross_amount, discount_amount, net_amount
)
SELECT
    TO_CHAR(o.order_date, 'YYYYMMDD')::INTEGER AS date_key,
    dc.customer_key,
    dp.product_key,
    oi.quantity,
    p.unit_price,
    oi.discount,
    oi.quantity * p.unit_price AS gross_amount,
    (oi.quantity * p.unit_price) * (oi.discount / 100) AS discount_amount,
    (oi.quantity * p.unit_price) * (1 - oi.discount / 100) AS net_amount
```

FROM oltp.order_items oi

JOIN oltp.orders o ON oi.order_id = o.order_id

JOIN oltp.products p ON oi.product_id = p.product_id

JOIN dwh.dim_customer dc ON o.customer_id = dc.customer_id AND dc.is_current = TRUE

JOIN dwh.dim_product dp ON p.product_id = dp.product_id;

6. **Проверка целостности данных**

6.1. Выполните проверочные запросы:

-- Проверка: нет ли фактов без соответствующих измерений

```
SELECT COUNT(*) AS orphaned_facts
FROM dwh.fact_sales fs
LEFT JOIN dwh.dim_date d ON fs.date_key = d.date_key
WHERE d.date_key IS NULL;
```

-- Проверка: подсчёт общего количества строк

```
SELECT COUNT(*) AS fact_count FROM dwh.fact_sales;
SELECT COUNT(*) AS order_item_count FROM oltp.order_items;
```

7. **Выполнение аналитического запроса**

7.1. Напишите запрос для верификации схемы:

```
SELECT
    d.year,
    d.month_name,
    c.customer_segment,
    p.category,
    SUM(fs.quantity) AS total_quantity,
    SUM(fs.net_amount) AS total_revenue,
    AVG(fs.net_amount / fs.quantity) AS avg_price,
    COUNT(DISTINCT fs.customer_key) AS unique_customers
```

```
FROM dwh.fact_sales fs
JOIN dwh.dim_date d ON fs.date_key = d.date_key
JOIN dwh.dim_customer c ON fs.customer_key = c.customer_key
JOIN dwh.dim_product p ON fs.product_key = p.product_key
GROUP BY d.year, d.month_name, c.customer_segment, p.category
ORDER BY d.year, d.month_name, total_revenue DESC;
```

Пример выполнения задания

Исходные OLTP-данные (схема: customers → orders → order_items ← products)

После выполнения всех шагов получается следующая схема «звезда»:

Таблица измерений dim_customer**:**

customer_key	**customer_id**	**full_name**	**city**	**customer_segment**	**is_current**
1	1	Иван Петров	Москва	Gold	TRUE
2	2	Мария Сидорова	СПб	Standard	TRUE
3	3	Алексей Иванов	Москва	Silver	TRUE

Таблица измерений dim_product**:**

product_key	product_id	product_name	category	unit_price
1	1	Ноутбук	Электроника	55000
2	2	Мышь	Электроника	1500
3	3	Клавиатура	Электроника	3500
4	4	Книга Python	Книги	1200

Таблица фактов fact_sales **(фрагмент):**

sales_key	date_key	customer_key	product_key	quantity	unit_price	discount_percent	net_amount
1	20240110	1	1	1	55000	0	55000
2	20240110	1	2	2	1500	10	2700
3	20240110	2	3	1	3500	0	3500
4	20240215	1	1	1	55000	5	52250
5	20240215	1	4	1	1200	0	1200

Индивидуальные задания

Каждый вариант представляет собой описание бизнес-области и исходных OLTP-таблиц. Студент должен самостоятельно спроектировать схему «звезда» (определить факты и измерения, их атрибуты и связи) и реализовать её в PostgreSQL.

Вариант 1. Розничные продажи

Исходные таблицы: stores (магазины), employees (сотрудники), products (товары), sales_transactions (продажи), sales_lines (строки продаж). Построить схему для анализа выручки по магазинам, сотрудникам, товарам и времени.

Вариант 2. Логистика и доставка

Исходные таблицы: warehouses (склады), drivers (водители), vehicles (транспорт), deliveries (доставки), delivery_items (позиции доставки). Построить схему для анализа эффективности доставки (время, расстояние, загрузка).

Вариант 3. Интернет-магазин (отзывы)

Исходные таблицы: users, products, reviews, review_ratings (оценки по критериям). Построить схему для анализа满意度 клиентов (средняя оценка, динамика оценок, топ товаров).

Вариант 4. Банковские транзакции

Исходные таблицы: customers, accounts, cards, transactions, transaction_types. Построить схему для анализа оборота по клиентам, счетам, типам транзакций и времени.

Вариант 5. Образовательная платформа

Исходные таблицы: students, courses, instructors, enrollments, lesson_completions. Построить схему для анализа прогресса студентов, популярности курсов и активности по времени.

Вариант 6. Медицинские визиты

Исходные таблицы: patients, doctors, departments, visits, diagnoses, visit_procedures. Построить схему для анализа загрузки врачей, частоты диагнозов и сезонности визитов.

Вариант 7. Телеком (звонки)

Исходные таблицы: subscribers, tariffs, calls, call_details (длительность, тарификация). Построить схему для анализа длительности и стоимости звонков по абонентам, тарифам и времени.

Вариант 8. Производство продукции

Исходные таблицы: factories, products, production_batches, batch_quality_checks. Построить схему для анализа объёмов производства и процента брака по заводам и временным периодам.

Вариант 9. HR-аналитика

Исходные таблицы: employees, departments, positions, salary_payments, performance_reviews. Построить схему для анализа динамики зарплат, текучести кадров и эффективности сотрудников.

Вариант 10. Подписки и платежи

Исходные таблицы: subscribers, subscription_plans, payments, payment_statuses. Построить схему для анализа MRR (Monthly Recurring Revenue), оттока и среднего чека.

Вариант 11. Туристические бронирования

Исходные таблицы: tourists, tours, hotels, bookings, booking_services (доп. услуги). Построить схему для анализа загрузки туров, популярности отелей и сезонности.

Вариант 12. Ритейл (возвраты товаров)

Исходные таблицы: customers, products, sales, returns, return_reasons. Построить схему для анализа процента возвратов по товарам, причинам и времени.

Вариант 13. Складской учёт

Исходные таблицы: warehouses, products, incoming_orders, outgoing_orders, inventory_movements. Построить схему для анализа оборачиваемости запасов и движения товаров.

Вариант 14. Маркетплейс (отзывы продавцов)

Исходные таблицы: sellers, buyers, orders, order_items, seller_ratings. Построить схему для анализа рейтинга продавцов, объёмов продаж и географии.

Вариант 15. Спортивные соревнования

Исходные таблицы: teams, players, matches, match_stats (очки, фолы, время). Построить схему для анализа результативности команд и игроков по сезонам.

Требования к отчёту

1. Титульный лист с названием работы, ФИО, группой, номером варианта.
2. Цель и задачи (из методички).
3. Теоретическая часть (понятие схемы «звезда», отличие от OLTP, компоненты схемы).
4. Ход выполнения работы:

- ER-диаграмма исходной OLTP-схемы (можно текстовое описание или нарисованная диаграмма).
- Определение зерна таблицы фактов.
- SQL-код создания всех таблиц DWH (измерения и факт).
- SQL-код заполнения таблиц (с комментариями).
- Обоснование выбора SCD-типа для каждого измерения.

5. Результаты:
   - Скриншоты содержимого таблиц (первые 5-10 строк каждой).
   - Пример аналитического запроса с результатом выполнения.
   - Проверка целостности (отсутствие «сиротских» записей).
6. Ответы на контрольные вопросы.
7. Вывод — преимущества полученной схемы «звезда» для аналитики по сравнению с исходной OLTP-схемой.

Контрольные вопросы

1. Чем схема «звезда» отличается от схемы «снежинка»? В каких случаях снежинка предпочтительнее?

2. Почему таблица фактов обычно растёт быстрее, чем таблицы измерений?

3. Что такое «зерно» (granularity) таблицы фактов и почему его важно определить до проектирования?

4. Какие меры в таблице фактов являются аддитивными, полуаддитивными и неаддитивными?

5. Как бы вы реализовали SCD Type 2 для измерения dim_customer, если клиент меняет город проживания?

6. Зачем в измерении dim_date хранить отдельные колонки (год, месяц, день), если можно извлекать из full_date на лету?

7. Что произойдёт с аналитикой, если в таблицу фактов добавится строка с внешним ключом, которого нет в измерении?

8. В каких ситуациях оправдано хранение нескольких таблиц фактов (например, fact_sales и fact_inventory) в одной схеме?

9. Как конвейер ETL должен обрабатывать новые измерения, появившиеся в источнике (новые категории товаров)?

10. Почему в схеме «звезда» не рекомендуется хранить текстовые описания в таблице фактов?

11. Какой тип соединения (JOIN) используется при заполнении фактов из OLTP-таблиц и почему важно убедиться в его корректности?

12. Как можно переиспользовать измерение dim_date для разных таблиц фактов (продажи, логистика, HR)?

13. Что такое «конформированные измерения» (conformed dimensions) и зачем они нужны в корпоративном хранилище?

14. Как изменится схема, если в таблице фактов нужно анализировать не только фактические продажи, но и прогнозные?

15. Как с помощью схемы «звезда» можно реализовать аналитику «на начало периода» (например, остаток товаров на начало месяца)?

Лабораторная работа №6.
Медленно меняющиеся измерения (SCD Type 2)

Цель работы: реализовать логику отслеживания истории изменения атрибутов измерений с использованием подхода SCD Type 2 (ведение записей с полями valid_from, valid_to, is_current) на примере таблицы клиентов или товаров.

Задачи:

1. Изучить классификацию медленно меняющихся измерений (SCD Type 0–6).
2. Создать эталонную таблицу измерений с поддержкой SCD Type 2 (поля valid_from, valid_to, is_current).
3. Реализовать SQL-процедуру (или Python-скрипт), которая принимает источник изменённых данных и выполняет:
   - закрытие предыдущей активной записи (установку valid_to = текущая дата, is_current = FALSE)
   - вставку новой записи с новыми атрибутами, valid_from = текущая дата, valid_to = '9999-12-31', is_current = TRUE
4. Обеспечить обработку ситуации, когда изменений фактически нет (избегать дублирования записей).
5. Протестировать работу механизма на наборе тестовых обновлений (вставка новых записей, изменение атрибутов существующих).
6. Продемонстрировать возможность построения аналитики как «на момент» (с использованием valid_from/valid_to), так и актуального состояния (is_current = TRUE).

Теоретическая часть

Медленно меняющиеся измерения (Slowly Changing Dimensions, SCD)

В хранилищах данных атрибуты измерений (например, адрес клиента, название товара) могут изменяться со временем. Способ отслеживания этих изменений определяет тип SCD. Наиболее распространены три типа.

Тип SCD	Описание	Когда использовать
SCD Type 0	Фиксированные значения – изменения игнорируются	Идентификаторы, дата рождения
SCD Type 1	Перезапись старого значения новым (история теряется)	Исправление опечаток, обновление телефона
SCD Type 2	Хранение полной истории: создаётся новая версия записи	Адрес проживания, должность, тарифный план
SCD Type 3	Хранение только предыдущего значения (доп. колонка old_value)	Если важно знать только «было – стало»
SCD Type 4	История выносится в отдельную таблицу (мини-измерение)	Для часто меняющихся атрибутов
SCD Type 6	Гибрид Type 2 + Type 3 (история + текущее значение в отдельной колонке)	Сложные аналитические сценарии

Структура таблицы для SCD Type 2

```
CREATE TABLE dim_customer_scd2 (
    surrogate_key SERIAL PRIMARY KEY,     -- искусственный ключ
    business_key INTEGER NOT NULL,        -- естественный ключ (customer_id)
    full_name VARCHAR(100),
    city VARCHAR(50),
    customer_segment VARCHAR(20),
    valid_from DATE NOT NULL,
    valid_to DATE NOT NULL,               -- '9999-12-31' для активной записи
    is_current BOOLEAN DEFAULT TRUE
```

);

Логика обновления при изменении атрибута customer_segment:

1. **Найти активную запись** для данного business_key (is_current = TRUE).
2. **Сравнить** все значимые атрибуты (кроме valid_* и surrogate_key) с новыми значениями.
3. **Если есть различия:**
   - Закрыть старую запись: UPDATE SET valid_to = CURRENT_DATE, is_current = FALSE
   - Вставить новую: INSERT ... (new values, valid_from = CURRENT_DATE, valid_to = '9999-12-31', is_current = TRUE)
4. **Если различий нет** – ничего не делать.

Реализация на SQL (функция/процедура) vs Python:

Подход	Преимущества	Недостатки
SQL (PL/pgSQL)	Выполняется внутри БД, атомарность, производительность	Сложнее отлаживать, привязанность к конкретной СУБД
Python + psycopg2	Гибкость, простота отладки, можно использовать Pandas	Дополнительный слой, сетевые вызовы

Типичный пайплайн обновления измерения:

Источник (OLTP) → выявление изменений → SCD2-обновление → DWH измерение (с историей)

Методика выполнения работы

1. **Подготовка окружения**
   1.1. Создайте каталог lab6_scd_type2
   1.2. Внутри создайте подкаталоги: sql, scripts, data, logs

1.3. Запустите PostgreSQL в Docker-контейнере (если не запущен):

docker run -d --name postgres_scd -e POSTGRES_USER=etl_user -e POSTGRES_PASSWORD=etl_pass -e POSTGRES_DB=dwh_db -v $(pwd)/data:/var/lib/postgresql/data -p 5432:5432 postgres:15

2. **Создание исходных таблиц**

2.1. Подключитесь к БД и создайте схему source для имитации источников:

```
CREATE SCHEMA IF NOT EXISTS source;
CREATE SCHEMA IF NOT EXISTS dwh;
```

2.2. Создайте таблицу источника source.customers, которая будет обновляться (имитация OLTP):

```
CREATE TABLE source.customers (
    customer_id INTEGER PRIMARY KEY,
    full_name VARCHAR(100),
    city VARCHAR(50),
    customer_segment VARCHAR(20),
    last_updated DATE DEFAULT CURRENT_DATE
);
```

2.3. Заполните таблицу источника начальными данными:

```
INSERT INTO source.customers (customer_id, full_name, city, customer_segment) VALUES
(1, 'Иван Петров', 'Москва', 'Gold'),
(2, 'Мария Сидорова', 'СПб', 'Standard'),
(3, 'Алексей Иванов', 'Москва', 'Silver');
```

3. **Создание целевой таблицы измерения с SCD Type 2**

3.1. Выполните SQL-скрипт:

```
CREATE TABLE dwh.dim_customer_scd2 (
    surrogate_key SERIAL PRIMARY KEY,
    customer_id INTEGER NOT NULL,
    full_name VARCHAR(100),
    city VARCHAR(50),
    customer_segment VARCHAR(20),
    valid_from DATE NOT NULL,
    valid_to DATE NOT NULL,
    is_current BOOLEAN DEFAULT TRUE
);

CREATE INDEX idx_customer_business ON dwh.dim_customer_scd2(customer_id, is_current);
```

4. **Реализация SCD Type 2 на SQL (через хранимую функцию)**

   4.1. Создайте функцию merge_customer_scd2(), которая принимает данные из источника и обновляет измерение:

```
CREATE OR REPLACE FUNCTION merge_customer_scd2()
RETURNS VOID AS $$
DECLARE
    rec RECORD;
    current_rec RECORD;
BEGIN
    FOR rec IN SELECT customer_id, full_name, city, customer_segment FROM source.customers LOOP
        -- Найти активную запись в измерении
        SELECT * INTO current_rec FROM dwh.dim_customer_scd2
        WHERE customer_id = rec.customer_id AND is_current = TRUE;

        IF NOT FOUND THEN
```

```
            -- Нет записи – просто вставляем
            INSERT INTO dwh.dim_customer_scd2 (customer_id, full_name, city, customer_segment, valid_from, valid_to, is_current)
            VALUES (rec.customer_id, rec.full_name, rec.city, rec.customer_segment, CURRENT_DATE, '9999-12-31', TRUE);
        ELSE
            -- Проверяем, изменились ли значимые атрибуты
            IF (current_rec.full_name <> rec.full_name OR
               current_rec.city <> rec.city OR
               current_rec.customer_segment <> rec.customer_segment) THEN

               -- Закрыть старую запись
               UPDATE dwh.dim_customer_scd2
               SET valid_to = CURRENT_DATE - 1, is_current = FALSE
               WHERE surrogate_key = current_rec.surrogate_key;

               -- Вставить новую
               INSERT INTO dwh.dim_customer_scd2 (customer_id, full_name, city, customer_segment, valid_from, valid_to, is_current)
               VALUES (rec.customer_id, rec.full_name, rec.city, rec.customer_segment, CURRENT_DATE, '9999-12-31', TRUE);
            END IF;
        END IF;
    END LOOP;
END;
$$ LANGUAGE plpgsql;
```

4.2. Выполните первичную загрузку:

```
SELECT merge_customer_scd2();
```

5. **Тестирование работы SCD Type 2**

5.1. Внесите изменения в источник:

UPDATE source.customers SET city = 'Казань', last_updated = CURRENT_DATE WHERE customer_id = 1;

INSERT INTO source.customers (customer_id, full_name, city, customer_segment) VALUES (4, 'Ольга Смирнова', 'Новосибирск', 'Gold');

5.2. Снова выполните функцию SELECT merge_customer_scd2();

5.3. Проверьте содержимое измерения:

SELECT surrogate_key, customer_id, full_name, city, customer_segment, valid_from, valid_to, is_current

FROM dwh.dim_customer_scd2 ORDER BY customer_id, valid_from;

5.4. Ожидаемый результат: для customer_id=1 появились две записи – старая (Москва) закрыта, новая (Казань) активна.

6. **Реализация SCD Type 2 на Python (альтернативный вариант)**

6.1. Создайте файл scripts/scd2_python.py:

```
import psycopg2
import logging
from datetime import date

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

def merge_customer_scd2():
    conn = psycopg2.connect("host=localhost dbname=dwh_db user=etl_user password=etl_pass")
    cur = conn.cursor()
```

```
        # Получить все записи из источника
        cur.execute("SELECT customer_id, full_name, city, customer_segment FROM source.customers")
        source_rows = cur.fetchall()

        for row in source_rows:
            customer_id, full_name, city, segment = row

            # Найти активную запись
            cur.execute("""
                SELECT surrogate_key, full_name, city, customer_segment
                FROM dwh.dim_customer_scd2
                WHERE customer_id = %s AND is_current = TRUE
            """, (customer_id,))
            current = cur.fetchone()

            if not current:
                cur.execute("""
                    INSERT INTO dwh.dim_customer_scd2
                    (customer_id, full_name, city, customer_segment, valid_from, valid_to, is_current)
                    VALUES (%s, %s, %s, %s, %s, %s, %s)
                """, (customer_id, full_name, city, segment, date.today(), '9999-12-31', True))
                logger.info(f"Inserted new customer {customer_id}")
            else:
                # Сравнение атрибутов (индекс 1,2,3)
                if (current[1] != full_name or current[2] != city or current[3] != segment):
                    # Закрыть старую
```

```
                cur.execute("""
                    UPDATE dwh.dim_customer_scd2
                    SET valid_to = %s, is_current = FALSE
                    WHERE surrogate_key = %s
                """, (date.today() - timedelta(days=1), current[0]))
                # Вставить новую
                cur.execute("""
                    INSERT INTO dwh.dim_customer_scd2
                    (customer_id, full_name, city, customer_segment, valid_from, valid_to, is_current)
                    VALUES (%s, %s, %s, %s, %s, %s, %s)
                """, (customer_id, full_name, city, segment, date.today(), '9999-12-31', True))
                logger.info(f"Updated customer {customer_id}")
        conn.commit()
        cur.close()
        conn.close()

    if __name__ == "__main__":
        merge_customer_scd2()
```

7. **Аналитические запросы для проверки**

7.1. Получить актуальное состояние:

```
SELECT customer_id, full_name, city, customer_segment FROM dwh.dim_customer_scd2 WHERE is_current = TRUE;
```

7.2. Получить состояние на определённую дату (например, на '2025-01-01'):

```
SELECT customer_id, full_name, city, customer_segment
FROM dwh.dim_customer_scd2
```

```
WHERE valid_from <= '2025-01-01' AND valid_to >= '2025-01-01';
```

7.3. Соединение с таблицей фактов для анализа «на момент»:

```
SELECT f.*, d.full_name, d.city
FROM fact_sales f
JOIN dwh.dim_customer_scd2 d ON f.customer_id = d.customer_id
WHERE f.order_date BETWEEN d.valid_from AND d.valid_to;
```

8. **Добавление логирования и обработки ошибок**

8.1. Создайте таблицу dwh.scd2_load_log:

```
CREATE TABLE dwh.scd2_load_log (
    log_id SERIAL PRIMARY KEY,
    load_timestamp TIMESTAMP DEFAULT NOW(),
    customer_id INTEGER,
    action VARCHAR(10),  -- 'INSERT', 'UPDATE', 'NOCHANGE'
    changed_fields TEXT
);
```

8.2. Расширьте функцию SQL, добавив вставку в лог после каждого изменения.

Пример выполнения задания

Исходные данные источника:

customer_id	full_name	city	customer_segment
1	Иван Петров	Москва	Gold
2	Мария Сидорова	СПб	Standard

После первого запуска SCD2:

surrogate_key	customer_id	city	is_current	valid_from	valid_to
1	1	Москва	TRUE	2024-01-01	9999-12-31
2	2	СПб	TRUE	2024-01-01	9999-12-31

Обновление источника:

UPDATE source.customers SET city = 'Казань', customer_segment = 'Platinum' WHERE customer_id = 1;

После второго запуска SCD2:

surrogate_key	customer_id	city	is_current	valid_from	valid_to
1	1	Москва	FALSE	2024-01-01	2024-06-15
2	2	СПб	TRUE	2024-01-01	9999-12-31
3	1	Казань	TRUE	2024-06-16	9999-12-31

Аналитический запрос «актуальные клиенты» вернёт: customer_id 1 – Казань, customer_id 2 – СПб.

Индивидуальные задания

Каждый вариант описывает предметную область и набор атрибутов, подлежащих отслеживанию по SCD Type 2. Студент должен реализовать схему измерения с историей и процедуру обновления (на SQL или Python). Дополнительно требуется предусмотреть обработку массовой загрузки (несколько записей за раз) и логирование.

Вариант 1. Измерение «Сотрудник»

Атрибуты: employee_id (ключ), full_name, department, position, salary. Изменение department или position должно создавать новую версию. salary также отслеживать (изменение оклада — новая версия).

Вариант 2. Измерение «Продукт»

Атрибуты: product_id, product_name, category, unit_price, supplier_name. Изменение unit_price или supplier_name — новая версия.

Вариант 3. Измерение «Клиент» (финансовый)

Атрибуты: client_id, full_name, passport_number, address, credit_rating. Важно отслеживать историю адресов и кредитного рейтинга (новые версии при изменении).

Вариант 4. Измерение «Тарифный план» (телеком)

Атрибуты: tariff_id, tariff_name, monthly_fee, gb_limit, call_rate. При изменении любого атрибута, кроме имени, создавать новую версию.

Вариант 5. Измерение «Склад»

Атрибуты: warehouse_id, warehouse_name, location_city, manager_name, storage_capacity_sqm. Изменение manager_name или location_city требует новой версии.

Вариант 6. Измерение «Студент» (образование)

Атрибуты: student_id, full_name, group_name, scholarship_amount, admission_year. При изменении group_name или scholarship_amount — новая версия.

Вариант 7. Измерение «Автомобиль» (страхование)

Атрибуты: vehicle_id, license_plate, brand, model, year, insured_value. Каждое изменение insured_value создаёт новую версию.

Вариант 8. Измерение «Поставщик» (ритейл)

Атрибуты: supplier_id, company_name, tax_number, payment_terms_days, bank_account. Изменение payment_terms_days или bank_account — новая версия.

Вариант 9. Измерение «Абонент» (услуги связи)

Атрибуты: subscriber_id, phone_number, tariff_id, balance, contract_status. Отслеживать историю tariff_id и contract_status.

Вариант 10. Измерение «Оборудование» (производство)

Атрибуты: equipment_id, equipment_name, location_shop, maintenance_interval_days, last_maintenance_date. Изменение location_shop или maintenance_interval_days — новая версия.

Вариант 11. Измерение «Торговая точка»

Атрибуты: store_id, store_name, region, city, store_type, manager_id. Новая версия при изменении manager_id или store_type.

Вариант 12. Измерение «Проект» (IT-компания)

Атрибуты: project_id, project_name, project_manager, budget, status. Отслеживать историю project_manager, budget, status.

Вариант 13. Измерение «Счёт» (банк)

Атрибуты: account_id, account_number, account_type, interest_rate, currency. Изменение interest_rate или currency даёт новую версию.

Вариант 14. Измерение «Лекарство» (фармацевтика)

Атрибуты: drug_id, drug_name, manufacturer, price, requires_prescription. Новая версия при изменении price или manufacturer.

Вариант 15. Измерение «Услуга» (клининг)

Атрибуты: service_id, service_name, price_per_hour, cleaning_type, is_active. Отслеживать историю price_per_hour и is_active.

Дополнительное задание для всех вариантов:

Реализовать обработку «мягкого удаления» – когда запись перестаёт существовать в источнике, устанавливать valid_to = текущая дата и is_current = FALSE для последней активной версии, но не удалять физически.

Требования к отчёту

1. Титульный лист с названием работы, ФИО, группой, номером варианта.
2. Цель и задачи (из методички).
3. Теоретическая часть – описание SCD Type 2, сравнение с другими типами, структура таблицы с историей.
4. Ход выполнения работы:
   - SQL-код создания таблиц источника и измерения (или Python-скрипт).
   - Реализация SCD2-логики (функция или Python-функция) с комментариями.
   - Скрипт тестовых обновлений источника (несколько изменений).
5. Результаты:
   - Содержимое измерения после начальной загрузки.
   - Содержимое измерения после каждого обновления источника.
   - Пример аналитического запроса «состояние на дату» с результатом.
   - Лог загрузок (если реализован).
6. Ответы на контрольные вопросы.
7. Вывод – какие преимущества даёт SCD Type 2 для анализа изменений, какие сложности возникли.

Контрольные вопросы

1. Почему SCD Type 2 называют «сохраняющим полную историю» и в каких бизнес-задачах это критически важно?

2. Чем отличается естественный (business key) от суррогатного ключа в измерении с историей?

3. Какие недостатки есть у SCD Type 2 с точки зрения размера таблицы и производительности?

4. Как бы вы реализовали SCD Type 2 для миллиона записей с частыми (ежедневными) изменениями?

5. Зачем нужны поля valid_from и valid_to? Почему для активной записи valid_to устанавливают в '9999-12-31'?

6. Как корректно обработать ситуацию, когда в источнике запись удалена (физически отсутствует)?

7. В чём разница между valid_to = CURRENT_DATE - 1 и valid_to = CURRENT_DATE при закрытии записи?

8. Как с помощью SCD Type 2 можно получить снапшот измерения на произвольную дату за прошлый год?

9. Какие проблемы могут возникнуть при параллельном выполнении двух обновлений одного и того же business_key?

10. Как можно оптимизировать процесс обновления, если в источнике изменились только 5% записей из миллиона?

11. В чём преимущество реализации SCD2 на уровне базы данных (PL/pgSQL) перед реализацией на Python?

12. Что такое «атрибуты-выключатели» (SCD Type 2 с дополнительным флагом) и зачем они нужны?

13. Как в схеме «звезда» таблица фактов связывается с разными версиями измерения? Приведите пример SQL.

14. Можно ли комбинировать SCD Type 2 и SCD Type 1 для разных атрибутов одного измерения?

15. Что такое «SCD Type 6» и для каких задач он был придуман?

Лабораторная работа №7.
Apache Airflow: первый DAG

Цель работы: установить Apache Airflow в Docker, создать и запустить простой DAG (Directed Acyclic Graph), состоящий из двух последовательных задач: первая задача выводит текущую дату, вторая – сохраняет дату в текстовый файл.

Задачи:

1. Развернуть Apache Airflow в Docker-контейнерах с использованием docker-compose (официальный образ).
2. Познакомиться с веб-интерфейсом Airflow (UI): просмотр DAG'ов, запуск, логирование.
3. Создать директорию dags и разместить в ней первый Python-скрипт, реализующий DAG.
4. Определить в DAG'е две задачи (оператора) с использованием BashOperator или PythonOperator.
5. Установить зависимости между задачами: вторая задача должна выполняться только после успешного завершения первой.
6. Запустить DAG вручную через UI или командную строку и проверить выполнение.
7. Проанализировать логи выполнения и убедиться, что файл с датой создан.

Теоретическая часть

Apache Airflow — платформа для программного создания, планирования и мониторинга конвейеров обработки данных (workflow). Широко используется в инжиниринге данных для оркестрации ETL/ELT-пайплайнов.

Основные понятия Airflow:

Понятие	Описание
DAG (Directed Acyclic Graph)	Набор задач с определёнными зависимостями (направленный ациклический граф)
Operator	Кирпичик, выполняющий одну операцию: BashOperator, PythonOperator, PostgresOperator, EmailOperator и др.
Task	Экземпляр оператора в DAG'е
Dependency	Связь между задачами (task1 >> task2 – task2 выполняется после task1)
Scheduler	Планировщик, запускающий DAG'и по расписанию
Web Server	Веб-интерфейс для просмотра и управления DAG'ами
Executor	Механизм выполнения задач (LocalExecutor, CeleryExecutor, KubernetesExecutor)

Зачем Airflow в инжиниринге данных:

- **Оркестрация** – управление последовательностью извлечения, трансформации, загрузки.
- **Планирование** – cron-подобные расписания (schedule_interval).
- **Мониторинг** – визуальное отслеживание успешных и упавших задач.
- **Повторные попытки** – автоматические retry при временных сбоях.
- **Логирование** – доступ к логам каждой задачи через UI.

Архитектура Airflow (Production):

Web Server → Scheduler → Metadata DB (Postgres/MySQL)

↘ Executor → Workers (запускают задачи)

Для разработки достаточно запустить все компоненты в Docker (официальный docker-compose.yaml). Задачи по умолчанию выполняются

внутри того же контейнера (LocalExecutor), что удобно для лабораторных работ.

Простой DAG на Python:

```
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
from datetime import datetime

default_args = {
   'owner': 'data_engineer',
   'start_date': datetime(2025, 1, 1),
   'retries': 1
}

dag = DAG('my_first_dag', default_args=default_args, schedule_interval=None)

task1 = BashOperator(task_id='print_date', bash_command='date', dag=dag)
task2 = BashOperator(task_id='save_date', bash_command='echo {{ ds }} > /tmp/date.txt', dag=dag)

task1 >> task2  # task1 выполняется перед task2
```

Ключевые моменты:

- schedule_interval=None – DAG не запускается автоматически, только вручную.
- {{ ds }} – макрос Airflow (дата выполнения в формате YYYY-MM-DD).
- task1 >> task2 – установка зависимости (синтаксический сахар для set_downstream).

Методика выполнения работы

1. **Подготовка рабочего окружения**

1.1. Создайте каталог lab7_airflow

1.2. Внутри создайте подкаталоги: dags, logs, plugins, config

1.3. Скачайте официальный docker-compose.yaml для Airflow:

curl -LfO 'https://airflow.apache.org/docs/apache-airflow/2.7.1/docker-compose.yaml'

1.4. При необходимости отредактируйте файл:

- В разделе AIRFLOW__CORE__LOAD_EXAMPLES установите false'
- В разделе AIRFLOW__CORE__DAGS_FOLDER должно быть /opt/airflow/dags
- Смонтируйте локальную папку dags в контейнер (это уже есть в стандартном compose)

2. **Запуск Airflow в Docker**

2.1. Выполните команду для инициализации (создания пользователя и настроек):

docker-compose up airflow-init

2.2. Запустите все сервисы:

docker-compose up -d

2.3. Проверьте, что контейнеры работают:

docker ps

Должны быть запущены: airflow-scheduler, airflow-webserver, airflow-worker, postgres, redis.

2.4. Откройте веб-интерфейс: http://localhost:8080 (логин: airflow, пароль: airflow).

3. **Создание первого DAG'а**

3.1. В папке dags создайте файл first_dag.py

3.2. Напишите следующий код:

```
from airflow import DAG
from airflow.operators.bash import BashOperator
from datetime import datetime

default_args = {
    'owner': 'student',
    'depends_on_past': False,
    'start_date': datetime(2025, 1, 1),
    'retries': 1,
    'retry_delay': 5
}

dag = DAG(
    'lab7_print_and_save_date',
    default_args=default_args,
    description='Печать даты и сохранение в файл',
    schedule_interval=None,      # без расписания (ручной запуск)
    catchup=False,
    tags=['lab7']
)

task_print_date = BashOperator(
    task_id='print_current_date',
    bash_command='echo "Current date: $(date +%Y-%m-%d %H:%M:%S)"',
```

```
    dag=dag
)

task_save_date_to_file = BashOperator(
    task_id='save_date_to_tmp',
    bash_command='date +%Y-%m-%d > /tmp/airflow_date.txt && echo "Saved to /tmp/airflow_date.txt"',
    dag=dag
)

# Определяем порядок
task_print_date >> task_save_date_to_file
```

3.3. Сохраните файл. Airflow автоматически подхватит новый DAG в течение 1-2 минут.

4. **Запуск DAG'а через веб-интерфейс**

   4.1. В UI (http://localhost:8080) найдите DAG lab7_print_and_save_date.

   4.2. Переведите переключатель DAG в положение "On".

   4.3. Нажмите на кнопку "Trigger DAG" (стрелка ▶️) для ручного запуска.

   4.4. Перейдите в раздел "Graph View" – увидите две задачи с зависимостью.

   4.5. Нажмите на задачу save_date_to_tmp → "Log" – просмотрите вывод.

5. **Проверка результата**

   5.1. Убедитесь, что файл создан в контейнере:

   docker exec -it lab7_airflow_scheduler_1 cat /tmp/airflow_date.txt

   (имя контейнера может отличаться; используйте docker ps для уточнения)

5.2. При желании смонтируйте дополнительный том, чтобы файл сохранялся на хост.

6. **Модификация DAG для использования PythonOperator**

6.1. Создайте второй DAG – first_dag_python.py с PythonOperator:

```
from airflow import DAG
from airflow.operators.python import PythonOperator
from datetime import datetime
import os

def print_date(**context):
    current_date = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    print(f"Current date: {current_date}")
    return current_date

def save_date_to_file(**context):
    # Получаем возвращённое значение из предыдущего таска через XCom
    date_str = context['task_instance'].xcom_pull(task_ids='print_date_task')
    with open('/tmp/airflow_date_python.txt', 'w') as f:
        f.write(date_str)
    print(f"Saved: {date_str}")

default_args = {'owner': 'student', 'start_date': datetime(2025, 1, 1)}

dag = DAG('lab7_python_operator', default_args=default_args, schedule_interval=None)

task1 = PythonOperator(task_id='print_date_task', python_callable=print_date, dag=dag)
```

task2 = PythonOperator(task_id='save_date_task', python_callable=save_date_to_file, dag=dag)

task1 >> task2

6.2. Запустите и проверьте.

7. **Изучение веб-интерфейса**

7.1. Перейдите в "Grid View" – история запусков.

7.2. Откройте "Code" – исходный код DAG'a.

7.3. Изучите "Task Duration", "Gantt", "Landing Times" для анализа производительности.

Пример выполнения задания

DAG-файл (dags/first_dag.py**) полностью соответствует приведённому выше коду.**

Результат выполнения через веб-интерфейс:

- Статус print_current_date – success (зелёный кружок).
- Статус save_date_to_tmp – success.
- Лог save_date_to_tmp содержит:

[2025-05-12 14:30:15,123] {subprocess.py:75} INFO - Running command: date +%Y-%m-%d > /tmp/airflow_date.txt && echo "Saved to /tmp/airflow_date.txt"

[2025-05-12 14:30:15,456] {subprocess.py:86} INFO - Output:

Saved to /tmp/airflow_date.txt

- В контейнере файл /tmp/airflow_date.txt содержит 2025-05-12.

Индивидуальные задания

Каждый вариант требует модификации DAG'a: добавления новых операторов, использования разных операторов (Bash, Python, файловых), изменения зависимостей и обработки данных.

Вариант 1. Сложение двух чисел

DAG из трёх задач: 1) сгенерировать случайное число (PythonOperator) → сохранить в XCom; 2) сгенерировать второе число; 3) сложить их и вывести результат (BashOperator).

Вариант 2. Обработка списка городов

Задача 1: создать список городов (PythonOperator). Задача 2: для каждого города запустить BashOperator с echo "Hello, {city}" (использовать динамические задачи, Dynamic Task Mapping).

Вариант 3. Копирование файла внутри контейнера

Задача 1: создать текстовый файл /tmp/data.txt со строкой "Airflow test". Задача 2: скопировать его в /tmp/copy_data.txt. Задача 3: проверить содержимое копии (BashOperator).

Вариант 4. Загрузка данных из URL (wget)

Задача 1: скачать CSV-файл с публичного URL (например, https://people.sc.fsu.edu/~jburkardt/data/csv/hw_200.csv) в /tmp/file.csv. Задача 2: вывести количество строк в файле (wc -l). Задача 3: архивировать в /tmp/file.csv.gz.

Вариант 5. Запись в PostgreSQL (опционально – с развёрнутым Postgres)

Задача 1: создать таблицу temp_messages (PostgresOperator). Задача 2: вставить в неё текущую дату. Задача 3: прочитать и вывести количество записей.

Вариант 6. Генерация JSON и парсинг

Задача 1: PythonOperator генерирует JSON-объект {"date": "2025-05-12", "value": 42} и сохраняет в /tmp/data.json. Задача 2: BashOperator выводит значение value через jq. Задача 3: удаляет файл.

Вариант 7. Параллельные задачи (branching)

Задача 1: проверяет день недели (PythonOperator). Если день чётный (пн, ср, пт) → задача A, иначе → задача B. Обе задачи пишут разные сообщения в файл.

Вариант 8. Обработка ошибок с повторными попытками

Настроить DAG с задачей, которая с вероятностью 50% падает (генерирует исключение в PythonOperator). Установить retries=3 и экспоненциальную задержку. Пронаблюдать повторные попытки в логах.

Вариант 9. Использование XCom для обмена данными

Задача 1: получает текущий курс USD/RUB через публичное API (requests). Задача 2: читает курс из XCom и сохраняет его в файл. Задача 3: отправляет уведомление (EmailOperator – можно заглушкой).

Вариант 10. Условная зависимость (BranchPythonOperator)

Если размер файла /tmp/info.txt (создаётся в первой задаче) больше 100 байт, то запустить задачу "обработка", иначе – "пропустить". Использовать BranchPythonOperator.

Вариант 11. Планирование по расписанию

Создать DAG с schedule_interval='5 9 * * *' (ежедневно в 9:05). Задача – логировать "Daily report generated". В отчёте показать, как проверить следующий scheduled запуск.

Вариант 12. Использование DockerOperator (продвинутый)

Задача 1: запускает контейнер с Python, который выполняет скрипт print("Hello from Docker"). Требуется наличие Docker в хосте и настройка docker сети.

Вариант 13. Ожидание внешнего файла (FileSensor)

DAG начинается с FileSensor, который ждёт появления файла /tmp/ready.txt (создаётся вручную или отдельным процессом). После появления – задача "обработка файла".

Вариант 14. Отправка HTTP-запроса (SimpleHttpOperator)

Задача 1: отправляет GET-запрос на публичное API (например, http://api.weatherstack.com/current – тестовый ключ). Задача 2: сохраняет ответ (JSON) в файл. Задача 3: выводит поле temperature.

Вариант 15. Декомпозиция ETL в три задачи

Извлечение: из dags/data/input.csv (файл создать вручную) прочитать данные (PythonOperator). Трансформация: агрегировать по группам (Pandas). Загрузка: сохранить результат в /tmp/output.parquet.

Дополнительное задание для всех вариантов:

Добавить в DAG теги (tags=['lab7', 'variant_N']) и описание (doc_md). Включить в DAG on_failure_callback, который будет логировать ошибку в отдельный файл в папке logs/.

Требования к отчёту

1. Титульный лист с названием работы, ФИО, группой, номером варианта.
2. Цель и задачи (из методички).
3. Теоретическая часть – что такое Airflow, DAG, Operator; архитектура; зачем оркестрация.
4. Ход выполнения работы:
   - Команды запуска Airflow в Docker (скриншот работающих контейнеров).
   - Листинг Python-кода DAG'a (с комментариями).
   - Скриншоты веб-интерфейса: список DAG'ов, граф, лог одной из задач.
5. Результаты:
   - Вывод команды проверки содержимого файла в контейнере.
   - Скриншот успешного выполнения DAG (статус "success").
   - Для индивидуального задания – демонстрация специфической логики (например, вызов API, ветвление).

6. Ответы на контрольные вопросы.
7. Вывод – в чём преимущество использования Airflow для управления пайплайнами, что удалось реализовать.

Контрольные вопросы

1. Что такое DAG и почему он должен быть ациклическим? Что произойдёт, если создать цикл в зависимостях?

2. В чём разница между BashOperator и PythonOperator? Когда предпочтительнее каждый?

3. Зачем нужен schedule_interval=None и как задать ежедневный запуск в 3 часа ночи?

4. Что такое XCom и для каких сценариев он применяется?

5. Как Airflow обрабатывает повторные попытки (retries) и как это настраивается?

6. Где Airflow хранит метаданные о выполненных DAG'ax? Какой БД это может быть?

7. Как посмотреть логи выполнения конкретной задачи через веб-интерфейс?

8. В чём смысл параметра catchup=False? Что произойдёт, если включить catchup=True для DAG'a с start_date год назад?

9. Как остановить DAG, который завис или выполняется слишком долго?

10. Можно ли в одном DAG'e использовать операторы для разных систем (Bash, Python, Postgres, Docker)? Почему?

11. Как передать данные из одной задачи в другую без XCom? Приведите альтернативы.

12. Как настроить уведомление по электронной почте при падении задачи?

13. Как запустить DAG не по расписанию, а только по событию (например, при появлении файла)?

14. В каких сценариях инжиниринга данных Airflow незаменим, а в каких его лучше заменить на что-то другое (например, Kafka Streams)?

15. Как смонтировать локальную папку в контейнер Airflow, чтобы сохранять результаты выполнения задач (например, /tmp/results) на хост-машину?

Лабораторная работа №8.
ETL в Airflow с PostgreSQLOperator

Цель работы: автоматизировать процесс загрузки данных из CSV-файла в staging-таблицу PostgreSQL с помощью DAG в Apache Airflow, используя FileSensor для контроля появления файла и PostgresOperator для выполнения SQL-команд загрузки.

Задачи:

1. Настроить подключение Airflow к PostgreSQL (добавление Connection в веб-интерфейсе).
2. Создать staging-таблицу в PostgreSQL для приёма данных.
3. Разработать DAG, включающий следующие задачи:
   - FileSensor – ожидание появления CSV-файла в заданной директории.
   - create_staging_table (PostgresOperator) – создание staging-таблицы (если не существует).
   - truncate_staging (PostgresOperator) – очистка staging-таблицы перед загрузкой.
   - copy_data (PostgresOperator или PythonOperator с COPY) – загрузка данных из CSV в staging.
   - validate_count (PostgresOperator) – проверка количества загруженных записей.
4. Обработать возможные ошибки (отсутствие файла, ошибки COPY, дублирование данных).
5. Запустить DAG вручную и убедиться в корректной загрузке данных в staging-слой.
6. Настроить логирование и уведомления (опционально).

Теоретическая часть

ETL-пайплайн с Airflow и PostgreSQL

В предыдущих лабораторных работах мы научились контейнеризовать скрипты, работать с PostgreSQL и создавать простые DAG. Теперь объединим эти знания: создадим пайплайн, который автоматически загружает данные из внешнего CSV-файла в staging-таблицу при появлении файла.

Ключевые компоненты DAG:

Компонент	Назначение
FileSensor	Сенсор, который проверяет наличие файла в указанном пути. Выполняется периодически, пока файл не появится или не истечёт таймаут.
PostgresOperator	Выполняет SQL-запрос (или скрипт) в базе данных PostgreSQL.
Connection	Настройки подключения к БД в Airflow (хранятся зашифрованными в метабазе).

Способы загрузки CSV в PostgreSQL через Airflow:

1. **Через SQL** COPY – самый производительный способ. Требует, чтобы файл был доступен на сервере БД или внутри контейнера Airflow. COPY выполняется от имени пользователя PostgreSQL.
2. **Через** \copy **метакоманду psql** – аналогично, но требует клиента psql.
3. **Через PythonOperator и Pandas** – более гибко, но медленнее.
4. **Через** CSVToPostgresOperator – оператор из apache-airflow-providers-postgres (обёртка над COPY).

   В лабораторной работе используем COPY внутри PostgresOperator.

   Пример SQL для COPY:

   ```
   COPY staging.my_table (column1, column2, ...)
   FROM '/path/to/file.csv'
   DELIMITER ','
   CSV HEADER;
   ```

Важно: Файл должен быть доступен по пути внутри контейнера PostgreSQL. Поскольку Airflow и PostgreSQL могут работать в разных

контейнерах, необходимо либо смонтировать общий том, либо скопировать файл в контейнер БД перед COPY (например, с помощью BashOperator и docker cp). В учебных целях удобно запускать PostgreSQL и Airflow в одной docker-compose сети и смонтировать общую директорию для файлов.

FileSensor – сенсор, который проверяет существование файла каждые poke_interval секунд. Параметры:

- filepath – путь к файлу (внутри контейнера Airflow, где работает сенсор).
- poke_interval – интервал проверки (по умолчанию 60 с).
- timeout – максимальное время ожидания.

Типичная структура DAG для загрузки:

Start → FileSensor (ждёт файл) → Create/Truncate staging → COPY data → Validate → End

Методика выполнения работы

1. **Подготовка окружения**

   1.1. Создайте каталог lab8_airflow_etl

   1.2. Внутри создайте подкаталоги: dags, logs, plugins, config, data (для входных CSV), sql

   1.3. Скопируйте в data/ тестовый CSV-файл sales_staging.csv (создайте вручную или с помощью скрипта).

   1.4. Разверните Airflow с PostgreSQL, как в ЛР №7, но добавьте в docker-compose.yaml дополнительный сервис PostgreSQL для хранилища (если его нет) и общий volume для данных.

2. **Настройка подключения к PostgreSQL в Airflow**

   2.1. Откройте веб-интерфейс Airflow (http://localhost:8080)

   2.2. Перейдите в Admin → Connections → Add (+)

   2.3. Заполните:

- Connection Id: postgres_dwh
- Connection Type: Postgres
- Host: postgres (имя сервиса в docker-compose)
- Schema: staging_db
- Login: dwh_user
- Password: dwh_pass
- Port: 5432

2.4. Нажмите Save.

3. **Создание staging-таблицы в PostgreSQL**

3.1. Подключитесь к контейнеру PostgreSQL (например, через docker exec -it) и создайте схему и таблицу:

```
CREATE SCHEMA IF NOT EXISTS staging;
CREATE TABLE IF NOT EXISTS staging.sales (
    transaction_id INTEGER PRIMARY KEY,
    sale_date DATE,
    product_name VARCHAR(100),
    quantity INTEGER,
    unit_price NUMERIC(10,2),
    total_amount NUMERIC(10,2)
);
```

3.2. Либо выполните этот SQL через PostgresOperator в DAG (задача create_table).

4. **Создание DAG с FileSensor и PostgresOperator**

4.1. В папке dags создайте файл etl_staging_dag.py:

```
from airflow import DAG
from airflow.sensors.filesystem import FileSensor
```

```
from airflow.providers.postgres.operators.postgres import PostgresOperator
from datetime import datetime
from airflow.operators.dummy import DummyOperator

default_args = {
    'owner': 'data_engineer',
    'start_date': datetime(2025, 1, 1),
    'retries': 1,
}

with DAG(
    dag_id='etl_load_staging_from_csv',
    default_args=default_args,
    schedule_interval=None,
    catchup=False,
    tags=['staging', 'postgres']
) as dag:

    wait_for_file = FileSensor(
        task_id='wait_for_csv_file',
        filepath='/opt/airflow/data/sales_staging.csv',  # путь внутри контейнера airflow
        poke_interval=10,
        timeout=300,
        mode='poke'
    )

    create_table = PostgresOperator(
        task_id='create_staging_table',
        postgres_conn_id='postgres_dwh',
```

```
        sql="""
        CREATE SCHEMA IF NOT EXISTS staging;
        CREATE TABLE IF NOT EXISTS staging.sales (
            transaction_id INTEGER PRIMARY KEY,
            sale_date DATE,
            product_name VARCHAR(100),
            quantity INTEGER,
            unit_price NUMERIC(10,2),
            total_amount NUMERIC(10,2)
        );
        """
    )

    truncate_table = PostgresOperator(
        task_id='truncate_staging_table',
        postgres_conn_id='postgres_dwh',
        sql="TRUNCATE TABLE staging.sales;"
    )

    copy_data = PostgresOperator(
        task_id='copy_csv_to_staging',
        postgres_conn_id='postgres_dwh',
        sql="""
        COPY staging.sales(transaction_id, sale_date, product_name, quantity,
unit_price, total_amount)
        FROM '/opt/airflow/data/sales_staging.csv'
        DELIMITER ','
        CSV HEADER;
        """
```

 # Внимание: путь к файлу должен быть доступен PostgreSQL, а не Airflow.

 # Если PostgreSQL запущен отдельным контейнером, нужно смонтировать том с данными в оба контейнера.

 # Упрощённо: можно скопировать файл в контейнер PostgreSQL через BashOperator, либо использовать PythonOperator с чтением файла и вставкой.

)

validate_count = PostgresOperator(
task_id='validate_row_count',
postgres_conn_id='postgres_dwh',
sql="""
SELECT COUNT(*) FROM staging.sales;
"""
)

wait_for_file >> create_table >> truncate_table >> copy_data >> validate_count

5. **Решение проблемы доступности файла для PostgreSQL**

5.1. Поскольку COPY выполняется внутри PostgreSQL, файл должен быть виден серверу БД. Один из способов – смонтировать одну и ту же директорию хоста в контейнеры Airflow и PostgreSQL.

5.2. В docker-compose.yaml для сервиса postgres добавьте:

volumes:
- ./data:/opt/airflow/data

Для сервиса airflow-worker и airflow-scheduler также смонтируйте ./data:/opt/airflow/data.

5.3. Альтернативно, используйте BashOperator для копирования файла в /var/lib/postgresql/data внутрь контейнера PostgreSQL (но это менее чисто).

6. **Запуск и тестирование DAG**

   6.1. Убедитесь, что файл sales_staging.csv лежит в ./data/.

   6.2. Через веб-интерфейс Airflow включите DAG etl_load_staging_from_csv и запустите "Trigger DAG".

   6.3. Наблюдайте за выполнением: сенсор должен обнаружить файл, затем задачи создадут таблицу, очистят и загрузят данные.

   6.4. Проверьте результат через psql: SELECT * FROM staging.sales;

7. **Добавление обработки ошибок и логирования**

   7.1. Добавьте задачу check_if_file_empty (PythonOperator), которая проверяет, не пустой ли файл, и завершает DAG с ошибкой, если пустой.

   7.2. Используйте on_failure_callback для отправки уведомления (например, в лог).

   7.3. Логируйте детали загрузки (количество строк, список ошибок) в отдельную таблицу staging.load_log.

Пример выполнения задания

Содержимое data/sales_staging.csv**:**

transaction_id,sale_date,product_name,quantity,unit_price,total_amount
101,2025-05-10,Ноутбук,1,55000,55000
102,2025-05-10,Мышь,2,1500,3000
103,2025-05-11,Клавиатура,1,3500,3500

После выполнения DAG таблица staging.sales **содержит три строки.**

Лог задачи copy_csv_to_staging **(пример):**

[2025-05-12 15:20:01] {postgres_operator.py:100} INFO - Executing SQL: COPY staging.sales(...) FROM '/opt/airflow/data/sales_staging.csv' DELIMITER ',' CSV HEADER;

[2025-05-12 15:20:01] {postgres_operator.py:105} INFO - COPY 3

Проверочный запрос validate_row_count **возвращает** 3**.**

Индивидуальные задания

Каждый вариант определяет структуру CSV-файла, необходимые трансформации при загрузке и дополнительные проверки качества данных.

Вариант 1. Клиенты

CSV: client_id,name,email,city,registration_date.

Загрузка в staging.clients. Дополнительно: привести email к нижнему регистру через SQL-функцию перед вставкой, отбросить записи с пустым city.

Вариант 2. Товары

CSV: product_id,product_name,category,price,stock.

Загрузка в staging.products. Установить CHECK (stock >= 0). Отбраковать строки с отрицательной ценой.

Вариант 3. Заказы

CSV: order_id,customer_id,order_date,total_amount,status.

Добавить задачу update_dim_date – из order_date извлечь год, месяц, день и вставить в измерение dim_date, если его там нет (SCD 0).

Вариант 4. Транзакции банка

CSV: txn_id,account_id,txn_date,amount,currency.

Перед загрузкой сконвертировать валюту в USD по курсу из отдельной таблицы exchange_rates (получить через JOIN в SQL загрузки).

Вариант 5. Логи посещений

CSV: log_id,user_id,page_visited,timestamp,ip_address.

Загрузка в staging.web_logs. Отбрасывать строки с ip_address из чёрного списка (заданного в отдельной таблице).

Вариант 6. Студенты и оценки

CSV: student_id,student_name,subject,score,exam_date.

Загрузить, вычислить процентную шкалу: если score < 50, добавить колонку grade='F', иначе 'P'. Загружать с трансформацией (использовать INSERT INTO ... SELECT с CASE).

Вариант 7. Сотрудники

CSV: emp_id,emp_name,department,salary,hire_date.

Перед загрузкой проверить, что salary не превышает 2* среднюю зарплату по департаменту (среднее вычислять из уже загруженных данных). Отклонённые строки сохранить в staging.rejected_salaries.

Вариант 8. Продажи

CSV: sale_id,date,store_id,product_id,quantity,price.

Загрузить, но добавить вычисляемую колонку total = quantity * price. Удалить строки с quantity <= 0.

Вариант 9. Авиабилеты

CSV: ticket_id,passenger_name,flight_no,departure_date,price.

Проверить, что departure_date не в прошлом (по сравнению с датой загрузки). Отбраковать.

Вариант 10. Медицинские записи

CSV: record_id,patient_id,diagnosis,treatment_cost,visit_date.

После загрузки запустить задачу "аудит": найти записи с treatment_cost > 50000 и записать их в отдельный лог.

Вариант 11. Отзывы

CSV: review_id,product_id,rating,review_text,review_date.

Загрузить, но заменить недопустимые символы в review_text (например, удалить HTML-теги через regexp_replace). Создать полнотекстовый индекс после загрузки.

Вариант 12. Транзакции криптовалют

CSV: txn_hash,from_address,to_address,amount,block_timestamp.

Перед загрузкой проверить, что from_address != to_address. Дубляжи по txn_hash игнорировать (ON CONFLICT DO NOTHING).

Вариант 13. IQ-тесты

CSV: person_id,age,test_date,iq_score,education_level.

Отбросить выбросы: IQ < 50 или > 160. Загрузить только чистые записи, битые сохранить в staging.rejected_iq.

Вариант 14. Подписки

CSV: sub_id,user_id,plan_name,start_date,end_date,price.

Если end_date NULL или пусто, установить как '9999-12-31'. Добавить колонку is_active (TRUE если end_date = '9999-12-31').

Вариант 15. Журнал посещаемости

CSV: employee_id,check_in,check_out,date.

После загрузки вычислить hours_worked = EXTRACT(EPOCH FROM (check_out - check_in))/3600. Добавить ограничение: часы не более 16.

Дополнительное задание для всех вариантов:

Реализовать инкрементальную загрузку: DAG должен запоминать максимальный ID (или дату) из предыдущей загрузки и загружать только новые строки (без повторной загрузки уже загруженных). Использовать Variable в Airflow для хранения состояния.

Требования к отчёту

1. Титульный лист с названием работы, ФИО, группой, номером варианта.
2. Цель и задачи (из методички).
3. Теоретическая часть – описание FileSensor, PostgresOperator, способов загрузки CSV в PostgreSQL.
4. Ход выполнения работы:
   - Содержимое docker-compose.yaml (добавленные volume).
   - Настройка Connection в Airflow (скриншот).

- Листинг DAG-файла с комментариями.
- Команды создания staging-таблицы (если не через DAG).

5. Результаты:
   - Скриншот веб-интерфейса с успешным выполнением всех задач DAG.
   - Скриншот лога задачи copy_data.
   - Вывод SELECT из staging-таблицы после загрузки.
   - Результаты проверки целостности (количество строк, отклонённые записи).
6. Ответы на контрольные вопросы.
7. Вывод – как Airflow помогает автоматизировать загрузку данных, какие проблемы возникли при синхронизации доступа к файлам между контейнерами.

Контрольные вопросы

1. В чём разница между FileSensor и FileWaitSensor? Когда использовать сенсор вместо простой проверки в PythonOperator?

2. Почему COPY в PostgreSQL требует, чтобы файл был виден серверу БД, а не только контейнеру Airflow?

3. Как можно обойти ограничение доступности файла для COPY, не монтируя общие тома?

4. Что произойдёт, если во время выполнения COPY в CSV окажется строка с неверным форматом даты?

5. Как настроить PostgresOperator для выполнения SQL из внешнего файла (.sql), а не строки в коде?

6. Какие преимущества даёт использование truncate перед загрузкой? В каких сценариях лучше использовать merge (UPSERT)?

7. Как в FileSensor задать маску для ожидания файлов с расширением .csv и как отличить готовые файлы от частично загруженных (например, по расширению .tmp)?

8. Как с помощью Airflow реализовать повторную загрузку (retry) при ошибке COPY?

9. Какие метрики можно собирать в процессе загрузки и хранить в отдельной таблице логов?

10. Можно ли использовать PostgresHook напрямую в PythonOperator для загрузки данных более гибко (например, с предобработкой в Pandas)? Приведите пример.

11. Как убедиться, что загруженные данные не содержат дубликатов по ключевому полю?

12. В чём отличие PostgresOperator от SQLExecuteQueryOperator?

13. Как настроить параллельную загрузку нескольких CSV-файлов из одной папки с помощью одного DAG?

14. Как задать в DAG параметр timeout для сенсора и что произойдёт при его истечении?

15. Как с помощью Airflow обрабатывать ошибки, когда CSV-файл присутствует, но имеет неверную кодировку?

Лабораторная работа №9.
Работа с очередью сообщений Kafka

Цель работы: настроить producer и consumer для работы с Apache Kafka: producer генерирует потоковые события (например, данные с датчиков, логи, транзакции), а consumer читает сообщения из Kafka и сохраняет их в ClickHouse для дальнейшего аналитического хранения.

Задачи:

1. Развернуть Apache Kafka (с Zookeeper) и ClickHouse в Docker-контейнерах.
2. Создать Kafka-топик для передачи событий.
3. Реализовать producer на Python (библиотека kafka-python или confluent-kafka), который генерирует сообщения в топик.
4. Реализовать consumer на Python, который читает сообщения из того же топика.
5. Настроить ClickHouse: создать базу данных и таблицу для приёма событий.
6. Модифицировать consumer так, чтобы он вставлял полученные сообщения в ClickHouse (вставка батчами).
7. Запустить producer и consumer параллельно, убедиться в сохранении данных в ClickHouse.
8. Добавить обработку ошибок, логирование и настройки (например, параметры подключения через переменные окружения).

Теоретическая часть

Apache Kafka — распределённая платформа для потоковой передачи сообщений (streaming). Используется для построения пайплайнов реального времени, соединяя источники данных с системами-потребителями.

Ключевые понятия Kafka:

Термин	Описание
Producer	Отправитель сообщений в Kafka
Consumer	Получатель сообщений из Kafka
Topic	Логический канал (очередь), куда публикуются сообщения
Partition	Разбиение топика для параллелизма и масштабирования
Broker	Сервер Kafka (узел кластера)
Offset	Уникальный идентификатор сообщения в партиции

Поточная обработка (streaming):

В отличие от batch, стриминг обрабатывает данные «по мере поступления» с минимальной задержкой. Kafka выступает буфером между producer (источник) и consumer (получатель), обеспечивая устойчивость к сбоям.

ClickHouse — колоночная OLAP-СУБД, оптимизированная для аналитических запросов на больших объёмах данных. Отлично подходит для хранения потоковых событий.

Типичная архитектура:

Producer (Python) → Kafka (topic) → Consumer (Python) → ClickHouse (таблица)

Библиотеки:

- kafka-python – чистая реализация, проста в использовании.
- confluent-kafka – обёртка над librdkafka, быстрее и надёжнее (рекомендуется для production).

Producer (минимальный пример):

from kafka import KafkaProducer

```
import json

producer = KafkaProducer(bootstrap_servers='localhost:9092',
                         value_serializer=lambda v: json.dumps(v).encode('utf-8'))
producer.send('my_topic', {'event': 'test', 'value': 42})
producer.flush()
```

Consumer (минимальный пример):

```
from kafka import KafkaConsumer

consumer = KafkaConsumer('my_topic', bootstrap_servers='localhost:9092',
                         auto_offset_reset='earliest',
                         value_deserializer=lambda x: json.loads(x.decode('utf-8')))
for msg in consumer:
    print(msg.value)
```

ClickHouse таблица для событий:

```
CREATE TABLE events (
    timestamp DateTime DEFAULT now(),
    event_type String,
    value Float64,
    metadata String
) ENGINE = MergeTree()
ORDER BY timestamp;
```

Методика выполнения работы

1. **Подготовка окружения**
   1.1. Создайте каталог lab9_kafka_clickhouse
   1.2. Внутри создайте подкаталоги: producer, consumer, data (для persistence Kafka), sql

1.3. Напишите docker-compose.yaml, запускающий Kafka (с Zookeeper) и ClickHouse:

```
version: '3.8'
services:
  zookeeper:
    image: confluentinc/cp-zookeeper:latest
    container_name: zookeeper
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
    ports:
      - "2181:2181"

  kafka:
    image: confluentinc/cp-kafka:latest
    container_name: kafka
    depends_on:
      - zookeeper
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://localhost:9092
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
    ports:
      - "9092:9092"

  clickhouse:
    image: clickhouse/clickhouse-server:latest
    container_name: clickhouse
    ports:
      - "8123:8123"
```

- "9000:9000"

1.4. Запустите контейнеры: docker-compose up -d

2. **Установка Python-библиотек**

2.1. Создайте requirements.txt со следующим содержимым:

kafka-python

clickhouse-driver

python-dotenv

2.2. Установите в виртуальном окружении: pip install -r requirements.txt

3. **Создание топика в Kafka**

3.1. Подключитесь к контейнеру Kafka:

docker exec -it kafka kafka-topics --create --topic events --bootstrap-server localhost:9092 --partitions 3 --replication-factor 1

3.2. Проверьте список топиков:

docker exec -it kafka kafka-topics --list --bootstrap-server localhost:9092

4. **Настройка ClickHouse**

4.1. Подключитесь к ClickHouse:

docker exec -it clickhouse clickhouse-client

4.2. Создайте базу данных и таблицу для событий:

```
CREATE DATABASE IF NOT EXISTS streaming;
CREATE TABLE streaming.user_events (
    event_id UUID DEFAULT generateUUIDv4(),
    event_timestamp DateTime,
    user_id UInt32,
```

```
    event_type String,
    event_value Float64,
    received_at DateTime DEFAULT now()
) ENGINE = MergeTree()
ORDER BY (event_timestamp, user_id);
```

5. **Реализация producer**

5.1. В папке producer создайте файл producer.py:

```
import os
import time
import json
import random
from datetime import datetime
from kafka import KafkaProducer

BOOTSTRAP_SERVERS = os.getenv('KAFKA_BOOTSTRAP', 'localhost:9092')
TOPIC = 'events'

producer = KafkaProducer(
    bootstrap_servers=BOOTSTRAP_SERVERS,
    value_serializer=lambda v: json.dumps(v, default=str).encode('utf-8')
)

def generate_event():
    return {
        'event_timestamp': datetime.now().isoformat(),
        'user_id': random.randint(1, 100),
        'event_type': random.choice(['click', 'view', 'purchase', 'scroll']),
        'event_value': round(random.uniform(0, 100), 2)
```

```
    }

if __name__ == '__main__':
    print("Producer started. Sending events to", TOPIC)
    try:
        while True:
            event = generate_event()
            future = producer.send(TOPIC, value=event)
            record_metadata = future.get(timeout=5)
            print(f"Sent: {event} | partition={record_metadata.partition} offset={record_metadata.offset}")
            time.sleep(1)  # регулируем частоту
    except KeyboardInterrupt:
        print("Producer stopped.")
    finally:
        producer.flush()
        producer.close()
```

6. **Реализация consumer с сохранением в ClickHouse**

6.1. В папке consumer создайте файл consumer.py:

```
python
import os
import json
from kafka import KafkaConsumer
from clickhouse_driver import Client
import logging

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)
```

```
    BOOTSTRAP_SERVERS = os.getenv('KAFKA_BOOTSTRAP', 'localhost:9092')
    TOPIC = 'events'
    GROUP_ID = 'clickhouse_consumer'

    # ClickHouse connection
    ch_client = Client(host='localhost', port=9000, database='streaming')

    consumer = KafkaConsumer(
        TOPIC,
        bootstrap_servers=BOOTSTRAP_SERVERS,
        group_id=GROUP_ID,
        auto_offset_reset='earliest',
        enable_auto_commit=True,
        value_deserializer=lambda x: json.loads(x.decode('utf-8'))
    )

    batch = []
    BATCH_SIZE = 50

    def insert_batch(batch):
        if not batch:
            return
        # преобразуем в список кортежей для вставки
        data = [(ev['event_timestamp'], ev['user_id'], ev['event_type'], ev['event_value'])
                for ev in batch]
        ch_client.execute(
            'INSERT INTO user_events (event_timestamp, user_id, event_type, event_value) VALUES',
```

```
            data
        )
        logger.info(f"Inserted {len(data)} rows into ClickHouse")

    if __name__ == '__main__':
        logger.info("Consumer started, listening to topic " + TOPIC)
        try:
            for msg in consumer:
                event = msg.value
                batch.append(event)
                if len(batch) >= BATCH_SIZE:
                    insert_batch(batch)
                    batch.clear()
        except KeyboardInterrupt:
            logger.info("Shutting down consumer...")
            if batch:
                insert_batch(batch)
            consumer.close()
            ch_client.disconnect()
```

6.2. Проверьте, что таблица в ClickHouse заполняется.

7. **Запуск и тестирование**

7.1. Откройте два терминала: в первом запустите producer, во втором – consumer.

```
bash
# Terminal 1
cd lab9_kafka_clickhouse/producer
python producer.py
bash
# Terminal 2
```

cd lab9_kafka_clickhouse/consumer

python consumer.py

7.2. Через несколько секунд проверьте данные в ClickHouse:

SELECT count() FROM streaming.user_events;

SELECT * FROM streaming.user_events LIMIT 10;

8. **Дополнительная настройка (опционально)**

8.1. Добавьте переменные окружения через .env файл.

8.2. Настройте consumer на чтение с определённой партиции и указанного смещения.

8.3. Реализуйте обработку ошибок (например, при недоступности ClickHouse — сохранить в резервный файл).

Пример выполнения задания

Producer отправляет события:

Sent: {'event_timestamp': '2025-05-12T16:20:15.123456', 'user_id': 42, 'event_type': 'click', 'event_value': 7.5} | partition=1 offset=0

Sent: {'event_timestamp': '2025-05-12T16:20:16.234567', 'user_id': 13, 'event_type': 'view', 'event_value': 0.0} | partition=0 offset=1

Consumer выводит лог:

2025-05-12 16:20:45,123 - INFO - Consumer started, listening to topic events

2025-05-12 16:20:50,456 - INFO - Inserted 50 rows into ClickHouse

Результат в ClickHouse:

SELECT count() FROM streaming.user_events;

Индивидуальные задания

Каждый вариант требует изменения типа генерируемых событий, структуры таблицы ClickHouse и, возможно, логики обработки.

Вариант 1. Логи сервера

Producer генерирует логи в формате: timestamp, level (INFO/WARN/ERROR), service_name, message. Consumer сохраняет в ClickHouse, добавляет поле severity_score (INFO=1, WARN=2, ERROR=3).

Вариант 2. Датчики температуры

Producer: каждый пакет содержит sensor_id, temperature_celsius, humidity, timestamp. Consumer вычисляет среднюю температуру за последние 10 сообщений и отправляет в отдельную таблицу (агрегат).

Вариант 3. Транзакции пользователей

Producer: user_id, transaction_id, amount, currency, timestamp. Consumer конвертирует сумму в USD (фиксированный курс) перед записью в ClickHouse.

Вариант 4. Клики по веб-страницам

Producer: session_id, page_url, element_id, timestamp. Consumer группирует клики по сессии и каждые 100 событий записывает агрегированную статистику в ClickHouse (число кликов на страницу).

Вариант 5. Финансовые котировки

Producer: ticker (AAPL, GOOGL и т.д.), price, volume, timestamp. Consumer проверяет аномалии: если изменение цены за 5 секунд > 2%, отправляет уведомление (в stdout) и сохраняет событие в таблицу alerts.

Вариант 6. Данные смарт-часов

Producer: user_id, heart_rate, steps, calories, timestamp. Consumer сохраняет в ClickHouse, а также вычисляет скользящее среднее пульса за последние 10 записей (используя дедубликацию по user_id и времени).

Вариант 7. Отслеживание заказов в доставке

Producer: order_id, status (created, packed, shipped, delivered), location, timestamp. Consumer создаёт таблицу фактов доставок, обновляя последний статус заказа в другой таблице (CDC-подход).

Вариант 8. Игровые события

Producer: player_id, event_type (kill, death, item_collected), weapon_id, score, timestamp. Consumer агрегирует статистику по игрокам в материализованное представление ClickHouse (убийства, смерти, отношение K/D).

Вариант 9. Поток геолокации такси

Producer: driver_id, lat, lon, speed, timestamp. Consumer находит «подозрительные» зоны (скопление машин со скоростью 0) и сохраняет их в отдельную таблицу idle_zones.

Вариант 10. Социальные лайки

Producer: user_id, post_id, action (like, dislike), timestamp. Consumer ведёт счётчик лайков на пост (использует таблицу-словарь) и раз в минуту записывает итог в ClickHouse.

Вариант 11. Данные метеостанций

Producer: station_id, wind_speed, pressure, temperature, timestamp. Consumer отбрасывает выбросы (wind_speed > 200 км/ч) и вставляет валидные данные в ClickHouse.

Вариант 12. Лог действий администратора

Producer: admin_id, action_type (login, export, delete_user), affected_resource, timestamp. Consumer в реальном времени проверяет «опасные» действия (delete_user) и пишет их в отдельную таблицу audit_alerts.

Вариант 13. Поток курсов валют

Producer: currency_pair (EURUSD, GBPUSD), bid, ask, timestamp. Consumer вычисляет спред (ask - bid) и сохраняет пары со спредом > определённого порога в таблицу high_spread.

Вариант 14. Данные с производственной линии

Producer: machine_id, temperature, vibration, power_consumption, timestamp. Consumer использует скользящее окно для обнаружения предаварийного состояния (значения > медианы + 3*IQR) и отправляет в Kafka другой топик alerts.

Вариант 15. Поток новостных заголовков

Producer: source, headline, keywords (список), published_at, timestamp. Consumer извлекает упоминания заданных ключевых слов (например, "Airflow", "Kafka") и сохраняет совпадения в ClickHouse, игнорируя остальные.

Дополнительное задание для всех вариантов:

Реализовать graceful shutdown: при получении SIGINT (Ctrl+C) consumer должен дообработать оставшиеся сообщения и корректно закрыть соединения. Также добавить мониторинг lag'a consumer с помощью kafka-consumer-groups CLI.

Требования к отчёту

1. Титульный лист с названием работы, ФИО, группой, номером варианта.
2. Цель и задачи (из методички).
3. Теоретическая часть – основы Kafka (топики, продюсеры, консьюмеры), архитектура stream processing, ClickHouse как аналитическое хранилище.
4. Ход выполнения работы:
   - Содержимое docker-compose.yaml (скриншот или листинг).
   - Команды создания топика и таблицы ClickHouse.
   - Листинг producer.py с пояснениями.
   - Листинг consumer.py с пояснениями.
5. Результаты:
   - Скриншот работающего producer (вывод в консоль).
   - Скриншот consumer (логи вставки в ClickHouse).
   - SQL-запрос к ClickHouse с результатом (количество записей, пример данных).
   - Вывод команды проверки lag (если реализовано).
6. Ответы на контрольные вопросы.

7. Вывод – преимущества использования Kafka для асинхронной обработки и интеграции с ClickHouse.

Контрольные вопросы

1. Чем отличается Kafka от традиционных очередей сообщений (RabbitMQ, ActiveMQ)?

2. Зачем в Kafka используется Zookeeper (или KRaft в новых версиях) и что он хранит?

3. Что такое партиции топика и как они влияют на параллельную обработку?

4. Как consumer группирует сообщения и почему это важно для сохранения порядка?

5. Какие стратегии назначения смещений (offset) существуют у consumer? Чем отличается earliest от latest?

6. Почему в примере мы используем батчирование при вставке в ClickHouse?

7. Как гарантировать, что сообщение из Kafka будет обработано consumer'ом ровно один раз (exactly-once semantics)?

8. Как масштабировать consumer'ов, если сообщений становится слишком много?

9. Что такое consumer group rebalance и как оно влияет на обработку?

10. Как producer может подтвердить, что сообщение записано в Kafka (acknowledgement)?

11. Какие типы сериализации сообщений обычно используются в Kafka (JSON, Avro, Protobuf)? В чём преимущества Avro?

12. Как запустить Kafka без Zookeeper (KRaft mode)? В чём отличия?

13. Как изменить количество партиций уже существующего топика?

14. Как с помощью kafka-console-consumer быстро проверить, что данные попадают в топик?

15. Какие альтернативы Kafka для потоковой обработки данных существуют (Apache Pulsar, Redpanda, RabbitMQ Streams) и в чём их особенности?

Лабораторная работа №10.
Great Expectations для качества данных

Цель работы: настроить проверки качества данных с использованием библиотеки Great Expectations: определить набор ожиданий (expectations) для проверки уникальности ключевых полей, отсутствия null-значений и соответствия типов, выполнить проверку на тестовом датасете и сгенерировать HTML-отчёт.

Задачи:

1. Установить библиотеку Great Expectations (GE) и инициализировать проект.
2. Подключиться к источнику данных (CSV-файл, Pandas DataFrame или база данных).
3. Создать Expectation Suite (набор ожиданий), включающий как минимум:
   - проверку на отсутствие null в заданных колонках
   - проверку на уникальность значений в колонке или комбинации колонок
   - проверку допустимых диапазонов значений (для числовых полей)
   - проверку формата (например, email, дата)
4. Запустить проверку (Validation) на датасете с «дефектами» (пропуски, дубликаты).
5. Сгенерировать HTML-отчёт (Data Docs) и проанализировать результаты.
6. Настроить автоматическую генерацию отчёта после каждого запуска проверки.

Теоретическая часть

Great Expectations (GE) — библиотека с открытым исходным кодом для валидации, документирования и профилирования данных. В инжиниринге данных GE используется для обеспечения качества данных на всех этапах ETL/ELT-пайплайна.

Ключевые понятия:

Понятие	Описание
Expectation	Утверждение о данных (например, «колонка не содержит null», «значения уникальны»).
Expectation Suite	Набор ожиданий, объединённый для конкретного набора данных.
Validator	Объект, который проверяет данные на соответствие Expectation Suite.
Checkpoint	Конфигурация для выполнения проверки (набор данных + suite + действия после проверки).
Data Docs	HTML-отчёт с результатами валидации.

Типовые ожидания (Expectations):

- expect_column_values_to_not_be_null – проверка на отсутствие null.
- expect_column_values_to_be_unique – уникальность значений в колонке.
- expect_column_values_to_be_between – значения в заданном диапазоне.
- expect_column_values_to_match_regex – соответствие регулярному выражению.
- expect_column_pair_values_to_be_equal – равенство двух колонок.
- expect_table_row_count_to_be_between – количество строк в таблице.

Архитектура работы GE:

Data Source (CSV/SQL/Pandas) → Datasource → Data Asset → Batch

↓

Expectation Suite ← (профилирование или ручное создание)

↓

Validator (Batch + Suite) → Validation Result → Data Docs (HTML)

Интеграция в ETL:

Проверки качества могут быть встроены в пайплайн после загрузки в staging, перед трансформацией или после загрузки в gold-слой. При провале проверки пайплайн может остановиться (raise_exception = true) или продолжить, но записать факт ошибки в лог.

Методика выполнения работы

1. **Подготовка рабочего окружения**

   1.1. Создайте каталог lab10_great_expectations

   1.2. Внутри создайте подкаталог data для входных файлов

   1.3. Создайте виртуальное окружение и установите зависимости:

   ```
   python -m venv venv
   source venv/bin/activate  # Windows: venv\Scripts\activate
   pip install great_expectations pandas
   ```

   1.4. Инициализируйте проект Great Expectations:

   great_expectations init

   (При выполнении эта команда создаст структуру каталогов: expectations/, checkpoints/, data_docs/, great_expectations.yml)

2. **Подготовка тестового датасета**

   2.1. Создайте в каталоге data/ файл customers_dirty.csv со следующим содержимым:

   ```
   customer_id,name,email,age,city
   1,Иван Петров,ivan@example.com,35,Москва
   2,Мария Сидорова,maria@example.com,,СПб
   3,Иван Петров,ivan@example.com,35,Москва
   4,,petr@example.com,25,Казань
   5,Анна Смирнова,anna@example.com,-5,Москва
   6,Ольга Кузнецова,olga,120,Новосибирск
   ```

7,Сергей Иванов,sergey@example.com,42,

(Присутствуют: дубликат строк 1 и 3, пропуски в email и city, отрицательный возраст, слишком большой возраст, пропущенное имя)

3. **Создание Datasource и Data Asset через CLI**

   3.1. Запустите интерактивный режим создания Datasource:

   great_expectations datasource new

   Выберите тип Files based (CSV, Excel, etc.) → укажите путь к папке data

   3.2. Создайте Expectation Suite через CLI:

   great_expectations suite new

   Выберите интерактивный режим, укажите имя suite: customers_validation_suite

4. **Добавление ожиданий через Python-скрипт**

   4.1. Создайте файл create_expectations.py в корне проекта:

```
import great_expectations as gx
from great_expectations.core.expectation_suite import ExpectationSuite
from great_expectations.core import ExpectationConfiguration

context = gx.get_context()

# Загружаем датасет как PandasDataSource
datasource = context.sources.add_pandas('customers_datasource')
data_asset = datasource.add_csv_asset('customers_asset', filepath_or_buffer='data/customers_dirty.csv')

batch = data_asset.get_batch()
batch.columns
```

```
# Создаём Expectation Suite
suite = context.add_expectation_suite('customers_validation_suite')

# 1. customer_id должен быть уникальным
suite.add_expectation(
    ExpectationConfiguration(
        expectation_type='expect_column_values_to_be_unique',
        kwargs={'column': 'customer_id'}
    )
)

# 2. В колонке name не должно быть null
suite.add_expectation(
    ExpectationConfiguration(
        expectation_type='expect_column_values_to_not_be_null',
        kwargs={'column': 'name'}
    )
)

# 3. email должен соответствовать регулярному выражению для email
suite.add_expectation(
    ExpectationConfiguration(
        expectation_type='expect_column_values_to_match_regex',
        kwargs={'column': 'email', 'regex': r'^[^@]+@[^@]+\.[^@]+$'}
    )
)

# 4. age должен быть в диапазоне 0–100
suite.add_expectation(
```

```
    ExpectationConfiguration(
        expectation_type='expect_column_values_to_be_between',
        kwargs={'column': 'age', 'min_value': 0, 'max_value': 100}
    )
)

# 5. city не должен быть null
suite.add_expectation(
    ExpectationConfiguration(
        expectation_type='expect_column_values_to_not_be_null',
        kwargs={'column': 'city'}
    )
)

# 6. Количество строк таблицы должно быть от 5 до 10
suite.add_expectation(
    ExpectationConfiguration(
        expectation_type='expect_table_row_count_to_be_between',
        kwargs={'min_value': 5, 'max_value': 10}
    )
)

print("Expectation suite created:", suite.name)
```

4.2. Запустите скрипт: python create_expectations.py

5. **Запуск валидации (Checkpoint)**

5.1. Создайте Checkpoint с помощью CLI:

great_expectations checkpoint new customers_checkpoint

Отредактируйте автоматически созданный файл checkpoints/customers_checkpoint.yml:

```
name: customers_checkpoint
config_version: 1.0
class_name: SimpleCheckpoint
run_name_template: "%Y%m%d-%H%M%S-customers-validation"
validations:
  - batch_request:
      datasource_name: customers_datasource
      data_asset_name: customers_asset
    expectation_suite_name: customers_validation_suite
action_list:
  - name: store_validation_result
    action:
      class_name: StoreValidationResultAction
  - name: store_evaluation_params
    action:
      class_name: StoreEvaluationParametersAction
  - name: update_data_docs
    action:
      class_name: UpdateDataDocsAction
```

5.2. Запустите Checkpoint:

```
great_expectations checkpoint run customers_checkpoint
```

6. **Просмотр HTML-отчёта**

6.1. Откройте сгенерированный отчёт в браузере:

```
great_expectations docs build
```

По умолчанию отчёт доступен по адресу: file://path/to/great_expectations/uncommitted/data_docs/local_site/index.html

6.2. Изучите детали: какие ожидания провалились, на каких строках.

7. **Исправление данных и повторная проверка (опционально)**

   7.1. Создайте копию customers_dirty.csv как customers_clean.csv и исправьте нарушения:

   – Удалите дубликат строки 3.
   – Добавьте имя для customer_id 4.
   – Замените email "olga" на "olga@example.com".
   – Замените age -5 на 25, age 120 на 35.
   – Добавьте city для customer_id 7.

   7.2. Создайте новый Data Asset customers_clean_asset, указывающий на новый файл.

   7.3. Запустите проверку с тем же Expectation Suite и убедитесь, что все ожидания выполнены.

Пример выполнения задания

Фрагмент результата валидации (в консоли):

Validation completed successfully.

Checkpoint: customers_checkpoint

Expectation Suite: customers_validation_suite

- expect_column_values_to_be_unique (customer_id): ❌ Failed (1 unexpected)

- expect_column_values_to_not_be_null (name): ❌ Failed (1 unexpected)

- expect_column_values_to_match_regex (email): ❌ Failed (2 unexpected)

- expect_column_values_to_be_between (age): ❌ Failed (2 unexpected)

- expect_column_values_to_not_be_null (city): ❌ Failed (1 unexpected)

- expect_table_row_count_to_be_between: ✓ Passed

HTML-отчёт (Data Docs) содержит:

- Сводку: процент успешных/проваленных ожиданий.
- Детали по каждому ожиданию: список "неожиданных" значений.
- Timeline и метаданные запуска.

После исправления данных – все проверки зелёные.

Индивидуальные задания

Каждый вариант определяет набор требований к качеству данных (expectations) для конкретного датасета. Студент должен самостоятельно создать Expectation Suite, применить его к «грязному» датасету, сгенерировать отчёт, а затем предложить способ очистки данных для прохождения проверок.

Вариант 1. Транзакции интернет-магазина

Датасет: transactions.csv (transaction_id, user_id, amount, date, status). Ожидания: transaction_id уникален, нет null в user_id и amount, amount > 0, status in ('pending','completed','cancelled'), date в формате YYYY-MM-DD.

Вариант 2. Студенты и оценки

Датасет: grades.csv (student_id, full_name, subject, score, exam_date). Ожидания: комбинация (student_id, subject) уникальна, score между 0 и 100, exam_date не в будущем, full_name не пустое.

Вариант 3. Продукты на складе

Датасет: inventory.csv (product_code, product_name, quantity, price, last_restock). Ожидания: product_code уникален, quantity ≥ 0, price > 0, last_restock не null и не позже текущей даты.

Вариант 4. Клиенты банка

Датасет: clients.csv (client_id, passport, full_name, birth_date, credit_score). Ожидания: passport уникален, birth_date между 1900 и текущим годом, credit_score между 300 и 850, full_name не содержит цифр.

Вариант 5. Логи доступа

Датасет: access_logs.csv (log_id, user_ip, endpoint, response_time, timestamp). Ожидания: log_id уникален, user_ip соответствует формату IPv4, response_time ≥ 0, timestamp не null, endpoint не пуст.

Вариант 6. Заказы доставки

Датасет: deliveries.csv (delivery_id, order_id, courier_name, delivery_date, delivery_status). Ожидания: delivery_id уникален, order_id не null, delivery_date не в прошлом (если статус не delivered) или не в будущем (если delivered), delivery_status in ('pending','in_transit','delivered').

Вариант 7. Отзывы пользователей

Датасет: reviews.csv (review_id, user_id, product_id, rating, review_text, date). Ожидания: review_id уникален, rating от 1 до 5, review_text не пустое (если rating < 3, текст обязателен), date не null.

Вариант 8. Движение денег по счетам

Датасет: ledger.csv (entry_id, account_id, amount, currency, entry_date). Ожидания: entry_id уникален, amount ≠ 0, currency in ('USD','EUR','RUB'), entry_date между 2020-01-01 и сегодня.

Вариант 9. Сотрудники компании

Датасет: employees.csv (emp_id, email, phone, hire_date, department). Ожидания: emp_id уникален, email соответствует формату, phone содержит 11 цифр (для России), hire_date не null, department in list (предзадан).

Вариант 10. Книги в библиотеке

Датасет: books.csv (isbn, title, author, year, pages). Ожидания: isbn уникален (13 цифр или 10), year между 1500 и текущим, pages > 0, title и author не пустые.

Вариант 11. Прогноз погоды

Датасет: weather.csv (station_id, date, temperature_c, humidity, wind_speed). Ожидания: комбинация (station_id, date) уникальна, temperature_c между -50 и +50, humidity между 0 и 100, wind_speed >= 0.

Вариант 12. Транзакции криптобиржи

Датасет: crypto_trades.csv (trade_id, user_id, symbol, quantity, price, timestamp). Ожидания: trade_id уникален, quantity > 0, price > 0, timestamp не старше 5 минут от времени проверки (для актуальности).

Вариант 13. Телефонные звонки

Датасет: calls.csv (call_id, caller_number, callee_number, duration_sec, start_time). Ожидания: call_id уникален, caller_number и callee_number различны, duration_sec >= 0, start_time не null и не в будущем.

Вариант 14. Данные медицинских исследований

Датасет: clinical_trials.csv (trial_id, patient_id, test_name, result_value, normal_range). Ожидания: trial_id уникален, результат не выходит за пределы normal_range (задаётся регуляркой, например "10-20"), patient_id не null.

Вариант 15. Посты в блоге

Датасет: blog_posts.csv (post_id, url_slug, title, published_at, author_email). Ожидания: post_id и url_slug уникальны, url_slug содержит только латиницу и дефисы, published_at не null (если статус опубликован), author_email валиден.

Дополнительное задание для всех вариантов:

Создать второй Expectation Suite, который содержит только пороговые ожидания (например, 95% успеха). Настроить Checkpoint так, чтобы при провале критических ожиданий (например, уникальности первичного ключа) пайплайн выбрасывал исключение, а при провале некритичных – только логировал и продолжал выполнение. Реализовать это через параметры "catch_exceptions": False и "result_format": "BOOLEAN_ONLY".

Требования к отчёту

1. Титульный лист с названием работы, ФИО, группой, номером варианта.
2. Цель и задачи (из методички).
3. Теоретическая часть – назначение Great Expectations, структура ожиданий, концепции Data Docs.
4. Ход выполнения работы:

- Команды инициализации GE и создания Datasource.
- Листинг скрипта создания Expectation Suite (или CLI-команды).
- Описание добавленных ожиданий (список с обоснованием).
- Команда запуска Checkpoint (или Python-код).

5. Результаты:
   - Скриншот HTML-отчёта (Data Docs) с отображением проваленных ожиданий.
   - Таблица с перечислением «неожиданных» значений для каждого проваленного ожидания.
   - Демонстрация исправленного датасета (если выполняли), который проходит проверки.
6. Ответы на контрольные вопросы.
7. Вывод – как Great Expectations помогает контролировать качество данных, какие типовые проблемы были выявлены, как интеграция GE в ETL-пайплайн повышает надёжность.

Контрольные вопросы

1. Что такое Expectation Suite и чем он отличается от Checkpoint?

2. Как GE обрабатывает большие объёмы данных (несколько гигабайт)? Есть ли ограничения?

3. Какие типы Datasource поддерживаются в GE (Pandas, SQLAlchemy, Spark)?

4. Как можно автоматически сгенерировать Expectation Suite на основе существующих данных (профилирование)?

5. Что означает параметр result_format в ExpectationConfiguration? Какие значения он принимает?

6. Как настроить GE для работы с базой данных PostgreSQL, а не с CSV-файлами?

7. В чём разница между expect_column_values_to_be_unique и expect_compound_columns_to_be_unique?

8. Как задать ожидание, что две колонки имеют одинаковое количество значений или одинаковое распределение?

9. Как интегрировать GE в пайплайн Apache Airflow (использовать оператор GreatExpectationsOperator)?

10. Что такое Data Docs и как их можно автоматически публиковать на веб-сервер (например, S3)?

11. Как пропустить проверку на определённых строках (например, исключить строки с пометкой "test")?

12. Как GE помогает при миграции данных или изменении схемы?

13. Можно ли использовать GE для проверки качества данных на этапе производства (production) без остановки пайплайна?

14. Каковы альтернативы Great Expectations для валидации данных (pandas-profiling, Deequ, dbt tests)?

15. Как в GE реализовать пользовательское ожидание (custom expectation) на Python?

Лабораторная работа №11.
DBT (Data Build Tool)

Цель работы: выполнить трансформацию данных в стиле ELT (Extract-Load-Transform) с использованием dbt (data build tool) и SQL: определить dbt-модели, настроить тесты качества данных, сгенерировать документацию и выполнить запуск трансформаций.

Задачи:

1. Установить dbt и настроить подключение к целевой базе данных (PostgreSQL или ClickHouse).
2. Создать новый dbt-проект с правильной структурой папок.
3. На основе существующих сырых таблиц (staging-слой) написать dbt-модели (SQL-файлы) для трансформации:
   - очистка и приведение типов
   - создание таблиц измерений (dimensions)
   - создание таблицы фактов (fact table) по схеме «звезда»
4. Добавить тесты для моделей (unique, not_null, accepted_values, relationships).
5. Сгенерировать документацию проекта (dbt docs generate) и просмотреть её в веб-интерфейсе.
6. Выполнить dbt run, проверить корректность созданных объектов в БД.
7. Использовать макросы (macros) для повторяющихся логик (например, вычисление возраста, форматирование дат).

Теоретическая часть

DBT (data build tool) — инструмент для трансформации данных внутри хранилища, реализующий подход ELT. В отличие от классического ETL, где трансформация происходит во внешнем движке (Python, Spark), в ELT данные сначала загружаются в хранилище (например, в staging), а затем преобразуются SQL-запросами непосредственно в БД с помощью dbt.

Основные компоненты dbt:

Компонент	Описание
Модель (Model)	SQL-файл с SELECT-запросом, который создаёт таблицу или представление.
Тест (Test)	Проверка качества данных (unique, not_null, accepted_values, relationships и пользовательские).
Макрос (Macro)	Многоразовый SQL-код с использованием Jinja.
Документация	Описание моделей и колонок в YAML-файлах, генерирует HTML-сайт.
Слои (Staging, Intermediate, Marts)	Рекомендуемая структура папок для организации трансформаций.

Типичная структура dbt-проекта:

```
my_dbt_project/
├── models/
│   ├── staging/          # сырые данные (минимальная очистка)
│   ├── intermediate/     # промежуточные агрегаты
│   └── marts/            # готовые витрины (факты и измерения)
├── tests/                # пользовательские тесты
├── macros/               # макросы
├── seeds/                # CSV-файлы для загрузки справочников
├── dbt_project.yml       # конфигурация проекта
└── profiles.yml          # подключения к БД
```

Ключевые команды dbt:

Команда	Назначение
dbt init	Создание нового проекта
dbt run	Выполнение всех моделей

Команда	Назначение
dbt test	Запуск тестов
dbt docs generate	Генерация документации
dbt docs serve	Запуск локального веб-сервера с документацией
dbt build	Комбинация run + test

Преимущества dbt:

- Декларативность (модели — это SQL, а не императивный код)
- Управление зависимостями (автоматическое разрешение порядка выполнения через ref())
- Тестирование и документация «из коробки»
- Интеграция с Airflow и другими оркестраторами.

Методика выполнения работы

1. **Подготовка окружения**

   1.1. Создайте каталог lab11_dbt

   1.2. Установите dbt для PostgreSQL (или другой адаптер):

   pip install dbt-postgres

   1.3. Запустите PostgreSQL в Docker (как в ЛР №2) и создайте базу dwh_db, схему raw (для исходных таблиц) и dbt_transformed (куда dbt будет писать результат).

   1.4. Создайте исходные таблицы в схеме raw с тестовыми данными:

```
CREATE SCHEMA raw;
CREATE TABLE raw.clients (
    client_id INTEGER,
    name VARCHAR(100),
    email VARCHAR(100),
    birth_date DATE,
```

```
    city VARCHAR(50),
    segment VARCHAR(20)
);

INSERT INTO raw.clients VALUES
(1, 'Иван Петров', 'ivan@example.com', '1985-06-15', 'Москва', 'Gold'),
(2, 'Мария Сидорова', 'maria@example.com', '1990-11-20', 'СПб', 'Silver'),
(3, 'Алексей Иванов', 'alex@example.com', '2000-01-05', 'Москва', 'Standard'),
(4, NULL, 'olga@example.com', '1995-03-10', 'Казань', 'Gold'),
(5, 'Петр Смирнов', 'petr@example.com', '1988-07-22', NULL, 'Silver');

CREATE TABLE raw.orders (
    order_id INTEGER,
    client_id INTEGER,
    order_date DATE,
    amount DECIMAL(10,2),
    status VARCHAR(20)
);

INSERT INTO raw.orders VALUES
(101, 1, '2025-01-15', 5500, 'completed'),
(102, 2, '2025-01-20', 3200, 'completed'),
(103, 1, '2025-02-10', 7800, 'completed'),
(104, 3, '2025-02-15', 1200, 'cancelled'),
(105, 4, '2025-03-01', 4200, 'completed'),
(106, 5, '2025-03-05', 3000, 'pending');
```

2. **Инициализация dbt-проекта**

2.1. В каталоге lab11_dbt выполните:

dbt init my_retail_project

При запросе выберите адаптер postgres, введите параметры подключения (хост localhost, порт 5432, пользователь dwh_user, пароль dwh_pass, база dwh_db, схема dbt_transformed).

2.2. Перейдите в папку проекта: cd my_retail_project

3. **Настройка профиля и источников**

3.1. Отредактируйте profiles.yml (обычно в ~/.dbt/ или создайте в корне проекта). Убедитесь, что схема назначения указана правильно.

3.2. Создайте файл models/staging/sources.yml для описания источников (сырых таблиц):

```
yaml
version: 2
sources:
  - name: raw
    schema: raw
    tables:
      - name: clients
      - name: orders
```

4. **Создание staging-моделей**

4.1. В папке models/staging/ создайте stg_clients.sql:

```
select
    client_id,
    coalesce(name, 'Unknown') as client_name,
    lower(trim(email)) as email,
    birth_date,
```

```
    coalesce(city, 'Unknown') as city,
    segment
from {{ source('raw', 'clients') }}
where client_id is not null
```

4.2. Создайте stg_orders.sql:

```
select
    order_id,
    client_id,
    order_date,
    amount,
    status
from {{ source('raw', 'orders') }}
where status != 'cancelled'   -- исключаем отменённые заказы
```

5. **Создание промежуточной модели (intermediate)**

5.1. В папке models/intermediate/ создайте int_clients_orders.sql:

```
select
    c.client_id,
    c.client_name,
    c.city,
    c.segment,
    o.order_id,
    o.order_date,
    o.amount
from {{ ref('stg_clients') }} c
inner join {{ ref('stg_orders') }} o on c.client_id = o.client_id
```

6. **Создание витрины (mart) – таблица фактов и измерения**

6.1. В папке models/marts/ создайте dim_client.sql (измерение клиента):

```
select distinct
    client_id,
    client_name,
    city,
    segment,
    case
        when extract(year from age(current_date, birth_date)) < 30 then 'Young'
        when extract(year from age(current_date, birth_date)) < 60 then 'Middle'
        else 'Senior'
    end as age_group
from {{ ref('stg_clients') }}
```

6.2. Создайте fact_sales.sql (факт продаж):

```
select
    order_id as sales_key,
    client_id,
    order_date,
    amount,
    amount * 0.2 as estimated_tax
from {{ ref('int_clients_orders') }}
```

7. **Добавление тестов**

7.1. Создайте models/marts/_sources.yml (или models/schema.yml) с описанием и тестами:

```
version: 2
models:
  - name: dim_client
    description: "Измерение клиента с возрастной группой"
```

```
    columns:
      - name: client_id
        description: "Уникальный идентификатор клиента"
        tests:
          - unique
          - not_null
      - name: client_name
        tests:
          - not_null
  - name: fact_sales
    columns:
      - name: sales_key
        tests:
          - unique
          - not_null
      - name: amount
        tests:
          - not_null
          - accepted_values:
              values: [0, 0.1, 0.5, 1] # неприменимо, но для примера
```

(В реальном примере тест accepted_values лучше заменить на dbt_utils.accepted_range или свой макрос.)

8. **Создание макроса**

8.1. В папке macros/ создайте format_currency.sql:

```
{% macro format_currency(amount, currency='USD') %}
  case
    when '{{ currency }}' = 'USD' then {{ amount }} * 1.0
    when '{{ currency }}' = 'EUR' then {{ amount }} * 1.05
    else {{ amount }}
```

```
    end
{% endmacro %}
```

8.2. Используйте макрос в модели fact_sales, заменив amount на {{ format_currency('amount', 'USD') }}.

9. **Запуск dbt-проекта**

9.1. Выполните последовательно:

```
dbt run        # выполнить модели
dbt test       # запустить тесты
dbt docs generate
dbt docs serve # открыть документацию в браузере
```

9.2. Проверьте, что в схеме dbt_transformed появились таблицы: stg_clients, stg_orders, int_clients_orders, dim_client, fact_sales.

Пример выполнения задания

Исходные данные (raw.clients и raw.orders) – показаны выше.

Результат модели dim_client **после** dbt run**:**

client_id	client_name	city	segment	age_group
1	Иван Петров	Москва	Gold	Middle
2	Мария Сидорова	СПб	Silver	Middle
3	Алексей Иванов	Москва	Standard	Young
4	Unknown	Казань	Gold	Middle
5	Петр Смирнов	Unknown	Silver	Middle

Результат fact_sales**:**

sales_key	client_id	order_date	amount	estimated_tax
101	1	2025-01-15	5500	1100
102	2	2025-01-20	3200	640
103	1	2025-02-10	7800	1560
105	4	2025-03-01	4200	840
106	5	2025-03-05	3000	600

Тесты: проверка unique и not_null на client_id и sales_key проходит; not _null на client_name обнаруживается проблема с client_id=4 (но после coalesce ошибки нет).

Документация: после dbt docs serve открывается сайт со схемой линий, описанием моделей и колонок.

Индивидуальные задания

Каждый вариант описывает предметную область и требования к трансформации. Студент должен создать dbt-проект с моделями staging, intermediate, marts, добавить тесты и документацию.

Вариант 1. Продажи книжного магазина

Исходные таблицы: raw.books (book_id, title, author, price), raw.sales (sale_id, book_id, sale_date, quantity). Требуется создать модель fact_sales (сумма продаж по дням) и dim_book. Добавить тест на уникальность book_id и неотрицательность quantity.

Вариант 2. Аренда велосипедов

Исходные таблицы: raw.stations (station_id, station_name, city), raw.trips (trip_id, start_station_id, end_station_id, duration_sec, start_time). Построить модели: stg_trips (перевести duration в минуты), dim_station, fact_trips (количество поездок по станциям).

Вариант 3. Курсы валют

Исходные таблицы: raw.rates (currency_code, rate_to_usd, date), raw.transactions (txn_id, amount_local, currency, txn_date). Трансформировать в fact_transactions_usd (сумма в USD) с макросом конвертации. Добавить тест на уникальность txn_id.

Вариант 4. Посещаемость веб-сайта

Исходные: raw.page_views (view_id, user_id, page_url, timestamp, session_id). Создать промежуточную модель с извлечением домена из page_url, витрину – количество просмотров по дням и странам (страну извлекать из IP – статический seed-файл).

Вариант 5. Складской учёт

Исходные: raw.products (product_id, name, category), raw.stock_movements (movement_id, product_id, quantity, movement_date, type (in/out)). Построить модель current_stock (расчётный остаток на последнюю дату). Использовать оконные функции.

Вариант 6. Отзывы и рейтинги

Исходные: raw.reviews (review_id, user_id, product_id, rating, review_text, date). Добавить проверку rating от 1 до 5, а также уникальность пары (user_id, product_id) в модели dim_review. Создать модель fact_ratings (средний рейтинг по продуктам за месяц).

Вариант 7. Заказы в общепите

Исходные: raw.orders (order_id, table_id, order_time), raw.order_items (item_id, order_id, menu_item_id, quantity, price). Создать модели: dim_menu_item, fact_order_details (сумма заказа), а также проверить уникальность order_id в итоговой витрине.

Вариант 8. Расписание самолётов

Исходные: raw.flights (flight_id, airline, departure_airport, arrival_airport, scheduled_departure, scheduled_arrival). Построить модель с вычислением длительности полёта в минутах, модель fact_flights_delayed (рейсы с задержкой > 15 мин), используя макрос для вычисления разницы времени.

Вариант 9. Банковские счета

Исходные: raw.accounts (account_id, client_id, open_date, balance), raw.transactions (txn_id, account_id, txn_date, amount, txn_type). Создать модель fact_clients_balance (суммарный баланс по клиентам), добавить тест, что balance >= 0.

Вариант 10. Подписки на сервисы

Исходные: raw.subscriptions (sub_id, user_id, plan_id, start_date, end_date), raw.users (user_id, name, email). Построить SCD Type 2 для измерения dim_user (активность подписки) – создать несколько версий пользователя при смене плана.

Вариант 11. Данные с IoT-датчиков

Исходные: raw.sensors (sensor_id, location), raw.readings (reading_id, sensor_id, ts, temperature, humidity). Построить промежуточную модель с агрегацией по часам (средняя температура, влажность), витрину – суточные аномалии (температура > 30°C). Добавить тест "не более 1 аномалии в день".

Вариант 12. Видеохостинг

Исходные: raw.videos (video_id, title, channel_id, duration_sec), raw.views (view_id, video_id, user_id, view_date, watch_time_sec). Создать модели: fact_video_popularity (суммарное время просмотра по видео), процент просмотра от длительности (watch_time/duration_sec). Обеспечить тест watched_ratio <= 1.

Вариант 13. Криптовалютный портфель

Исходные: raw.assets (asset_id, ticker, name), raw.trades (trade_id, asset_id, trade_type (buy/sell), quantity, price_usd, trade_date). Построить витрину portfolio_value (количество активов на руках с учётом всех сделок). Использовать макрос для вычисления текущего остатка.

Вариант 14. Социальная сеть

Исходные: raw.users (user_id, name, registration_date), raw.posts (post_id, user_id, content, post_date), raw.likes (like_id, post_id, user_id, like_date). Создать модели: fact_engagement (количество лайков и постов по дням), dim_user_age

(активность пользователя: количество постов > 10 – активный). Добавить тесты.

Вариант 15. HR-аналитика (рекрутинг)

Исходные: raw.candidates (candidate_id, name, skill, experience_years), raw.interviews (interview_id, candidate_id, interview_date, score, status). Построить staging (очистка: округление score до целого), intermediate – лучший скилл по среднему score, mart – таблица перспективных кандидатов (score > 8 и опыт > 3 лет).

Дополнительное задание для всех вариантов:

Реализовать пользовательский тест (custom test) в SQL, который проверяет, что разница между суммой детальных записей и итоговой суммой в витрине не превышает 1%. Подключить его к соответствующим моделям и запустить dbt test.

Требования к отчёту

1. Титульный лист с названием работы, ФИО, группой, номером варианта.
2. Цель и задачи (из методички).
3. Теоретическая часть – что такое dbt, ELT, основные концепции.
4. Ход выполнения работы:
   - Структура созданного dbt-проекта (скриншот дерева каталогов).
   - Листинги всех моделей SQL (staging, intermediate, marts) с комментариями.
   - Содержимое YAML-файлов с тестами и документацией.
   - Макросы (если использовались).
5. Результаты:
   - Вывод команды dbt run --select ... (скриншот или текст).
   - Вывод команды dbt test (результаты тестов).

- Скриншот сгенерированной документации (главная страница, граф линий).
- Примеры данных из результирующих таблиц (SELECT * LIMIT 10).

6. Ответы на контрольные вопросы.
7. Вывод – почему dbt удобен для трансформаций в стиле ELT, как тесты и документация повышают надёжность.

Контрольные вопросы

1. Чем отличается подход ELT от ETL? Почему dbt реализует именно ELT?

2. Как dbt определяет порядок выполнения моделей? Что такое ref() и зачем он нужен?

3. В чём разница между материализациями table, view, incremental, ephemeral?

4. Как в dbt тестировать модели на уникальность составного ключа (двух колонок)?

5. Как написать пользовательский тест (generic test) с использованием макросов?

6. Как с помощью dbt документировать не только модели, но и источники (sources)?

7. Что такое seeds в dbt и для чего они применяются?

8. Как в dbt обрабатывать изменяющиеся данные (инкрементальные модели)? Приведите синтаксис is_incremental().

9. Как запустить только определённую модель или подпапку (--select, --exclude)?

10. Как интегрировать dbt с Airflow (например, через DbtRunOperator)?

11. Какие преимущества даёт использование dbt с Git (ветки, CI/CD)?

12. Как в dbt задать конфигурацию модели (materialized, schema, tags) внутри SQL-файла через {{ config(...) }}?

13. Что такое dbt Cloud и чем он отличается от dbt Core?

14. Как выполнять dbt-модели параллельно для ускорения?

15. Можно ли использовать dbt с нереляционными хранилищами (например, BigQuery, Snowflake)? Какие адаптеры существуют?

Лабораторная работа №12.
Инкрементальная загрузка (CDC-подход)

Цель работы: реализовать инкрементальную загрузку данных (delta load) с использованием метода Change Data Capture (CDC) на основе полей timestamp или hash-столбца для обновления только новых и изменённых записей в целевой таблице.

Задачи:

1. Изучить принципы инкрементальной загрузки: полная (full refresh) vs дельта (delta).
2. Создать исходную таблицу-источник и целевую таблицу в PostgreSQL.
3. Реализовать два подхода к CDC:
   - **на основе timestamp** – загрузка записей, изменённых после последнего запуска
   - **на основе hash-столбца** – сравнение хеша всей строки для обнаружения изменений
4. Написать SQL-скрипт (или Python-скрипт) для выполнения инкрементальной загрузки (UPDATE / INSERT).
5. Настроить хранение состояния последней загрузки (водяной знак – watermark).
6. Выполнить тестовые изменения в источнике (вставка новых строк, обновление существующих, удаление – если требуется) и убедиться, что целевая таблица обновляется корректно.
7. Сравнить производительность полной и инкрементальной загрузки на объёме данных.

Теоретическая часть

Инкрементальная загрузка (Delta Load) – подход к переносу данных, при котором загружаются только новые и изменённые записи после предыдущей загрузки. В отличие от полной загрузки (truncate & insert),

инкрементальная экономит время и ресурсы при работе с большими объёмами данных.

Методы CDC (Change Data Capture):

Метод	Описание	Преимущества	Недостатки
Timestamp (дата изменения)	В исходной таблице есть столбец updated_at. Загружаются записи с updated_at > last_load_time.	Простота, не требует доп. структур	Зависит от точности времени; не видит удаления
Hash-столбец	Вычисляется хеш всех значимых колонок. При изменении хеш меняется.	Обнаруживает любые изменения	Требует вычисления на каждом цикле
Триггеры / лог изменений	Триггеры пишут изменения в отдельную таблицу.	Гарантированное обнаружение, без пропусков	Сложность, нагрузка на источник
Логический декодинг (PostgreSQL logical replication)	Читает WAL-логи.	Нулевое влияние на источник	Сложная настройка

Алгоритм на основе timestamp:

1. Храним last_load_timestamp (например, в служебной таблице или файле).
2. При загрузке: SELECT * FROM source WHERE updated_at > last_load_timestamp.
3. Вставляем / обновляем целевые записи (INSERT ... ON CONFLICT DO UPDATE – upsert).
4. Обновляем last_load_timestamp = максимальный updated_at из выбранных записей.

Алгоритм на основе hash:

1. Храним хеши строк или updated_at.
2. Для источника вычисляем MD5 (или другой хеш) от конкатенации всех колонок.
3. Сравниваем с хешем в целевой таблице – если различается, обновляем.

Upsert в PostgreSQL (ON CONFLICT):

```
INSERT INTO target (id, col1, col2, updated_at)
VALUES (1, 'a', 'b', NOW())
ON CONFLICT (id) DO UPDATE SET
   col1 = EXCLUDED.col1,
   col2 = EXCLUDED.col2,
   updated_at = EXCLUDED.updated_at;
```

Водяной знак (watermark) – значение времени или ID, до которого данные уже загружены. Хранится в служебной таблице.

Методика выполнения работы

1. **Подготовка окружения**
   1.1. Создайте каталог lab12_incremental_load
   1.2. Внутри создайте подкаталоги: sql, scripts, logs
   1.3. Запустите PostgreSQL в Docker (как в ЛР №2).
   1.4. Создайте базу данных cdc_db, схемы source и target.
2. **Создание таблиц в источнике и приёмнике**
   2.1. Выполните SQL-скрипт для создания источника:

```
CREATE SCHEMA source;
CREATE TABLE source.orders (
   order_id INTEGER PRIMARY KEY,
   customer_name VARCHAR(100),
   amount DECIMAL(10,2),
   status VARCHAR(20),
```

```
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

INSERT INTO source.orders (order_id, customer_name, amount, status) VALUES
(1, 'Иван Петров', 1500, 'pending'),
(2, 'Мария Сидорова', 3200, 'completed'),
(3, 'Алексей Иванов', 780, 'pending');
```

2.2. Создайте целевую таблицу (с возможностью upsert):

```
CREATE SCHEMA target;
CREATE TABLE target.orders (
    order_id INTEGER PRIMARY KEY,
    customer_name VARCHAR(100),
    amount DECIMAL(10,2),
    status VARCHAR(20),
    last_updated TIMESTAMP
);
```

2.3. Создайте служебную таблицу для хранения watermark:

sql

```
CREATE TABLE target.load_watermark (
    table_name VARCHAR(100),
    last_load_timestamp TIMESTAMP
);
INSERT INTO target.load_watermark VALUES ('orders', '1900-01-01'::TIMESTAMP);
```

3. **Реализация инкрементальной загрузки через timestamp (SQL-скрипт)**

3.1. Создайте файл sql/incremental_load_timestamp.sql:

```
DO $$
DECLARE
    last_load TIMESTAMP;
    current_max TIMESTAMP;
BEGIN
    -- 1. Получить последнее время загрузки
    SELECT last_load_timestamp INTO last_load
    FROM target.load_watermark WHERE table_name = 'orders';
    RAISE NOTICE 'Last load: %', last_load;

    -- 2. Инкрементальная загрузка (upsert)
    INSERT INTO target.orders (order_id, customer_name, amount, status, last_updated)
    SELECT
        order_id,
        customer_name,
        amount,
        status,
        updated_at
    FROM source.orders
    WHERE updated_at > last_load
    ON CONFLICT (order_id) DO UPDATE SET
        customer_name = EXCLUDED.customer_name,
        amount = EXCLUDED.amount,
        status = EXCLUDED.status,
        last_updated = EXCLUDED.last_updated;

    -- 3. Обновить watermark
    SELECT COALESCE(MAX(updated_at), last_load) INTO current_max
```

```
    FROM source.orders WHERE updated_at > last_load;

    UPDATE target.load_watermark
    SET last_load_timestamp = current_max
    WHERE table_name = 'orders';

    RAISE NOTICE 'New watermark: %', current_max;
END $$;
```

3.2. Выполните скрипт в psql: \i sql/incremental_load_timestamp.sql

4. **Тестирование изменений в источнике**

4.1. Добавьте новые записи и измените существующие в source.orders:

UPDATE source.orders SET amount = 1700, status = 'confirmed', updated_at = CURRENT_TIMESTAMP WHERE order_id = 1;

INSERT INTO source.orders (order_id, customer_name, amount, status) VALUES (4, 'Ольга Кузнецова', 2400, 'pending');

4.2. Повторно выполните скрипт инкрементальной загрузки.

4.3. Проверьте, что целевая таблица содержит обновлённую сумму для order_id=1 и новую запись order_id=4, а order_id=2,3 не перезаписывались.

5. **Реализация инкрементальной загрузки через hash-столбец (Python-скрипт)**

5.1. В папке scripts/ создайте incremental_hash_load.py:

```
import psycopg2
import hashlib
from datetime import datetime
```

```
    conn = psycopg2.connect("host=localhost dbname=cdc_db user=etl_user password=etl_pass")
    cur = conn.cursor()

    # Добавим столбец hash_row в целевую таблицу, если его нет
    cur.execute("ALTER TABLE target.orders ADD COLUMN IF NOT EXISTS row_hash VARCHAR(32)")

    # Получить все строки из источника
    cur.execute("SELECT order_id, customer_name, amount, status, updated_at FROM source.orders")
    source_rows = cur.fetchall()

    inserted = 0
    updated = 0

    for row in source_rows:
        order_id, name, amount, status, updated_at = row
        # Вычисляем хеш
        hash_str = hashlib.md5(f"{name}{amount}{status}".encode()).hexdigest()

        # Проверим существование записи в целевой таблице
        cur.execute("SELECT row_hash FROM target.orders WHERE order_id = %s", (order_id,))
        existing = cur.fetchone()

        if existing is None:
            cur.execute("""
                INSERT INTO target.orders (order_id, customer_name, amount, status, last_updated, row_hash)
```

```
                VALUES (%s, %s, %s, %s, %s, %s)
            """, (order_id, name, amount, status, updated_at, hash_str))
            inserted += 1
        elif existing[0] != hash_str:
            cur.execute("""
                UPDATE target.orders
                SET customer_name=%s, amount=%s, status=%s, last_updated=%s,
row_hash=%s
                WHERE order_id=%s
            """, (name, amount, status, updated_at, hash_str, order_id))
            updated += 1

    conn.commit()
    print(f"Inserted: {inserted}, Updated: {updated}")
    cur.close()
    conn.close()
```

5.2. Запустите скрипт: python scripts/incremental_hash_load.py

5.3. Добавьте ещё одно изменение в источник (например, измените status) и запустите скрипт снова – убедитесь, что только изменённая запись обновилась.

6. **Реализация удаления (опционально)**

   6.1. Для обработки удалений в источнике можно использовать флаг is_deleted или логирование первичных ключей удалённых записей. Реализуйте механизм, который добавляет в целевую таблицу deleted_flag и устанавливает его в TRUE, если записи больше нет в источнике.

6.2. Сравнение полного списка ключей источника и цели – затратно для больших таблиц; рекомендуется использовать отдельную таблицу изменений или логическую репликацию.

7. **Сравнение полной vs инкрементальной загрузки**

   7.1. Напишите скрипт для полной загрузки: TRUNCATE target.orders; INSERT INTO target.orders SELECT * FROM source.orders;

   7.2. Засеките время выполнения полной загрузки и инкрементальной (при небольшом количестве изменений).

   7.3. Сделайте вывод об эффективности.

Пример выполнения задания

Исходное состояние:

- source.orders: 3 записи.
- target.orders: пусто (первый запуск загрузит все).
- load_watermark: 1900-01-01.

После первого запуска (timestamp-метод):

- Загружены все 3 записи в target.
- Watermark обновлён до максимального updated_at среди загруженных (например, '2025-05-12 10:00:00').

Изменение источника:

UPDATE source.orders SET amount=1900, updated_at=NOW() WHERE order_id=1;

INSERT INTO source.orders VALUES (5, 'Новый клиент', 500, 'pending', NOW());

Второй запуск:

- Выбираются записи с updated_at > '2025-05-12 10:00:00' (order_id=1 и 5).
- Для order_id=1 выполняется UPDATE в target, для 5 – INSERT.
- Watermark обновляется до нового максимума.

Результат в target.orders после второго запуска (timestamp):

order_id	customer_name	amount	status	last_updated
1	Иван Петров	1900	pending	2025-05-12 10:05:00
2	Мария Сидорова	3200	completed	2025-05-12 10:00:00
3	Алексей Иванов	780	pending	2025-05-12 10:00:00
5	Новый клиент	500	pending	2025-05-12 10:05:00

Hash-метод даст аналогичный результат, но обнаружит изменения даже если updated_at не изменился.

Индивидуальные задания

Каждый вариант задаёт структуру источника и требования к инкрементальной загрузке (какие поля считать изменяющимися, нужно ли обрабатывать удаления, какой метод использовать). Студент реализует скрипт инкрементальной загрузки на основе timestamp или hash.

Вариант 1. Транзакции интернет-магазина

Таблица source.transactions (txn_id, user_id, amount, status, txn_date, updated_at). Загружать инкрементально по updated_at. Добавить индекс на updated_at. Написать полную и дельта-загрузку, сравнить время.

Вариант 2. Клиенты с SCD Type 2

Таблица source.customers (cust_id, name, phone, address, updated_at). Целевая таблица должна хранить историю версий (surrogate_key, valid_from, valid_to). Инкрементальная загрузка: при изменении атрибутов закрывать старую версию и создавать новую.

Вариант 3. Логи доступа (большой объём)

Таблица source.access_log (log_id, user_ip, endpoint, response_time, log_timestamp). Реализовать инкрементальную загрузку с batch-размером 1000 строк и использовать хранение последнего log_id вместо timestamp (watermark по числовому ключу).

Вариант 4. Инвентаризация склада

Таблица source.inventory (product_id, quantity, last_modified). При каждой загрузке обновлять целевую таблицу. Если количество изменилось, записывать в отдельную таблицу inventory_history (product_id, old_quantity, new_quantity, changed_at). Сделать атомарно.

Вариант 5. Обработка удалений (soft delete)

Таблица source.employees (emp_id, name, department, is_deleted, deleted_at). Инкрементально загружать только активных сотрудников (is_deleted = false). При удалении в источнике (is_deleted становится true) – установить в целевой таблице is_active = false.

Вариант 6. Использование хеширования для детекта изменений

Таблица source.products (prod_id, name, price, category). Нет полей updated_at. Реализовать инкрементальную загрузку на основе сравнения MD5-хеша полей (name, price, category). Добавить в целевую таблицу столбец row_hash.

Вариант 7. Многопоточная загрузка (Python)

Таблица source.orders объёмом 100 000 строк. Реализовать инкрементальную загрузку на Python с использованием ThreadPoolExecutor для параллельной обработки изменённых записей. Замерить ускорение.

Вариант 8. Совмещение timestamp и удаления

Таблица source.payments (payment_id, order_id, amount, status, updated_at, deleted_flag). Инкрементальная загрузка должна учитывать не только updated_at, но и помечать записи как удалённые в target, если в источнике deleted_flag = true.

Вариант 9. Инкрементальная загрузка в ClickHouse

Перенести данные из PostgreSQL source в ClickHouse target. ClickHouse не поддерживает UPDATE; нужно вставлять новые версии строк и использовать FINAL при чтении. Реализовать метод с использованием ReplacingMergeTree.

Вариант 10. Загрузка с компенсацией часовых поясов

Таблица source.events (event_id, event_timestamp_utc, event_type, payload). В водяном знаке хранить UTC-время. При загрузке учитывать возможные системные задержки: загружать записи с event_timestamp_utc > last_load - INTERVAL '5 minutes' (overlap).

Вариант 11. Импорт из CSV с инкрементальным флагом

Вместо БД источник – CSV-файлы, которые обновляются. Реализовать механизм: отслеживать last_modified файла, загружать только новые строки по сравнению с предыдущим снапшотом (через хеш или позицию в файле).

Вариант 12. CDC через логическую репликацию PostgreSQL

Настроить publication на таблице source и subscription на другой базе (той же или другой). Реализовать приём изменений (INSERT/UPDATE/DELETE) и применять к целевой таблице. Упрощённо: использовать pg_recvlogical или Python-библиотеку psycopg3 с COPY ... FROM.

Вариант 13. Дельта загрузка с окном (windowed)

Таблица source.sensor_readings (reading_id, sensor_id, value, reading_time). Загружать только данные за последние 24 часа (скользящее окно). Водяной знак – reading_time. Старые данные не обновлять.

Вариант 14. Обработка изменений с помощью триггера

В источнике создать триггер, который при каждом INSERT/UPDATE/DELETE пишет ключ изменённой записи в служебную таблицу changed_pks. При инкрементальной загрузке читать changed_pks и загружать только эти записи (после чего очищать). Сравнить с timestamp-методом.

Вариант 15. Универсальный загрузчик (для любой таблицы)

Написать на Python функцию incremental_load(source_table, target_table, pk_columns, watermark_column, batch_size=1000), которая генерирует query для инкрементальной загрузки динамически. Протестировать на двух разных таблицах из вариантов выше.

Дополнительное задание для всех вариантов:

Реализовать оповещение телеграм-ботом о результатах загрузки: сколько записей вставлено, обновлено, удалено, и время выполнения. Настроить отправку сообщения после каждого запуска.

Требования к отчёту

1. Титульный лист с названием работы, ФИО, группой, номером варианта.
2. Цель и задачи (из методички).
3. Теоретическая часть – описание CDC, методов инкрементальной загрузки (timestamp vs hash), upsert, watermark.
4. Ход выполнения работы:
   - SQL-скрипты создания исходных и целевых таблиц.
   - Листинг реализации инкрементальной загрузки (SQL или Python) с комментариями.
   - Скрипты для внесения тестовых изменений в источник.
5. Результаты:
   - Содержимое целевой таблицы до и после инкрементальной загрузки.
   - Выводы логов (количество вставленных/обновлённых строк).
   - Сравнение времени выполнения полной и инкрементальной загрузки (в виде таблицы).
   - Демонстрация работы watermark (запись в служебной таблице).
6. Ответы на контрольные вопросы.

7. Вывод – в чём преимущества инкрементальной загрузки, какой метод (timestamp или hash) оказался удобнее для выбранного варианта.

Контрольные вопросы

1. В чём разница между полной (full refresh) и инкрементальной (delta) загрузкой?

2. Почему метод на основе timestamp не может надёжно обнаружить удалённые записи?

3. Как работает механизм ON CONFLICT в PostgreSQL и какие существуют стратегии (DO NOTHING, DO UPDATE)?

4. Что такое «водяной знак» и где его безопасно хранить в контексте ETL?

5. Какие проблемы могут возникнуть, если в источнике нет поля updated_at, но данные меняются?

6. Какой хеш-алгоритм рекомендуется использовать для сравнения строк и почему?

7. Как обрабатывать очень частые обновления (миллисекунды) с timestamp? Может ли запись быть пропущена?

8. Как built-in функции pg_last_committed_xact могут помочь в CDC?

9. В каких сценариях инкрементальная загрузка не даёт выигрыша по сравнению с полной?

10. Как избежать дубликатов при сбое инкрементальной загрузки на середине (транзакционность)?

11. Как с помощью ROW_NUMBER() и оконных функций можно загрузить только последние версии rows?

12. Что такое логический декодинг (logical decoding) в PostgreSQL? Когда его используют вместо триггеров?

13. Как изменится логика, если целевая таблица имеет SCD Type 2?

14. Как реализовать идемпотентность инкрементальной загрузки (повторный запуск не навредит)?

15. Как протестировать инкрементальную загрузку в среде с высокой конкуренцией за обновление строк источника?

Лабораторная работа №13.
Мониторинг пайплайна с Prometheus и Grafana

Цель работы: экспортировать метрики ETL-пайплайна (время выполнения, количество обработанных строк, статус успеха/ошибки) в Prometheus, настроить сбор метрик и визуализировать их в виде графиков и дашбордов в Grafana.

Задачи:

1. Изучить основные концепции мониторинга: метрики, экспортеры, сбор (scrape), визуализация.
2. Развернуть Prometheus и Grafana в Docker-контейнерах.
3. В существующий ETL-скрипт на Python добавить экспорт метрик с использованием клиентской библиотеки Prometheus (prometheus_client).
4. Определить следующие метрики:
   - etl_duration_seconds – гистограмма времени выполнения отдельных этапов.
   - etl_rows_processed_total – счётчик количества обработанных строк.
   - etl_last_run_timestamp – временная метка последнего успешного запуска.
   - etl_job_status – метка состояния (0 – успех, 1 – ошибка).
5. Настроить Prometheus (файл prometheus.yml) на сбор метрик с ETL-скрипта (экспорт через HTTP-эндпоинт).
6. В Grafana создать дашборд с графиками:
   - Длительность выполнения по шагам (линейный график).
   - Количество обработанных строк по запускам (барчарт).
   - Индикатор статуса последнего запуска.
7. Запустить ETL-скрипт несколько раз с разными объёмами данных и наблюдать за обновлением метрик в Grafana.

8. Настроить алертинг (Alertmanager) для уведомления при падении пайплайна или превышении времени выполнения.

Теоретическая часть

Мониторинг в инжиниринге данных

Для надёжной работы ETL/ELT-пайплайнов необходимо отслеживать их состояние, производительность и качество данных. Prometheus – система сбора и хранения метрик, Grafana – инструмент визуализации.

Ключевые понятия:

Термин	Описание
Метрика (Metric)	Числовая характеристика системы (счётчик, гистограмма, gauge).
Экспортер (Exporter)	Процесс, который собирает метрики с приложения и отдаёт их в формате Prometheus.
Scrape	Процесс периодического опроса экспортера Prometheus'ом.
Gauge	Метрика, значение которой может увеличиваться и уменьшаться (текущая температура, количество активных соединений).
Counter	Счётчик, значение которого только растёт (общее число обработанных строк).
Histogram	Гистограмма, измеряющая распределение значений (время выполнения).

Типы метрик для ETL-пайплайна:

Метрика	Тип	Назначение
etl_duration_seconds	Histogram	Время выполнения (p50, p95, p99)

Метрика	Тип	Назначение
etl_rows_processed	Counter	Количество строк на каждом этапе
etl_errors_total	Counter	Количество ошибок
etl_last_success_timestamp	Gauge	UNIX-время последнего успеха
etl_job_status	Gauge	1 – работает, 0 – остановлен

Архитектура:

ETL Script (Python + prometheus_client) → HTTP /metrics → Prometheus (scrape) → Grafana (визуализация)

Пример экспорта метрик в Python:

```
from prometheus_client import start_http_server, Counter, Histogram, Gauge
import time
import random

ROWS_PROCESSED = Counter('etl_rows_processed_total', 'Total rows processed')
DURATION = Histogram('etl_duration_seconds', 'ETL job duration')
LAST_RUN = Gauge('etl_last_run_timestamp', 'Last successful run timestamp')

start_http_server(8000)

while True:
    start = time.time()
    for i in range(random.randint(100, 1000)):
        ROWS_PROCESSED.inc()
        time.sleep(0.001)
```

```
    DURATION.observe(time.time() - start)
    LAST_RUN.set(time.time())
    time.sleep(60)
```

Методика выполнения работы

1. **Подготовка окружения**

   1.1. Создайте каталог lab13_monitoring

   1.2. Внутри создайте подкаталоги: etl_script, prometheus, grafana

   1.3. Напишите docker-compose.yaml, запускающий Prometheus и Grafana:

```
version: '3.8'
services:
  prometheus:
    image: prom/prometheus:latest
    container_name: prometheus
    volumes:
      - ./prometheus/prometheus.yml:/etc/prometheus/prometheus.yml
    ports:
      - "9090:9090"
    command:
      - '--config.file=/etc/prometheus/prometheus.yml'
  grafana:
    image: grafana/grafana:latest
    container_name: grafana
    ports:
      - "3000:3000"
    environment:
      - GF_SECURITY_ADMIN_PASSWORD=admin
```

   1.4. Создайте файл prometheus/prometheus.yml:

```
global:
  scrape_interval: 15s
scrape_configs:
  - job_name: 'etl_job'
    static_configs:
      - targets: ['host.docker.internal:8000']
```

2. **Создание ETL-скрипта с экспортом метрик**

2.1. В папке etl_script создайте виртуальное окружение и установите зависимости:

```
pip install prometheus-client pandas psycopg2-binary
```

2.2. Создайте файл etl_with_metrics.py:

```
import time
import random
from prometheus_client import start_http_server, Counter, Histogram, Gauge, Summary
import pandas as pd

# Определение метрик
ROWS_READ = Counter('etl_rows_read_total', 'Total rows read from source')
ROWS_WRITTEN = Counter('etl_rows_written_total', 'Total rows written to target')
ETL_DURATION = Histogram('etl_duration_seconds', 'Duration of ETL job', buckets=[1, 5, 10, 30, 60])
ETL_LAST_SUCCESS = Gauge('etl_last_success_timestamp', 'Last successful run timestamp')
ETL_STATUS = Gauge('etl_job_status', 'Job status (1=success, 0=failure)')
```

```
def simulate_extract():
    # Имитация чтения данных (CSV или БД)
    time.sleep(random.uniform(0.5, 2.0))
    rows = random.randint(100, 1000)
    for _ in range(rows):
        ROWS_READ.inc()
    return rows

def simulate_transform():
    time.sleep(random.uniform(0.5, 1.5))
    # трансформация

def simulate_load(rows):
    time.sleep(random.uniform(0.5, 1.0))
    for _ in range(rows):
        ROWS_WRITTEN.inc()

def run_etl():
    start_time = time.time()
    try:
        rows = simulate_extract()
        simulate_transform()
        simulate_load(rows)
        duration = time.time() - start_time
        ETL_DURATION.observe(duration)
        ETL_LAST_SUCCESS.set(time.time())
        ETL_STATUS.set(1)
        print(f"ETL succeeded: {rows} rows, {duration:.2f}s")
    except Exception as e:
        ETL_STATUS.set(0)
```

```
        print(f"ETL failed: {e}")

if __name__ == '__main__':
    # Запуск HTTP-сервера для метрик на порту 8000
    start_http_server(8000)
    print("Metrics server running on port 8000")
    # Эмуляция периодического запуска (например, каждые 30 секунд)
    while True:
        run_etl()
        time.sleep(30)
```

2.3. Запустите скрипт: python etl_with_metrics.py

3. **Запуск Prometheus и Grafana**

3.1. Из каталога lab13_monitoring запустите:

docker-compose up -d

3.2. Проверьте, что Prometheus доступен: http://localhost:9090. В Target-ах должно отображаться etl_job с состоянием UP.

4. **Настройка Grafana**

4.1. Откройте http://localhost:3000 (логин: admin, пароль: admin).

4.2. Добавьте источник данных Prometheus:

- Connection

  URL: http://prometheus:9090 (или http://localhost:9090 если из браузера)
- Нажмите "Save & Test".

4.3. Создайте дашборд:

- Нажмите "Create Dashboard" → "Add new panel".

- В запросе введите метрику: rate(etl_rows_read_total[1m]) – для отображения скорости чтения.
- Добавьте второй график: histogram_quantile(0.95, sum(rate(etl_duration_seconds_bucket[5m])) by (le)) – 95-й перцентиль времени выполнения.
- Третий график: etl_last_success_timestamp – можно преобразовать в "Time since last success" с помощью time() - etl_last_success_timestamp.
- Четвёртая панель – Stat (singlestat) для etl_job_status (отображать текст "OK" или "FAIL").

4.4. Сохраните дашборд.

5. **Тестирование и наблюдение**

5.1. Запустите ETL-скрипт на несколько минут.

5.2. В Grafana в реальном времени наблюдайте рост счётчиков, гистограмму времени выполнения.

5.3. Попробуйте вызвать ошибку (например, раскомментируйте строку raise Exception в run_etl), чтобы увидеть изменение статуса.

6. **Настройка алертинга (опционально)**

6.1. В Prometheus добавьте правило алерта в prometheus.yml:

```
rule_files:
  - "alerts.yml"
```

Создайте prometheus/alerts.yml:

```
groups:
  - name: etl_alerts
    rules:
      - alert: ETLJobFailed
```

```
expr: etl_job_status == 0
for: 1m
annotations:
  summary: "ETL job failed"
```

6.2. Настройте Alertmanager для отправки уведомлений (email, webhook). Или просто просматривайте алерты в Prometheus UI.

Пример выполнения задания

Метрики, экспортируемые ETL-скриптом:

```
# HELP etl_duration_seconds Duration of ETL job
# TYPE etl_duration_seconds histogram
etl_duration_seconds_bucket{le="1.0"} 2
etl_duration_seconds_bucket{le="5.0"} 7
etl_duration_seconds_sum 12.34
etl_duration_seconds_count 5

# HELP etl_rows_read_total Total rows read from source
# TYPE etl_rows_read_total counter
etl_rows_read_total 12345

# HELP etl_job_status Job status (1=success,0=failure)
# TYPE etl_job_status gauge
etl_job_status 1
```

График в Grafana (результат выполнения):

- График etl_duration_seconds показывает, что последние 5 запусков укладываются в 2–4 секунды.
- Счётчик etl_rows_read_total увеличивается на ~800 строк за запуск.
- Панель статуса зелёная (OK) при успешном выполнении.

Вывод: Prometheus + Grafana позволяют оперативно отслеживать здоровье ETL-пайплайна, выявлять замедления и аномалии.

Индивидуальные задания

Каждый вариант описывает специфику ETL-пайплайна и требует добавления уникальных метрик, а также настройки визуализации и алертов.

Вариант 1. Пайплайн загрузки из API

В пайплайне есть этапы: запрос к API, парсинг JSON, запись в БД. Добавить
метрики: api_response_time_seconds, api_http_status, json_parse_errors_total. В Grafana отобразить гистограмму времени ответа API.

Вариант 2. Пайплайн с Airflow

Запускать ETL-скрипт через Airflow. В скрипт добавить метрики для каждого таска (сенсор, трансформация, загрузка). Prometheus должен собирать метрики из скрипта. В Grafana настроить дашборд с графиком времени выполнения каждого таска.

Вариант 3. Пайплайн с Kafka-потоком

Consumer читает сообщения из Kafka и вставляет в ClickHouse. Метрики: kafka_consumer_lag, records_consumed_total, insert_clickhouse_duration_seconds. Сделать дашборд с лагом consumer.

Вариант 4. Мониторинг качества данных

В ETL добавить проверки (unique, not_null) и экспортировать метрики: dq_failed_checks_total (с лейблом check_name). Настроить алерт, если количество ошибок > 0.

Вариант 5. Пайплайн с обработкой файлов

Скрипт обрабатывает файлы в папке: сенсор ожидает файл, читает, архивирует. Метрики: files_processed_total, file_size_bytes, processing_duration_seconds. Гистограмма размера файлов.

Вариант 6. Использование Pushgateway

Вместо pull-модели (Prometheus скрейпит) использовать Pushgateway для сбора метрик из короткоживущих ETL-джобов. Реализовать push метрик через prometheus_client.push_to_gateway().

Вариант 7. Мониторинг ресурсов контейнера (cAdvisor)

Добавить в docker-compose сервис cAdvisor и настроить сбор метрик CPU/памяти контейнера с ETL-скриптом. В Grafana добавить графики использования ресурсов.

Вариант 8. Алертинг на длительность выполнения

Настроить правило алерта в Prometheus: если 95-й перцентиль времени выполнения ETL за последние 5 минут превышает 10 секунд – отправить в Slack (через webhook). Реализовать заглушку для теста.

Вариант 9. ETL с разными стадиями (Extract, Transform, Load)

Каждая стадия экспортирует свои метрики: stage_duration_seconds{stage="extract"}, аналогично для transform и load. В Grafana создать stacked-панель, показывающую вклад каждой стадии.

Вариант 10. Метрики загрузки в S3 (или MinIO)

ETL загружает результат в S3-совместимое хранилище (MinIO). Добавить метрики: s3_upload_duration_seconds, s3_upload_bytes_total, s3_upload_errors_total. Настроить дашборд.

Вариант 11. Мониторинг инкрементальной загрузки (из ЛР12)

Добавить метрики: incremental_rows_inserted, incremental_rows_updated, incremental_watermark_lag_seconds (разница между текущим временем и последним загруженным обновлением в источнике).

Вариант 12. Пайплайн с параллельной обработкой (multiprocessing)

ETL использует пул процессов для обработки данных. Нужно экспортировать метрики из каждого процесса (использовать multiprocessing.Queue для отправки метрик в главный процесс). Сделать гистограмму времени обработки на процесс.

Вариант 13. Мониторинг очереди задач (Redis / RabbitMQ)

ETL берёт задания из очереди. Метрики: queue_length (Gauge), tasks_processed_total, task_waiting_time_seconds (гистограмма). Настроить алерт при queue_length > 100.

Вариант 14. Экспорт бизнес-метрик

Помимо технических метрик, экспортировать бизнес-метрики: daily_revenue_total, new_customers_count (получаемые из данных). Отобразить их в Grafana как отдельные панели (Stat, Time series). Добавить алерт при падении выручки на 20%.

Вариант 15. Интеграция с OpenTelemetry

Вместо prometheus_client использовать OpenTelemetry SDK (экспорт в Prometheus через OTLP). Реализовать трассировку (traces) для каждого этапа ETL и экспорт метрик через Prometheus. В Grafana добавить связь с Tempo (если есть).

Дополнительное задание для всех вариантов:

Настроить в Grafana анимированный "State timeline" для статуса пайплайна за последние 24 часа и экспорт дашборда в JSON. Предоставить JSON-файл в отчёте.

Требования к отчёту

1. Титульный лист с названием работы, ФИО, группой, номером варианта.
2. Цель и задачи (из методички).
3. Теоретическая часть – мониторинг, метрики, Prometheus, Grafana.
4. Ход выполнения работы:
   - Содержимое docker-compose.yaml и prometheus.yml.
   - Листинг ETL-скрипта с экспортом метрик (указать добавленные метрики и их типы).
   - Команды запуска.
5. Результаты:

- Скриншот веб-интерфейса Prometheus с метриками (например, etl_rows_read_total).
- Скриншот дашборда Grafana с графиками (минимум три панели).
- Демонстрация работы алерта (если реализован).
- Результаты нагрузки (таблица времён выполнения).

6. Ответы на контрольные вопросы.
7. Вывод – как мониторинг повышает надёжность пайплайна, какие инсайты удалось получить из добавленных метрик.

Контрольные вопросы

1. В чём разница между метриками типа Counter и Gauge? Приведите примеры использования каждого в ETL.

2. Что такое гистограмма и зачем нужны _bucket, _sum, _count?

3. Как Prometheus находит целевые эндпоинты для сбора метрик (service discovery)?

4. Почему в Python-скрипте для экспорта метрик запускается отдельный HTTP-сервер на порту 8000?

5. Как можно экспортировать метрики из программы, которая завершается после одного запуска (batch job)?

6. Какие значения по умолчанию имеет гистограмма и как настроить свои bucket'ы?

7. Как в Prometheus запросе получить среднее значение метрики за последний час?

8. Что такое rate() и irate() в PromQL? В чём разница?

9. Как в Grafana настроить уведомление (алерт) на превышение порога времени выполнения?

10. Как защитить эндпоинт /metrics (например, базовой аутентификацией)?

11. Как с помощью лейблов (labels) различать метрики для разных этапов пайплайна (например, stage="extract")?

12. Можно ли использовать один HTTP-сервер для экспорта метрик из нескольких параллельно работающих экземпляров ETL? Как избежать конфликта?

13. Какие альтернативы Prometheus существуют для мониторинга (Zabbix, Datadog, CloudWatch)?

14. Как Grafana получает данные из Prometheus и как часто обновляются графики?

15. Как с помощью метрик можно обнаружить утечку памяти или рост потребления CPU в ETL-скрипте?

Лабораторная работа №14.
Развертывание в облаке (Yandex Cloud)

Цель работы: создать serverless ETL-пайплайн на платформе Yandex Cloud: загрузить CSV-файл в Object Storage, настроить триггер, автоматически запускающий облачную функцию на Python, которая обработает данные из загруженного файла и сохранит результат.

Задачи:

1. Создать в Yandex Cloud бакет (bucket) в Object Storage для хранения входящих и обработанных файлов.
2. Сгенерировать статические ключи доступа для работы с Object Storage через AWS SDK (boto3).
3. Создать сервисный аккаунт и назначить ему необходимые роли (для вызова функции и доступа к бакету).
4. Разработать на Python скрипт для загрузки тестового CSV-файла в созданный бакет.
5. Написать облачную функцию (Cloud Function) на Python, которая при загрузке файла в бакет:
   - скачивает файл по событию триггера
   - выполняет простую трансформацию данных (агрегацию, очистку)
   - сохраняет результат в другой каталог / бакет
6. Создать триггер для Object Storage, который будет вызывать функцию при создании нового объекта в бакете.
7. Протестировать весь пайплайн: загрузить CSV и проверить появление обработанного результата.

Теоретическая часть

Serverless в инжиниринге данных

Serverless-архитектура позволяет запускать код в облаке без управления серверами. Для Data Engineer это означает возможность создавать эластичные

ETL-пайплайны, которые масштабируются автоматически и требуют оплаты только за фактическое время выполнения.

Компоненты Yandex Cloud в этой работе:

Компонент	Аналог в других облаках	Назначение
Object Storage	AWS S3, Google Cloud Storage	Хранение файлов (CSV, Parquet, JSON)
Cloud Functions	AWS Lambda, Google Cloud Functions	Бессерверный запуск кода в ответ на события
Trigger (Object Storage)	S3 Event Notification	Автоматический запуск функции при загрузке файла
Service Account	IAM Role	Безопасное управление правами доступа
Static Access Keys	AWS Access Keys	Аутентификация через API (совместимость с S3)

Архитектура serverless ETL:

User/Client → Upload CSV → Object Storage Bucket

│

↓ (event)

Trigger (Object Storage)

│

↓ (invoke)

Cloud Function (Python)

│

├─ download file

├─ process with pandas

└─ upload result

│

↓

Object Storage (processed/)

Ключевые преимущества для Data Engineer:

- Нет необходимости администрировать сервер
- Автоматическое масштабирование под нагрузку
- Оплата только за время выполнения (в бесплатном гранте Yandex Cloud — десятки тысяч вызовов бесплатно)
- Простота интеграции через S3-совместимый API

Методика выполнения работы

1. **Подготовка облака**

   1.1. Зарегистрируйтесь в Yandex Cloud и создайте платежный аккаунт (можно с пробным грантом).

   1.2. Создайте каталог (например, data-engineering-lab).

   1.3. Установите Yandex Cloud CLI и инициализируйте профиль:

   yc init

   1.4. Установите Python-библиотеку boto3 для работы с Object Storage (S3-совместимый API):

   pip install boto3 pandas

2. **Создание бакета в Object Storage**

   2.1. В консоли управления выберите сервис **Object Storage**.

   2.2. Нажмите **Создать бакет**, укажите уникальное имя (например, etl-lab-username-bucket), выберите параметры доступа (рекомендуется **Ограниченный доступ**).

   2.3. Запомните или запишите имя бакета и эндпоинт https://storage.yandexcloud.net.

3. **Создание сервисного аккаунта и ключей доступа**

3.1. Перейдите в **IAM → Сервисные аккаунты → Создать сервисный аккаунт**. Имя: etl-sa.

3.2. Назначьте роли: storage.editor (для доступа к бакету) и serverless.functions.invoker (для вызова функции).

3.3. На вкладке **Ключи** этого сервисного аккаунта создайте **статический ключ доступа** (для S3). Сохраните key_id и secret.

3.4. **Важно:** статические ключи нужно хранить в безопасном месте, но для учебной цели запишите их во временный .env файл (не коммитьте в Git!).

4. **Загрузка тестового CSV в бакет через boto3**

4.1. Создайте локально папку проекта lab14_yc_serverless и внутри – файл upload_to_storage.py:

```
import boto3
from botocore.client import Config
from dotenv import load_dotenv
import os
import pandas as pd
from io import StringIO

load_dotenv()

AWS_ACCESS_KEY_ID = os.getenv('AWS_ACCESS_KEY_ID')
AWS_SECRET_ACCESS_KEY = os.getenv('AWS_SECRET_ACCESS_KEY')
BUCKET_NAME = 'etl-lab-username-bucket'
ENDPOINT_URL = 'https://storage.yandexcloud.net'

session = boto3.session.Session()
s3 = session.client(
```

```
    service_name='s3',
    endpoint_url=ENDPOINT_URL,
    aws_access_key_id=AWS_ACCESS_KEY_ID,
    aws_secret_access_key=AWS_SECRET_ACCESS_KEY
)

# Создаем тестовый DataFrame
data = {
    'transaction_id': [1, 2, 3, 4, 5],
    'date': ['2025-05-01', '2025-05-01', '2025-05-02', '2025-05-02', '2025-05-03'],
    'product': ['Laptop', 'Mouse', 'Keyboard', 'Laptop', 'Mouse'],
    'quantity': [1, 2, 1, 1, 3],
    'price': [55000, 1500, 3500, 55000, 1500],
    'total': [55000, 3000, 3500, 55000, 4500]
}
df = pd.DataFrame(data)

# Сохраняем DataFrame в CSV (в памяти)
csv_buffer = StringIO()
df.to_csv(csv_buffer, index=False)

# Загружаем в бакет
object_key = 'raw/sales_202505.csv'
s3.put_object(Bucket=BUCKET_NAME, Key=object_key, Body=csv_buffer.getvalue())
print(f'Uploaded {object_key} to bucket {BUCKET_NAME}')
```

4.2. Создайте .env файл:

AWS_ACCESS_KEY_ID=вставить_свой_key_id

AWS_SECRET_ACCESS_KEY=вставить_свой_secret

4.3. Запустите скрипт и убедитесь, что файл появился в бакете (проверить в консоли или через CLI).

5. **Создание облачной функции**

5.1. В консоли выберите сервис **Cloud Functions → Создать функцию**.

5.2. Имя функции: etl-processor.

5.3. После создания перейдите на вкладку **Редактор**, выберите среду выполнения python311 (или python312).

5.4. Напишите код функции, который обрабатывает CSV из события Object Storage:

```
import boto3
import pandas as pd
import io
from datetime import datetime

# Инициализация клиента S3
session = boto3.session.Session()
s3 = session.client(
    service_name='s3',
    endpoint_url='https://storage.yandexcloud.net'
)
# Имя исходного бакета (замените на своё)
SOURCE_BUCKET = 'etl-lab-username-bucket'
DEST_BUCKET = 'etl-lab-username-bucket'
PROCESSED_PREFIX = 'processed/'

def handler(event, context):
```

```
    """
    Функция вызывается триггером при загрузке объекта в бакет.
    Ожидаем, что event содержит messages с информацией о файле.
    """
    print('Received event:', event)

    for message in event['messages']:
        # Извлекаем детали события
        details = message['details']
        object_key = details['object_id']  # ключ загруженного файла

        # Проверяем, что файл находится в папке raw и имеет расширение .csv
        if not object_key.startswith('raw/') or not object_key.endswith('.csv'):
            print(f'Skipping file {object_key}: not in raw/ or not CSV')
            continue

        # Скачиваем файл из бакета
        response = s3.get_object(Bucket=SOURCE_BUCKET, Key=object_key)
        content = response['Body'].read()
        df = pd.read_csv(io.BytesIO(content))
        print(f'Loaded {len(df)} rows from {object_key}')

        # Трансформация данных (агрегация по дате)
        agg_df = df.groupby('date').agg({
            'total': 'sum',
            'quantity': 'sum',
            'transaction_id': 'count'
```

```
        }).rename(columns={'transaction_id':
'num_transactions'}).reset_index()

        # Добавляем метаданные
        agg_df['processed_at'] = datetime.now().isoformat()

        # Сохраняем результат как CSV
        output_buffer = io.StringIO()
        agg_df.to_csv(output_buffer, index=False)

        # Формируем путь для сохранения:
processed/aggregated_<original_file>.csv
        base_name = object_key.split('/')[-1].replace('.csv', '')
        result_key = f'{PROCESSED_PREFIX}aggregated_{base_name}.csv'
        s3.put_object(Bucket=DEST_BUCKET, Key=result_key,
Body=output_buffer.getvalue())
        print(f'Saved aggregated result to {result_key}')

    return {'statusCode': 200, 'body': 'Processing completed'}
```

5.5. Укажите **Точку входа**: index.handler (если файл называется index.py).

5.6. В разделе **Параметры → Переменные окружения** добавьте:

- AWS_ACCESS_KEY_ID (значение из статического ключа)
- AWS_SECRET_ACCESS_KEY
- AWS_DEFAULT_REGION = ru-central1

Процесс добавления переменных описан в документации Cloud Functions.

5.7. Создайте **версию функции** (нажмите «Создать версию»). Если код содержит библиотеку pandas:

- Для Python 3.11 и выше pandas предустановлена — достаточно просто указать pandas в **requirements.txt** (вкладка «Дополнительные настройки»).
- Для старых версий pandas не предустановлена — нужно создать ZIP-архив с кодом и requirements.txt, содержащим pandas, boto3.

6. **Создание триггера на Object Storage**

   6.1. В сервисе **Cloud Functions** перейдите на вкладку **Триггеры**.

   6.2. Нажмите **Создать триггер**.

   6.3. Заполните:

   - Имя: storage-trigger
   - Тип: **Object Storage**
   - Бакет: выберите ваш созданный ранее бакет
   - Типы событий: **Создание объекта**
   - Запускаемый ресурс: выберите созданную функцию etl-processor
   - Сервисный аккаунт: выберите ранее созданный etl-sa (или другой с правами на вызов функции).

   6.4. Нажмите **Создать**.

7. **Тестирование пайплайна**

   7.1. Запустите скрипт upload_to_storage.py — новый CSV появится в бакете в папке raw/.

   7.2. Откройте логи функции в консоли (Cloud Functions → Ваша функция → Логи). Вы должны увидеть строки Loaded X rows from ... и Saved aggregated result to ....

   7.3. Перейдите в ваш бакет и откройте папку processed/. Найдите там CSV-файл с агрегированными данными.

   7.4. Скачайте результат и убедитесь в корректности трансформации.

Пример выполнения задания

Исходный CSV (загруженный пользователем):

transaction_id	date	product	quantity	price	total
1	2025-05-01	Laptop	1	55000	55000
2	2025-05-01	Mouse	2	1500	3000
3	2025-05-02	Keyboard	1	3500	3500
4	2025-05-02	Laptop	1	55000	55000
5	2025-05-03	Mouse	3	1500	4500

Обработанный результат (Cloud Function):

date	total	quantity	num_transactions	processed_at
2025-05-01	58000	3	2	2026-05-12T14:30:25.123456
2025-05-02	58500	2	2	2026-05-12T14:30:25.123456
2025-05-03	4500	3	1	2026-05-12T14:30:25.123456

Лог функции (фрагмент):

2026-05-12 14:30:23 Received event: {...}

2026-05-12 14:30:23 Loaded 5 rows from raw/sales_202505.csv

2026-05-12 14:30:24 Saved aggregated result to processed/aggregated_sales_202505.csv

2026-05-12 14:30:24 Processing completed

Индивидуальные задания

Каждый вариант описывает специфику входного файла, необходимую трансформацию и формат вывода. Студент должен адаптировать код облачной функции под свой вариант.

Вариант 1. Фильтрация транзакций

Входной CSV: txn_id,user_id,amount,status,currency. Трансформация: отфильтровать транзакции с amount > 10000, сохранить в папку filtered/ как CSV.

Вариант 2. Конвертация валют

Входной CSV: product,price_usd,quantity,sale_date. Трансформация: добавить колонку price_rub (используя фиксированный курс 1 USD = 95 RUB). Результат сохранить в converted/ как CSV.

Вариант 3. Вычисление скользящего среднего

Входной CSV: date,metric_value. Трансформация: добавить колонку moving_avg_3 (скользящее среднее за 3 дня). Сохранить в analysis/.

Вариант 4. Обнаружение дубликатов

Входной CSV: id,field1,field2,timestamp. Трансформация: удалить дубликаты по id, оставив запись с максимальным timestamp. Сохранить в deduplicated/ как CSV.

Вариант 5. Разделение по категориям

Входной CSV: order_id,category,amount,order_date. Трансформация: разделить данные на несколько CSV-файлов (по одному на категорию) в папке by_category/.

Вариант 6. Очистка текста

Входной CSV: review_id,username,review_text,rating. Трансформация: привести review_text к нижнему регистру, удалить пунктуацию и сохранить в cleaned_reviews/.

Вариант 7. Аномалии (Z-score)

Входной CSV: sensor_id,reading,timestamp. Трансформация: пометить строки, где чтение отличается от среднего более чем на 3 стандартных отклонения (аномалии), сохранить в anomalies/.

Вариант 8. Выгрузка в Parquet

Входной CSV: любой. Трансформация: преобразовать CSV в формат Parquet (более эффективное хранение). Сохранить в parquet/ с тем же именем.

Вариант 9. JSON → CSV

На вход подаётся не CSV, а JSON-файл (массив объектов). Трансформация: распарсить JSON и сохранить как CSV в converted/ (поменять функцию соответственно).

Вариант 10. Сравнение с эталоном

Входной CSV: product_code,stock. В бакете уже есть файл reference/prices.csv (product_code,price). Трансформация: объединить (join) с эталоном, добавить колонку total_value = stock * price. Сохранить в enriched/.

Вариант 11. Pivot-таблица

Входной CSV: date,category,sales. Трансформация: создать сводную таблицу (pivot), где строки – даты, столбцы – категории, значения – сумма sales. Сохранить как pivot_table.csv в папке aggregated/.

Вариант 12. Логирование результатов в другую таблицу

В дополнение к сохранению CSV, направить итоги агрегации (общее количество строк, сумма) в лог. Использовать print() или формальное логирование средствами Cloud Functions (вывод в stdout).

Вариант 13. Инкрементальная обработка

Функция должна обрабатывать только файлы, чьё имя содержит текущую дату (YYYY-MM-DD). Если файл старый — игнорировать. Сохранять результат с суффиксом _processed.

Вариант 14. Ограничение по размеру

Если входной CSV превышает 5 МБ, функция не обрабатывает его, а записывает уведомление в errors/size_exceeded.txt. Обработку реализовать через проверку размера объекта в S3.

Вариант 15. Переодическая переобработка

Триггер срабатывает не только при создании, но и при обновлении объекта (событие UpdateObject). При обновлении функция перезаписывает предыдущий результат в processed/.

Дополнительное задание (для всех вариантов):

Добавить в облачную функцию генерацию **файла-маркера** (ready.txt), который создаётся в бакете после успешной обработки. Это уведомляет downstream-системы о готовности данных.

Требования к отчёту

1. Титульный лист с названием работы, ФИО, группой, номером варианта.
2. Цель и задачи (из методички).
3. Теоретическая часть – serverless, Object Storage, Cloud Functions, триггеры.
4. Ход выполнения работы:
   - Скриншоты создания бакета и сервисного аккаунта в консоли.
   - Листинг скрипта upload_to_storage.py.
   - Код облачной функции (целиком) с комментариями.
   - Конфигурация триггера (скриншот из консоли).
5. Результаты:
   - Скриншот успешной загрузки файла в бакет (папка raw/).
   - Лог выполнения Cloud Function (скопированный текст или скриншот).
   - Скриншот результата обработки (файл в папке processed/ или другой по варианту).

- Вывод SELECT * FROM processed_data (или содержимое CSV-результата).

6. Ответы на контрольные вопросы.
7. Вывод – в чем преимущества serverless-подхода для ETL, какие потенциальные проблемы могут возникнуть при масштабировании.

Контрольные вопросы

1. В чем основное отличие между serverless-функцией и веб-приложением на постоянно запущенном сервере?

2. Почему для работы с Yandex Object Storage из Python используется AWS SDK (boto3)?

3. Какая роль у сервисного аккаунта в контексте облачной функции? Какие роли нужно выдать для доступа к бакету?

4. Как Cloud Function получает данные о событии (какой файл загрузили)?

5. Что будет, если в функцию одновременно придут два события (два файла загружены в бакет)?

6. Как ограничить максимальное время выполнения функции и что делать, если обработка файла занимает больше времени?

7. Какие еще типы триггеров, кроме Object Storage, могут быть полезны для Data Engineer?

8. Как защитить статические ключи доступа от компрометации (помимо .env)?

9. Какой максимальный размер файла можно обработать в Cloud Function? Что делать, если файл больше?

10. Как отлаживать ошибки в облачной функции, если нет локального запуска?

11. Как настроить Dead Letter Queue (DLQ) для обработки сообщений, которые не удалось обработать?

12. В чем разница между хранением обработанных данных обратно в Object Storage и записью в Managed ClickHouse?

13. Как можно автоматизировать развертывание Cloud Function через Terraform или YC CLI?

14. Как проверить, что функция действительно обработала файл (кроме просмотра логов)?

15. Какие альтернативы Yandex Cloud существуют для serverless ETL (AWS Lambda, Google Cloud Functions, Яндекс.Облако vs AWS S3)?

Лабораторная работа №15.
CI/CD для пайплайнов с GitHub Actions

Цель работы: настроить автоматическое тестирование ETL-скриптов с использованием GitHub Actions: при каждом пуше в репозиторий запускать workflow, который устанавливает зависимости, выполняет линтеры и запускает pytest на тестовых данных.

Задачи:

1. Создать репозиторий на GitHub и клонировать его локально.
2. Структурировать ETL-проект с поддержкой тестов (папка src, tests, requirements.txt).
3. Написать простой ETL-скрипт на Python с функцией трансформации (например, очистка и агрегация).
4. Написать unit-тесты с использованием pytest для проверки логики трансформации на sample-данных.
5. Создать файл GitHub Actions workflow (.github/workflows/ci.yml), который:
   - запускается при push в ветки main и при pull request
   - устанавливает Python и зависимости из requirements.txt
   - выполняет линтер (например, flake8 или black --check)
   - запускает pytest и генерирует отчёт
6. Проверить, что при успешных тестах workflow зелёный, а при падении – красный.
7. Добавить в workflow шаг для проверки качества кода (например, pylint или bandit для безопасности).

Теоретическая часть

CI/CD (Continuous Integration / Continuous Deployment) – практика автоматизации сборки, тестирования и развёртывания приложений. Для инженера данных CI/CD означает, что каждый коммит в репозиторий с ETL-

кодом автоматически проверяется на корректность, что предотвращает попадание сломанных пайплайнов в production.

GitHub Actions – встроенная платформа CI/CD в GitHub. Workflow описывается в YAML-файле в папке .github/workflows/. Каждый workflow состоит из событий (push, pull_request, schedule), джобов (jobs) и шагов (steps).

Ключевые понятия:

Термин	Описание
Workflow	Автоматизируемый процесс, состоящий из одного или нескольких job'ов.
Job	Группа шагов, выполняющихся на одном раннере (runner).
Step	Отдельная команда или action, выполняемая в job'е.
Runner	Виртуальная машина, на которой выполняется workflow (Ubuntu, Windows, macOS).
Action	Переиспользуемый блок кода (например, actions/checkout@v4, actions/setup-python@v5).

Типичный CI-пайплайн для ETL-проекта:

push → checkout code → setup Python → install dependencies → lint → test → (optional) build Docker → (optional) deploy to staging

Почему CI важен для ETL:

- Обнаруживает ошибки в логике трансформации до развёртывания.
- Проверяет, что код совместим с разными версиями Python и библиотек.
- Автоматизирует проверку стиля кода (PEP8).
- Позволяет безопасно рефакторить и добавлять новые фичи.

Методика выполнения работы

1. **Подготовка локального репозитория**

   1.1. Создайте новый репозиторий на GitHub (например, etl-ci-demo).

   1.2. Клонируйте его локально:

   git clone https://github.com/your-username/etl-ci-demo.git

   cd etl-ci-demo

   1.3. Создайте структуру проекта:

   ├── .github/workflows/

   ├── src/

   │ └── etl_transform.py

   ├── tests/

   │ └── test_etl_transform.py

   ├── data/

   │ └── sample_input.csv

   ├── requirements.txt

   └── README.md

2. **Разработка ETL-скрипта**

   2.1. В файл src/etl_transform.py поместите:

```
import pandas as pd


def clean_data(df):
    # Удаление строк с пропусками в важных колонках
    df = df.dropna(subset=['transaction_id', 'amount'])
    # Фильтрация отрицательной суммы
    df = df[df['amount'] > 0]
    # Приведение типов
    df['date'] = pd.to_datetime(df['date'], errors='coerce')
    df = df.dropna(subset=['date'])
    return df
```

```
def aggregate_by_date(df):
    agg = df.groupby(df['date'].dt.date).agg(
        total_amount=('amount', 'sum'),
        transaction_count=('transaction_id', 'count')
    ).reset_index()
    agg.rename(columns={'date': 'date'}, inplace=True)
    return agg

def run_etl(input_path, output_path):
    df = pd.read_csv(input_path)
    df_clean = clean_data(df)
    df_agg = aggregate_by_date(df_clean)
    df_agg.to_csv(output_path, index=False)
```

2.2. В файл data/sample_input.csv добавьте тестовые данные:

```
transaction_id,date,amount
1,2025-05-01,100
2,2025-05-01,-50
3,2025-05-02,200
4,,300
5,2025-05-03,0
```

(Содержит некорректные строки для проверки очистки)

3. **Написание тестов с pytest**

3.1. Установите pytest и pandas:

```
pip install pytest pandas
```

3.2. В requirements.txt укажите:

```
pandas
```

pytest

flake8

3.3. Создайте tests/test_etl_transform.py:

```
import pytest
import pandas as pd
from src.etl_transform import clean_data, aggregate_by_date

def test_clean_data_removes_negative_and_null():
    input_df = pd.DataFrame({
        'transaction_id': [1, 2, 3, 4],
        'date': ['2025-01-01', '2025-01-01', None, '2025-01-02'],
        'amount': [100, -20, 300, 0]
    })
    result = clean_data(input_df)
    # после очистки должны остаться только строки с положительным amount и непустой датой
    assert len(result) == 1  # только первая строка (id=1)
    assert result.iloc[0]['transaction_id'] == 1

def test_aggregate_by_date():
    df = pd.DataFrame({
        'date': pd.to_datetime(['2025-01-01', '2025-01-01', '2025-01-02']),
        'amount': [100, 50, 200],
        'transaction_id': [1, 2, 3]
    })
    agg = aggregate_by_date(df)
    assert agg['total_amount'].iloc[0] == 150
    assert agg['transaction_count'].iloc[0] == 2
    assert agg['total_amount'].iloc[1] == 200
```

4. **Создание GitHub Actions workflow**

4.1. Создайте папку .github/workflows/ и внутри файл ci.yml:

```
name: CI for ETL pipeline

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  test:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        python-version: ['3.9', '3.10', '3.11']

    steps:
    - uses: actions/checkout@v4

    - name: Set up Python ${{ matrix.python-version }}
      uses: actions/setup-python@v5
      with:
        python-version: ${{ matrix.python-version }}

    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install -r requirements.txt
```

```
    - name: Lint with flake8
      run: |
        flake8 src/ tests/ --count --select=E9,F63,F7,F82 --show-source --statistics
        flake8 src/ tests/ --count --exit-zero --max-complexity=10 --max-line-length=127 --statistics

    - name: Test with pytest
      run: |
        pytest tests/ --verbose --tb=short
```

4.2. Добавьте файл .flake8 в корень для конфигурации линтера (опционально):

```
ini
[flake8]
max-line-length = 120
exclude = .git,__pycache__,venv
```

5. **Фиксация и проверка workflow**

5.1. Выполните команды:

```
git add .
git commit -m "Add ETL script, tests and CI workflow"
git push origin main
```

5.2. Перейдите на GitHub → ваш репозиторий → вкладка Actions. Вы увидите запущенный workflow.

5.3. Если тесты прошли, появится зелёная галочка. Если нет – красный крестик и логи ошибок.

6. **Добавление дополнительных шагов (опционально)**

6.1. Добавьте шаг для проверки покрытия тестами (pytest-cov):

```
- name: Test with coverage
  run: |
    pip install pytest-cov
    pytest tests/ --cov=src --cov-report=xml
- name: Upload coverage to Codecov (optional)
  uses: codecov/codecov-action@v4
  with:
    file: ./coverage.xml
```

6.2. Добавьте статический анализатор безопасности bandit:

```
yaml
- name: Security linter (bandit)
  run: |
    pip install bandit
    bandit -r src/ -ll
```

Пример выполнения задания

Результат успешного CI-запуска (фрагмент лога):

```
Run pytest tests/ --verbose --tb=short
============================= test session starts ==============================
platform linux -- Python 3.10.12, pytest-8.0.0, pluggy-1.0.0
rootdir: /home/runner/work/etl-ci-demo
collected 2 items

tests/test_etl_transform.py::test_clean_data_removes_negative_and_null PASSED
tests/test_etl_transform.py::test_aggregate_by_date PASSED

============================== 2 passed in 0.32s ===============================
```

Зелёный статус на GitHub Actions.

При внесении ошибки (например, удалить строку фильтрации amount > 0**):** один тест падает, workflow становится красным, и автор получает уведомление.

Индивидуальные задания

Каждый вариант требует реализации ETL-функции и набора тестов для неё, а затем настройки CI-пайплайна, который автоматически проверяет эти тесты.

Вариант 1. Фильтрация выбросов IQR

Функция remove_outliers_iqr(df, column) удаляет строки, где значение в column выходит за пределы 1.5*IQR. Написать тесты: проверка при нормальном распределении, выбросах, пустом датафрейме.

Вариант 2. Конвертация единиц измерения

Функция convert_units(df, from_col, to_col, conversion_factor). Тесты: корректность конвертации, обработка отрицательных значений, отсутствие деления на ноль.

Вариант 3. Заполнение пропусков медианой

Функция fill_na_with_median(df, columns). Тесты: медиана считается правильно, исходные не-NaN не меняются, выбор колонок.

Вариант 4. Проверка формата email

Функция validate_email(df, email_col) добавляет колонку email_valid (bool) на основе регулярного выражения. Тесты: валидные email, невалидные, пропуски.

Вариант 5. Оконные функции (SQL-like)

Функция add_lag_column(df, group_col, order_col, value_col, lag=1). Тесты: корректность лага для групп, обработка границ (первая строка -> NaN).

Вариант 6. Дедупликация с сохранением последней

Функция deduplicate_last(df, key_col, order_col). Тесты: дубликаты удаляются, остаётся запись с максимальным order_col, исходный порядок не важен.

Вариант 7. Python-скрипт для загрузки API

ETL извлекает данные из JSONPlaceholder (тестовое API) и сохраняет в CSV. Unit-тесты должны мокать requests.get и проверять обработку ошибок, пагинацию.

Вариант 8. Интеграция с PostgreSQL

Функция write_to_db(df, table_name, conn_string) вставляет DataFrame в БД. Тесты используют in-memory SQLite или контейнер с PostgreSQL (через pytest fixtures). CI должен поднимать тестовую БД через Docker Compose.

Вариант 9. Генерация md5-хеша строки

Функция add_hash_column(df, columns_to_hash, new_col='row_hash'). Тесты: одинаковые строки дают одинаковый хеш, изменение порядка колонок – хеш меняется.

Вариант 10. Проверка ссылочной целостности

Функция check_referential_integrity(df, foreign_key, reference_df, reference_key). Тесты: находит сиротские записи, возвращает процент несоответствий.

Вариант 11. Разделение на train/test

Функция split_dataset(df, target_col, train_ratio=0.7, random_state=42). Тесты: пропорции соблюдаются, стратификация по целевому столбцу (опционально).

Вариант 12. Агрегация с несколькими метриками

Функция aggregate_metrics(df, group_cols, agg_dict). Тесты: например, {'sales': 'sum', 'count': 'count', 'price': 'mean'}.

Вариант 13. Обнаружение сдвига распределения (простой тест Колмогорова-Смирнова)

Функция compare_distributions(df, col, ref_df, threshold=0.05). Тесты: возвращает True/False, проверяет на идентичных и разных распределениях.

Вариант 14. Логирование метрик выполнения

Функция-декоратор @log_etl_metrics, которая логирует время выполнения и количество строк на входе/выходе. Тесты: проверка, что логи пишутся (можно мокать logging).

Вариант 15. JSON-валидация по схеме

Функция validate_json_schema(json_data, schema). Тесты: валидные/невалидные JSON, проверка подсхем, тип полей.

Дополнительное задание для всех вариантов:

Добавить в CI workflow шаг, который автоматически вычисляет покрытие тестами (pytest-cov) и проверяет, что оно не ниже 80%. Если покрытие ниже – workflow падает. Результат покрытия выводить в виде аннотации к Pull Request.

Требования к отчёту

1. Титульный лист с названием работы, ФИО, группой, номером варианта.
2. Цель и задачи (из методички).
3. Теоретическая часть – CI/CD, GitHub Actions, тестирование ETL.
4. Ход выполнения работы:
   - Структура репозитория (скриншот дерева).
   - Листинг кода ETL-функции (из src/).
   - Листинг тестов (из tests/).
   - Содержимое файла .github/workflows/ci.yml.
   - Команды и процесс коммита.
5. Результаты:
   - Скриншот успешного запуска workflow на GitHub Actions.
   - Скриншот лога с прохождением тестов.
   - При наличии покрытия – отчёт coverage.

   - Демонстрация падения workflow при внесении ошибки (по желанию).
6. Ответы на контрольные вопросы.
7. Вывод – как CI/CD повышает надёжность разработки ETL, что было самым сложным при настройке.

Контрольные вопросы

1. Что такое CI/CD и зачем он нужен в проектах по инжинирингу данных?

2. Какие события могут запускать workflow в GitHub Actions?

3. В чём разница между push и pull_request триггерами?

4. Как в workflow можно протестировать код на нескольких версиях Python?

5. Для чего нужен шаг actions/checkout@v4?

6. Что делает pytest? Как написать тест, который проверяет, что функция возвращает DataFrame с определёнными колонками?

7. Как запустить тесты, требующие доступа к базе данных или внешнему API, внутри CI-среды?

8. Как просмотреть логи упавшего workflow и найти ошибку?

9. Зачем в CI-пайплайне запускать линтер (flake8) и анализатор безопасности (bandit)?

10. Как секретные переменные (пароли, ключи API) передать в workflow безопасно?

11. Что такое матричная стратегия (matrix strategy) для параллельного запуска job'ов?

12. Как ограничить запуск workflow только для изменённых файлов (paths-ignore)?

13. Как настроить уведомления о результате CI (email, Slack, Telegram)?

14. Как в GitHub Actions кэшировать зависимости (pip cache) для ускорения сборки?

15. Как можно развернуть ETL-скрипт в облачную функцию после успешного прохождения тестов (CI/CD с деплоем)?

Лабораторная работа №16.
Feature store для ML (Feast)

Цель работы: подготовить features для модели машинного обучения с использованием feature store Feast: зарегистрировать источник данных (файл или таблицу), определить feature views, материализовать features в онлайн-хранилище (Redis) и получить их для инференса.

Задачи:

1. Изучить концепцию feature store и его роль в MLOps.
2. Установить Feast и запустить локальную инфраструктуру (PostgreSQL для онлайнового хранилища или Redis через Docker).
3. Подготовить исторические данные (CSV-файл или таблицу в БД) с признаками для обучения модели.
4. Определить feature definitions (сущности, источники данных, feature views) в конфигурационных файлах или Python-коде.
5. Настроить материализацию признаков в онлайн-хранилище для низколатентного доступа.
6. Закрепить (materialize) features за выбранный временной диапазон.
7. Написать скрипт для получения online features по ключу сущности (например, customer_id) и продемонстрировать их использование.
8. Показать, как полученные features могут быть переданы в ML-модель для предсказания.

Теоретическая часть

Feature store – центральное хранилище для управления, версионирования и доставки признаков (features) в ML-пайплайнах. Решает проблему разрыва между обучением модели (используются исторические данные) и инференсом (нужны актуальные признаки с минимальной задержкой).

Основные компоненты feature store:

Компонент	Назначение
Source (источник)	Место хранения сырых данных (файл, таблица в БД, Kafka).
Entity (сущность)	Ключ, по которому объединяются признаки (например, customer_id, product_id).
Feature View	Логическое описание набора признаков, их преобразований и временных окон.
Online Store	Быстрое (In-Memory / KV) хранилище для онлайн-признаков (Redis, DynamoDB).
Offline Store	Хранилище для больших исторических данных (Parquet, BigQuery, Snowflake).
Materialization	Процесс копирования признаков из offline в online хранилище за определённый интервал.

Feast (Feature Store для машинного обучения) – open-source проект с поддержкой локального запуска, интеграцией с cloud и автономной работой. Основные шаги работы с Feast:

1. **Define** – описать сущности, источники данных, feature views в файле feature_store.yaml и Python-файлах.
2. **Apply** – применить конфигурацию к хранилищу метаданных.
3. **Materialize** – перенести признаки из offline в online хранилище.
4. **Retrieve** – получить features для обучения (offline) или инференса (online).

Архитектура типового использования:

Historical data (CSV/Parquet) → Feast offline store → Training data (модель)

↑

New data (Kafka/API) → Feast ingestion → Online store → Feature retrieval (инференс)

Пример определения feature view в Feast:

```
from feast import Entity, FeatureView, FeatureService, Source, Field
from feast.types import Float32, Int64
from datetime import timedelta

customer_entity = Entity(name="customer", join_keys=["customer_id"])

customer_source = FileSource(
    path="data/customers.parquet",
    event_timestamp_column="event_timestamp",
)

customer_features = FeatureView(
    name="customer_features",
    entities=[customer_entity],
    ttl=timedelta(days=30),
    schema=[
        Field(name="age", dtype=Int64),
        Field(name="total_spent", dtype=Float32),
        Field(name="avg_cart_value", dtype=Float32),
    ],
    source=customer_source,
)
```

Методика выполнения работы

1. **Подготовка окружения**

   1.1. Создайте каталог lab16_feast

   1.2. Установите Feast и необходимые библиотеки:

```
pip install feast pandas redis
```

1.3. Запустите Redis (онлайн-хранилище) в Docker:

```
docker run -d --name feast-redis -p 6379:6379 redis
```

2. **Инициализация Feast-проекта**

2.1. Выполните команду:

```
feast init feature_repo
cd feature_repo
```

2.2. Отредактируйте feature_store.yaml для использования Redis в качестве online store:

```
project: fraud_detection
registry: data/registry.db
provider: local
online_store:
  type: redis
  connection_string: "localhost:6379"
offline_store:
  type: file
```

3. **Подготовка исходных данных**

3.1. В папке data/ создайте файл customer_features.parquet (можно из CSV). Используем Pandas для генерации:

```
import pandas as pd
import numpy as np
from datetime import datetime, timedelta

np.random.seed(42)
n = 100
data = {
  'customer_id': [f'cust_{i}' for i in range(1, n+1)],
```

```
    'event_timestamp': [datetime(2024, 5, 1) + timedelta(days=np.random.randint(0,30)) for _ in range(n)],
    'age': np.random.randint(18, 70, n),
    'total_spent': np.round(np.random.uniform(100, 10000, n), 2),
    'avg_cart_value': np.round(np.random.uniform(10, 500, n), 2),
    'num_transactions': np.random.randint(1, 100, n)
}
df = pd.DataFrame(data)
df.to_parquet('data/customer_features.parquet', index=False)
```

4. **Определение feature view и сущности**

4.1. Создайте файл features.py в корне проекта:

```
from datetime import timedelta
import pandas as pd
from feast import Entity, FeatureView, FeatureService, Field, FileSource, ValueType
from feast.types import Int64, Float32

# Определяем сущность (ключ)
customer_entity = Entity(
    name="customer",
    join_keys=["customer_id"],
    description="Customer identifier",
)

# Источник данных - parquet файл
customer_source = FileSource(
    name="customer_parquet_source",
    path="data/customer_features.parquet",
    timestamp_field="event_timestamp",
```

```
    file_format="parquet",
)

# Feature view - набор признаков
customer_features = FeatureView(
    name="customer_features",
    entities=[customer_entity],
    ttl=timedelta(days=30),
    schema=[
        Field(name="age", dtype=Int64),
        Field(name="total_spent", dtype=Float32),
        Field(name="avg_cart_value", dtype=Float32),
        Field(name="num_transactions", dtype=Int64),
    ],
    source=customer_source,
    online=True,
)
```

4.2. (Опционально) Добавьте FeatureService для группировки фичей:

```
python
customer_feature_service = FeatureService(
    name="customer_service", features=[customer_features],
)
```

5. **Применение конфигурации и материализация**

5.1. Выполните команды Feast:

```
feast apply   # регистрация сущностей, view и т.д.
feast materialize-incremental 2024-06-01   # материализовать все данные до 2024-06-01
```

(Замените дату на актуальную позже, чем максимальная дата в данных)

6. **Получение online features**

6.1. Создайте файл get_online_features.py:

```
import pandas as pd
from feast import FeatureStore

store = FeatureStore(repo_path=".")

# Запрос признаков для конкретных сущностей
entity_rows = [
    {"customer": "cust_1"},
    {"customer": "cust_2"},
    {"customer": "cust_5"}
]

features = store.get_online_features(
    features=[
        "customer_features:age",
        "customer_features:total_spent",
        "customer_features:avg_cart_value",
        "customer_features:num_transactions",
    ],
    entity_rows=entity_rows,
).to_df()

print("Retrieved online features:")
print(features)
```

6.2. Запустите скрипт, убедитесь, что выданы корректные значения.

7. **Интеграция с ML-моделью (симуляция)**

7.1. Напишите мини-модель, которая предсказывает склонность к оттоку на основе полученных признаков.

```
import pickle
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split

# Предположим, что у нас есть исторические данные с меткой churn
# Загружаем training set из того же parquet (добавим случайную метку для демо)
df_full = pd.read_parquet("data/customer_features.parquet")
df_full['churn'] = (df_full['total_spent'] < 500) & (df_full['num_transactions'] < 5)
X = df_full[['age', 'total_spent', 'avg_cart_value', 'num_transactions']]
y = df_full['churn'].astype(int)
X_train, X_test, y_train, y_test = train_test_split(X, y)
model = RandomForestClassifier().fit(X_train, y_train)

# Сохраняем модель
with open("model.pkl", "wb") as f:
    pickle.dump(model, f)

# Используем онлайн-признаки для предсказания
import pickle
with open("model.pkl", "rb") as f:
    clf = pickle.load(f)
preds = clf.predict(features[['age', 'total_spent', 'avg_cart_value', 'num_transactions']])
features['churn_prediction'] = preds
print("Predictions with online features:")
print(features[['customer', 'age', 'total_spent', 'churn_prediction']])
```

8. **Автоматизация через Airflow (опционально)**

   8.1. Покажите, как добавить задачу материализации в DAG Airflow с использованием FeastOperator или BashOperator (команда feast materialize).

Пример выполнения задания

Исходные данные (customer_features.parquet**)** – 100 записей, каждая с полями:

- customer_id (строковый идентификатор)
- event_timestamp (дата события)
- age, total_spent, avg_cart_value, num_transactions

После выполнения feast apply **и** feast materialize-incremental в Redis записываются признаки для всех customer.

Запрос get_online_features.py **возвращает DataFrame:**

	customer	age	total_spent	avg_cart_value	num_transactions
0	cust_1	25	5300.2	276.8	32
1	cust_2	43	1210.5	105.4	12
2	cust_5	31	8760.0	420.9	56

Модель предсказывает churn_prediction = 1 для cust_2 (риск оттока).

Вывод: Feast предоставляет унифицированный способ доступа к признакам как для обучения (offline), так и для инференса (online), что устраняет расхождение признаков (training–serving skew).

Индивидуальные задания

Каждый вариант описывает предметную область и набор признаков. Студент должен создать parquet-файл, определить feature views, материализовать и получить признаки для одного-двух ключей сущности.

Вариант 1. Признаки клиента банка

Сущность: client_id.

Признаки: age, income, credit_score, number_of_loans, has_mortgage. Выдать признаки для трёх клиентов.

Вариант 2. Признаки транзакций для фрод-модели

Сущность: transaction_id.

Признаки:amount, merchant_category, time_since_last_txn, txn_count_1h, is_foreign. Материализовать за последние сутки.

Вариант 3. Поведенческие признаки пользователя с мобильного приложения

Сущность: user_id.

Признаки: session_duration_avg, button_clicks_count_7d, push_enabled, app_version. Источник – ежедневные паркеты.

Вариант 4. Признаки товара для recommender system

Сущность: product_id.

Признаки: price, category, avg_rating, number_of_reviews, days_since_last_purchase.

Вариант 5. Риск-факторы по водителям (страхование)

Сущность: driver_id.

Признаки: age, driving_experience_years, accident_count_last_year, fine_points, car_power.

Вариант 6. IoT-датчики (индустриальное оборудование)

Сущность: sensor_id.

Признаки: avg_temperature_1h, max_vibration, rpm, power_consumption, maintenance_flag. Использовать оконные агрегации через Feast (request source).

Вариант 7. Оптимизация доставки

Сущность: delivery_person_id.

Признаки: total_deliveries_today, avg_delivery_time_min, current_orders_count, vehicle_type, zone.

Вариант 8. Медицинская диагностика

Сущность: patient_id.

Признаки: age, bmi, blood_pressure_systolic, cholesterol_level, smoker.

Вариант 9. Credit scoring на основе истории

Сущность: application_id.

Признаки: annual_income, debt_to_income, loan_amount, credit_history_length, num_inquiries.

Вариант 10. Предсказание оттока телеком-абонентов

Сущность: subscriber_id.

Признаки: call_minutes_per_month, data_usage_gb, num_complaints, months_since_last_payment, contract_type.

Вариант 11. Аналитика спортивных игроков

Сущность: player_id.

Признаки: goals_per_match, assists, tackles, fitness_score, injured.

Вариант 12. Признаки для модели ценообразования

Сущность: flight_route.

Признаки: distance, demand_index, competitor_price, days_until_departure, season.

Вариант 13. Хотел-рекомендации

Сущность: hotel_id.

Признаки: star_rating, review_score, distance_to_center, free_wifi, price_night_usd.

Вариант 14. Признаки качества кода для предсказания багов

Сущность: commit_hash.

Признаки: num_lines_added, num_lines_deleted, files_changed, author_experience_days, num_previous_bugs.

Вариант 15. Финансовые мульти-активы

Сущность: asset_id.

Признаки: price_ma_7d, volume, volatility, rsi, market_cap.

Дополнительное задание для всех вариантов:

Добавить в пайплайн Feast FeatureService для группировки трёх разных feature views (например, базовые признаки + агрегированные за 7 дней + поведенческие). Материализовать все три и проверить совместное получение через get_online_features.

Требования к отчёту

1. Титульный лист с названием работы, ФИО, группой, номером варианта.
2. Цель и задачи (из методички).
3. Теоретическая часть – назначение feature store, компоненты Feast, offline/online store.
4. Ход выполнения работы:
   - Команды инициализации Feast, запуска Redis.
   - Код генерации данных (parquet).
   - Определение feature view и сущности (листинг features.py).
   - Команды feast apply, feast materialize.
5. Результаты:
   - Скриншот вывода feast apply (или лог).
   - Скриншот/текст результата get_online_features.py.
   - Демонстрация интеграции с ML-моделью (предсказания для выбранных сущностей).
   - (Опционально) Grafana или дашборд мониторинга feature store.
6. Ответы на контрольные вопросы.
7. Вывод – как feature store решает проблему управления признаками и почему это важно для MLOps.

Контрольные вопросы

1. Что такое training–serving skew и как feature store помогает его избежать?

2. Чем отличается online store от offline store в Feast?

3. Какую роль выполняет entity в Feast? Приведите пример сложного составного ключа.

4. Для чего нужна материализация (materialization) признаков?

5. Как обновить уже материализованные признаки после добавления новых данных?

6. Какие online store поддерживаются Feast (кроме Redis)?

7. Как Feast гарантирует консистентность признаков во время инференса?

8. Можно ли использовать Feast без онлайн-хранилища (только для тренировки)?

9. Как в Feast реализовать вычисление скользящего окна (например, средний чек за последние 7 дней)?

10. Как подключить Feast к Apache Kafka для стриминговых feature views?

11. Что такое feature retrieval и как он выполняется в Feast с помощью get_online_features?

12. Как обеспечить авторизацию к Feast-метаданным и онлайн-стореджу?

13. Как интегрировать Feast с оркестратором (Airflow, Prefect) для периодической материализации?

14. Какие альтернативы Feast существуют (Tecton, Hopsworks, Amazon SageMaker Feature Store)?

15. Как можно тестировать feature views (юнит-тесты на фикстурах) в CI/CD пайплайне?

Лабораторная работа №17.

Интеграция с BI-инструментом (Apache Superset)

Цель работы: построить интерактивный дашборд на основе витрины данных (gold-слоя) с использованием Apache Superset: подключиться к хранилищу данных (PostgreSQL), создать набор визуализаций (диаграммы, таблицы, фильтры) и опубликовать дашборд.

Задачи:

1. Развернуть Apache Superset в Docker-контейнере.
2. Подготовить витрину данных в PostgreSQL (таблица фактов и измерения) для анализа продаж или другой предметной области.
3. Подключить Superset к PostgreSQL как источник данных (Database Connection).
4. Создать Dataset (набор данных) на основе SQL-запроса или таблицы витрины.
5. Построить несколько типов визуализаций:
   - временной ряд (линейная диаграмма продаж по дням)
   - столбчатая диаграмма (топ товаров/категорий)
   - круговую диаграмму (доля по регионам)
   - таблицу с агрегированными показателями
6. Добавить фильтры (по дате, категории, региону) для интерактивности.
7. Объединить визуализации в дашборд, настроить расположение и размеры.
8. Экспортировать дашборд в JSON (для воспроизводимости) или сделать скриншоты.
9. (Опционально) Настроить автоматическое обновление дашборда и права доступа.

Теоретическая часть

BI-инструменты в архитектуре данных

BI (Business Intelligence) – это конечный этап пайплайна обработки данных, где подготовленные и агрегированные данные визуализируются для принятия бизнес-решений. Apache Superset – open-source BI-платформа с богатыми возможностями визуализации, поддержкой множества баз данных и встроенным SQL-редактором.

Место BI в DWH:

Raw → Staging → Core (DWH) → Marts/Gold → BI Dashboard

Ключевые понятия Superset:

Понятие	Описание
Database Connection	Подключение к БД (PostgreSQL, ClickHouse, MySQL, др.).
Dataset	Логическое представление данных (таблица или SQL-запрос), основа для визуализаций.
Chart	Отдельный график или визуализация (линейный, столбчатый, heatmap и т.д.).
Dashboard	Набор chart'ов, объединённых на одной странице с фильтрами и макетом.
SQL Lab	Встроенный SQL-редактор для исследования данных и создания датасетов.

Типы визуализаций (для ETL-отчётности):

- **Временные ряды** – тренды продаж, активность пользователей.
- **Барчарты** – сравнение категорий, топ-10 товаров.
- **Пирожковые диаграммы** – доли рынка, распределение.
- **Таблицы с условным форматированием** – ключевые метрики.
- **Карты (геоданные)** – продажи по регионам.

Рекомендации по проектированию дашборда:

- Одна главная метрика (KPI) вверху.
- Фильтры (глобальные) для всего дашборда.
- Связанные визуализации (клик на столбце фильтрует другие).
- Минимум анимации и ярких цветов.

Методика выполнения работы

1. **Подготовка окружения**
   1.1. Создайте каталог lab17_superset
   1.2. Внутри создайте подкаталоги: docker, data (для PostgreSQL persistence), dashboards
   1.3. Напишите docker-compose.yaml для запуска Superset и PostgreSQL:

```
version: '3.8'
services:
  postgres:
    image: postgres:15
    container_name: superset_postgres
    environment:
      POSTGRES_USER: superset
      POSTGRES_PASSWORD: superset
      POSTGRES_DB: superset_meta
    ports:
      - "5433:5432"
    volumes:
      - ./data/postgres:/var/lib/postgresql/data

  superset:
```

```
image: apache/superset:latest
container_name: superset
environment:
  - SUPERSET_SECRET_KEY=your-secret-key-change-this
ports:
  - "8088:8088"
depends_on:
  - postgres
command: >
  sh -c "superset fab create-admin &&
        superset db upgrade &&
        superset init &&
        superset run -p 8088 --with-threads --reload --debugger"
```

1.4. Запустите: docker-compose up -d

1.5. Дождитесь инициализации (около минуты). Откройте Superset: http://localhost:8088 (логин: admin, пароль: admin).

2. **Подготовка витрины данных в PostgreSQL**

2.1. Запустите отдельный PostgreSQL для витрины (или используйте тот же, что для метаданных Superset). Для учебных целей создадим новую БД dwh_db в контейнере postgres.

2.2. Подключитесь к postgres контейнеру:

```
docker exec -it superset_postgres psql -U superset -d superset_meta
```

2.3. Создайте схему gold и таблицу sales_summary:

```
CREATE SCHEMA gold;
CREATE TABLE gold.sales_summary (
  sale_date DATE,
```

```
    product_category VARCHAR(50),
    region VARCHAR(50),
    total_sales DECIMAL(12,2),
    quantity_sold INTEGER,
    num_orders INTEGER
);
```

2.4. Заполните тестовыми данными (последние 30 дней, 3 категории, 3 региона):

```
INSERT INTO gold.sales_summary (sale_date, product_category, region, total_sales, quantity_sold, num_orders)
SELECT
    generate_series('2025-04-01'::DATE, '2025-05-01'::DATE, '1 day'::INTERVAL) AS sale_date,
    category,
    region,
    (random() * 10000)::DECIMAL(10,2) AS total_sales,
    (random() * 100)::INTEGER AS quantity_sold,
    (random() * 20)::INTEGER AS num_orders
FROM (VALUES ('Electronics'), ('Clothing'), ('Books')) AS cat(category),
     (VALUES ('Moscow'), ('SPb'), ('Kazan')) AS reg(region);
```

3. **Подключение Superset к PostgreSQL витрине**

3.1. В Superset: перейдите в **Data → Databases → + Database**.

3.2. Введите параметры подключения:

- Host: postgres (имя сервиса в docker-compose)
- Port: 5432
- Database: superset_meta (или dwh_db – если создавали отдельную)
- Username: superset
- Password: superset

- (Установите галочку **Allow Csv Upload** и **Expose in SQL Lab**)

3.3. Нажмите **Test Connection**, затем **Connect**.

4. **Создание Dataset**

4.1. Перейдите в **Data → Datasets → + Dataset**.

4.2. Выберите базу данных (только что добавленную), схему gold, таблицу sales_summary.

4.3. Нажмите **Create Dataset**.

5. **Построение визуализаций**

5.1. **Линейный график (продажи по дням)**:

- Нажмите **+ Create → Chart** → выберите sales_summary.
- Выберите тип визуализации **Time-series Line Chart**.
- **Time Column**: sale_date, **Aggregation**: sum для total_sales.
- **Group by**: product_category (для цветных линий).
- Настройте оси, заголовок, сохраните как Sales over time.

5.2. **Столбчатая диаграмма (топ товаров по выручке)**:

- **Bar Chart** → product_category как X-axis, SUM(total_sales) как Y-axis.
- Добавьте сортировку по убыванию.
- Сохраните как Sales by category.

5.3. **Круговая диаграмма (доля регионов)**:

- **Pie Chart** → region как Dimension, SUM(total_sales) как Metric.
- Сохраните как Revenue share by region.

5.4. **Таблица с агрегатами**:

- **Table** → группировка по product_category, region.

- Показать SUM(total_sales), SUM(quantity_sold), AVG(num_orders).
- Сохраните как Detailed table.

6. **Создание дашборда и добавление фильтров**

   6.1. Нажмите **Dashboards** → **+ Dashboard** → название Sales Dashboard.

   6.2. Нажмите **Edit Dashboard**, затем **Add Chart** → добавьте все созданные визуализации.

   6.3. Добавьте глобальные фильтры:

   - Нажмите **Filters** → **+ Add filter**.
   - Выберите колонку product_category → тип **Value** (или Checkbox).
   - Аналогично для region и sale_date (диапазон дат).

   6.4. Настройте расположение (перетаскивайте графики), задайте размеры.

   6.5. Сохраните дашборд.

7. **Тестирование и экспорт**

   7.1. Пощёлкайте фильтры, убедитесь, что все диаграммы обновляются.

   7.2. Экспортируйте дашборд: **...** (три точки) → **Export Dashboard**.

   7.3. Сделайте скриншоты итогового дашборда для отчёта.

8. **Дополнительно: настройка SQL Lab**

   8.1. Перейдите в **SQL Lab** → **SQL Editor** и выполните запрос к витрине, например, для проверки.

   8.2. Сохраните результат как новый Dataset на основе запроса.

Пример выполнения задания

Исходные данные (gold.sales_summary) – 3 категории * 3 региона * 30 дней = 270 записей.

Дашборд включает:

- Временной ряд продаж за апрель-май (цветные линии по категориям).
- Top categories (бары) – Electronics ≈ 70k, Clothing ≈ 55k, Books ≈ 40k.
- Доля регионов – Москва 45%, СПб 30%, Казань 25%.
- Таблица с детальными цифрами.

Filter по категории "Electronics" – все визуализации обновляются, показывая только электронику.

Скриншот дашборда (примерно):

(В отчёте студент вставит реальный скриншот)

Индивидуальные задания

Каждый вариант задаёт структуру витрины данных. Студент должен создать соответствующую таблицу в PostgreSQL, загрузить данные (можно сгенерировать скриптом), построить дашборд из 4–5 визуализаций.

Вариант 1. Продажи книжного магазина

Витрина: gold.book_sales (sale_date, genre, store, revenue, books_sold, discount). Построить график выручки по жанрам (столбчатая диаграмма), временной ряд по магазинам, долю скидок по жанру.

Вариант 2. Логистика доставки

Витрина: gold.delivery_metrics (delivery_date, city, courier_company, avg_delivery_min, deliveries_count, revenue). Визуализировать динамику среднего времени доставки, топ-5 городов по количеству доставок.

Вариант 3. Аналитика звонков контакт-центра

Витрина: gold.call_center_stats (date, operator_name, calls_handled, avg_wait_time_sec, satisfaction_score). Линейный график удовлетворённости по датам, гистограмма операторов, круговую диаграмму по часам пиковой нагрузки.

Вариант 4. Спортивные мероприятия (футбол)

Витрина: gold.match_stats (match_date, home_team, away_team, home_goals, away_goals, attendance). Создать топ команд по забитым голам, столбчатую диаграмму посещаемости, таблицу результатов.

Вариант 5. Гостиничный бизнес

Витрина: gold.hotel_bookings (booking_date, hotel_name, city, occupancy_rate, avg_daily_rate, revenue_per_room). Показать динамику загрузки, топ отелей по выручке, heatmap по дням недели.

Вариант 6. Рекламные кампании

Витрина: gold.ad_campaigns (day, campaign_name, spend, impressions, clicks, conversions). Рассчитать CTR и CPA на уровне кампаний, визуализировать тренды.

Вариант 7. Управление запасами

Витрина: gold.inventory (date, product_sku, warehouse, stock_quantity, reorder_threshold). Показать остатки по складам, сигнальные лампы для позиций ниже порога, линейный график движения запасов.

Вариант 8. IT-инфраструктура (мониторинг)

Витрина: gold.it_metrics (timestamp, service_name, response_time_ms, error_rate, cpu_usage). Временной ряд задержки, круговую по ошибкам, top-3 сервиса по CPU.

Вариант 9. Себестоимость продукции

Витрина: gold.cost_analysis (production_date, product_type, material_cost, labor_cost, overhead, total_cost). Столбчатая диаграмма структуры затрат, тренд себестоимости.

Вариант 10. Энергопотребление (IoT)

Витрина: gold.energy (reading_time, building_name, floor, kwh, temperature_c). Линейный график потребления по зданиям, хитмап по дням/часам, среднее потребление на этаж.

Вариант 11. Финансы персонала

Витрина: gold.salary_facts (month, department, position, avg_salary, headcount, bonus_pool). Гистограмма зарплат по отделам, пироговая доля бонусов, динамика headcount.

Вариант 12. Эффективность производства

Витрина: gold.manufacturing (date, line_id, shift, output_qty, defect_qty, downtime_min). Рассчитать OEE (Overall Equipment Effectiveness), создать дашборд с дефектами по сменам.

Вариант 13. Аналитика отзывов (NPS)

Витрина: gold.nps (review_date, product_id, region, score (0-10), comment_length). Группировать по промоутерам/нейтральным/критикам, показать динамику NPS и среднюю длину комментария.

Вариант 14. Транзакции по картам лояльности

Витрина: gold.loyalty (txn_date, customer_segment, store, amount_spent, loyalty_points_earned). Построить сегментацию по тратам (Scatter plot), топ-10 магазинов по баллам.

Вариант 15. Абитуриенты университета

Витрина: gold.admissions (year, faculty, exam_score_avg, applicants_count, accepted_count). Динамика конкурса по факультетам, столбчатая диаграмма % принятых.

Дополнительное задание для всех вариантов:

Настроить **Refresh** дашборда (автоматическое обновление каждые 5 минут) и создать **alert** через Superset (например, если выручка упала ниже порога – отправить email через встроенный email-бэкенд или заглушку).

Требования к отчёту

1. Титульный лист с названием работы, ФИО, группой, номером варианта.
2. Цель и задачи (из методички).
3. Теоретическая часть – BI-инструменты, Superset, место BI в DWH.
4. Ход выполнения работы:
   - docker-compose.yaml (листинг).
   - SQL-скрипты создания витрины и вставки данных.

- Пошаговое описание (скриншоты) создания подключения, датасета, визуализаций и дашборда.

5. Результаты:
   - Скриншот дашборда целиком (в режиме просмотра).
   - Скриншоты отдельных визуализаций (2-3).
   - Демонстрация работы фильтров (скриншот с применённым фильтром).
   - Файл экспортированного дашборда (JSON) прилагается к отчёту или ссылка.
6. Ответы на контрольные вопросы.
7. Вывод – как BI-инструмент помогает интерпретировать данные, какие типы графиков оказались наиболее полезны для выбранной предметной области.

Контрольные вопросы

1. Какое место занимает BI-инструмент в архитектуре хранилища данных (слои)?

2. Чем отличается SQL Lab от обычного просмотра дашборда в Superset?

3. Как в Superset создать вычисляемое поле (например, profit = sales - cost) внутри Dataset?

4. Какие существуют способы фильтрации данных в дашборде (глобальные фильтры, кросс-фильтрация)?

5. Как можно в Superset реализовать «дребезг» (drill down) – клик на категории открывает детализацию?

6. Почему для дашборда важно использовать агрегированные данные (витрину), а не сырые транзакции?

7. Как в Superset настроить автоматическое обновление дашборда каждые 5 минут?

8. Как поделиться дашбордом с коллегой без пароля администратора (роли, публичный доступ)?

9. Какие типы визуализаций лучше всего показывают временные тренды, а какие – структуру?

10. В чём преимущество Apache Superset перед Tableau (кроме цены)?

11. Как использовать кэширование в Superset для ускорения загрузки?

12. Можно ли в Superset визуализировать географические данные (карты)? Приведите пример.

13. Как в дашборде добавить текстовое описание метрик (коэффициенты, комментарии)?

14. Как экспортировать дашборд в PDF или изображение?

15. Как проверить, что дашборд действительно использует данные из витрины, а не напрямую из staging?

Лабораторная работа №18.

Итоговый проект: сквозной пайплайн обработки данных

Цель работы: собрать и реализовать сквозной ETL/ELT-пайплайн, охватывающий все ключевые этапы инжиниринга данных: извлечение из API (или другого источника), трансформацию и загрузку в хранилище, оркестрацию, контроль качества, мониторинг и визуализацию в BI-дашборде.

Задачи:

1. Выбрать предметную область и определить бизнес-метрики для анализа.
2. Спроектировать архитектуру пайплайна (выбор инструментов: Airflow, dbt, Kafka, Pandas, Great Expectations, Prometheus, Superset и др.).
3. Реализовать извлечение данных из внешнего источника (REST API, файлы, база данных) с обработкой пагинации и повторными попытками.
4. Выполнить очистку и трансформацию данных (Pandas, SQL, dbt) с применением SCD Type 2, если необходимо.
5. Оркестрировать пайплайн с помощью Apache Airflow (DAG с задачами extract, transform, load, validate).
6. Настроить проверки качества данных (Great Expectations) на критических этапах.
7. Добавить мониторинг метрик (Prometheus + Grafana) для отслеживания времени выполнения, количества строк и статуса.
8. Загрузить итоговые данные в витрину (PostgreSQL / ClickHouse) и подключить BI-инструмент (Superset) для построения дашборда.
9. Обеспечить воспроизводимость через Docker и, возможно, CI/CD (GitHub Actions).
10. Подготовить отчёт и защитить проект.

Теоретическая часть

Сквозной пайплайн данных (end-to-end data pipeline) объединяет все этапы обработки – от получения сырых данных до предоставления агрегированной информации бизнес-пользователю. Итоговый проект синтезирует компетенции, полученные в предыдущих лабораторных работах.

Типовая архитектура сквозного пайплайна:

[Источник данных] → (Extract) → [Data Lake / Staging] → (Transform / Quality) → [DWH / Marts]

↑ ↓

(Airflow DAG) [BI Dashboard]

↑ ↓

(Monitoring + Alerting) (Business User)

Компоненты итогового проекта (студент выбирает подмножество сообразно теме):

Компонент	Возможные технологии	Зачем нужен
Оркестрация	Apache Airflow, Prefect	Управление зависимостями, повторные запуски
Извлечение	Python + requests, Kafka	Получение данных из API или очереди
Трансформация	dbt, Pandas, SQL	Очистка, агрегация, денормализация
Качество данных	Great Expectations, dbt tests	Проверка уникальности, полноты, форматов
Мониторинг	Prometheus + Grafana	Время выполнения, число строк, ошибки
Хранилище	PostgreSQL, ClickHouse	Staging, DWH, витрина

Компонент	Возможные технологии	Зачем нужен
Визуализация	Apache Superset, Tableau	Дашборды для бизнес-аналитики
Инфраструктура	Docker, docker-compose	Локальное развёртывание

Требования к проекту:

- Автоматический запуск по расписанию (например, раз в час / раз в день).
- Обработка ошибок и повторные попытки.
- Логирование и возможность проследить lineage данных.
- Воспроизводимость с помощью Docker.

Методика выполнения работы

1. **Выбор темы и проектирование**
   1.1. Выберите предметную область из списка индивидуальных заданий или предложите свою.
   1.2. Определите источники данных (публичное API, CSV-файлы, тестовая БД).
   1.3. Нарисуйте архитектурную схему пайплайна (в отчёте).
   1.4. Выберите технологии (можно ориентироваться на те, что изучались в курсе).
2. **Настройка окружения**
   2.1. Создайте репозиторий на GitHub.
   2.2. Напишите docker-compose.yaml, который поднимает:
   - PostgreSQL для хранилища (и метаданных Airflow)
   - Apache Airflow (с CeleryExecutor или LocalExecutor)
   - (опционально) Redis для брокера, Kafka
   - Prometheus и Grafana

- Superset (опционально)

2.3. Настройте переменные окружения и volumes для сохранения данных.

3. **Реализация извлечения данных (Extract)**

3.1. Напишите Python-скрипт, который выгружает данные из API (или генерирует тестовые данные).

3.2. Реализуйте пагинацию, rate limiting, повторные попытки (retry).

3.3. Сохраните сырые данные в staging-таблицу PostgreSQL или в файл (CSV/Parquet) в локальную файловую систему / S3.

4. **Трансформация и загрузка (Transform & Load)**

4.1. Используйте dbt для написания моделей очистки и агрегации (или Pandas-скрипты).

4.2. Создайте staging-слой, core-слой (измерения и факты), витрину (gold).

4.3. Если нужно – реализуйте SCD Type 2 для одного измерения.

4.4. Добавьте тесты dbt (unique, not_null, relationships) и, возможно, Great Expectations.

5. **Оркестрация с Airflow**

5.1. Создайте DAG (например, end_to_end_pipeline), состоящий из задач:

- extract_from_api (PythonOperator)
- check_data_quality (GreatExpectationsOperator или PythonOperator)
- dbt_run (BashOperator или DbtRunOperator)
- dbt_test (или отдельный шаг проверки)
- materialize_features (для Feast, если используется)
- (опционально) refresh_superset_dashboard

5.2. Настройте зависимости (>>), повторные попытки, уведомления об ошибках.

6. **Мониторинг и логирование**

6.1. В ETL-скрипты добавьте экспорт метрик в Prometheus (клиентская библиотека).

6.2. Настройте сбор метрик (например, время выполнения каждого таска, количество обработанных строк) через prometheus_client.

6.3. Импортируйте метрики в Grafana, создайте дашборд с графиками.

6.4. (Опционально) Настройте алертинг в Prometheus/Alertmanager.

7. **Визуализация в BI**

   7.1. Подключите Superset к витрине (таблица gold).

   7.2. Создайте дашборд с 3-5 визуализациями (линейный график, столбцы, таблица).

   7.3. Добавьте фильтры для интерактивного анализа.

8. **Тестирование и интеграция CI/CD**

   8.1. Напишите unit-тесты на логику трансформации (pytest).

   8.2. Настройте GitHub Actions для запуска тестов при пуше в репозиторий.

   8.3. При желании добавьте деплой (например, сборку Docker-образов и push).

9. **Запуск и документирование**

   9.1. Запустите пайплайн вручную (через Airflow UI или CLI).

   9.2. Убедитесь, что данные появляются в витрине и дашборд отображает актуальную информацию.

   9.3. Задокументируйте все шаги в отчёте и подготовьте презентацию для защиты.

Пример выполнения задания

Тема: Аналитика продаж интернет-магазина (имитация данных через API Random Data API).

Архитектура:

Random Data API → Airflow (extract) → PostgreSQL staging → dbt (transform) → gold.sales_summary → Superset dashboard

↑

Great Expectations (quality)

↑

Prometheus + Grafana (monitoring)

Краткое описание шагов (в примере):

- **Extract:** Python-оператор в Airflow загружает 1000 записей (order_id, customer_id, amount, date, status) из тестового API и сохраняет в таблицу staging.orders.
- **Quality:** Запускается Expectation Suite: проверка amount > 0, date не в будущем, status ∈ допустимых значений.
- **Transform (dbt):** Модель stg_orders очищает данные, int_sales_daily агрегирует по дням, gold.sales_summary содержит итоговую витрину.
- **Airflow:** DAG выполняется раз в час, при падении проверки качества отправляется уведомление в Telegram (через on_failure_callback).
- **Monitoring:** Счётчики etl_rows_processed, гистограмма etl_duration_seconds экспортируются в Prometheus; в Grafana панель с количеством ошибок за последний день.
- **Superset:** Дашборд с графиком выручки по дням, топ-товаров (если есть категории) и таблицей заказов со статусами.

Результат: Рабочий сквозной пайплайн, который можно продемонстрировать преподавателю.

Индивидуальные задания

Каждый вариант представляет сквозную тему, для которой студент должен спроектировать и реализовать полный пайплайн. Допускается использование любых технологий, изученных в курсе, с обоснованием выбора.

Вариант 1. Аналитика погоды и влияния на продажи

Источник: OpenWeatherMap API (текущая погода в городах) + свои сгенерированные продажи. Цель: выявить корреляцию между температурой/осадками и продажами определённых товаров.

Вариант 2. Мониторинг вакансий IT-рынка

Источник: публичное API hh.ru или HeadHunter. Собирать вакансии по ключевым технологиям (Python, Java, SQL). Витрина: средняя зарплата, количество вакансий по городам, динамика.

Вариант 3. Аналитика транспортных потоков (общественный транспорт)

Источник: API Яндекс.Транспорт или любой открытый API GTFS (расписание). Вычислять среднюю задержку, загрузку маршрутов по часам. Визуализация на карте (если Superset поддерживает).

Вариант 4. Финансовый агрегатор котировок

Источник: Alpha Vantage API (или Yahoo Finance) для акций (AAPL, GOOGL, др.). Трансформировать в ежедневные свечи (open, high, low, close, volume). Витрина: скользящие средние, RSI.

Вариант 5. Логи веб-сервера и анализ поведения пользователей

Источник: сгенерированные логи в формате Common Log Format (или реальные из предыдущих работ). Построить витрину: количество уникальных посетителей, сессии, самые популярные страницы. Применить оконные функции.

Вариант 6. Система рекомендаций для новостей (имитация)

Источник: набор статей и пользовательских кликов. Построить feature store (Feast) для признаков (время чтения, теги). Оркестрировать обновление онлайновых признаков каждый час.

Вариант 7. Отток клиентов телеком-оператора

Источник: синтетические данные о звонках, жалобах, платежах. Использовать инкрементальную загрузку с SCD Type 2 для адресов. Качество: проверять полноту данных перед обучением модели оттока (логистическая регрессия в отдельном скрипте).

Вариант 8. Аналитика доставки (логистическая компания)

Источник: данные о заказах (API или CSV). Этапы: загрузка в staging, очистка, геокодирование городов (через внешний API), вычисление среднего времени доставки по регионам. Визуализация на карте.

Вариант 9. Оптимизация складских запасов

Источник: транзакции продаж и остатков на складах (синтетические). Построить витрину для ABC-анализа: товары категорий A, B, C по оборачиваемости. Мониторинг времени выполнения пайплайна.

Вариант 10. Аналитика энергопотребления зданий (IoT)

Источник: данные с датчиков (эмулировать через скрипт, отправляющий в Kafka). Потоковая обработка (Kafka -> Spark Streaming или Faust). Итог – витрина «пиковые часы» для каждого здания.

Вариант 11. Социальные сети (анализ тональности)

Источник: Twitter API (или тестовые твиты). Извлекать текст, вычислять тональность (VADER или другой библиотекой). Сохранять агрегированную статистику: тональность по часам, топ-10 слов.

Вариант 12. Медицинские исследования (клинические испытания)

Источник: сгенерированные данные пациентов, визитов, диагнозов. Требуется обеспечить качество: проверка диапазонов возраста, формата дат. Витрина: эффективность препарата по группам.

Вариант 13. Рейтинги гостиниц (отзывы)

Источник: API Tripadvisor или открытый датасет отзывов. Очистка текста, вычисление среднего рейтинга по городу, динамика. Использовать Great Expectations для проверки наличия рейтинга в диапазоне 1–5.

Вариант 14. Оптимизация маршрутов для курьерской службы

Источник: координаты заказов (генерировать). Построить модель расстояния между точками (гаверсинус). Витрина: суммарная длина маршрута по водителям за день. Мониторинг времени расчёта.

Вариант 15. Защита от мошенничества (фрод) на основе транзакций

Источник: данные транзакций (сумма, время, местоположение). Визуализировать аномалии (выбросы по сумме). Использовать dbt для

подготовки признаков, Feast для онлайнового получения подозрительных транзакций.

Дополнительное требование к каждому варианту:

Проект должен включать как минимум 5 технологий из списка: Airflow, dbt (или Pandas), Great Expectations (или dbt tests), Prometheus+Grafana, Superset, Docker. Студент обязан предоставить docker-compose.yml, воспроизводящий всё окружение одной командой.

Требования к отчёту

1. Титульный лист с темой проекта, ФИО, группой, номером варианта.
2. Введение – актуальность выбранной темы, бизнес-цели.
3. Архитектура – схема пайплайна с указанием инструментов и потоков данных.
4. Ход выполнения (разделы по этапам):
   - Описание источников данных.
   - Настройка окружения (Docker, Airflow, БД).
   - Реализация извлечения (код extract).
   - Трансформация и тесты качества (модели dbt/Python, expectations).
   - Оркестрация (DAG Airflow, скриншоты).
   - Мониторинг (метрики, дашборд Grafana).
   - BI-визуализация (дашборд Superset).
5. Результаты:
   - Скриншоты успешных запусков DAG.
   - Скриншоты графиков в Grafana.
   - Скриншоты дашборда Superset.
   - Примеры данных в витрине (выборка из таблицы).
6. Тестирование – описание тестов и CI/CD (если есть).
7. Заключение – достигнутые результаты, сложности, перспективы.
8. Ответы на контрольные вопросы (письменно).

9. Список использованных источников и ссылка на GitHub-репозиторий.

Контрольные вопросы

1. Почему в сквозном пайплайне важно разделять слои (staging, core, gold)?

2. Какой компонент вашего пайплайна наиболее критичен к отказам? Как вы обеспечили его отказоустойчивость?

3. Как часто должен перезапускаться пайплайн? Какие факторы влияют на выбор интервала?

4. Как вы гарантируете, что данные, отображаемые в дашборде, консистентны с данными, использованными для обучения модели?

5. Какие метрики вы мониторите в Prometheus? Как они помогают обнаружить деградацию пайплайна?

6. Как вы реализовали обработку ошибок в Airflow (retries, SLA, уведомления)?

7. Какие проверки качества (Great Expectations) оказались наиболее полезными для вашего источника данных?

8. Какой объём данных обрабатывает ваш пайплайн за один запуск? Оцените время выполнения.

9. Какие компромиссы между стоимостью разработки и производительностью вы выбрали?

10. Как можно масштабировать пайплайн при росте объёма данных в 100 раз?

11. Какие меры безопасности вы предприняли для API-ключей и паролей в коде?

12. Как вы тестировали отдельные модули (unit-тесты) без поднятия всей инфраструктуры?

13. Что бы вы улучшили в пайплайне, если бы работали над ним ещё месяц?

14. Как вы обеспечили воспроизводимость результатов (versioning данных, контейнеризация)?

15. Как ваш пайплайн может быть интегрирован в существующую инфраструктуру компании (например, существует ли каталог данных, DataHub)?

СПИСОК ЛИТЕРАТУРЫ

Основная литература

1. Reis J., Housley M. Fundamentals of Data Engineering: Plan and Build Robust Data Systems. – O'Reilly Media, 2022. – 450 с. (Охватывает архитектуру, жизненный цикл данных, выбор инструментов, современные практики.)

2. Kimball R., Ross M. The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling. – 3rd ed. – Wiley, 2013. – 608 с. (Классика по проектированию хранилищ данных, схема «звезда», медленно меняющиеся измерения.)

3. Inmon W. H. Building the Data Lakehouse. – Technics Publications, 2021. – 150 с. (Концепция объединения озёр и хранилищ данных, Data Lakehouse.)

4. Garcia-Molina H., Ullman J. D., Widom J. Database Systems: The Complete Book. – 2nd ed. – Pearson, 2018. – 1248 с. (Главы по SQL, транзакциям, нормализации, индексам – база для работы с DWH.)

Дополнительная литература по инструментам

5. Apache Airflow Documentation. – URL: https://airflow.apache.org/docs/ (актуальная версия). (Официальная документация по оркестрации DAG, операторам, сенсорам, best practices.)

6. dbt Documentation. – URL: https://docs.getdbt.com/ (Полное руководство по трансформациям в стиле ELT, тестированию, документации.)

7. Great Expectations Documentation. – URL: https://docs.greatexpectations.io/ (Валидация качества данных, expectation suites, Data Docs.)

8. Apache Kafka Documentation. – URL: https://kafka.apache.org/documentation/ (Потоковая передача сообщений, producer/consumer, Kafka Streams.)

9. Prometheus Documentation. – URL: https://prometheus.io/docs/ (Сбор метрик, экспортеры, PromQL, Alertmanager.)

10. Grafana Documentation. – URL: https://grafana.com/docs/ (Визуализация, дашборды, источники данных.)

11. Apache Superset Documentation. – URL: https://superset.apache.org/docs/ (BI-инструмент, подключения к БД, создание дашбордов.)

12. Feast Documentation. – URL: https://docs.feast.dev/ (Feature store для машинного обучения, материализация, онлайн/офлайн хранилища.)

13. Docker Documentation. – URL: https://docs.docker.com/ (Контейнеризация приложений, Dockerfile, docker-compose.)

14. Pandas Documentation. – URL: https://pandas.pydata.org/docs/ (Очистка и трансформация данных в Python.)

Интернет-ресурсы и курсы

15. Data Engineering Cookbook (by Andreas Kretz). – URL: https://github.com/andkret/Cookbook (Практические рецепты по ETL, архитектуре, инструментам.)

16. Designing Data-Intensive Applications (Martin Kleppmann). – O'Reilly, 2017. – 616 с. (Глубокое понимание распределённых систем, репликации, партиционирования, транзакций.)

17. Stanford CS245: Database Principles (курс лекций). – URL: https://web.stanford.edu/class/cs245/ (Теоретические основы баз данных для инженеров данных.)

18. DataTalks.Club Data Engineering Zoomcamp. – URL: https://github.com/DataTalksClub/data-engineering-zoomcamp (Бесплатный практический курс с проектами: Terraform, Airflow, dbt, GCP, Kafka.)

19. Awesome Data Engineering (список ресурсов). – URL: https://github.com/igorbarinov/awesome-data-engineering (Кураторский список книг, блогов, инструментов, конференций.)

Нормативные и справочные материалы

20. ISO/IEC 20748-1:2017 Information technology – Learning analytics interoperability – Part 1: Reference model.

21. GDPR (General Data Protection Regulation) – Регламент Евросоюза о защите данных.