Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ Информатика и системы управления

КАФЕДРА Системы обработки информации и управления

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОЙ РАБОТЕ

НА ТЕМУ:

Модель конкурентного доступа к таблице базы данных на многоядерном процессоре

Студент ИУ5-73Б _____ Пермяков Д.К.
(Группа) (Подпись, дата) (И.О.Фамилия)

Руководитель курсовой работы _____ Чёрненький М.В.
(Подпись, дата) (И.О.Фамилия)

Консультант _____ _____
(Подпись, дата) (И.О.Фамилия)

2024г.

Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

УТВЕРЖДАЮ
Заведующий кафедрой _____
(Индекс)
_____ _____
(И.О.Фамилия)
« _____ » _____ 2024 г.

ЗАДАНИЕ
на выполнение курсовой работы

по дисциплине __Имитационное моделирование дискретных процессов_____

Студенты группы _____*ИУ5-73Б*_____

_____*Пермяков Дмитрий Кириллович*_____
(Фамилия, имя, отчество)

Тема курсовой работы **Модель конкурентного доступа к таблице базы данных на многоядерном процессоре**

Направленность КР (учебная, исследовательская, практическая, производственная, др.)
_____**учебная**_____
Источник тематики (кафедра, предприятие, НИР) ____*кафедра*_____

График выполнения работы: 25% к ___ нед., 50% к ___ нед., 75% к __ нед., 100% к ___ нед.

Задание *__Разработать имитационную модель конкурентного доступа к таблице базы данных на многоядерном процессоре, провести эксперимент*

Оформление курсовой работы:
Расчетно-пояснительная записка на __23__ листах формата А4.

Дата выдачи задания «___ » _____ 2024 г.

Студент ____ИУ5-73Б___ _____ Пермяков Д.К.
(Группа) (Подпись, дата) (И.О.Фамилия)

Руководитель курсовой работы _____ Чёрненький М.В.
(Подпись, дата) (И.О.Фамилия)

Примечание: Задание оформляется в двух экземплярах: один выдается студенту, второй хранится на кафедре.

АННОТАЦИЯ

В данной проектной работе разработана имитационная модель конкурентного доступа к таблице базы данных, управляемая реляционная СУБД, с акцентом на использование многоядерных процессоров. Основное внимание уделяется анализу производительности системы при одновременном выполнении множества запросов, что является критически важным для обеспечения высокой пропускной способности и минимизации времени отклика.

В ходе исследования разрабатывается обоснованная модель, которая учитывает различные аспекты конкурентного доступа, такие как количество одновременно работающих потоков (пользователей), типы выполняемых операций (чтение, запись), а также механизмы блокировок, функцию распределения нагрузки на сервер и время работы обработчиков. Проводится анализ использования многоядерного процессора на эффективность обработки запросов, что позволяет выявить узкие места.

Работа охватывает несколько сценариев, включая различные уровни конкуренции за ресурсы, что позволяет оценить влияние параллелизма на производительность базы данных. Исследуется сравнительный анализ, основанный на распределении нагрузки между несколькими ядрами процессора.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Архитектура численного моделирования рабочих процессов в системах конкурентного доступа к данным определяет основные модули, процессы, процедуры и функции, их иерархию, взаимосвязь, способ реализации необходимых характеристик сквозного процесса, внешних воздействий и задач оптимизационного поиска в соответствии с заданными критериями эффективности. Если в результате выбора и действия противоположно направленных критериев оптимизации оптимальное решение не может быть найдено, то ищется максимально гармонизированное решение, позволяющее учесть каждый выбранный критерий вне зависимости от порядка их применения и найти решение за установленное время с использованием выбранных аппаратных средств.

Современный уровень развития информационных технологий позволяет и предопределяет использование методов численного моделирования динамических рабочих процессов в системах конкурентного доступа. Для нее численное моделирование является одной из возможностей оценки качества системы и прогнозирования ее взаимодействия с внешней средой в виде пользовательских запросов. Метод численного моделирования позволяет существенно сократить затраты в ходе анализа, оценки, модификации и реализации архитектуры деятельности конкурентного доступа.

Современные серверные архитектуры нуждаются в быстрой обработке, поскольку им приходится работать с большими данными и время отклика обратно пропорционально удовлетворенности пользователей, что сказывается на размере аудитории и доходах от продукта.

В курсовой работе уделено вниманию асинхронному доступу к таблице базы данных на современных компьютерах, имеющих несколько ядер процессора. Как организован доступ к разделяемым данным и как увеличить количество обработанных запросов пользователей будет рассмотрено ниже.

1. Постановка задачи

В процессе работы СУБД, которая обслуживает веб-приложение, возникла проблема с производительностью. Система обрабатывает большое количество запросов от пользователей, и время отклика на запросы значительно увеличилось в последнее время. Это связано с тем, что одновременно обращается большое количество пользователей N, что создает конкуренцию за ресурсы базы данных.

Необходимо смоделировать конкурентный доступ к таблице базы данных, чтобы спрогнозировать производительность системы при высоком уровне параллельных запросов и предложить способы увеличения количества обработанных запросов за тот же интервал времени. Учитывать, что есть одна серверная машина, используемая для хранения базы данных, и процессор такой машины многоядерный.

Нужно сформулировать эмпирическою гипотезу H0 с 1 фактором, провести статистический отсеивающий эксперимент для ее проверки.

2. Цели и задачи проекта

- Оценить влияние количества ядер процессора на скорость выполнения запроса:
    - Измерить среднее время выполнения N запросов для 1 ядра, 4 ядер и 8 ядер.
    - Сравнить статистические замеры времени выполнения запросов для разного количества ядер.
- Определить способы разрешения конкуретного доступа к данным при большом количестве запросов:
    - Привести способы разрешения.
    - Сравнить их и сделать вывод.
- Предложить способы оптимизации конкурентного доступа к таблице базы данных с главной целью – увеличить количество обработанных пользовательских запросов в единицу времени.

3. Результаты моделирования

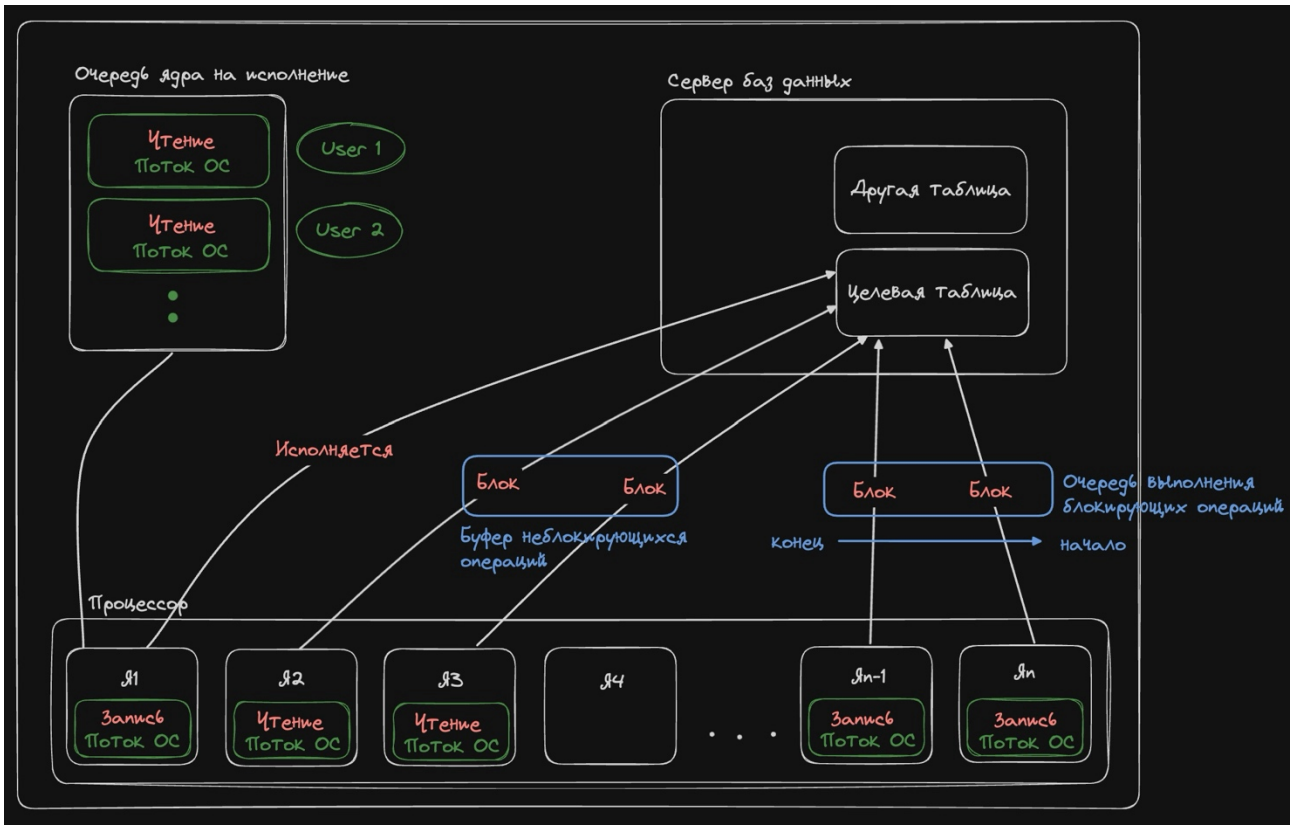

Рисунок 1 – Схема модели конкурентного доступа к таблице базы данных.

В упрощенной схеме у каждого ядра процессора есть своя очередь (в реальной модели есть планировщик, который управляет распределением ресурсов между программами), и мутирующие операции создают очередь выполнения операций с базой данных (на рис. 2 обозначено синим цветом). Неблокирующая операция, такая как чтение, может выполняться без блокировки, а запись с блокировкой.

Приведем логи работы модели. У нас есть некоторое количество пользователей, делающих разные типы запросов (чтение или запись) в базу данных. Симуляция очереди ожидания запроса организована с помощью семафора, где количество параллельных операций ограничено количеством ядер процессора машины.

Рисунок 2 – Логирование работы модели.

Время запросов пользователей имеет треугольное распределение (min=0.5; max=3.5; mod=1.0), а их количество случайное распределение, поскольку в небольшом интервале времени это значение действительно случайное.

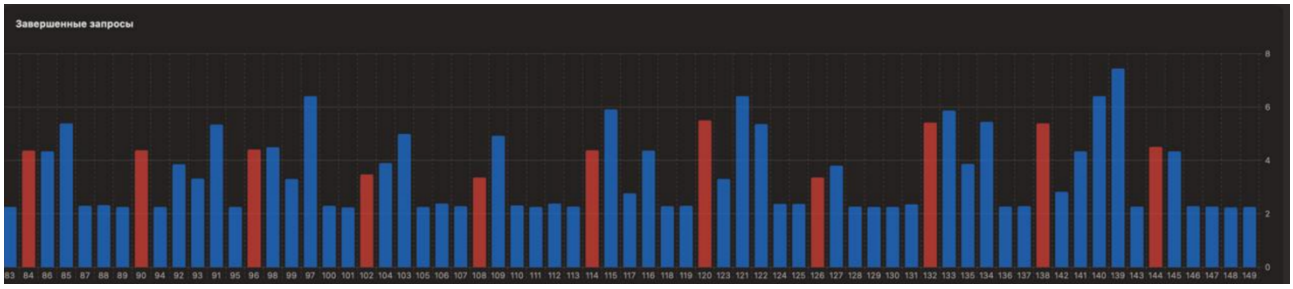

Рисунок 3. Треугольное распределение количества запросов пользователей.

Запросы на чтение и запись также поступают на исполнение в случайном порядке. Модель работает без таймаута на исполнение инструкций на ядре процессора, то есть если операция попала на исполнение, то она гарантированно сразу выполнится. Это имеет очевидные недостатки в случае IO-bound нагрузки, да и в случае CPU-нагрузки малые по размеру батчи инструкции будут ожидать длительные.

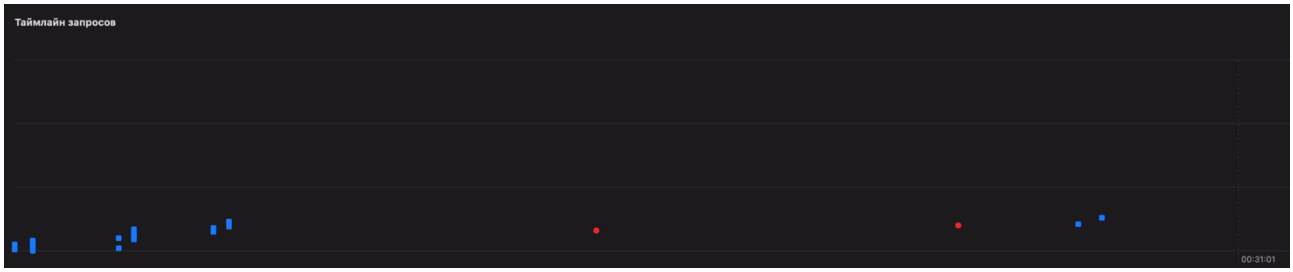

Рисунок 4 – Таймлайн запросов.

Также строим столбчатую диаграмму, отображающую количество читающих и записывающих операций в данный момент.

Рисунок 5 – Активность читающих и пишущих операций.

В модели асинхронного программирования важно оценивать, насколько много потоков ожидают такта процесса для исполнения. Для этого мы построили линейчатую диаграмму, изменяющуюся во времени и показывающая по оси Y динамику изменения размера очереди на исполнение.

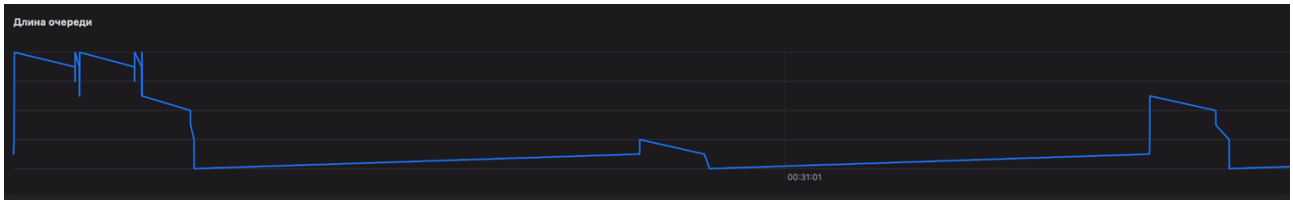

Рисунок 6 – Диаграмма длины очереди.

4. Статистическая гипотеза

Эмпирическая гипотеза H0 сформулирована с учетом двух факторов, связанных с конкурентным доступом к таблице базы данных:

1. Количество ядер процессора.

Сформулируем гипотезу H0 следующим образом: "Среднее время выполнения запросов к таблице базы данных зависит от количества ядер процессора при большой нагрузке".

Рассмотрим сервер с одним ядром. Очередь достигает максимального размера в 1 операцию, что соответствует использованию семафора в нашей модели.

Рисунок 7 – Сервер с одноядерным процессором. Длина очереди.

Количество запросов изменяется по линейному запросу, что в целом и логично, поскольку у нас может выполняться одна операция в определенный момент времени.

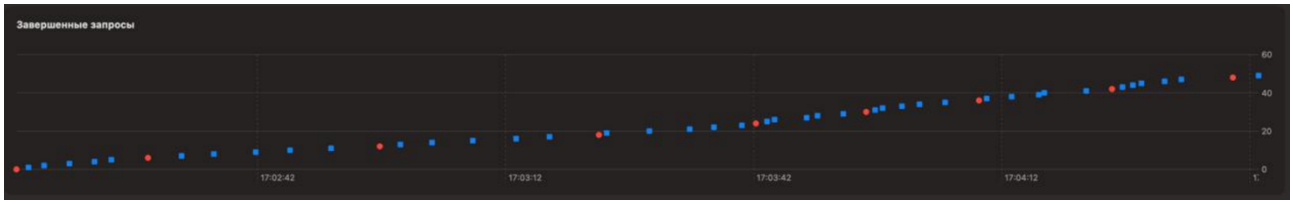

Рисунок 8 – Сервер с одноядерным процессором. Завершенные запросы.

Рассмотрим процессор с 4-мя ядрами. Очередь моментами достигает 3 операций, что удовлетворяет требованию семафора.

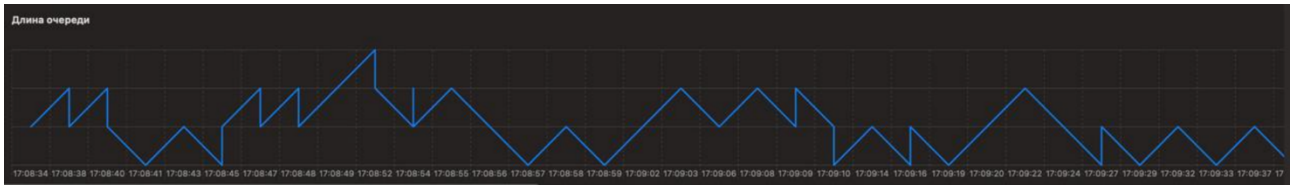

Рисунок 9 – Сервер с четырехядерным процессором. Длина очереди.

С 4 ядрами появилась кучность выполнения некоторых запросов на чтение, поскольку типы множества операций случайно, то кучность не выражена очень сильно. Время исполнения программы уменьшилось. Посмотрим, что будет в случае 8 ядер.

Рисунок 10 – Сервер с четырехядерным процессором. Завершенные запросы.

Длина очереди также возросла. Время исполнения программы уменьшилось.

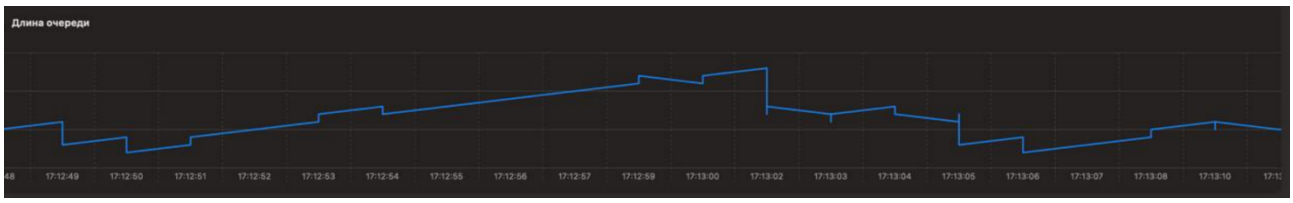

Рисунок 11 – Сервер с восьмиядерным процессором. Длина очереди.

Кучность выполнения запросов сильно возросла. Видно, как ожидающие операции на чтение выполняются батчем на ядрах процессора.

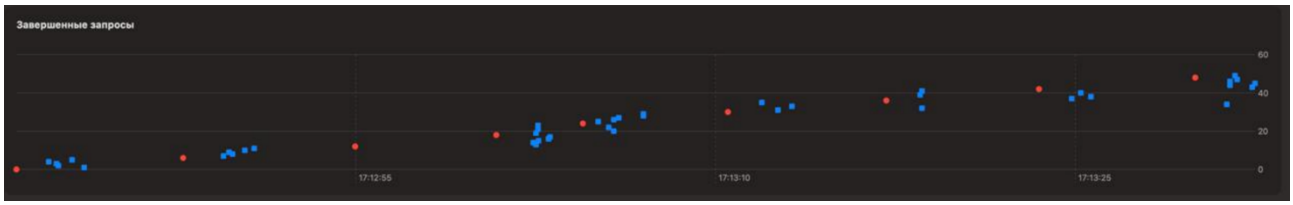

Рисунок 12 – Сервер с восьмиядерным процессором. Время завершения процессов.

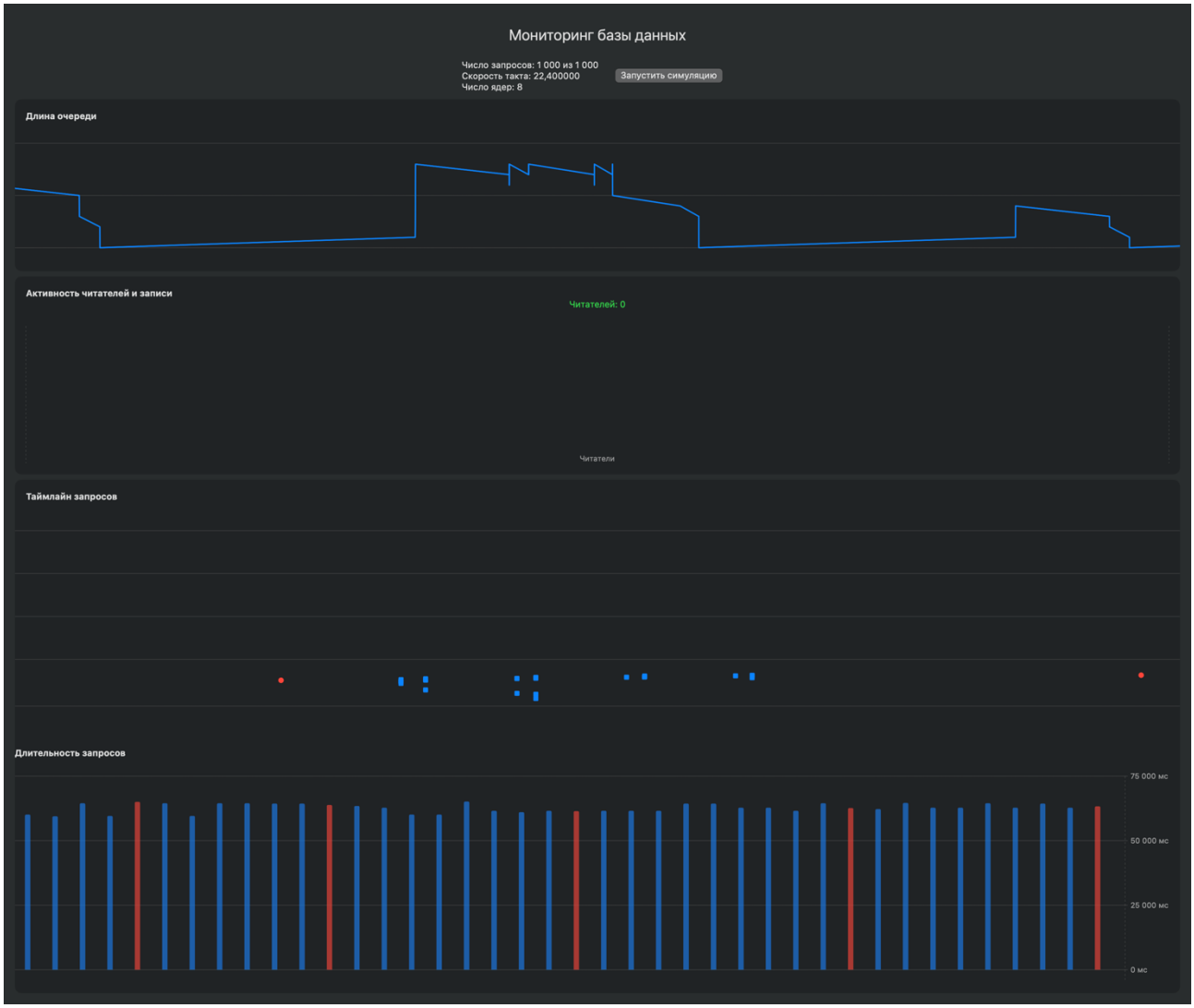

Рисунок 13 – Результат выполнения программы для 1000 запросов.

5. Рекомендации на основе выявленных влияющих факторов

Оптимизация асинхронного доступа к таблице базы данных может быть успешно реализована, учитывая, что некоторые оптимизации требуют не столько разработки, сколько обновления вычислительных мощностей машины. Это касается только отдельно взятого сервера. Шардирование не требует высокопроизводительная процессора, за счет чего размер кластера может быть достаточно большим, операции мутирования и чтения более изолированными, а стоимость более оптимальной, так как горизонтальное масштабирование обходится дешевле вертикального, и функция эффективности от цены лучше возрастает.

Стоит асинхронное получение данных таблицы базы данных рассматривать только в контексте многопользовательского доступа. Обеспечить разрешение большего количества запросов от разных пользователей возможно с помощью переключения контекста выполнения на уровне программы, а не операционной системы. Немалое количество языков программирования имеют коорутины – легковесные потоки, создающиеся на уровне программы и управляемые планировщиком языка.

Также стоит создать репликации базы данных, которые можно поставить в режим read-only, что позволит выполнять идемпотентные операции чтения без ожидания блокировок мутирующих операций. И для операции чтения можно создать индексы, чтобы быстрей выполнять работу.

Также можно увеличить тактовую частоту процессора, за счет чего будет больше выполняться инструкций процессора за секунду.

И последняя оптимизация – блокировать не всю таблицу базы данных, а отдельные строки, чтобы другие потоки ОС могли работать с этой же таблицей, но с другой ее частью. Это порождает первышение по памяти, но увеличивает скорость работы с данными базы данных.

Список использованных источников

1. PostgreSQL 16 изнутри [Электронный ресурс] // Documentation. URL: https://libgen.is/book/index.php?md5=42E04008C9ED1490FD4BF1A572B3843A (дата обращения 02.11.2024).

2. Многоядерные процессоры: как распределение нагрузки между ядрами увеличивает производительность [Электронный ресурс] // URL: https://www.mobilis.ru/news/mnogoyadernye-protsessory-kak-raspredelenie-nagruzki-mezhdu-yadrami-uvelichivaet-proizvoditelnost/ (дата обращения 02.11.2024).

3. Такие удивительные семафоры [Электронный ресурс] // URL: https://habr.com/ru/articles/261273/

4. Работа с таймерами в программировании на Swift [Электронный ресурс] // URL: https://swiftblog.org/rabota-s-tajmerami/

ПРИЛОЖЕНИЕ А

```swift
//
//  DatabaseMonitoringServer.swift
//  MulticoreDatabase
//
//  Created by Dmitriy Permyakov on 16.12.2024.
//

import SwiftUI
import Foundation

final class DatabaseMonitoringServer: ObservableObject {
    @Published private(set) var readersCount = 0
    @Published private(set) var completedLog: [User] = []
    @Published private(set) var queueLengths: [(Date, Int)] = []
    private(set) var data: [String] = []

    private let queue = DispatchQueue(label: "queue", attributes:
[.concurrent])
    private let counterQueue = DispatchQueue.main
    private let readSemaphore: DispatchSemaphore
    private var writeOperationWaiting = false

    init(codesCapacity: Int) {
        self.readSemaphore = DispatchSemaphore(value: codesCapacity)
    }

    func makeOperation(user: User) {
        switch user.operation {
        case .select:
            readOperation(user: user, index: 0)
        case .write:
            writeOperation(user: user, newName: "")
        }
    }

    private func writeOperation(user: User, newName: String) {
        writeOperationWaiting = true
        while readersCount > 0 {
            Thread.sleep(forTimeInterval: Constants.waitingForWriteTimeout)
        }
        writeOperationWaiting = false

        queue.async(flags: .barrier) { [weak self] in
            guard let self else { return }
            print("[\(Date().currentTime)]: 🔐 user #\(user.id)")

            // Имитируем длительность выполнения
            Thread.sleep(forTimeInterval: Constants.writeTimeout)
            print("[\(Date().currentTime)]: 🔓 user #\(user.id)")

            // Логируем пользователя
            counterQueue.sync {
                var tempUser = user
```

```
                tempUser.finishTime = Date()
                self.completedLog.append(tempUser)
            }
        }
    }

    private func readOperation(user: User, index: Int) {
        queue.async { [weak self] in
            guard let self else { return }
            readSemaphore.wait()
            incremantReadersCount(user: user)
            Thread.sleep(forTimeInterval: Constants.readTimeout)

            // Логируем пользователя
            counterQueue.sync {
                var tempUser = user
                tempUser.finishTime = Date()
                self.completedLog.append(tempUser)
            }

            readSemaphore.signal()
            decrementReadersCount(user: user)
        }
    }

    private func incremantReadersCount(user: User) {
        counterQueue.sync { [weak self] in
            guard let self else { return }
            readersCount += 1
            queueLengths.append((Date(), readersCount))
            print("[\(Date().currentTime)]: ✅ user #\(user.id) joined (users:
\(readersCount))")
        }
    }

    private func decrementReadersCount(user: User) {
        counterQueue.sync { [weak self] in
            guard let self else { return }
            readersCount -= 1
            queueLengths.append((Date(), readersCount))
            print("[\(Date().currentTime)]: ❌ user #\(user.id) quit (users:
\(self.readersCount))")
        }
    }
}

actor DatabaseServer {
    private(set) var queueLengths: [(date: String, Int)] = []
    private(set) var completedLog: [Agent] = []
    private(set) var activeReaders = 0
    private(set) var isWriting = false

    private var currentQueueLength = 0
    private let readerSemaphore: DispatchSemaphore
    private let writerSemaphore: DispatchSemaphore
    private var writerIsWaiting = false
```

```
    private var semaphorePausedReadAgents: Set<Agent> = []

    init(capacity: Int) {
        readerSemaphore = DispatchSemaphore(value: capacity)
        writerSemaphore = DispatchSemaphore(value: 1)
    }

    func reset() {
        queueLengths.removeAll()
        completedLog.removeAll()
        semaphorePausedReadAgents.removeAll()
    }

    func processRequest(agent: Agent) async {
        print(agent.startWork)
        let timeout = agent.operation == .read ? Constants.readTime :
Constants.writeTime
        try? await Task.sleep(for: .seconds(timeout))
        print(agent.endWork(date: Date()))

        // Сохраняем успешный запрос
        var completedAgent = agent
        completedAgent.finishTime = Date()
        completedLog.append(completedAgent)
    }

    func addToQueue(agent: Agent) async {
        currentQueueLength += 1
        queueLengths.append((Date().currentTime, currentQueueLength))

        if agent.operation == .write {
            await handleWrite(agent: agent)
        } else {
            await handleRead(agent: agent)
        }

        currentQueueLength -= 1
        queueLengths.append((Date().currentTime, currentQueueLength))
    }

    private func handleWrite(agent: Agent) async {
        writerIsWaiting = true
        while activeReaders > 0 || !semaphorePausedReadAgents.isEmpty {
            print("[\(Date().currentTime)] ✍️ ⏸️ Агент #\(agent.id) ждёт
записи")
            try? await Task.sleep(for: .seconds(1))
        }
        writerIsWaiting = false

        while writerSemaphore.wait(timeout: .now() + 1) == .timedOut {
            print("[\(Date().currentTime)] 👀 ⚙️ Агент #\(agent.id) не хватила
потока. Ждёт освобождения")
            try? await Task.sleep(for: .seconds(Constants.pauseTime))
        }

        isWriting = true
```

```
        print("[\(Date().currentTime)] 🔐 Агент #\(agent.id) Блокирует для
записи 🔐")

        await processRequest(agent: agent)

        isWriting = false
        writerSemaphore.signal()
        print("[\(Date().currentTime)] 🔓 Агент #\(agent.id) Разблокировал 🔓")
    }

    private func handleRead(agent: Agent) async {
        while isWriting || writerIsWaiting {
            print("[\(Date().currentTime)] 👀 ⏸️ Агент #\(agent.id) ждёт в
очереди на чтение")
            try? await Task.sleep(for: .seconds(Constants.pauseTime))
        }

        while readerSemaphore.wait(timeout: .now() + 0.5) == .timedOut {
            semaphorePausedReadAgents.insert(agent)
            print("[\(Date().currentTime)] 👀 ⚙️ Агент #\(agent.id) не хватила
потока. Ждёт освобождения")
            try? await Task.sleep(for: .seconds(Constants.pauseTime))
        }

        // Удаляем отложенную задачу если она там есть
        if let index = semaphorePausedReadAgents.firstIndex(where: { $0.id ==
agent.id }) {
            print("[\(Date().currentTime)] 👀 ⚙️ Агент #\(agent.id) УДАЛЁНА из
отложенных")
            semaphorePausedReadAgents.remove(at: index)
        }
        activeReaders += 1
        print("[\(Date().currentTime)] 👀 ➕ Агент #\(agent.id) захватил
очередь. (всего: \(activeReaders))")

        await processRequest(agent: agent)
        activeReaders -= 1

        readerSemaphore.signal()
        print("[\(Date().currentTime)] 👀 ➖ Агент #\(agent.id) освободил
очередь. (осталось: \(activeReaders))")
    }
}

// MARK: - Constants

enum Constants {
    /// Время выполнения одного процесса
    static let processTime = 3
    /// Время симуляции (в секундах)
    static let simTime: TimeInterval = 30
    /// Средний интервал между запросами
    static let requestInterval: TimeInterval = 1
    /// Число ядер
    static let codesCapacity = 5
    /// Число пользователей
```

```
    static let numbersOfUsers = 50
    /// Время паузы между запросами
    static let pauseTime = 1
    static let sharingTime = 1
    static let readTime = 1
    static let writeTime = 3
}

@Observable
final class SimulationMonitor {
    private(set) var queueLengths: [(date: String, Int)] = []
    private(set) var activeReaders = 0
    private(set) var isWriting = false
    private(set) var completedLog: [Agent] = []
    private(set) var showLoader = false

    @ObservationIgnored
    private var task: Task<Void, Never>?
    @ObservationIgnored
    private var server = DatabaseServer(capacity: Constants.codesCapacity)

    private func reset() async {
        queueLengths = []
        completedLog = []
        activeReaders = 0
        await server.reset()
    }

    func runSimulation(numUsers: Int) async {
        guard task == nil else { return }
        await reset()

        // Мониторинг активности читателей и писателей
        task = Task {
            while true {
                let activeReaders = await server.activeReaders
                let isWriting = await server.isWriting
                await MainActor.run {
                    self.activeReaders = activeReaders
                    self.isWriting = isWriting
                }
                try? await Task.sleep(for: .seconds(0.2))
            }
        }

        // Генерация запросов
        await generateRequests(numUsers: numUsers)

        // Завершение симуляции
        await MainActor.run {
            showLoader = true
        }

        try? await Task.sleep(nanoseconds: UInt64(Constants.simTime) *
1_000_000_000)
        print("Симуляция завершена.")
        let queueLengths = await server.queueLengths
```

```
        let completedLog = await server.completedLog
        await MainActor.run {
            self.queueLengths = queueLengths
            self.completedLog = completedLog
            showLoader = false
        }
        task?.cancel()
        task = nil
    }

    private func generateRequests(numUsers: Int) async {
        for requestId in 0..<numUsers {
            try? await Task.sleep(for: .seconds(Constants.requestInterval))
            Task {
                let agent = Agent(
                    id: String(requestId),
                    operation: requestId % 6 == 0 ? .write : .read,
                    arrivalTime: Date()
                )
                await server.addToQueue(agent: agent)
            }
        }
    }
}

struct ActivityChart: View {
    var activeReaders: Int
    var isWriting: Bool

    var body: some View {
        VStack {
            HStack {
                Text("Читателей: \(activeReaders)")
                    .foregroundColor(.green)
                Spacer()
                Text(isWriting && activeReaders == 0 ? "Запись активна" : "Нет
записи")
                    .foregroundColor(isWriting ? .red : .gray)
            }
            .padding()

            Chart {
                BarMark(
                    x: .value("Активность", "Читатели"),
                    y: .value("Читатели", activeReaders)
                )
                .foregroundStyle(.green)

                BarMark(
                    x: .value("Активность", "Запись"),
                    y: .value("Запись", isWriting && activeReaders == 0 ? 1 :
0)
                )
                .foregroundStyle(.red)
            }
            .frame(height: 200)
        }
```

```
        .padding()
        .background(.background, in: .rect(cornerRadius: 10))
        .overlay(
            Text("Активность читателей и записи")
                .font(.headline)
                .padding(),
            alignment: .topLeading
        )
    }
}


struct CompletedRequestsChart: View {
    var completedLog: [Agent]

    var body: some View {
        GeometryReader { geo in
            VStack(alignment: .leading, spacing: 0) {
                Text("Завершенные запросы")
                    .font(.headline)
                    .padding()

                ScrollView(.horizontal) {
                    Chart {
                        ForEach(completedLog, id: \.id) { agent in
                            PointMark(
                                x: .value("Время завершения",
agent.finishTime!),
                                y: .value("Номер запроса", Int(agent.id) ?? 0)
                            )
                            .foregroundStyle(agent.operation == .write ? .red :
.blue)
                            .symbol(agent.operation == .write ? .circle :
.square)
                        }
                    }
                    .frame(width: geo.size.width - 32)
                    .padding()
                }
            }
        }
        .frame(minHeight: 250)
        .background(.background, in: .rect(cornerRadius: 10))
    }
}


struct QueueLengthChart: View {
    var queueLengths: [(date: String, Int)]

    var body: some View {
        VStack(alignment: .leading, spacing: 0) {
            Text("Длина очереди")
                .font(.headline)
                .padding()
            ScrollView(.horizontal) {
                HStack {
```

```
                    Chart {
                        ForEach(queueLengths, id: \.date) { point in
                            LineMark(
                                x: .value("Время", point.date),
                                y: .value("Длина очереди", point.1)
                            )
                            .foregroundStyle(.blue)
                        }
                    }
                    .frame(width: CGFloat(queueLengths.count) * 40)
                    .padding()
                }
            }
        }
        .frame(minHeight: 250)
        .background(.background, in: .rect(cornerRadius: 10))
    }
}
```