Федеральное государственное автономное образовательное учреждение высшего образования «Национальный исследовательский университет ИТМО»

# Лабораторная работа 3

**по дисциплине «Хранилища и базы данных»**

Выполнил:
Бобряков Кирилл, гр. № P4114
Преподаватель:
Королёва Юлия Александровна

Санкт-Петербург
2023

## Задание:

Развернуть несколько, например, три docker-контейнера как кластер, внутри каждого контейнера развернуть выбранную БД. Ограничить оперативную память 2 ГБ и память жёсткого диска 5 ГБ для каждого контейнера. Сымитировать «стрессовую» нагрузку, активно писать, а после запрашивать данные из БД. Фиксировать время выполнения запросов. Определить критический момент «падения» одного, двух и всех узлов нашего кластера. До «падения» последнего узла чтение должно быть корректным.

## Выполнение:

### Настройка кластера

Для начала необходимо было поднять и настроить кластер. Сперва создаем docker-compose файл с указанием одного мастер-узла и двух слейв-узлов.

После поднятия кластера при помощи команды *docker compose up -d* мы создадим базы данных для соответствующих узлов, и у нас появится возможность редактировать параметры инстансов postgresql.

Сначала необходимо настроить мастер-ноду, указав параметры репликации и размеры памяти в файле postgresql.conf.

```
# Replication
wal_level = replica
hot_standby = on
max_wal_senders = 10
max_replication_slots = 10
hot_standby_feedback = on

# Memory
effective_cache_size = 128kB
shared_buffers = 128kB                  # min 128kB
work_mem = 64kB                         # min 64kB
maintenance_work_mem = 1MB              # min 1MB
```

```
version: '3'

services:
  db_master:
    image: postgres
    restart: no
    environment:
      POSTGRES_PASSWORD: mypass
      POSTGRES_REPLICATION_USER: replication_user
      POSTGRES_REPLICATION_PASSWORD: replication_password
      POSTGRES_INITDB_ARG: --data-checksums
    volumes:
      - ./data/master:/var/lib/postgresql/data
    ports:
      - 5435:5432
    networks:
      - postgres-network
    deploy:
      resources:
        limits:
          cpus: '1'
          memory: 20M

  db_slave1:
    image: postgres
    restart: no
    environment:
      POSTGRES_PASSWORD: mypass
      POSTGRES_MASTER_HOST: db_master
      POSTGRES_MASTER_PORT: 5435
      POSTGRES_REPLICATION_USER: replication_user
      POSTGRES_REPLICATION_PASSWORD: replication_password
    volumes:
      - ./data/slave1:/var/lib/postgresql/data
    ports:
      - 5434:5432
    networks:
      - postgres-network
    deploy:
      resources:
        limits:
          cpus: '0.5'
          memory: 15M

networks:
  postgres-network:
```

После этого подключаемся к PostgreSQL при помощи *psql -u postgres*, и вводим следующие команды для создания пользователя для репликации и настройки слотов для репликации.

```
$ CREATE USER replication_user WITH REPLICATION ENCRYPTED PASSWORD
'replication_password';

$ SELECT * FROM pg_create_physical_replication_slot('replication_slot_slave1');
$ SELECT * FROM pg_create_physical_replication_slot('replication_slot_slave2');


$ SELECT * FROM pg_replication_slots;
```

Затем нам необходимо предоставить доступ этому пользователю, путем добавления записи для него в pg_hba.conf

```
# pg_hba.conf
host    replication     replication_user    0.0.0.0/0            trust
```

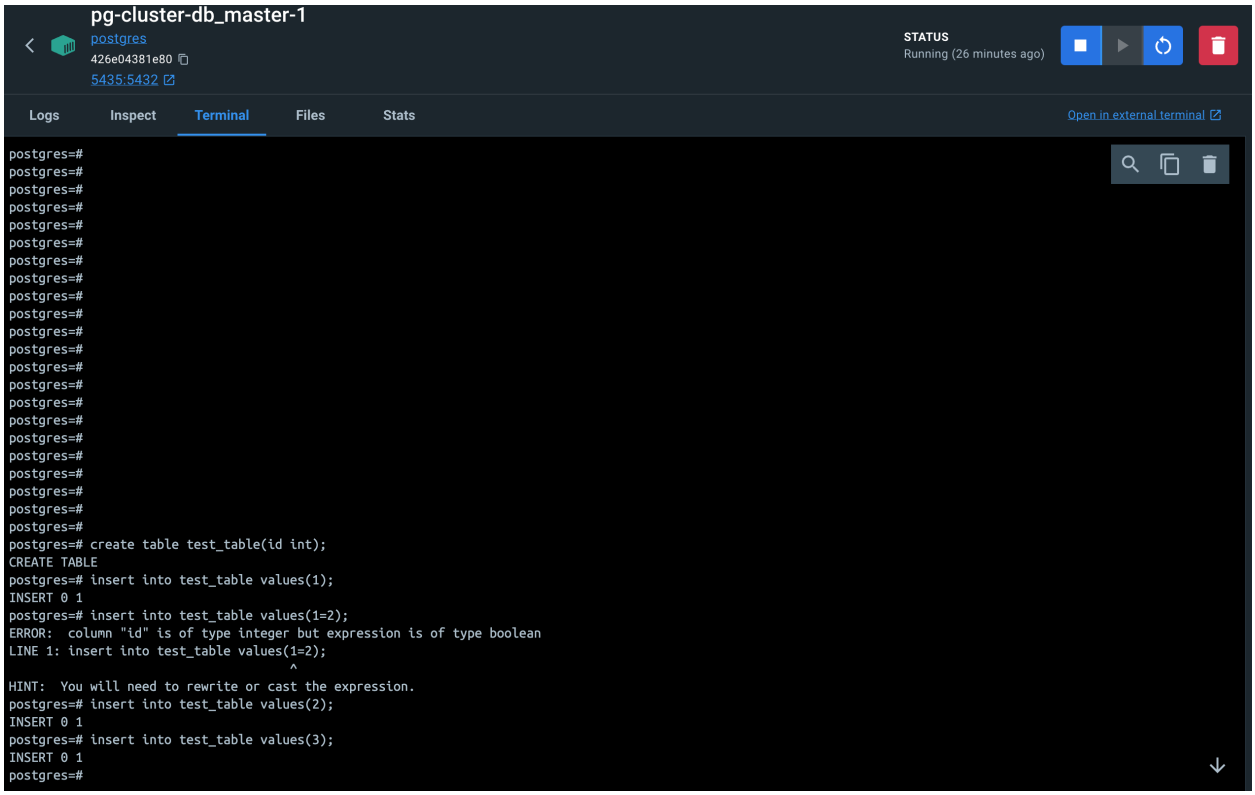
Затем нам необходимо скопировать весь этот конфиг для слейв-узлов. Выполняем это при помощи команды *pg_basebackup -D /tmp/postgresslave1 -S replication_slot_slave1 -X stream -P -U replication_user -Fp -R -h db_master.* Аналогичную команду необходимо сделать и для второго слейва.

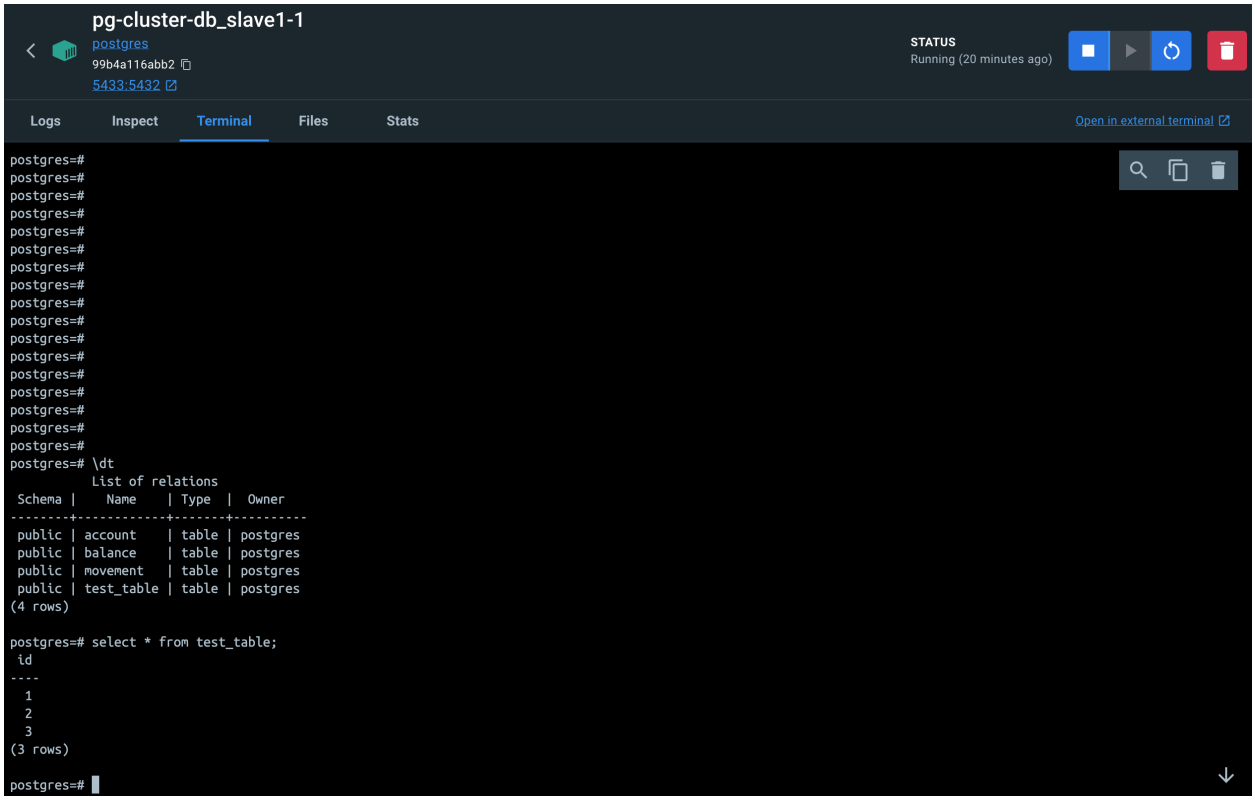
По итогам выполнения команды на мастер-сервере создадутся соответствующие директории для обоих слейвов - */tmp/postgresslave1*, */tmp/postgresslave2*. Их содержимое необходимо скопировать и заменить им содержимое директорий, созданных для баз slave1, slave2.

При этом необходимо заменить содержимое сгенерированного файла *postgresql.auto.conf*, указав информацию о подключении к мастер-узлу и команду восстановления

```
# postgresql.auto.conf
primary_conninfo = 'host=db_master port=5432 user=replication_user password=replica-
tion_password'
restore_command = 'cp /var/lib/postgresql/data/pg_wal/%f "%p"'
```

Таким образом, настройка завершена. Проверим, что репликация работает.



## Реализация скриптов для тестирования

После этого нам необходимо провести стресс-тестирование кластера. Для этого было написано несколько Python-скриптов – для записи на мастер ноду, чтения из любой из нод, проверки нод на доступность.

### main.py

```
from concurrent.futures import ThreadPoolExecutor
from datetime import timedelta, date
from random import randint, choice

import psycopg2
import csv
import time

db_config = {
    'host': 'localhost',
    'database': 'postgres',
    'user': 'postgres',
    'password': 'mypass',
    'port': '5435'
}

def create_test_subject_area():
    conn = psycopg2.connect(**db_config)
    cursor = conn.cursor()

    cursor.execute('''
        CREATE TABLE account (
            id SERIAL PRIMARY KEY,
            label VARCHAR(100),
            code VARCHAR(50),
            clientname VARCHAR(100),
            opendate DATE
        )
    ''')

    cursor.execute('''
        CREATE TABLE balance (
            id SERIAL PRIMARY KEY,
            account_id INTEGER,
            rest_type INTEGER,
            amount DECIMAL(10, 2)
        )
    ''')

    cursor.execute('''
        CREATE TABLE movement (
            id SERIAL PRIMARY KEY,
            account_id INTEGER,
            rest_type INTEGER,
            amount DECIMAL(10, 2)
        )
    ''')

    conn.commit()
    cursor.close()
    conn.close()

def simulate_stress_load():
    conn = psycopg2.connect(**db_config)
```

```
    cursor = conn.cursor()

    query = "SELECT * FROM your_table"
    iterations = 1000

    with open('execution_times.csv', 'w', newline='') as csvfile:
        writer = csv.writer(csvfile)
        writer.writerow(['Iteration', 'Execution Time (ms)'])

        for i in range(iterations):
            start_time = time.time()

            cursor.execute(query)

            end_time = time.time()
            execution_time = (end_time - start_time) * 1000

            writer.writerow([i+1, execution_time])

    cursor.close()
    conn.close()

def generate_random_date(start_date, end_date):
    time_between_dates = end_date - start_date
    days_between_dates = time_between_dates.days
    random_number_of_days = randint(0, days_between_dates)
    random_date = start_date + timedelta(days=random_number_of_days)
    return random_date

def write_random_rows(csv_writer):
    conn = psycopg2.connect(**db_config)
    cursor = conn.cursor()

    j = -1
    while True:
        j += 1
        for _ in range(100):
            label = f'Account-{randint(1, 100)}'
            code = f'Code-{randint(1000, 9999)}'
            clientname = f'Client-{randint(1, 50)}'
            opendate = generate_random_date(date(2022, 1, 1), date(2023, 12,
31))

            if _ == 99:
                start_time = time.time()

            cursor.execute("INSERT INTO account (label, code, clientname,
opendate) VALUES (%s, %s, %s, %s)",
                           (label, code, clientname, opendate))

            account_id = cursor.lastrowid  # Get the generated account ID

            rest_type = choice([0, 1, 2])
            amount = round(randint(100, 10000) / 100, 2)

            cursor.execute("INSERT INTO balance (account_id, rest_type,
amount) VALUES (%s, %s, %s)",
                           (account_id, rest_type, amount))

            rest_type = choice([0, 1, 2])
            amount = round(randint(100, 10000) / 100, 2)

            cursor.execute("INSERT INTO movement (account_id, rest_type,
amount) VALUES (%s, %s, %s)",
```

```
                            (account_id, rest_type, amount))

                if _ == 99:
                    end_time = time.time()
                    execution_time = (end_time - start_time) * 1000
                    csv_writer.writerow([j + 1, execution_time])

            conn.commit()

        cursor.close()
        conn.close()

if __name__ == '__main__':
    executor = ThreadPoolExecutor(max_workers=8)

    csv_file = open('execution_write_times.csv', 'w', newline='')
    csv_writer = csv.writer(csv_file)
    csv_writer.writerow(['Query', 'Execution Time (ms)'])
    for i in range(1,8):
        executor.submit(write_random_rows, csv_writer)

    executor.shutdown(wait=True)

    # create_test_subject_area()
```

Данный скрипт используется для создания предметной области:

- Счет (Идентификатор, Наименование, Код счета, Клиент, Дата открытия)
- Баланс (Идентификатор, Идентификатор счета, Тип остатка, Сумма)
- Движение (Идентификатор, Идентификатор счета, Тип остатка, Сумма)

И после этого он создает 8 потоков, запускающих метод записи случайных данных в эти таблицы, а также фиксирует время выполнения запроса и записывает в .csv файл.

## main-read.py

```
import csv
import sys
import time
from threading import current_thread

import psycopg2
from concurrent.futures import ThreadPoolExecutor

db_config = {
    'host': 'localhost',
    'database': 'postgres',
    'user': 'postgres',
    'password': 'mypass',
    'port': int(sys.argv[1])
}

def read_all_data(csv_writer):
    conn = psycopg2.connect(**db_config)
    cursor = conn.cursor()

    j = -1
    print('Start reading on' + sys.argv[1])
    while True:
        j += 1
        start_time = time.time()
        cursor.execute("SELECT * FROM movement")
        cursor.execute("SELECT * FROM balance")
        cursor.execute("SELECT * FROM account")

        cursor.execute("SELECT COUNT(*) FROM movement")
        movement_count = cursor.fetchone()
        cursor.execute("SELECT COUNT(*) FROM balance")
        balance_count = cursor.fetchone()
        cursor.execute("SELECT COUNT(*) FROM account")
        account_count = cursor.fetchone()

        end_time = time.time()
        execution_time = (end_time - start_time) * 1000
        if "0_0" in current_thread().name:
            csv_writer.writerow([j + 1, execution_time, movement_count[0] +
balance_count[0] + account_count[0]])

    cursor.close()
    conn.close()

    return movement_data

def read_random_data():
    executor = ThreadPoolExecutor(max_workers=8)

    csv_file = open('read_times' + sys.argv[1] + '-' + sys.argv[2] + '.csv',
'w', newline='')
    csv_writer = csv.writer(csv_file)
    csv_writer.writerow(['Query', 'Execution Time (ms)', 'Common count'])

    for i in range(8):
        executor.submit(read_all_data, csv_writer)

    executor.shutdown(wait=True)

if __name__ == '__main__':
    read_random_data()
```

Данный скрипт используется для чтения данных из БД. Он создает также 8 потоков, запускающих метод чтения всех данных из таблицы, фиксирует время выполнения запроса и общее количество всех записей.

**monitor-liveness.py**

```
import csv
import datetime
import subprocess
import time

db_host = 'localhost'
db_name = 'postgres'
db_user = 'postgres'

instances = ['5433', '5434', '5435']

def execute_pg_isready(port):
    command = ['pg_isready', '-h', db_host, '-p', port, '-d', db_name, '-u',
db_user]
    result = subprocess.run(command, capture_output=True, text=True)

    return result.stdout.strip()

if __name__ == '__main__':
    csv_file = open('liveness.csv', 'w', newline='')
    csv_writer = csv.writer(csv_file)
    csv_writer.writerow(['Times'] + instances)

    while True:
        result = []
        for i in instances:
            pg_isready_output = execute_pg_isready(i)
            result.append('1' if 'accepting connections' in pg_isready_output
else '0')
        csv_writer.writerow([str(datetime.datetime.now().time())] + result)
        time.sleep(1)
```

Последний скрипт используется для проверки каждой базы данных на доступность при помощи функции *pg_isready*. Каждую секунду он проверяет статус всех узлов, и фиксирует это в .csv файле.

## Тестирование кластера

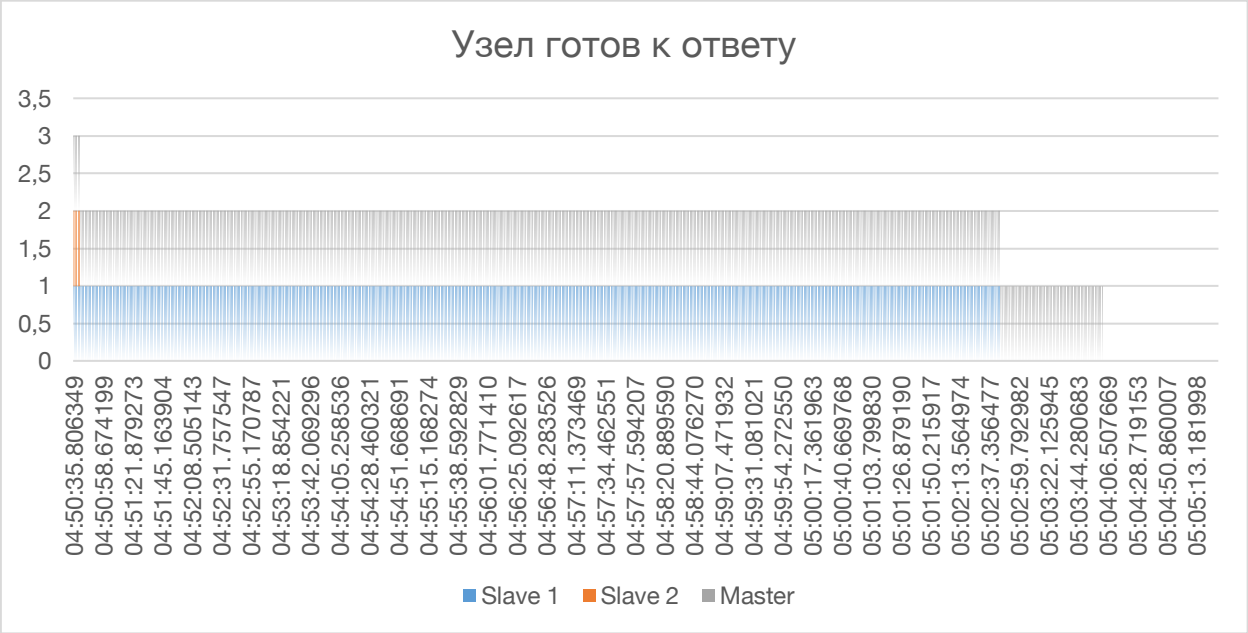
После всей подготовки приступаем к тестированию кластера:

1. Запускаем скрипт *monitor-liveness.py* для мониторинга статуса узлов
2. Затем запускаем скрипт *main.py* для записи в мастер-узел
3. После этого запускаем два раза скрипт *main-read.py* – для чтения из slave1 и slave2

После начала тестирования узел slave2 отваливается практически сразу, и остаются только два узла, которые без дополнительной нагрузки продолжают работать и достаточно успешно справляются со своей задачей.

Поэтому была добавлена дополнительная нагрузка на slave2 узел, после чего упал и он. Для падения мастер-узла пришлось также запустить несколько экземпляров скрипта чтения из него.

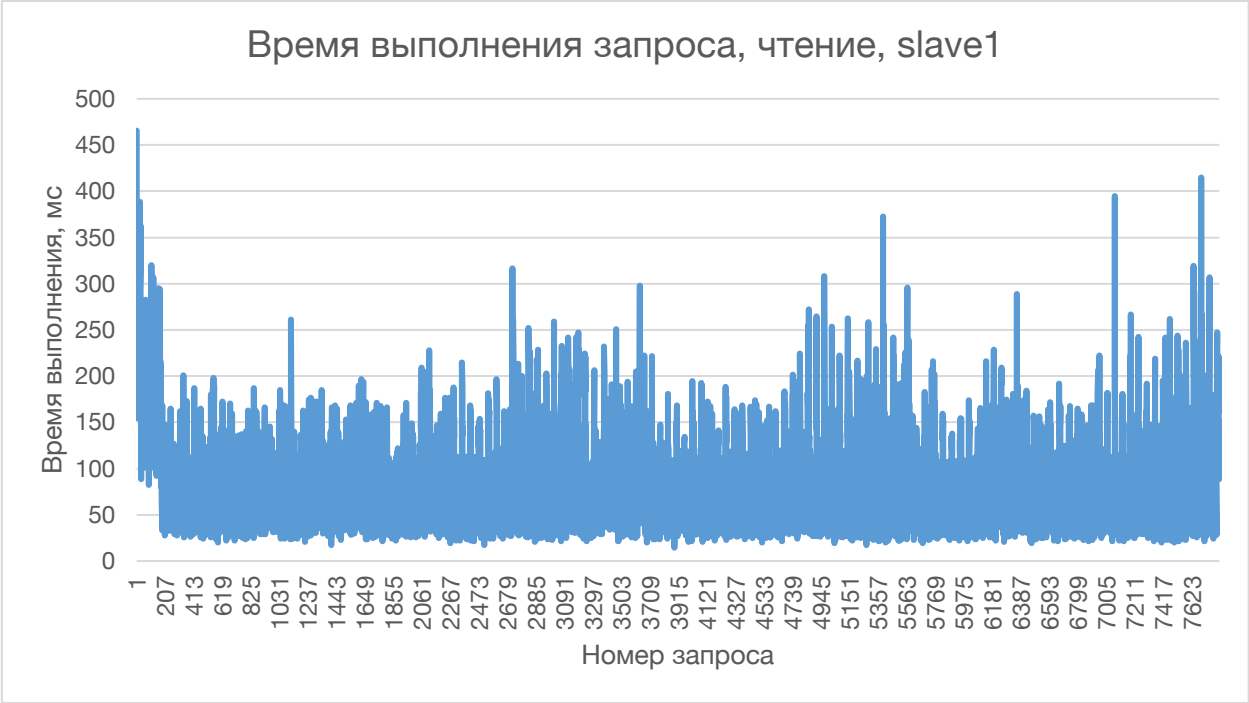
По итогу получаем следующие графики.



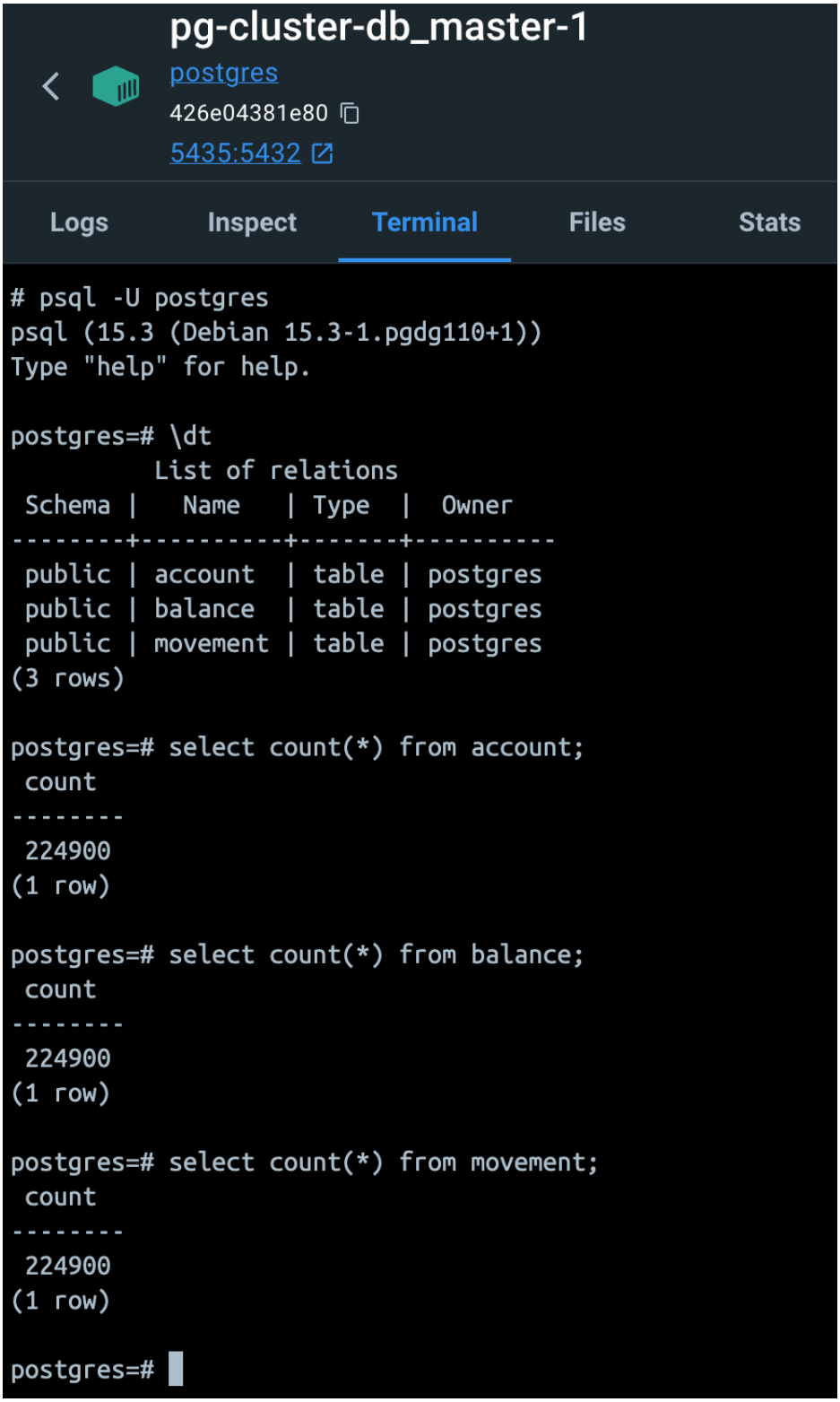

Из первого графки видно, что slave2 узел прожил совсем недолго, упав примерно в 04:50:50, slave2 упал приблизительно в 05:02:37, мастер-узел упал в 05:04:06.

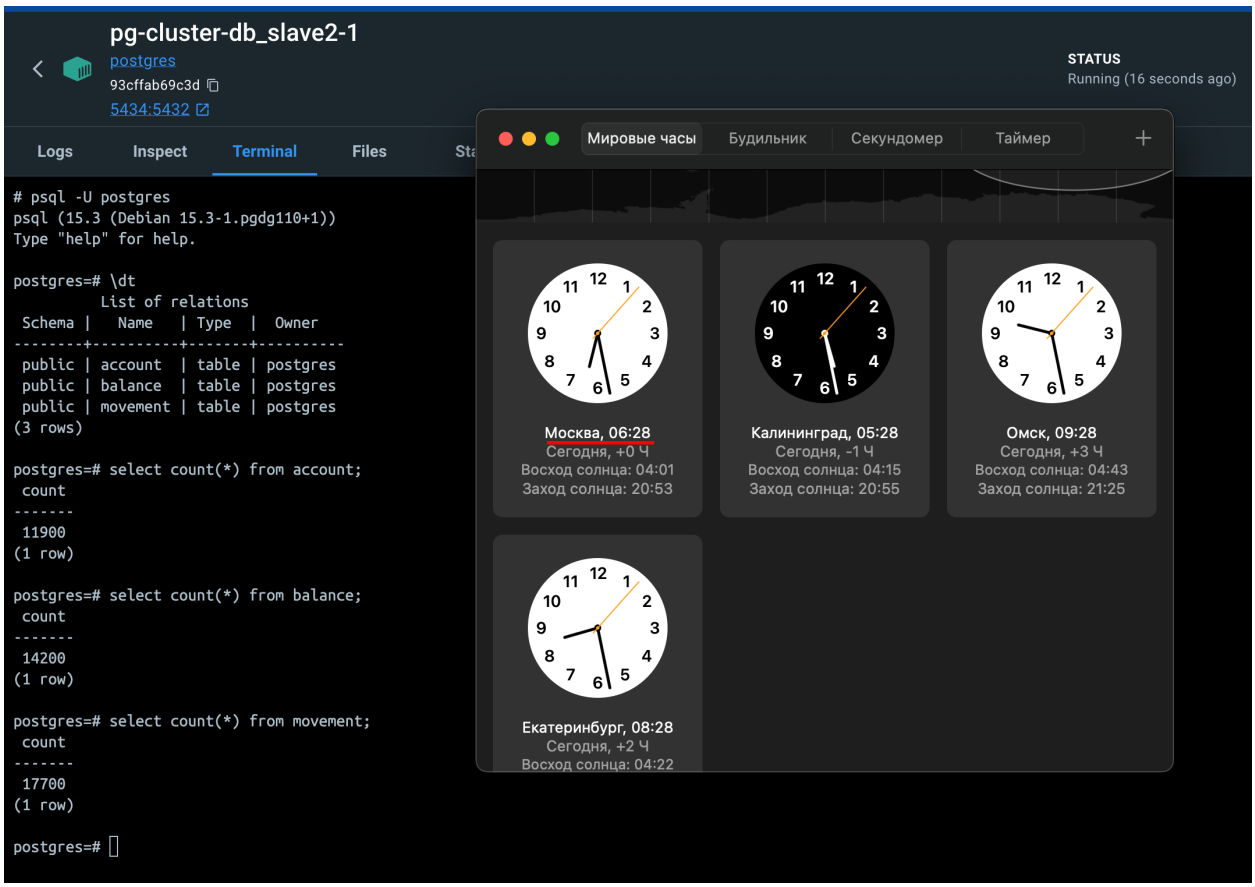
График ниже представляет время выполнения запроса на запись в мастер-узле. Запись выполнялась достаточно равномерно, за исключением некоторых выбросов.

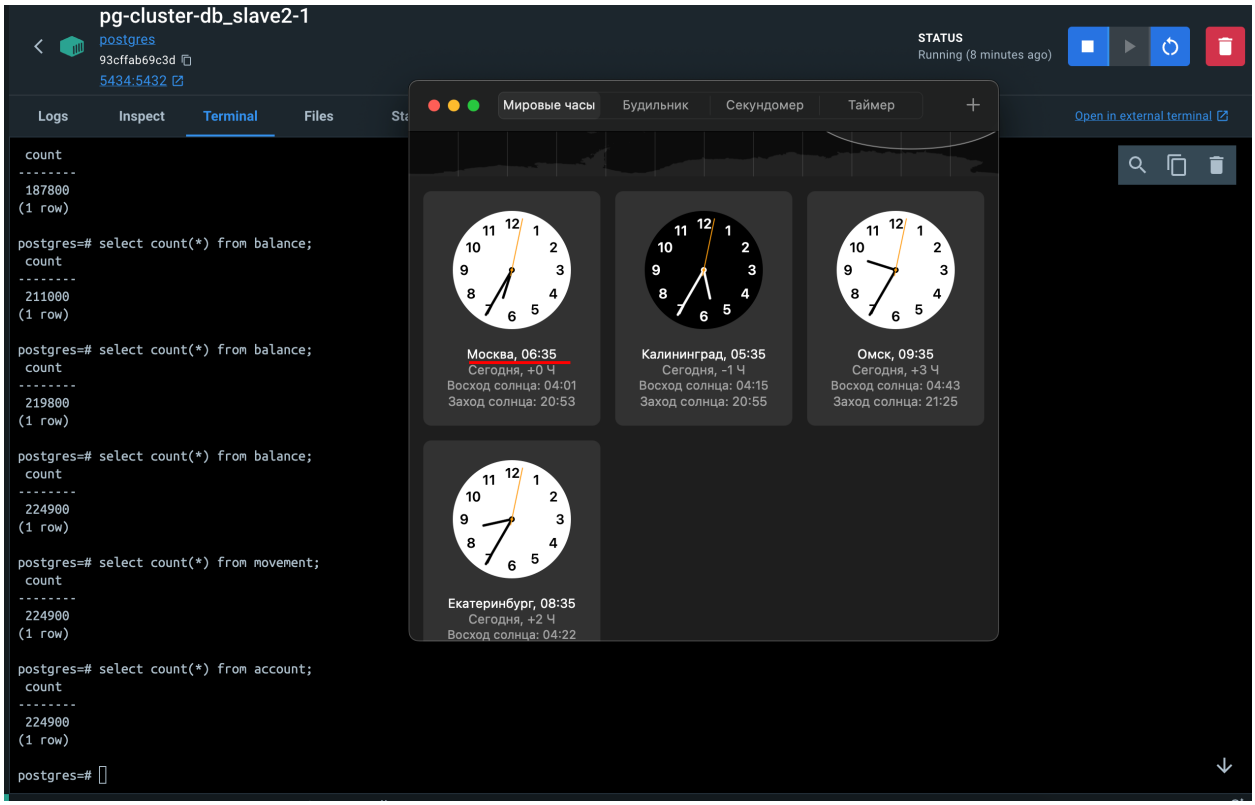
Ниже представлены графики времени выполнения запросов на чтение в slave-узлах. Из прожившего дольше slave1 узла видно, что наибольшее время выполнения запросов было в начале тестирования и перед самым его падением.




Также удостоверимся, что восстановление проходит корректно. Запустим поочередно каждый узел.






На восстановление узла slave2, упавшего практически сразу, понадобилось примерно 7-8 минут.

## Вывод:

В рамках выполнения данной лабораторной работы был настроен кластер из баз данных PostgreSQL, в котором один узел представлял собой основной – для записи и чтения, и два узла являлись реплицирующими, на которых было доступно только чтение.

Также было проведено тестирование данного кластера, и получены данные о поведении кластера при пиковой нагрузке.

СУБД PostgreSQL является достаточно оптимизированной, и даже при малых ресурсах хорошо справляется с нагрузкой, показывая примерно то же время выполнения запроса, а также успешно выполняет восстановление кластера после его падения.

В данном случае пришлось постараться, чтобы уронить кластер даже в условиях столь малого числа ресурсов (15 Мб ОЗУ, 0.5 CPU), что несомненно говорит о том, что базы данных способы справляться с большой нагрузкой, но при этом важно не забывать о правильной их настройке и стратегиях избегания и наступления, позволяющих либо превентивно решать проблему пиковых нагрузок, либо правильно действовать в критической ситуации.