

Сравнительный анализ производительности SQL и NoSQL СУБД для различных сценариев работы с данными

Мандронов Антон Андреевич

Приамурский государственный университет имени Шолом-Алейхема

Студент

Аннотация

В статье проводится сравнительный анализ производительности реляционной СУБД PostgreSQL и документоориентированной NoSQL-системы MongoDB в ключевых сценариях работы с данными. На основе синтетического датасета, имитирующего платформу для ведения блогов, выполнено тестирование операций массовой вставки, сложных выборок с соединениями и агрегаций. Результаты демонстрируют преимущество MongoDB при массовой вставке данных (превышение скорости на 86,7%), в то время как PostgreSQL значительно эффективнее выполняет сложные запросы (на 69,3%) и операции агрегации (на 83,6%). Исследование подтверждает, что выбор СУБД должен определяться спецификой workloads: MongoDB предпочтительна для задач с интенсивной записью, а PostgreSQL — для систем с сложными запросами и требованиями к целостности данных.

Ключевые слова: СУБД, PostgreSQL, MongoDB, производительность, NoSQL, реляционные базы данных, массовая вставка, сложные запросы, агрегация, сравнительный анализ.

Comparative analysis of the performance of SQL and NoSQL DBMSs for various data processing scenarios

Mandronov Anton Andreevich

Sholom-Aleichem Priamursky State University

Student

Abstract

This article presents a comparative analysis of the performance of the relational DBMS PostgreSQL and the document-oriented NoSQL system MongoDB in key data processing scenarios. Using a synthetic dataset simulating a blog platform, tests were conducted on bulk insertion, complex join queries, and aggregation operations. The results demonstrate MongoDB's advantage in bulk data insertion (86.7% faster), while PostgreSQL significantly outperforms in complex queries (69.3% faster) and aggregation operations (83.6% faster). The research confirms that the choice of DBMS should be determined by specific workload requirements: MongoDB is preferable for write-intensive tasks, while PostgreSQL is better suited for systems with complex queries and strict data integrity requirements.

Keywords: DBMS, PostgreSQL, MongoDB, performance, NoSQL, relational databases, bulk insertion, complex queries, aggregation, comparative analysis.

1 Введение

1.1 Актуальность

Актуальность темы обусловлена растущей популярностью NoSQL-СУБД и необходимостью объективного сравнения их с традиционными реляционными системами. В условиях, когда время отклика приложения является критическим фактором, выбор оптимальной базы данных напрямую влияет на производительность всей системы. Существует потребность в четком практическом понимании, в каких сценариях SQL-системы сохраняют преимущество, а когда NoSQL-решения являются более предпочтительными. Данное исследование призвано дать аргументированный ответ на этот вопрос.

1.2 Обзор исследований

В работе О. А. Ляшенко и др. проводится сравнительный анализ выполнения запросов в реляционной СУБД MySQL и документоориентированной MongoDB. Исследование выявляет сильные и слабые стороны каждой системы в различных сценариях работы с данными, что предоставляет практические рекомендации по их выбору [1]. В публикации А. Н. Важдая и А. А. Захаровой рассматривается потенциал PostgreSQL как суверенной технологии для систем искусственного интеллекта. Подчеркивается её надёжность, производительность и соответствие современным требованиям сложных вычислительных задач [2]. В исследовании В. Е. Сорокина анализируется механизм наследования таблиц в PostgreSQL и оценивается его эффективность для организации данных. Рассматриваются преимущества и ограничения этого подхода при работе с крупномасштабными информационными структурами [3]. В работе В. Е. Сорокина также исследуются методы хранения и эффективной обработки нечётких данных в СУБД PostgreSQL. Предлагаются решения для повышения производительности запросов к данным с неопределёнными или неточными значениями [4].

1.3 Цель исследования

Цель исследования – сравнить производительность SQL и NoSQL СУБД в ключевых сценариях работы с данными: массовая вставка, сложные выборки с соединениями и операции агрегации.

2 Материалы и методы

Для проведения сравнительного анализа производительности СУБД была применена современная реляционная система управления базами данных PostgreSQL 18.0 и документоориентированная NoSQL-система MongoDB 6.0.8. Для исследования был разработан синтетический датасет,

имитирующий реальную платформу для ведения блогов. Датасет содержит три основных сущности:

- Пользователи: 1000 уникальных записей
- Посты: 5000 текстовых публикаций (в среднем по 5 постов на пользователя)
- Комментарии: 20 000 текстовых сообщений (в среднем по 5 комментариев на пост)

Для автоматизации тестирования был использован язык программирования Python 3.12 со следующими ключевыми библиотеками:

- psycopg2-binary - драйвер для подключения к PostgreSQL;
- pymongo - драйвер для работы с MongoDB;
- faker - генерация реалистичных тестовых данных;
- time - точные замеры времени выполнения операций.

3 Результаты и обсуждения

Для проведения исследования необходимо установить PostgreSQL и MongoDB с сайтов mongodb.practicum-team.ru (рис.1) и www.postgresql.org (рис.2). При установке PostgreSQL создаётся пароль для подключения к базам данных. В рамках исследования использован пароль «postgres».

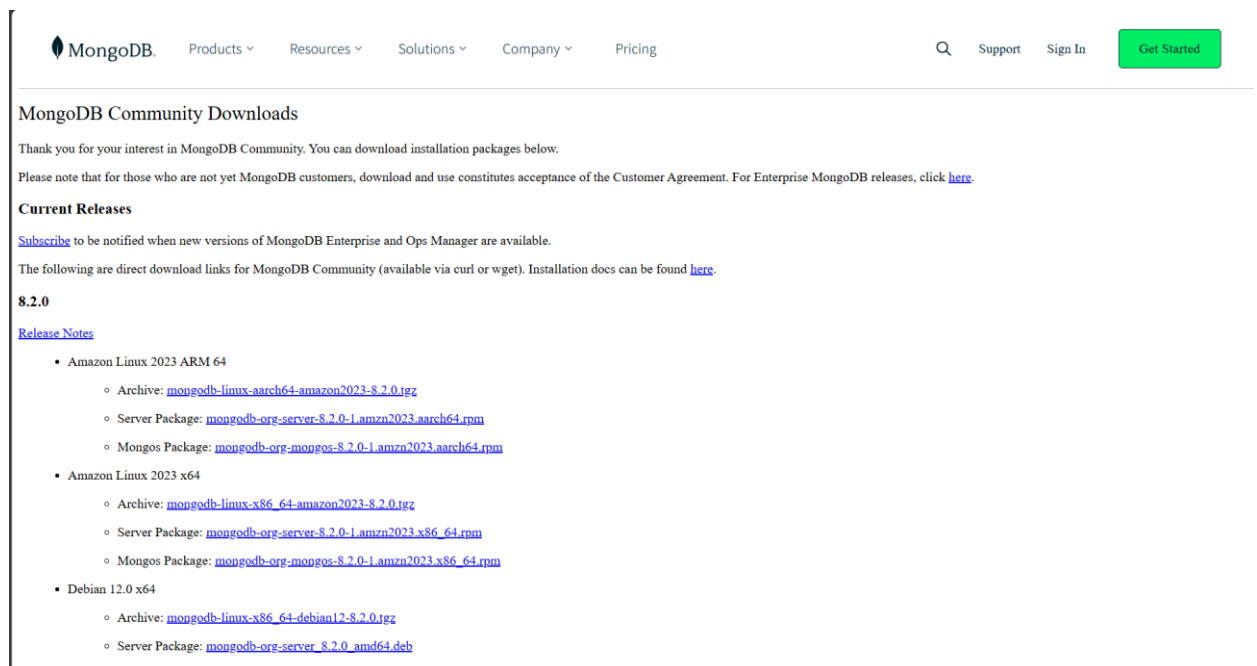


Рис 1. Сайт для установки MongoDB

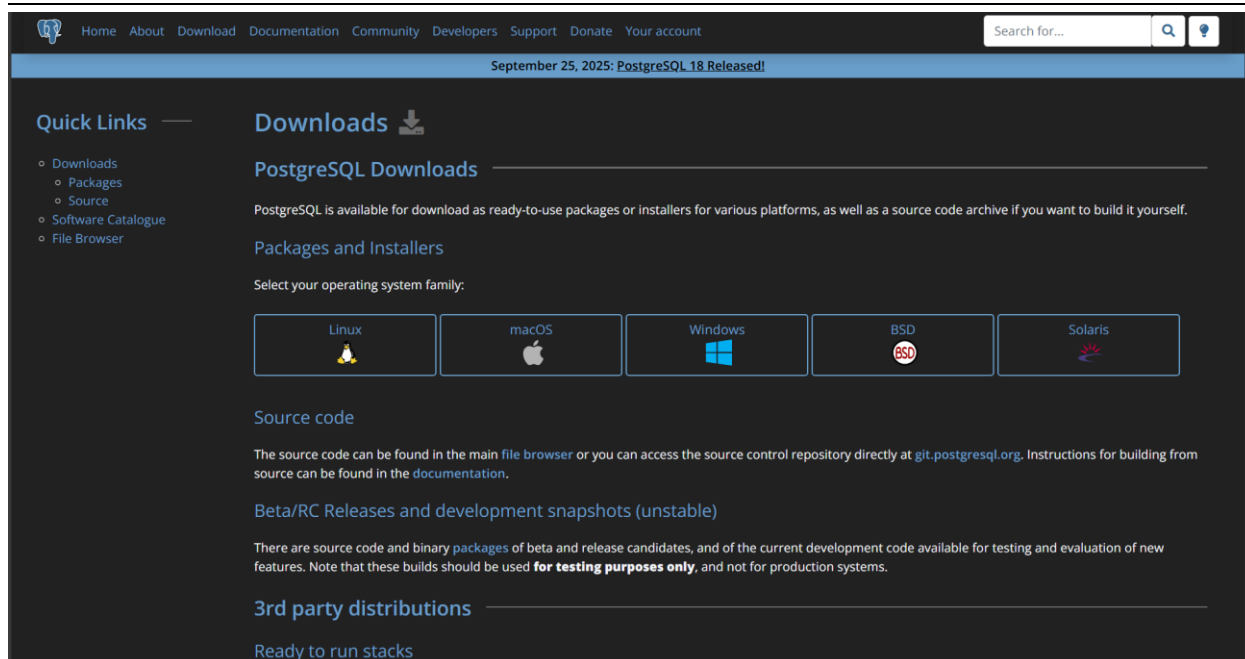


Рис 2. Сайт для установки PostgreSQL

Устанавливаем библиотеки для работы с базами данных в виртуальное окружение проекта (рис. 3).

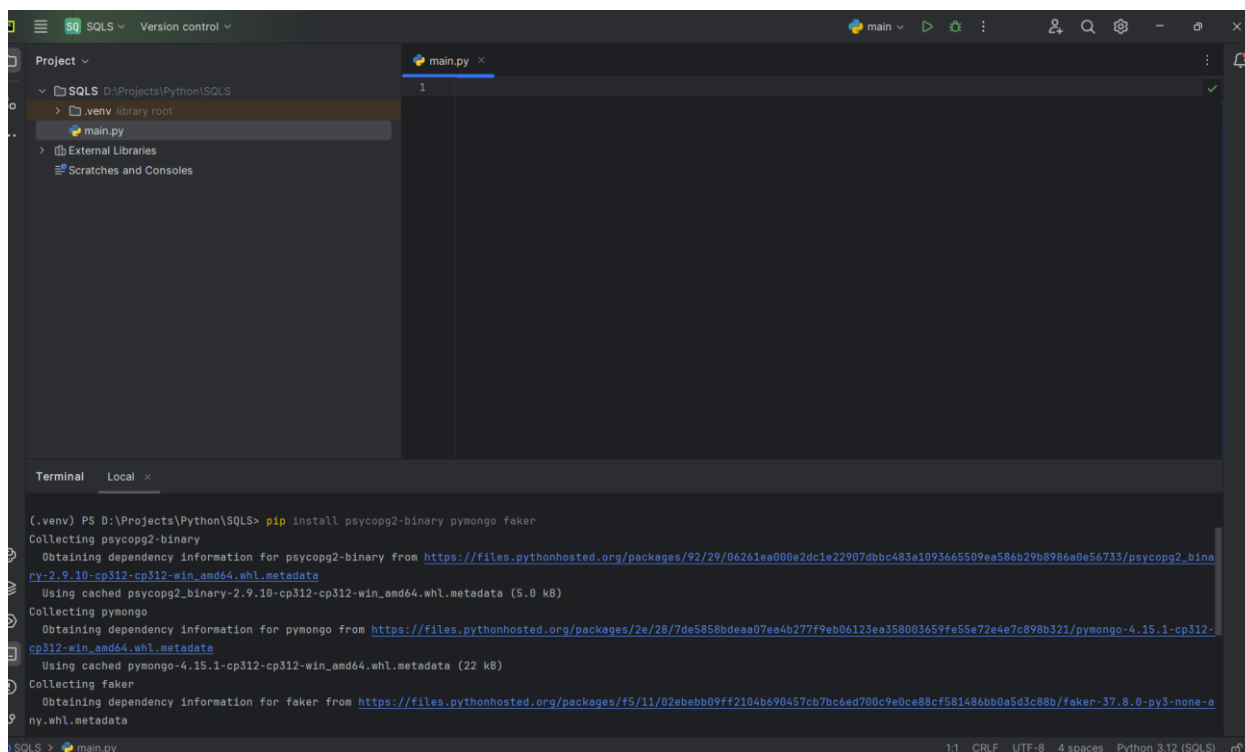


Рис 3. Установка зависимостей

Импортируем в код установленные библиотеки (рис.4).

```
import psycopg2
from pymongo import MongoClient
import time
from faker import Faker
```

Рис 4. Импортирование библиотек

Создадим класс FullPerformanceTester (рис.5) для проведения всех тестов с SQL и NoSQL СУБД. Конструктор класса будет принимать пароль для подключения к PostgreSQL.

```
class FullPerformanceTester:
    def __init__(self, postgres_password):
        self.fake = Faker()
        self.postgres_password = postgres_password
        self.results = {}
        self.num_users = 1000
        self.num_posts = 5000
        self.num_comments = 20000
```

Рис 5. Класс FullPerformanceTester

Реализуем методы setup_postgresql_full (рис.6) и setup_mongodb_full (рис.7) для установки подключения к СУБД и созданию баз данных пользователей, постов и комментариев.

```

def setup_postgresql_full(self):
    try:
        conn = psycopg2.connect(host='localhost', database='postgres', user='postgres', password=self.postgres_password, port=5432)
        conn.autocommit = True
        cursor = conn.cursor()
        cursor.execute("DROP DATABASE IF EXISTS full_test_db")
        cursor.execute("CREATE DATABASE full_test_db")
        print("✅ Базу full_test_db создана")
        cursor.close()
        conn.close()

        conn = psycopg2.connect(host='localhost', database='full_test_db', user='postgres', password=self.postgres_password, port=5432)
        cursor = conn.cursor()

        tables = [
            """CREATE TABLE users (id SERIAL PRIMARY KEY, username VARCHAR(50) UNIQUE NOT NULL, email VARCHAR(100) UNIQUE NOT NULL,
            created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP)""",
            """CREATE TABLE posts (id SERIAL PRIMARY KEY, user_id INTEGER NOT NULL, title VARCHAR(200) NOT NULL, content TEXT,
            views INTEGER DEFAULT 0, created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
            FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE)""",
            """CREATE TABLE comments (id SERIAL PRIMARY KEY, post_id INTEGER NOT NULL, user_id INTEGER NOT NULL, text TEXT NOT NULL,
            created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP, FOREIGN KEY (post_id) REFERENCES posts(id) ON DELETE CASCADE,
            FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE)"""]

        for table in tables:
            cursor.execute(table)

        indexes = [
            "CREATE INDEX idx_posts_user_id ON posts(user_id)",
            "CREATE INDEX idx_posts_created_at ON posts(created_at DESC)",
            "CREATE INDEX idx_comments_post_id ON comments(post_id)",
            "CREATE INDEX idx_comments_user_id ON comments(user_id)",
            "CREATE INDEX idx_users_username ON users(username)"]

        for index in indexes:
            cursor.execute(index)
        conn.commit()
        cursor.close()
        conn.close()
        return True

    except Exception as e:
        print(f"❌ Ошибка настройки PostgreSQL: {e}")
        return False

```

Рис 5. Метод setup_postgresql_full

```

def setup_mongodb_full(self):
    try:
        client = MongoClient(host='localhost', port=27017)
        client.drop_database('full_test_db')
        db = client['full_test_db']
        collections = ['users', 'posts', 'comments']
        for collection in collections:
            db.create_collection(collection)
        db.users.create_index([("username", 1)], unique=True)
        db.users.create_index([("email", 1)], unique=True)
        db.posts.create_index([("user_id", 1)])
        db.posts.create_index([("created_at", -1)])
        db.comments.create_index([("post_id", 1)])
        db.comments.create_index([("user_id", 1)])
        client.close()
        return True

    except Exception as e:
        print(f"❌ Ошибка настройки MongoDB: {e}")
        return False

```

Рис 6. Метод setup_mongodb_full

Устанавливаться соединение будет в отдельном методе setup_databases (рис.7).

```

1 usage

def setup_databases(self):
    postgres_ok = self.setup_postgresql_full()
    mongo_ok = self.setup_mongodb_full()
    return postgres_ok, mongo_ok

```

Рис 7. Метод setup_databases

В методе generate_complex_data (рис.8) будет прописана логика генерации данных для заполнения баз данных с помощью библиотеки faker.

```

def generate_complex_data(self):
    print(f"\n🔍 Генерируем сложные тестовые данные...")
    print(f" Пользователи: {self.num_users}")
    print(f" Посты: {self.num_posts}")
    print(f" Комментарии: {self.num_comments}")
    users = []
    for i in range(self.num_users):
        users.append({'id': i + 1, 'username': f"user_{i + 1}", 'email': f"user_{i + 1}@example.com",
                     'created_at': self.fake.date_time_this_year()})
    posts = []
    for i in range(self.num_posts):
        user_id = self.fake.random_int( min= 1, self.num_users)
        posts.append({'id': i + 1, 'user_id': user_id, 'title': self.fake.sentence(nb_words=8),
                     'content': self.fake.text(max_nb_chars=500), 'views': self.fake.random_int( min= 0, max= 10000),
                     'created_at': self.fake.date_time_this_year()})
    comments = []
    for i in range(self.num_comments):
        post_id = self.fake.random_int( min= 1, self.num_posts)
        user_id = self.fake.random_int( min= 1, self.num_users)
        comments.append({'id': i + 1, 'post_id': post_id, 'user_id': user_id, 'text': self.fake.text(max_nb_chars=200),
                        'created_at': self.fake.date_time_this_year()})
    print(f"✅ Генерировано: {len(users)} пользователей, {len(posts)} постов, {len(comments)} комментариев")
    return users, posts, comments

```

Рис 8. Метод generate_complex_data

Реализуем методы test_postgres_complex (рис.9) и test_mongodb_complex (рис.10) для проведения всех тестов: массовая вставка, сложные выборки с соединениями и операции агрегации.

```

def test_postgres_complex(self, users, posts, comments):
    try:
        conn = psycopg2.connect(host='localhost', database='full_test_db', user='postgres', password=self.postgres_password, port=5432)
        cursor = conn.cursor()
        results = {}
        print("\n TECT 1: Массовая вставка данных")
        start_time = time.time()
        user_values = [(u['id'], u['username'], u['email'], u['created_at']) for u in users]
        cursor.executemany("INSERT INTO users (id, username, email, created_at) VALUES (%s, %s, %s, %s)", user_values)
        post_values = [(p['id'], p['user_id'], p['title'], p['content'], p['views'], p['created_at']) for p in posts]
        cursor.executemany("INSERT INTO posts (id, user_id, title, content, views, created_at) VALUES (%s, %s, %s, %s, %s, %s)", post_values)
        comment_values = [(c['id'], c['post_id'], c['user_id'], c['text'], c['created_at']) for c in comments]
        cursor.executemany("INSERT INTO comments (id, post_id, user_id, text, created_at) VALUES (%s, %s, %s, %s, %s)", comment_values)
        conn.commit()
        insert_time = (time.time() - start_time) * 1000
        results['mass_insert'] = insert_time
        print(f" Mассовая вставка: {insert_time:.2f} мс")
        print("\n TECT 2: Сложные JOIN-запросы")
        join_query = """SELECT u.username, p.title, p.views, COUNT(c.id) as comment_count, (SELECT COUNT(*) FROM posts p2 WHERE p2.user_id = u.id) as user_post_count
FROM posts p JOIN users u ON p.user_id = u.id LEFT JOIN comments c ON p.id = c.post_id WHERE p.views > 100 GROUP BY u.id, p.id, u.username, p.title, p.views
ORDER BY p.views DESC LIMIT 20"""
        times = []
        for i in range(5):
            start_time = time.time()
            cursor.execute(join_query)
            results_data = cursor.fetchall()
            query_time = (time.time() - start_time) * 1000
            times.append(query_time)
            if i == 0:
                print(f" Первый запуск: {query_time:.2f} мс, найдено {len(results_data)} записей")
        avg_join_time = sum(times) / len(times)
        results['complex_join'] = avg_join_time
        print(f" Сложные JOIN: {avg_join_time:.2f} мс (среднее)")
        print("\n TECT 3: Агрегационные запросы")
        agg_query = """SELECT u.username, COUNT(p.id) as post_count, COUNT(c.id) as comment_count, AVG(p.views) as avg_views, MAX(p.views) as max_views
FROM users u LEFT JOIN posts p ON u.id = p.user_id LEFT JOIN comments c ON u.id = c.user_id GROUP BY u.id, u.username HAVING COUNT(p.id) > 8
ORDER BY post_count DESC LIMIT 15"""
        times = []
        for i in range(5):
            start_time = time.time()
            cursor.execute(agg_query)
            agg_results = cursor.fetchall()
            agg_time = (time.time() - start_time) * 1000
            times.append(agg_time)
            if i == 0:
                print(f" Первый запуск: {agg_time:.2f} мс")
        avg_agg_time = sum(times) / len(times)
        results['aggregation'] = avg_agg_time
        print(f" Агрегации: {avg_agg_time:.2f} мс (среднее)")
        cursor.close()
        conn.close()
        return results
    except Exception as e:
        print(f" Ошибка тестирования PostgreSQL: {e}")
        return None

```

Рис 9. Метод test_postgres_complex

```

def test_mongodb_complex(self, users, posts, comments):
    try:
        client = MongoClient(host='localhost', port=27017)
        db = client['full_test_db']
        results = {}

        print("\n📌 ТЕСТ 1: Массовая вставка данных")
        start_time = time.time()
        if users: db.users.insert_many(users)
        if posts: db.posts.insert_many(posts)
        if comments: db.comments.insert_many(comments)
        insert_time = (time.time() - start_time) * 1000
        results['mass_insert'] = insert_time
        print(f"✅ Массовая вставка: {insert_time:.2f} мс")

        print("\n📌 ТЕСТ 2: Сложные запросы с агрегацией")
        times = []
        for i in range(5):
            start_time = time.time()
            popular_posts = list(db.posts.find({"views": {"$gt": 100}}).sort("views", -1).limit(20))
            results_data = []
            for post in popular_posts:
                author = db.users.find_one({"_id": post['user_id']})
                comment_count = db.comments.count_documents({"post_id": post['id']})
                user_post_count = db.posts.count_documents({"user_id": post['user_id']})
                results_data.append({'username': author['username'] if author else 'Unknown', 'title': post['title'],
                                    'views': post['views'], 'comment_count': comment_count, 'user_post_count': user_post_count})
            query_time = (time.time() - start_time) * 1000
            times.append(query_time)

            if i == 0: print(f"📄 Первый запуск: {query_time:.2f} мс, найденно {len(results_data)} записей")
        avg_query_time = sum(times) / len(times)
        results['complex_join'] = avg_query_time
        print(f"✅ Сложные запросы: {avg_query_time:.2f} мс (среднее)")

        print("\n📌 ТЕСТ 3: Агрегационные pipeline")
        pipeline = [{"lookup": {"from": "posts", "localField": "id", "foreignField": "user_id", "as": "user_posts"}},
                    {"$lookup": {"from": "comments", "localField": "id", "foreignField": "user_id", "as": "user_comments"}},
                    {"$project": {"username": 1, "post_count": {"$size": "$user_posts", "comment_count": {"$size": "$user_comments"}, "avg_views": {"$avg": "$user_posts.views", "max_views": {"$max": "$user_posts.views"}}}},
                    {"$match": {"post_count": {"$gt": 0}}},
                    {"$sort": {"post_count": -1}},
                    {"$limit": 15}]

        times = []
        for i in range(5):
            start_time = time.time()
            agg_results = list(db.users.aggregate(pipeline))
            agg_time = (time.time() - start_time) * 1000
            times.append(agg_time)
            if i == 0:
                print(f"📄 Первый запуск: {agg_time:.2f} мс")
        avg_agg_time = sum(times) / len(times)
        results['aggregation'] = avg_agg_time
        print(f"✅ Агрегация: {avg_agg_time:.2f} мс (среднее)")
        client.close()
        return results

    except Exception as e:
        print(f"❌ Ошибка тестирования MongoDB: {e}")
        return None

```

Рис 10. Метод test_mongodb_complex

С помощью метода print_detailed_results (рис.11) выведем результаты всех тестов в виде таблицы, где будет видно, за какое время выполнили ту или иную операцию СУБД, кто выигрывает по скорости и насколько.

```

def print_detailed_results(self):
    print("\n" + "=" * 70)
    print("🔍 ДЕТАЛИЗИРОВАННЫЕ РЕЗУЛЬТАТЫ ТЕСТИРОВАНИЯ")
    print("=" * 70)
    if not self.results:
        print("❌ Нет результатов для отображения")
        return
    print(f"{'ТЕСТ':<25} {'PostgreSQL':<12} {'MongoDB':<12} {'ПОБЕДИТЕЛЬ':<12} {'ПРЕИМУЩЕСТВО':<12}")
    print("-" * 70)
    tests = [
        ('mass_insert', 'Массовая вставка'),
        ('complex_join', 'Сложные запросы'),
        ('aggregation', 'Агрегации')
    ]
    for test_key, test_name in tests:
        pg_time = self.results.get('PostgreSQL', {}).get(test_key)
        mongo_time = self.results.get('MongoDB', {}).get(test_key)
        if pg_time and mongo_time:
            if pg_time < mongo_time:
                winner = "PostgreSQL"
                advantage_pct = f"{{{mongo_time - pg_time} / mongo_time * 100:.1f}%"
            else:
                winner = "MongoDB"
                advantage_pct = f"{{{pg_time - mongo_time} / pg_time * 100:.1f}%"
            print(f"{'test_name':<25} {'pg_time':<10.2f}мс {'mongo_time':<10.2f}мс {'winner':<12} {'advantage_pct':<12}")
        elif pg_time:
            print(f"{'test_name':<25} {'pg_time':<10.2f}мс {'N/A':<12} {'PostgreSQL':<12} {'-':<12}")
        elif mongo_time:
            print(f"{'test_name':<25} {'N/A':<12} {'mongo_time':<10.2f}мс {'MongoDB':<12} {'-':<12}")
    print("\n" + "=" * 70)

```

Рис 11. Метод print_detailed_results

Метод run_full_performance_test (рис.12) сообщит о завершении всех тестов перед выводом результатов.

```

def run_full_performance_test(self):
    print("=" * 70)
    print("🚀 ПОЛНОЕ ТЕСТИРОВАНИЕ ПРОИЗВОДИТЕЛЬНОСТИ БАЗ ДАННЫХ")
    print("=" * 70)
    postgres_ok, mongo_ok = self.setup_databases()
    if not postgres_ok and not mongo_ok:
        print("❌ Не удалось настроить базы данных!")
        return
    users, posts, comments = self.generate_complex_data()
    if postgres_ok:
        pg_results = self.test_postgresql_complex(users, posts, comments)
        if pg_results:
            self.results['PostgreSQL'] = pg_results
            print("\n✅ PostgreSQL: все тесты завершены")
    if mongo_ok:
        mongo_results = self.test_mongodb_complex(users, posts, comments)
        if mongo_results:
            self.results['MongoDB'] = mongo_results
            print("\n✅ MongoDB: все тесты завершены")
    self.print_detailed_results()

```

Рис 12. Метод run_full_performance_test

Теперь создадим функцию `main` (рис.13), которая проведёт полное тестирование и передаст в объект класса пароль для подключения к PostgreSQL, и вызовем её.

```
def main():
    print("🔥 ЗАПУСК ПОЛНОГО ТЕСТИРОВАНИЯ ПРОИЗВОДИТЕЛЬНОСТИ")
    password = 'postgres'
    print("⌚ Запускаем полное тестирование... Это может занять несколько минут\n")
    tester = FullPerformanceTester(password)
    tester.run_full_performance_test()

main()
```

Рис 13. Функция `main`

Прежде чем запустить код, нужно убедиться, что серверы PostgreSQL и MongoDB активны. Для этого достаточно открыть диспетчер служб (рис.14) и найти в списке служб `postgresql-x64-18` и `MongoDB Server`.

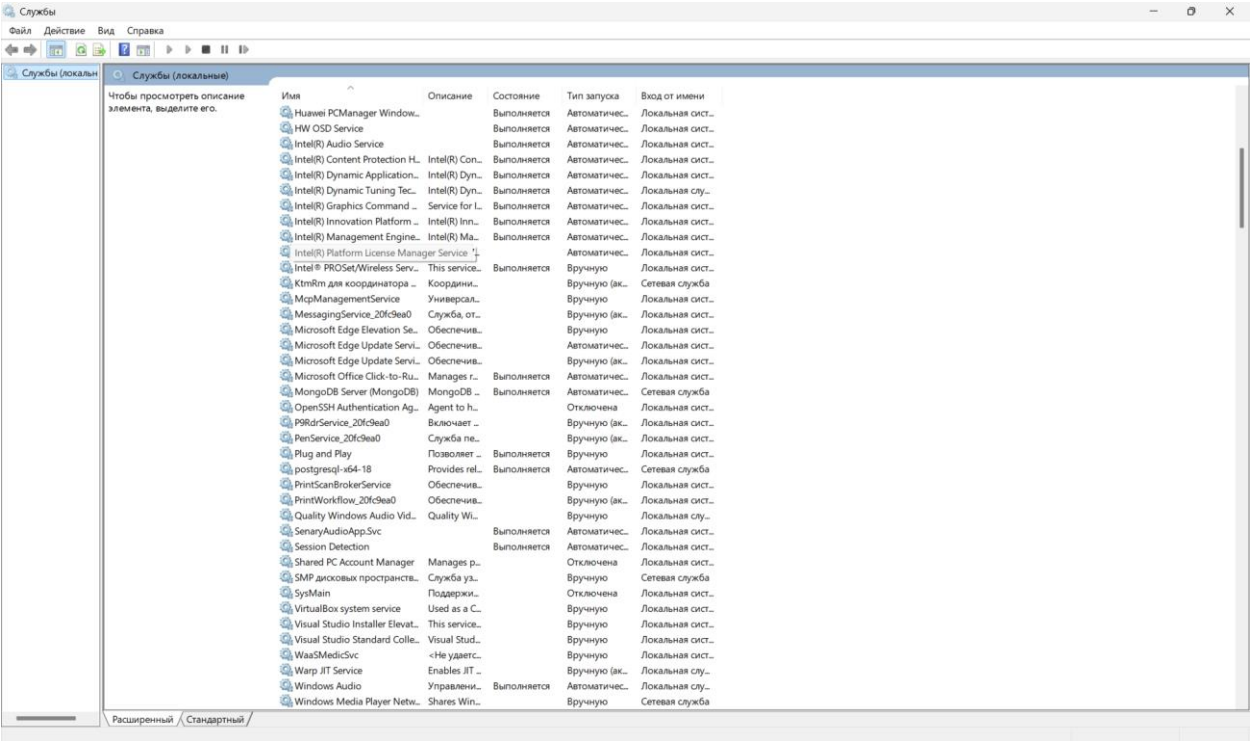


Рис 14. Диспетчер служб Windows

После проверки запускаем код и получаем результат в виде таблицы (табл. 1).

Таблица 1 – Время выполнения операций в SQL и NoSQL СУБД

ТЕСТ	PostgreSQL	MongoDB	ПОБЕДИТЕЛЬ	ПРЕИМУЩЕСТВО
Массовая вставка	3677.06 мс	490.15 мс	MongoDB	+86.7%

Сложные запросы	32.59 мс	106.09 мс	PostgreSQL	+69.3%
Агрегации	28.41 мс	173.32 мс	PostgreSQL	+83.6%

4 Выводы

Проведенное исследование выявило четкое разделение областей преимуществ СУБД: MongoDB значительно превосходит PostgreSQL в скорости массовой вставки данных (быстрее на 86.7%), что объясняется ее оптимизацией на быструю запись документов. Однако PostgreSQL демонстрирует полное доминирование в операциях, требующих сложных соединений и агрегаций, выполняя их на 69.3% и 83.6% быстрее соответственно, благодаря мощному оптимизатору запросов и реляционной модели данных. Таким образом, выбор СУБД должен определяться конкретными задачами: MongoDB предпочтительнее для сценариев с интенсивной записью неструктурированных данных, а PostgreSQL — для систем с сложными запросами и строгими требованиями к целостности данных.

Библиографический список

1. Ляшенко О. А., Конашков О. О., Солодкая Н. А. Сравнительный анализ выполнения запросов к серверам баз данных MySQL и MongoDB //Вестник Херсонского национального технического университета. 2019. №. 4 (71). С. 114-124.
2. Важдаев А. Н., Захарова А. А. Проект разработки и применения суверенных технологий баз данных PostgreSQL для систем искусственного интеллекта //Современное образование: интеграция образования, науки, бизнеса и власти. 2022. С. 93-96.
3. Сорокин В. Е. Об эффективности наследования таблиц в СУБД PostgreSQL //Программные продукты и системы. 2016. №. 3 (115). С. 15-23.
4. Сорокин В. Е. Хранение и эффективная обработка нечетких данных в СУБД PostgreSQL //Программные продукты и системы. 2017. Т. 30. №. 4. С. 609-618.