МИНИСТЕРСТВО ТРАНСПОРТА И КОММУНИКАЦИЙ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования

«Белорусский государственный университет транспорта»

Кафедра «Управление эксплуатационной работой и охрана труда»

Отчет  
по лабораторной работе №10

по дисциплине «Средства и технологии анализа и разработки информационных систем»

Выполнил Проверил

студент группы ГИ-31 зав. каф. УЭРиОТ

Сидорова А. А. Козлов В. Г.

Гомель, 2025

# Лабораторная работа №10

**РАЗРАБОТКА СИСТЕМЫ ОБРАБОТКИ СООБЩЕНИЙ С ИСПОЛЬЗОВАНИЕМ APACHE KAFKA**

**Цель работы:** Разработка системы асинхронной обработки сообщений с использованием Apache Kafka, включая создание продюсеров, консьюмеров и настройку их взаимодействия.

**Задание:**

1. **Установка и настройка Kafka**

• Запустите ZooKeeper и Kafka-брокер (локально или через Docker).

• Создайте топик user\_actions с 3 партициями.

1. **Разработка продюсера**

• Напишите приложение (Java/Python), отправляющее JSON-сообщения в топик. Пример сообщения: {"user\_id": 101, "action": "purchase", "timestamp": "2023-10-01T12:00:00"}

• Реализуйте ввод сообщений через консоль или чтение из файла.

**3. Разработка консьюмера**

• Напишите консьюмер, обрабатывающий сообщения из топика:

- Фильтрация по типу действия (например, "purchase").

- Вывод статистики в консоль (количество сообщений, частые действия).

**4. Обработка ошибок и масштабирование**

• Настройте Dead Letter Topic (DLT) для некорректных сообщений.

• Запустите 2 консьюмера в одной группе. Продемонстрируйте распределение сообщений между ними.

**5. Интеграция с внешней системой**

• Сохраняйте данные в PostgreSQL/MySQL.

***Задание 1.***

Исходный код docker-compose.yml:

version: '3.8'

services:

  zookeeper:

    image: confluentinc/cp-zookeeper:7.0.1

    container\_name: lab10\_zookeeper

    environment:

      ZOOKEEPER\_CLIENT\_PORT: 2181

      ZOOKEEPER\_TICK\_TIME: 2000

  kafka:

    image: confluentinc/cp-kafka:7.0.1

    container\_name: lab10\_kafka

    depends\_on:

      - zookeeper

    ports:

      - "9092:9092"

    environment:

      KAFKA\_BROKER\_ID: 1

      KAFKA\_ZOOKEEPER\_CONNECT: zookeeper:2181

      KAFKA\_ADVERTISED\_LISTENERS: PLAINTEXT://localhost:9092

      KAFKA\_OFFSETS\_TOPIC\_REPLICATION\_FACTOR: 1

      KAFKA\_AUTO\_CREATE\_TOPICS\_ENABLE: "true"

  postgres:

    image: postgres:15

    container\_name: lab10\_postgres

    restart: always

    ports:

      - "5432:5432"

    environment:

      POSTGRES\_USER: postgres

      POSTGRES\_PASSWORD: 1234

      POSTGRES\_DB: test\_db

    volumes:

      - pg\_data:/var/lib/postgresql/data

      - ./init.sql:/docker-entrypoint-initdb.d/init.sql

volumes:

  pg\_data:



Рисунок 10.1 – **Запуск ZooKeeper и Kafka через Docker**

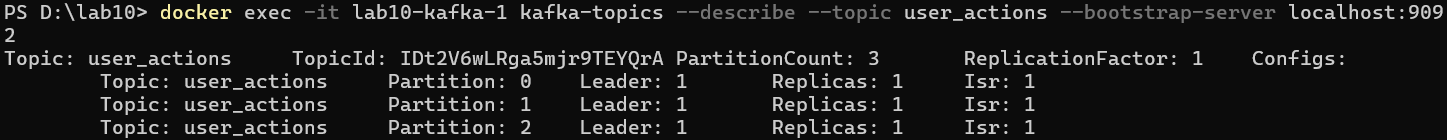


Рисунок 10.2 – **Создание топика user\_actions**

***Задание 2.***

Исходный код producer.py:

from kafka import KafkaProducer

import json

from datetime import datetime

producer = KafkaProducer(

    bootstrap\_servers='localhost:9092',

    value\_serializer=lambda v: json.dumps(v).encode('utf-8')

)

while True:

    user\_id = input("Введите user\_id: ")

    action = input("Введите действие (например, 'purchase'): ")

    message = {

        "user\_id": int(user\_id),

        "action": action,

        "timestamp": datetime.now().isoformat()

    }

    producer.send('user\_actions', message)

    print(f"[OK] Отправлено: {message}")

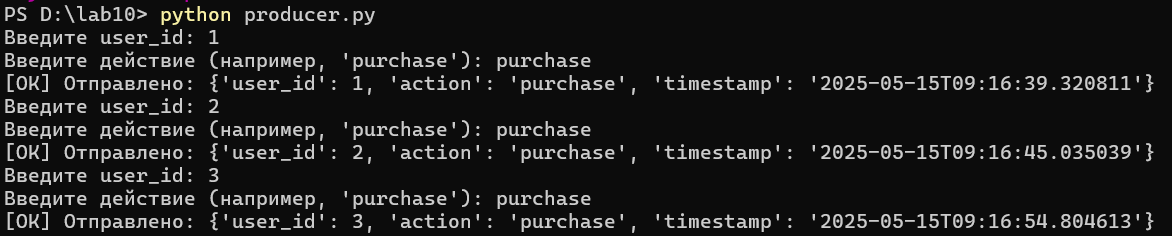


Рисунок 10.3 – **Работа producer.py**

***Задание 3.***

Исходный код consumer.py:

from kafka import KafkaConsumer

import json

from collections import Counter

import psycopg2

# Настройка подключения к PostgreSQL (находится в Docker)

conn = psycopg2.connect(

    dbname="test\_db", user="postgres", password="1234", host="localhost", port=5432

)

cursor = conn.cursor()

# Настройка Kafka-консьюмера

consumer = KafkaConsumer(

    'user\_actions',

    bootstrap\_servers='localhost:9092',

    value\_deserializer=lambda m: json.loads(m.decode('utf-8')),

    group\_id='action\_group'

)

action\_counter = Counter()

print("Слушаю сообщения...")

try:

    for message in consumer:

        try:

            data = message.value

            # Проверка структуры сообщения

            if "user\_id" not in data or "action" not in data or "timestamp" not in data:

                raise ValueError("Некорректное сообщение")

            # Обрабатываем только действия типа "purchase"

            if data['action'] == 'purchase':

                action\_counter[data['action']] += 1

                print(f"Покупка от пользователя {data['user\_id']} в {data['timestamp']}")

                print(f"Статистика: {dict(action\_counter)}")

                # Сохраняем сообщение в базу данных

                cursor.execute(

                    "INSERT INTO actions (user\_id, action, timestamp) VALUES (%s, %s, %s)",

                    (data['user\_id'], data['action'], data['timestamp'])

                )

                conn.commit()

        except Exception as e:

            print(f"[Ошибка] {e} — сообщение можно отправить в DLT (Dead Letter Topic)")

except KeyboardInterrupt:

    print("\nОстановлено пользователем.")

finally:

    cursor.close()

    conn.close()

    print("Соединение с БД закрыто.")

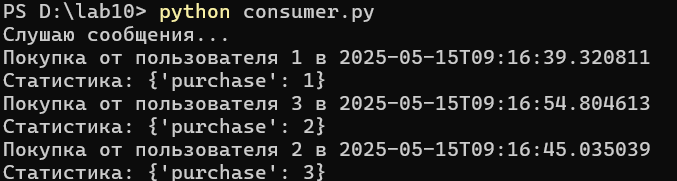


Рисунок 10.4 – **Работа consumer.py**

***Задание 4.***

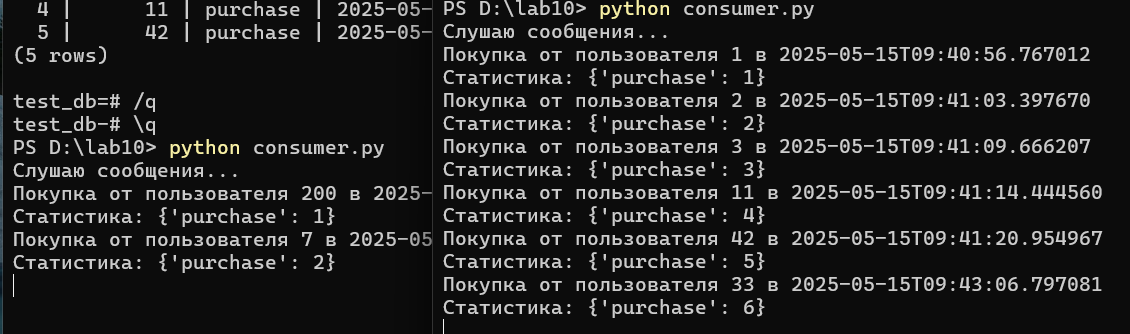


Рисунок 10.5 – **Результат выполнения задания 4**

***Задание 5.***

Исходный код init.sql:

CREATE TABLE IF NOT EXISTS actions (

    id SERIAL PRIMARY KEY,

    user\_id INT NOT NULL,

    action TEXT NOT NULL,

    timestamp TIMESTAMP NOT NULL

);

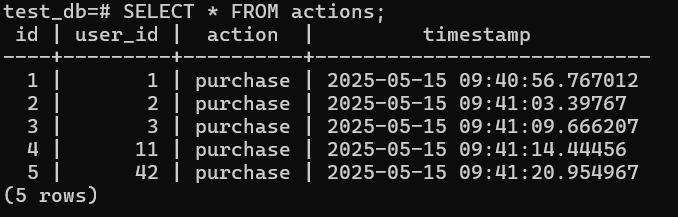


Рисунок 10.6 – **Результат заполнения базы данных**

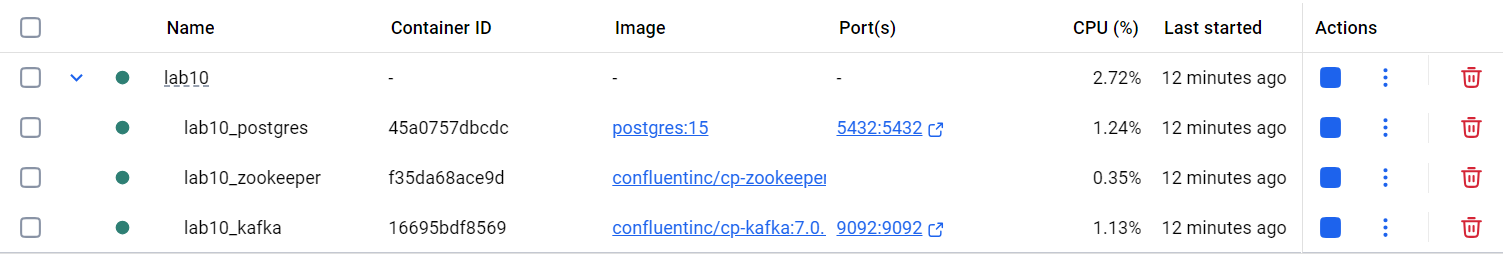


Рисунок 10.7 – **Конечный вид Docker-контейнера**

***Вывод:*** В результате выполнения лабораторной работы была реализована система асинхронной обработки сообщений с использованием Apache Kafka. Были изучены ключевые компоненты Kafka, такие как продюсеры, консьюмеры, топики и партиции, а также реализовано распределение нагрузки между потребителями и интеграция с базой данных. Kafka показала себя как мощный и масштабируемый инструмент для обработки потоков данных в реальном времени. Основные трудности возникли на этапе настройки окружения и понимания принципов работы партиций и групп консьюмеров, однако их преодоление позволило глубже разобраться в архитектуре распределённых систем.

***Контрольные вопросы.***

1. Зачем Kafka использует ZooKeeper?

До версии Kafka 2.8, Apache Kafka использовала ZooKeeper как внешний инструмент для координации и управления кластером. ZooKeeper хранил информацию о конфигурации брокеров, следил за их состоянием, помогал выбрать контроллера (главный брокер в кластере) и обеспечивал консенсус между узлами. Он также использовался для хранения информации о топиках, ACL и контроле за списками подписчиков (consumer groups). С версии Kafka 2.8 начался переход на KRaft (Kafka Raft) – встроенный механизм управления кластером без ZooKeeper, который делает Kafka более автономной и простой в развертывании. Но многие используют ZooKeeper до сих пор, особенно в продакшене.

1. Как обеспечить порядок сообщений в партиции?

Kafka гарантирует порядок сообщений внутри каждой партиции. Это означает, что если сообщения отправляются в одну и ту же партицию, то consumer прочитает их в том же порядке, в каком они были отправлены. Чтобы обеспечить этот порядок, нужно:

* Либо отправлять все сообщения без ключа, чтобы Kafka сама использовала round-robin (неуправляемо).
* Либо использовать один и тот же ключ при отправке — Kafka будет consistently отправлять такие сообщения в одну и ту же партицию.  
  Таким образом, гарантированный порядок обеспечивается только внутри одной партиции, а не всего топика.

1. В чём разница между auto.offset.reset=earliest и latest?

Параметр auto.offset.reset определяет, что делать, если offset ещё не сохранён (например, consumer только начал читать топик).

* earliest означает: читать все старые сообщения с начала партиции. Это полезно для тестов и дебага, когда хочется обработать все старые события.
* latest означает: читать только новые сообщения, которые поступят после запуска consumer'а. То есть старые игнорируются.

Пример: если в топике уже лежат 100 сообщений и ты запускаешь consumer впервые:

* earliest — он прочитает все 100.
* latest — он прочитает только те, что придут после запуска.

1. Как Kafka обеспечивает отказоустойчивость?

Kafka достигает отказоустойчивости с помощью репликации данных между брокерами. Каждый топик может иметь несколько реплик (копий), размещённых на разных брокерах. Одна из них – лидер, остальные – фолловеры. Все записи происходят сначала в лидера, а затем копируются фолловерам.  
Если лидер-брокер выходит из строя, Kafka автоматически выбирает нового лидера из оставшихся реплик – таким образом, сообщения не теряются и система продолжает работать. Кроме того, consumer'ы и producer'ы автоматически переподключаются к новым брокерам, если текущий становится недоступным. Это делает Kafka надёжной платформой для критичных к сбоям систем.