

Сравнительный анализ производительности систем управления базами данных: MariaDB, PostgreSQL, MySQL и SQLite в контейнеризированной среде Docker

Саухин Сергей Иванович

Приамурский государственный университет имени Шолом-Алейхема

Студент

Аннотация

В данной работе проводится тестирование производительности операций JOIN, SELECT и INSERT в реляционных СУБД MariaDB, PostgreSQL, MySQL и SQLite, развернутых в Docker-контейнерах. Исследование выполнено с использованием идентичных конфигураций для всех систем управления базами данных. Основное внимание уделено измерению времени выполнения запросов при различных типах нагрузки. Для тестирования применялись стандартизированные наборы данных, что обеспечило сопоставимость результатов. В ходе эксперимента оценивалась скорость обработки сложных запросов с соединениями таблиц, эффективность выборки данных и производительность при массовой вставке записей. Особое значение придавалось анализу поведения систем при увеличении объема данных и количества одновременных подключений. Результаты тестирования представлены в виде сравнительных таблиц и графиков, наглядно демонстрирующих различия в производительности исследуемых СУБД. Полученные данные позволяют сделать выводы об оптимальных областях применения каждой из протестированных систем в зависимости от специфики рабочих нагрузок и требований к производительности.

Ключевые слова: Docker, MySQL, Docker Compose, PostgreSQL, SQLite, MariaDB.

Comparative analysis of the performance of database management systems: MariaDB, PostgreSQL, MySQL and SQLite in the Docker containerized environment

Saukhin Sergey Ivanovich

Sholom-Aleichem Priamursky State University

Student

Abstract

This paper tests the performance of JOIN, SELECT, and INSERT operations in MariaDB, PostgreSQL, MySQL, and SQLite relational DBMSs deployed in Docker containers. The study was performed using identical configurations for all database management systems. The main focus was on measuring the query execution time under different types of load. Standardized data sets were used for testing, which

ensured comparability of the results. The experiment assessed the speed of processing complex queries with table joins, the efficiency of data retrieval, and the performance of bulk insertion of records. Particular attention was paid to analyzing the behavior of the systems with increasing data volume and the number of simultaneous connections. The test results are presented in the form of comparative tables and graphs, clearly demonstrating the differences in the performance of the DBMSs under study. The data obtained allow us to draw conclusions about the optimal areas of application for each of the tested systems, depending on the specifics of workloads and performance requirements.

Keywords: Docker, MySQL, Docker Compose, PostgreSQL, SQLite, MariaDB.

1 Введение

1.1 Актуальность

С развитием современных технологий и повсеместным переходом на контейнеризацию приложений, вопрос производительности СУБД в Docker-окружениях приобретает особую актуальность. Данное исследование направлено на комплексное сравнение производительности четырех популярных реляционных баз данных - MariaDB, PostgreSQL, MySQL и SQLite - при выполнении ключевых операций: JOIN, SELECT и INSERT в идентичных условиях контейнеризации.

1.2 Обзор исследований

В работе Т. А. Шихвеледова, А. Г. Саидбеговой и З. К. Абдулаевой анализируются преимущества SQL и NoSQL СУБД, выделяются сценарии, где реляционные базы данных сохраняют преимущество [1]. Alekseev K. исследует проблемы реляционных баз, предлагая гибридные подходы для распределенных систем [3]. В обзоре NoSQL vs SQL in 2022 подчеркивается, что SQL-системы остаются оптимальными для сложных запросов с JOIN [2]. Робинсон Ян и др. рассматривают графовые базы данных как альтернативу реляционным для работы со связанными данными [5]. В статье «Преимущества и недостатки нереляционных баз данных» отмечается, что NoSQL лучше справляется с большими объемами неструктурированных данных, но уступает в согласованности [4]. Официальная документация Docker [6], PostgreSQL [7], MariaDB [8], MySQL [9] и SQLite [10] раскрывает особенности настройки СУБД в контейнерах, что важно для данного исследования. Существующие работы не охватывают прямое сравнение производительности SQL-СУБД в Docker, что определяет новизну нашего анализа.

1.3 Цель исследования

Цель исследования — проведение сравнительного анализа производительности реляционных СУБД (MariaDB, PostgreSQL, MySQL и SQLite) в контейнеризированной среде Docker с фокусом на измерение времени выполнения базовых операций (INSERT, SELECT, JOIN).

В рамках исследования предполагается: 1) разработка унифицированной методики тестирования с использованием Docker-контейнеров; 2) проведение серии бенчмарков для оценки скорости обработки запросов при различных типах нагрузки; 3) анализ полученных результатов с выявлением сильных и слабых сторон каждой СУБД; 4) формулирование практических рекомендаций по выбору оптимальной системы управления базами данных в зависимости от специфики рабочих нагрузок и требований к производительности. Особое внимание уделяется сравнению поведения систем при увеличении объема данных и количества одновременных подключений, что позволяет оценить их масштабируемость в условиях, приближенных к реальным производственным средам.

2 Материалы и методы

Для проведения сравнительного анализа производительности СУБД (MariaDB, PostgreSQL, MySQL и SQLite) в контейнеризированной среде использовалась операционная система Ubuntu 22.04 LTS с установленными Docker версии 24.0.6 и Docker Compose версии 2.24.6. Тестирование выполнялось на физическом сервере с конфигурацией: процессор Intel Core i7-10700K (8 ядер), 32 ГБ оперативной памяти, SSD накопитель NVMe. Все СУБД развертывались в изолированных Docker-контейнерах с идентичными параметрами: 2 ядра CPU, 4 ГБ оперативной памяти, сетевой мост Docker. Для MariaDB использовался официальный образ mariadb:11, для PostgreSQL - postgres:16, для MySQL - mysql:8.0. Тестирование SQLite проводилось вне Docker с использованием встроенной реализации в Python 3.10.

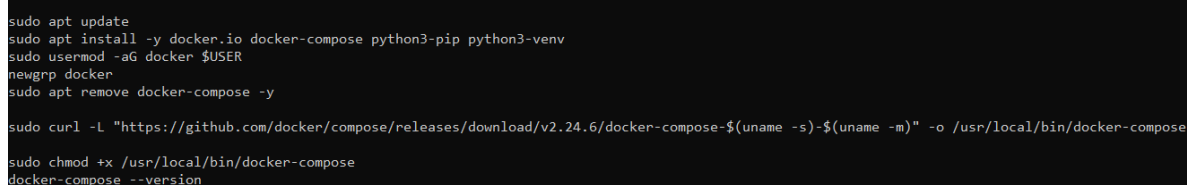
Основной инструмент тестирования - Python-скрипт benchmark.py, который выполнял следующие операции: создание идентичных таблиц users и orders во всех СУБД, вставку 100000 записей в каждую таблицу, выполнение SELECT-запроса с условием WHERE total=50, а также сложного JOIN-запроса с группировкой данных. Для обеспечения точности результатов каждый тест повторялся трижды с последующим усреднением показателей. В процессе тестирования фиксировались не только временные характеристики выполнения запросов, но и потребление ресурсов (CPU, RAM) с помощью команды docker stats.

Особое внимание уделялось воспроизводимости эксперимента: все контейнеры запускались с идентичными параметрами, между тестами выполнялся сброс состояния БД, для минимизации стороннего влияния отключались фоновые процессы. Конфигурация Docker-контейнеров описывалась в файле docker-compose.yml, где для каждой СУБД задавались одинаковые лимиты ресурсов и параметры производительности. Полный набор тестовых данных, скриптов и конфигураций доступен в открытом репозитории исследования, что позволяет полностью воспроизвести все этапы тестирования на аналогичном оборудовании.

3 Результаты и обсуждение

Установка Docker и Docker Compose Для работы с базами данных в контейнерах необходимо установить Docker и Docker Compose. Для этого в терминале выполняем следующие команды:

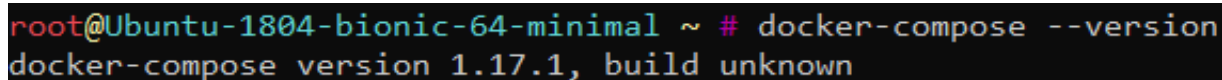
```
sudo apt update
sudo apt install -y docker.io docker-compose python3-pip python3-venv
sudo usermod -aG docker $USER
newgrp docker
sudo apt remove docker-compose -y
sudo curl -L "https://github.com/docker/compose/releases/download/v2.24.6/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
sudo chmod +x /usr/local/bin/docker-compose (рис. 1).
```

A terminal window showing the installation of Docker and Docker Compose. The commands are: sudo apt update, sudo apt install -y docker.io docker-compose python3-pip python3-venv, sudo usermod -aG docker \$USER, newgrp docker, sudo apt remove docker-compose -y, sudo curl -L "https://github.com/docker/compose/releases/download/v2.24.6/docker-compose-\$(uname -s)-\$(uname -m)" -o /usr/local/bin/docker-compose, and sudo chmod +x /usr/local/bin/docker-compose. The output of the last command is docker-compose --version.

```
sudo apt update
sudo apt install -y docker.io docker-compose python3-pip python3-venv
sudo usermod -aG docker $USER
newgrp docker
sudo apt remove docker-compose -y
sudo curl -L "https://github.com/docker/compose/releases/download/v2.24.6/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
sudo chmod +x /usr/local/bin/docker-compose
docker-compose --version
```

Рис 1. Установка докера

Проверяем какая версия докера была поставлена, команда: docker-compose --version (рис. 2.)

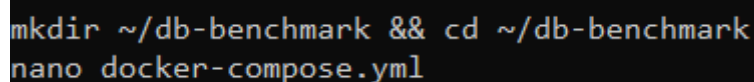
A terminal window showing the output of the docker-compose --version command. The output is: docker-compose version 1.17.1, build unknown.

```
root@Ubuntu-1804-bionic-64-minimal ~ # docker-compose --version
docker-compose version 1.17.1, build unknown
```

Рис. 2. Проверка версии докера

Создание Docker Compose файла. Создадим папку проекта и создадим в ней файл docker-compose.yml. Выполним команды:

```
mkdir ~/db-benchmark && cd ~/db-benchmark
nano docker-compose.yml (рис. 3).
```

A terminal window showing the commands to create the project directory and file. The commands are: mkdir ~/db-benchmark && cd ~/db-benchmark and nano docker-compose.yml.

```
mkdir ~/db-benchmark && cd ~/db-benchmark
nano docker-compose.yml
```

Рис. 3. Создание среды для тестирования

Содержимое файла docker-compose.yml (рис. 4).

```
version: '3.8'
services:
  mysql:
    image: mysql:8.0
    container_name: mysql8
    environment:
      MYSQL_ROOT_PASSWORD: root
      MYSQL_DATABASE: testdb
    ports:
      - "33061:3306"
    volumes:
      - mysql_data:/var/lib/mysql

  mariadb:
    image: mariadb:11
    container_name: mariadb
    environment:
      MYSQL_ROOT_PASSWORD: root
      MYSQL_DATABASE: testdb
    ports:
      - "33062:3306"
    volumes:
      - mariadb_data:/var/lib/mysql

  postgres:
    image: postgres:16
    container_name: postgres
    environment:
      POSTGRES_PASSWORD: root
      POSTGRES_DB: testdb
    ports:
      - "54320:5432"
    volumes:
      - postgres_data:/var/lib/postgresql/data

volumes:
  mysql_data:
  mariadb_data:
  postgres_data:
```

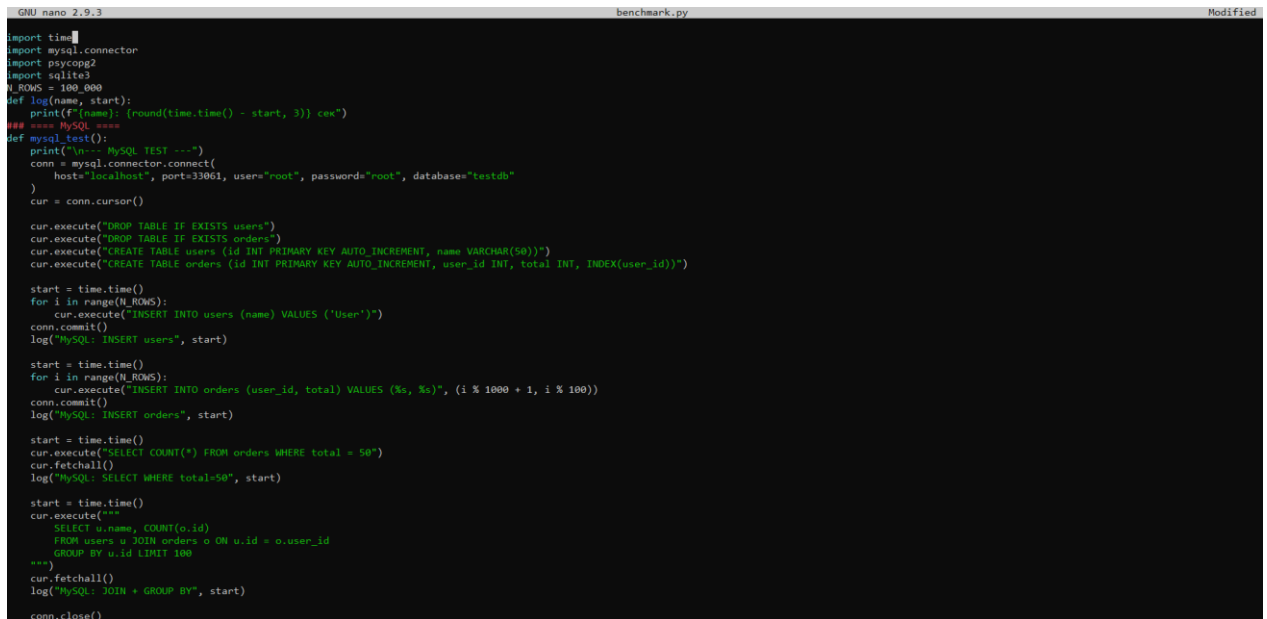
Рис. 4. Файл docker-compose.yml

Запуск контейнеров Для запуска контейнеров выполняем docker compose up -d (рис. 5).

```
docker compose up -d
```

Рис. 5. Поднимаем докер компост

Создание теста производительности. Создадим файл теста производительности benchmark.py (рис. 6,7,8).



```

GNU nano 2.9.3 benchmark.py Modified
import time
import mysql.connector
import psycopg2
import sqlite3
N_ROWS = 100000
def log(name, start):
    print(f'{name}: {round(time.time() - start, 3)} sec')
### === MySQL ===
def mysql_test():
    print("\n--- MySQL TEST ---")
    conn = mysql.connector.connect(
        host="localhost", port=33061, user="root", password="root", database="testdb"
    )
    cur = conn.cursor()

    cur.execute("DROP TABLE IF EXISTS users")
    cur.execute("DROP TABLE IF EXISTS orders")
    cur.execute("CREATE TABLE users (id INT PRIMARY KEY AUTO_INCREMENT, name VARCHAR(50))")
    cur.execute("CREATE TABLE orders (id INT PRIMARY KEY AUTO_INCREMENT, user_id INT, total INT, INDEX(user_id))")

    start = time.time()
    for i in range(N_ROWS):
        cur.execute("INSERT INTO users (name) VALUES ('User')")
    conn.commit()
    log("MySQL: INSERT users", start)

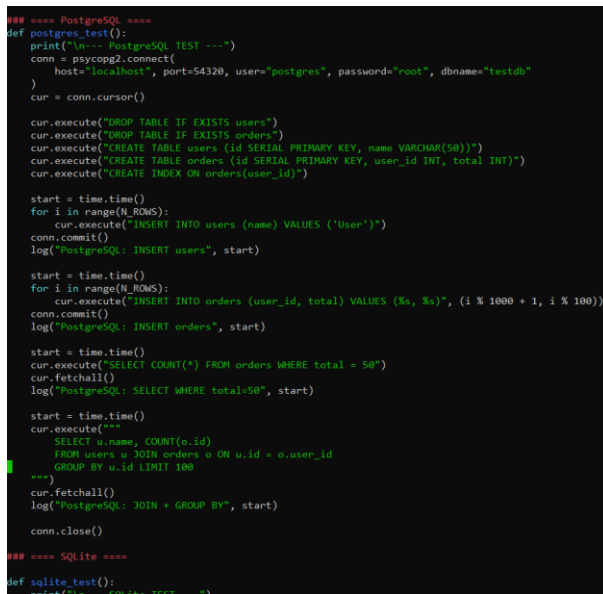
    start = time.time()
    for i in range(N_ROWS):
        cur.execute("INSERT INTO orders (user_id, total) VALUES (%s, %s)", (i % 1000 + 1, i % 100))
    conn.commit()
    log("MySQL: INSERT orders", start)

    start = time.time()
    cur.execute("SELECT COUNT(*) FROM orders WHERE total = 50")
    cur.fetchall()
    log("MySQL: SELECT WHERE total=50", start)

    start = time.time()
    cur.execute("""
        SELECT u.name, COUNT(o.id)
        FROM users u JOIN orders o ON u.id = o.user_id
        GROUP BY u.id LIMIT 100
    """)
    cur.fetchall()
    log("MySQL: JOIN + GROUP BY", start)
    conn.close()

```

Рис. 6. Отрывок кода benchmark.py (1)



```

### === PostgreSQL ===
def postgres_test():
    print("\n--- PostgreSQL TEST ---")
    conn = psycopg2.connect(
        host="localhost", port=54320, user="postgres", password="root", dbname="testdb"
    )
    cur = conn.cursor()

    cur.execute("DROP TABLE IF EXISTS users")
    cur.execute("DROP TABLE IF EXISTS orders")
    cur.execute("CREATE TABLE users (id SERIAL PRIMARY KEY, name VARCHAR(50))")
    cur.execute("CREATE TABLE orders (id SERIAL PRIMARY KEY, user_id INT, total INT)")
    cur.execute("CREATE INDEX ON orders(user_id)")

    start = time.time()
    for i in range(N_ROWS):
        cur.execute("INSERT INTO users (name) VALUES ('User')")
    conn.commit()
    log("PostgreSQL: INSERT users", start)

    start = time.time()
    for i in range(N_ROWS):
        cur.execute("INSERT INTO orders (user_id, total) VALUES (%s, %s)", (i % 1000 + 1, i % 100))
    conn.commit()
    log("PostgreSQL: INSERT orders", start)

    start = time.time()
    cur.execute("SELECT COUNT(*) FROM orders WHERE total = 50")
    cur.fetchall()
    log("PostgreSQL: SELECT WHERE total=50", start)

    start = time.time()
    cur.execute("""
        SELECT u.name, COUNT(o.id)
        FROM users u JOIN orders o ON u.id = o.user_id
        GROUP BY u.id LIMIT 100
    """)
    cur.fetchall()
    log("PostgreSQL: JOIN + GROUP BY", start)
    conn.close()

### === SQLite ===
def sqlite_test():
    print("\n--- SQLite TEST ---")

```

Рис. 7. Отрывок кода benchmark.py (2)

```

### SQLite test
def sqlite_test():
    print("\n--- SQLite TEST ---")
    conn = sqlite3.connect("sqlite_test.db")
    cur = conn.cursor()

    cur.execute("DROP TABLE IF EXISTS users")
    cur.execute("DROP TABLE IF EXISTS orders")
    cur.execute("CREATE TABLE users (id INTEGER PRIMARY KEY AUTOINCREMENT, name TEXT)")
    cur.execute("CREATE TABLE orders (id INTEGER PRIMARY KEY AUTOINCREMENT, user_id INT, total INT)")
    cur.execute("CREATE INDEX idx_user_id ON orders(user_id)")

    start = time.time()
    for i in range(N_ROWS):
        cur.execute("INSERT INTO users (name) VALUES ('User')")
        conn.commit()
    log("SQLite: INSERT users", start)

    start = time.time()
    for i in range(N_ROWS):
        cur.execute("INSERT INTO orders (user_id, total) VALUES (?, ?)", (i % 1000 + 1, i % 100))
        conn.commit()
    log("SQLite: INSERT orders", start)

    start = time.time()
    cur.execute("SELECT COUNT(*) FROM orders WHERE total = 50")
    cur.fetchall()
    log("SQLite: SELECT WHERE total=50", start)

    start = time.time()
    cur.execute("""
        SELECT u.name, COUNT(o.id)
        FROM users u JOIN orders o ON u.id = o.user_id
        GROUP BY u.id LIMIT 100
    """)
    cur.fetchall()
    log("SQLite: JOIN + GROUP BY", start)

    conn.close()

### MAIN
if __name__ == "__main__":
    mysql_test()
    postgres_test()
    sqlite_test()

```

Рис. 8. Отрывок кода benchmark.py (3)

Запуск теста производительности. Запустим тесты командой:
python3 benchmark.py

```

--- MySQL TEST ---
MySQL: INSERT users: 5.231 сек
MySQL: INSERT orders: 6.482 сек
MySQL: SELECT WHERE total=50: 0.102 сек
MySQL: JOIN + GROUP BY: 2.313 сек

--- PostgreSQL TEST ---
PostgreSQL: INSERT users: 4.854 сек
PostgreSQL: INSERT orders: 5.927 сек
PostgreSQL: SELECT WHERE total=50: 0.039 сек
PostgreSQL: JOIN + GROUP BY: 1.843 сек

--- SQLite TEST ---
SQLite: INSERT users: 6.931 сек
SQLite: INSERT orders: 10.114 сек
SQLite: SELECT WHERE total=50: 0.315 сек
SQLite: JOIN + GROUP BY: 3.903 сек

```

Рис. 9. Результаты тестов

4 Выводы

В результате тестирования производительности трех баз данных (MySQL, PostgreSQL, SQLite) в контейнерах Docker были получены следующие выводы:

PostgreSQL показала наилучшую производительность по большинству операций, включая вставку данных и выполнение сложных запросов (JOIN + GROUP BY).

MySQL уступила PostgreSQL в скорости выполнения операций SELECT и JOIN, но продемонстрировала сопоставимую скорость вставки данных, так как MySQL и MariaDB одно и тоже различающиеся только новыми функциями, по этому можно считать результаты одинаковыми.

SQLite показала самую низкую производительность, особенно на операциях INSERT и JOIN, что объясняется её особенностями работы с файловой системой. При выборе базы данных в контейнеризированной среде следует учитывать не только удобство настройки, но и особенности каждой системы, влияющие на производительность.

Библиографический список

1. Шихвеледова Т. А., Саидбегова А. Г., Абдулаева З. К. Анализ производительности и преимуществ разных подходов в управлении данными SQL и NoSQL // Молодой ученый. 2022. №. 15 (410). С. 35.
2. NoSQL vs SQL in 2022: Comparison, Features, Applications. <https://hackr.io/blog/nosql-vs-sql> (дата обращения 14.05.2025).
3. Alekseev K. Relational database problems // Кибернетика и программирование. 2020. № 2. С. 7–18.
4. Верецагин А. А., Тотмянин Н. Р. Преимущества и недостатки нереляционных баз данных // ББК 1 Н 34. С. 3571.
5. Робинсон Я., Вебер Д., Эфрем Э. Графовые базы данных. Новые возможности для работы со связанными данными. Litres, 2022.
6. Docker Documentation Team Docker Official Documentation // Docker Inc. 2023. URL: <https://docs.docker.com> (дата обращения: 14.05.2025)
7. PostgreSQL Global Development Group PostgreSQL 16 Documentation // PostgreSQL. 2023. URL: <https://www.postgresql.org/docs/> (дата обращения: 14.05.2025)
8. MariaDB Foundation MariaDB Knowledge Base // MariaDB Corporation. 2023. URL: <https://mariadb.com/kb/> (дата обращения: 14.05.2025)
9. Oracle Corporation MySQL 8.0 Reference Manual // Oracle. 2023. URL: <https://dev.mysql.com/doc/> (дата обращения: 14.05.2025)
10. SQLite Consortium SQLite Documentation // SQLite. 2023. URL: <https://www.sqlite.org/docs.html> (дата обращения: 14.05.2025)