



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 «Программное обеспечение ЭВМ и информационные технологии»

# РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

## К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ

### НА ТЕМУ:

*«Метод параллельного выполнения запросов к  
системе управления базами данных PostgreSQL в  
пределах одного соединения»*

Студент ИУ7-85Б  
(Группа)

\_\_\_\_\_  
(Подпись, дата) О. С. Платонова  
(И.О.Фамилия)

Руководитель

\_\_\_\_\_  
(Подпись, дата) М. В. Филиппов  
(И.О.Фамилия)

Консультант

\_\_\_\_\_  
(Подпись, дата) Ю. М. Гаврилова  
(И.О.Фамилия)

2022 г.

## **РЕФЕРАТ**

Расчетно-пояснительная записка: 34 страницы, 2 таблицы, 15 рисунков, 18 источников.

Ключевые слова: база данных, PostgreSQL, многопоточность, соединение.

# СОДЕРЖАНИЕ

РЕФЕРАТ .....	2
Введение.....	6
1 Аналитический раздел.....	8
1.1 Анализ СУБД.....	8
1.2 Архитектура PostgreSQL .....	9
1.2.1 Клиентская часть.....	9
1.2.2 Серверная часть.....	10
1.2.3 Хранилище данных .....	10
1.3 Соединение в PostgreSQL.....	11
1.4 Многопоточность .....	11
1.5 Классификация существующих решений.....	13
1.5.1 Пул соединений.....	13
1.5.2 Распараллеливание запросов .....	16
1.5.3 Сравнительный анализ методов .....	19
1.6 Вывод.....	19
2 Конструкторский раздел.....	21
2.1 Формализация задачи .....	21
2.1 Требования к разрабатываемому методу .....	21
2.2 Ключевые этапы работы метода.....	21
3 Технологический раздел.....	23
3.1 Обоснование программных средств .....	23
3.X Реализация основных этапов метода .....	23
3.X Внешний модуль.....	30
3.X Внешний пул соединений.....	31

3.X Тестирование программного продукта .....	32
3.X Вывод.....	33
4 Исследовательский раздел .....	34
4.1 Выполнение простого запроса без нагрузки БД .....	34
4.1.1 Сравнение разработанного метода с пулом соединений .....	35
4.2 Выполнение простого запроса с нагрузкой БД .....	36
4.3 Анализ памяти .....	39
4.4 Вывод.....	39
Вывод.....	41
Список использованных источников .....	42

## **ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ**

База данных (БД) — собрание данных, организованных в соответствии с концептуальной структурой, описывающей характеристики этих данных и взаимоотношения между ними, причем такое собрание данных, которое поддерживает одну или более областей применения [1].

Система управления базой данных (СУБД) — совокупность программных и лингвистических средств общего или специального назначения, обеспечивающих управление созданием и использованием баз данных [2].

Массивно-параллельная архитектура (massive parallel processing, MPP) — класс архитектур параллельных вычислительных систем. Главная особенность такой архитектуры состоит в том, что память физически разделена [3].

## Введение

В XXI веке человечество владеет невообразимым объемом данных. Знания, передававшиеся из поколения в поколение в течение многих тысячелетий, продолжают увеличиваться каждый день. Так, ежегодный прирост информации составляет 30% [4].

С появлением письменности, будь то шумерские таблички или берестяные грамоты, перед человечеством возникает вопрос хранения и обработки данных. Причем с развитием цивилизации, и, как следствие, увеличением документооборота, проблема хранения информации требует систематического решения. Например, в конце XX века данные крупной компании могли занимать несколько этажей, что требовало дополнительных кадров для работы с ними.

Первым этапом решения этого вопроса стало внедрение компьютеров. Многие операции с данными были упрощены, а быстрый рост информационных технологий привел к увеличению скорости работы над данными. Однако хранение информации в виде файлов на одном компьютере стало неэффективным. Во-первых, поиск файла в файловой системе был долгим. Во-вторых, хранение информации в одном файле затрудняло поиск необходимых данных.

Решение проблемы разрозненного хранения данных впервые было представлено на симпозиуме в 1964 году в Санта-Монике [5]. Хотя речь шла о внедрении баз данных в военные приложения, этот момент считается точкой отсчета истории базы данных. Их применение в работе компаний привело к увеличению скорости работы. А автоматизация основных процессов базы данных, таких как создание, просмотр, удаление, привело к созданию системы управления базы данных.

В 2022 году ни одна сфера жизни не обходится без компьютеризации. Организации используют базы и СУБД для перевода данных в электронный вид. Необходимость перевода заключается не столько в потребности сократить временные и материальные (кадровые) расходы, сколько в поддержании

конкурентоспособности. Переход компании в электронный вид дает возможность приобретения принципиально новых качеств, позволяющих иметь существенные преимущества над другими.

Из-за высокой популярности СУБД возникает вопрос об оптимизации ее работы. Так как один из самых распространенных способов увеличения производительности — параллельное выполнение, следует рассмотреть оптимизацию многопоточной программы [6]. Поскольку операция соединения с базой данных является одной из самых дорогостоящих, следует минимизировать количество соединений.

Целью данной работы является разработка и реализация метода параллельного выполнения запросов к СУБД PostgreSQL в пределах одного соединения. Для достижения поставленной цели необходимо решить следующие задачи:

1. Выполнить анализ предметной области и существующих методов реализации многопоточного доступа в МРР системах;
2. Разработать метод параллельного выполнения запросов к СУБД PostgreSQL в пределах одного соединения;
3. Реализовать программный модуль для СУБД PostgreSQL;
4. Выполнить сравнительный анализ стандартного метода обработки запросов к СУБД PostgreSQL с реализуемым методом.

## 1 Аналитический раздел

В данном разделе будет выполнен анализ существующих СУБД и приведена аргументация выбора PostgreSQL, изучены основные положения архитектуры и инструменты реализации соединения в PostgreSQL. Также будут рассмотрены существующие методы и подходы выполнения запросов к СУБД и выполнен сравнительный анализ на основе рассмотренных методов.

### 1.1 Анализ СУБД

В таблице 1.1 представлен рейтинг популярности СУБД, составленный компанией «DB-Engines» по состоянию на начало 2022 года [7].

Таблица 1.1 — Рейтинг популярности СУБД.

Рейтинг	СУБД	Модель БД
1.	Oracle	Реляционная
2.	MySQL	Реляционная
3.	Microsoft SQL Server	Реляционная
4.	PostgreSQL	Реляционная
5.	MongoDB	Документная
6.	Redis	«Ключ-значение»
7.	IBM Db2	Реляционная
8.	Elasticsearch	Поисковая система
9.	Microsoft Access	Реляционная
10.	SQLite	Реляционная

Согласно рейтингу, лидирующие позиции занимают реляционные модели баз данных. Данная работа будет основываться на объектно-реляционной СУБД PostgreSQL 14-ой версии, занимающей 4-ое место в рейтинге [7]. Выбор аргументирован такими преимуществами как доступность исходного кода и кроссплатформенность.



## 1.2 Архитектура PostgreSQL

Одной из наиболее сильных сторон PostgreSQL является архитектура, основанная на модели «клиент-сервер». Выделяют 3 основные подсистемы: клиентская часть, серверная часть и хранилище данных (рисунок 1.1) [8].

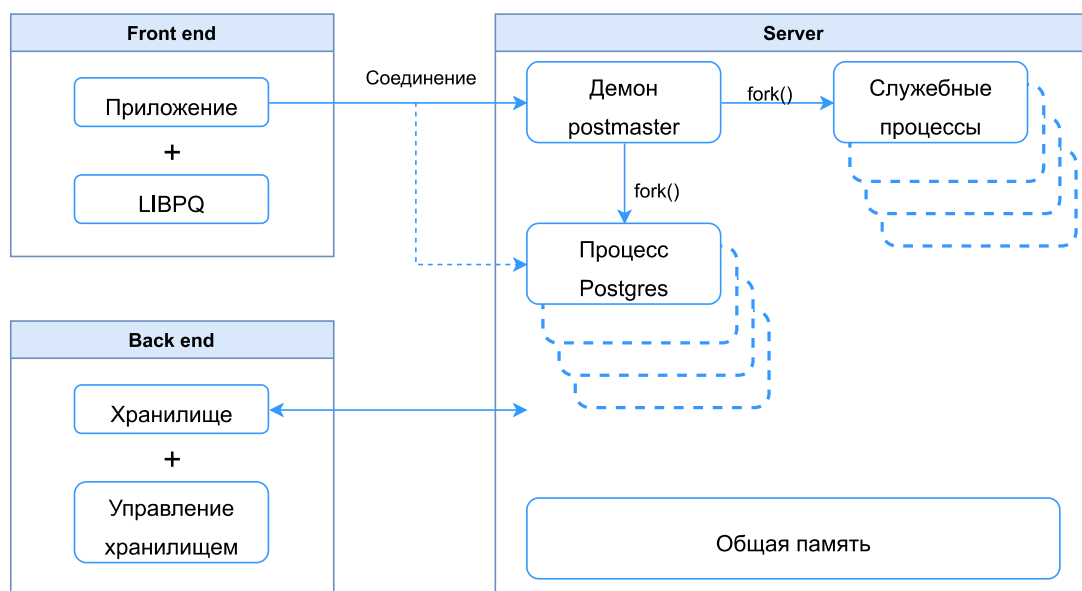


Рисунок 1.1 — «Клиент-серверная» архитектура PostgreSQL.

Представленная концепция взаимодействия позволяет исключать произвольный доступ клиентов к данным, тем самым поддерживая их целостность и повышая уровень безопасности. Также она дает возможность создания распределенных систем.

### 1.2.1 Клиентская часть

Клиентская часть состоит из пользовательского приложения и библиотеки libpq. Данная библиотека содержит набор функций, с помощью которых клиентские программы могут передавать запросы серверу PostgreSQL и принимать результаты этих запросов [9].

### 1.2.2 Серверная часть

Серверная часть, включающая в себя процесс-демон *postmaster*, серверные и служебные процессы, выполняет обработку запросов. Соединение, установленное клиентом, принимается демоном *postmaster*, который в дальнейшем с помощью системного вызова *fork()* создаст новый серверный процесс для обслуживания соединения данного клиента.

Между собой серверные процессы взаимодействуют с помощью семафоров и разделяемой памяти. Для всех серверных процессов доступна общая память, большую часть которой занимает буферный кэш. Он необходим для ускорения работы с данными на диске, обращение к которым реализовано через операционную систему [10]. Описанные механизмы позволяют обеспечить целостность данных при одновременном обращении к ним [9].

Помимо создания серверного процесса процесс-демон также порождает ряд служебных процессов.

### 1.2.3 Хранилище данных

Третья часть сформирована из хранилища данных и средств его управления. Хранение данных на диске организовано с помощью табличных пространств, указывающих расположение данных. Допускается обращение нескольких серверных процессов к информации хранилища одновременно. На рисунке 1.2 представлена организация хранилища [10].

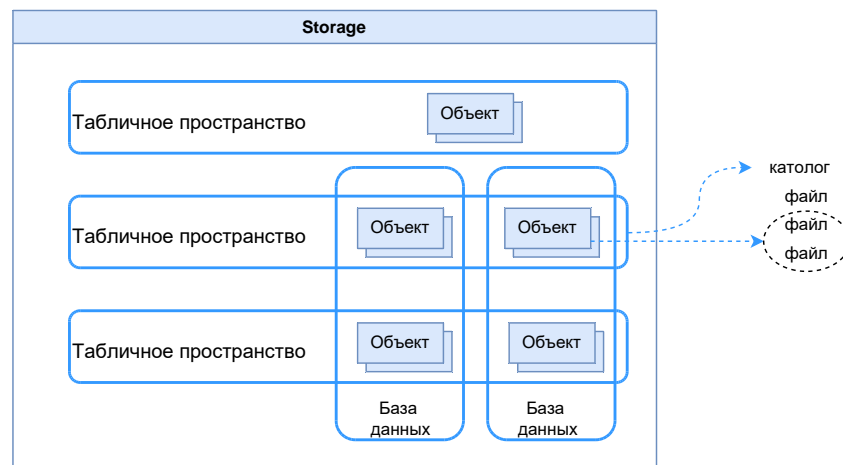


Рисунок 1.2. – Организация хранилища PostgreSQL.

### 1.3 Соединение в PostgreSQL

Как было рассмотрено выше, пользователь устанавливает соединение и посылает его серверному процессу *postmaster*. Каждое соединение представляется объектом *PGconn*, который можно получить от функций *PQconnectdb*, *PQconnectdbParams* или *PQsetdbLogin*.

PostgreSQL содержит инструменты для реализации многопоточности. Один из них — библиотека *libpq*, которая по умолчанию поддерживает повторные вызовы. Однако при многопоточности существует ограничение: «два потока не должны пытаться одновременно работать с одним объектом *PGconn*. В частности, не допускается параллельное выполнение команд из разных потоков через один объект соединения.» [9].

### 1.4 Многопоточность

Поскольку на сегодняшний день конкуренция происходит в области затрат и скорости, работа многих приложений основана на многопоточности, которая рассматривается как один из способов увеличения производительности.

С ростом объема БД наблюдается преимущество по времени многопоточной реализации. Так, при работе с базой данных, состоящей из 100000 записей, время выполнения запросов примерно в 1000 раз выше у

однопоточной программы. Также однопоточная модель показывает нестабильную работу на больших данных. На рисунке 1.3 приведены результаты сравнения работы однопоточной и многопоточной программ на больших данных [11].

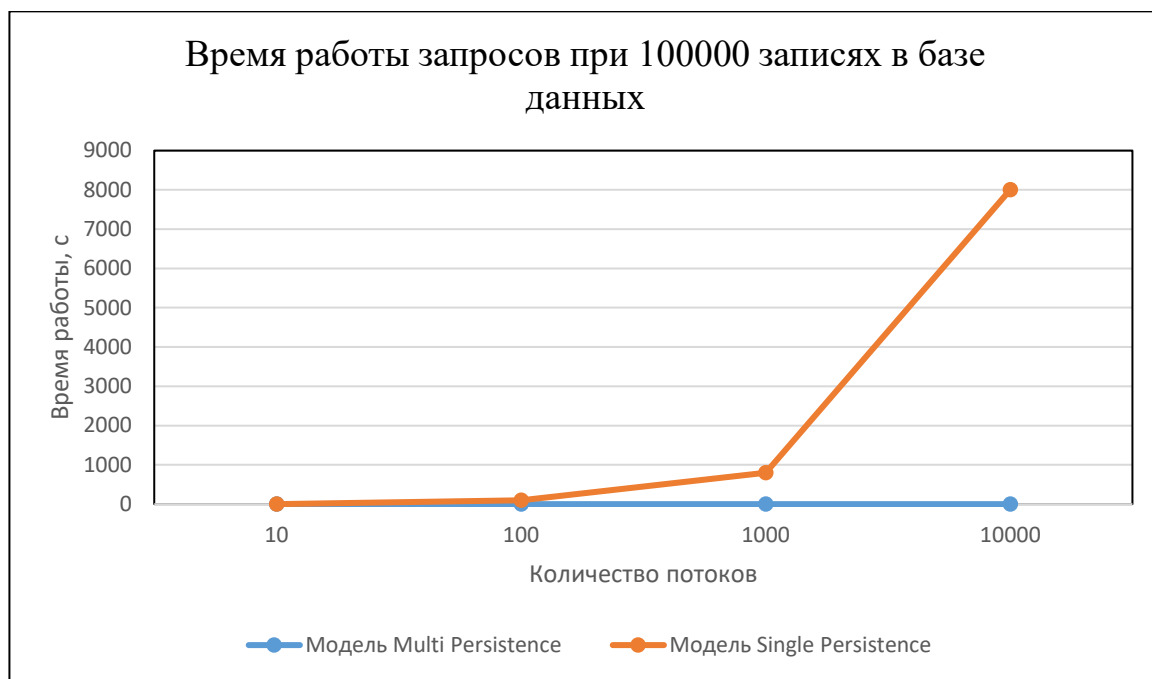


Рисунок 1.3 — Сравнение однопоточной и многопоточной программ, выполняющих запросы чтения БД.

Многопоточная реализация имеет ряд недостатков. Так, приложения, использующие PostgreSQL, вынуждены открывать новое соединение в каждом потоке. Поскольку операция подключения — одна из самых дорогостоящих (процесс подключения к БД занимает от 2 до 3 МБ [12]), рост количества потоков может привести к замедлению работы программы: повышенная нагрузка на системные ресурсы и значительное снижение производительности, особенно в многоядерных системах. Увеличение конкуренции при обращении множества процессов к ресурсам PostgreSQL также способно замедлять работу программы.

Открытие соединений на разных потоках может привести к проблеме превышения количества подключений на сервере и, как следствие, к долгому ожиданию дальнейших запросов или их отклонению.

## **1.5 Классификация существующих решений**

### **1.5.1 Пул соединений**

Объектный пул представляет из себя набор инициализированных и готовых к использованию объектов. При необходимости системы обратиться к объекту, вместо его создания будет взят экземпляр из пула. Особенно заметно повышение производительности, когда стоимость и скорость инициализации экземпляра высоки, а количество одновременно используемых объектов в любой момент времени является низким [13].

В PostgreSQL отсутствует встроенный пул соединений [9], однако допускается использование внешнего. Внешний пул может быть разработан с помощью средств библиотеки `libpq` или подключен в качестве дополнительной службы (`pgbouncer` или `pgpool`).

#### **1.5.1.1 Пул на основе `libpq`**

При инициализации пула выполняется установка необходимого количества соединений. Предельный размер определяется пользователем в зависимости от контекста задачи. После успешной инициализации из пула может быть извлечено свободное соединение для выполнения необходимых запросов к БД. После выполнения запросов соединение должно быть возвращено в пул. Если соединение было закрыто, его следует удалить из пула и вместо него создать новое [14].

Преимущество данного метода наглядно демонстрирует следующий пример. Устанавливается соединение для 10 клиентов, каждый из которых выполняет 10.000 запросов в БД. Если в среднем выполняется 486 транзакций в секунду, то реализация пула соединений (размером 25) позволяет увеличить это значение примерно на 60% – до 566 транзакций в секунду [15].

Главным недостатком внешнего пула являются затраты на его разработку. В том числе могут возникать трудности при его встраивании в крупный модуль, которые приведут к необходимости изменения архитектуры.

### 1.5.1.2 Пул в качестве внешней службы

PGBouncer и PGPool — дополнительные службы, которые можно включить отдельно для конкретного сервера БД. Их работа основана на оптимизации процесса подключения: многократное использование соединений. На рисунке 1.4 изображена архитектура приложения, использующего внешнюю службу.

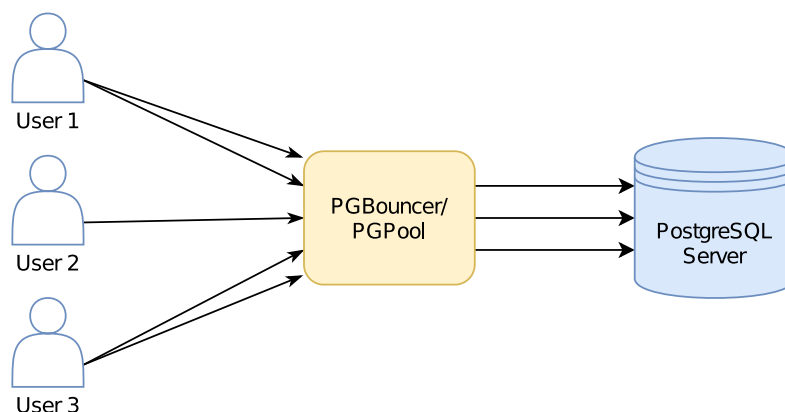


Рисунок 1.4 — Архитектура приложения, использующего внешнюю службу.

Вместо создания соединений с PostgreSQL, создается более «легкое» соединение с PGBouncer/PGPool, которые пользуются уже существующими соединениями с БД [9]. Внешний пул в данном ключе позволяет не только снижать потребление ресурсов PostgreSQL, но и выполнять обновление PostgreSQL без прерывания соединений с клиентом. Более того, службы позволяют временно приостанавливать соединения с целью перезапуска БД (после внесения изменений) или переключения на другую.

К существенным недостаткам служб следует отнести сложность диагностики: отсутствует код ошибки, а сообщение всегда одинаково не зависимо от того, где возникла проблема: на стороне клиента или сервера. Другой недостаток — однопоточная реализация самих служб. С одной стороны это делает их очень простыми, с другой — нагрузка службы на ядро может достигать 97%.

Основное ограничение описанных выше пулов соединений – их размер. Ограничение обусловлено значением параметра сервера `max_connections`, по умолчанию равное 100. Параметр определяет максимальное число одновременных подключений к серверу БД и может быть уменьшено [9]. Также следует обратить внимание на сложность расчета параметров пула: минимальное количество соединений, максимальное количество пулов соединений, максимальное время простоя, время ожидания соединения, количество попыток после тайм-аута. От настройки конфигурации зависит то, насколько увеличится или уменьшится пропускная способность транзакции.

### 1.5.1.3 Server pool

Хотя PostgreSQL не имеет встроенного пула подключений, он был реализован в коммерческой системе Postgres Pro Enterprise. Это объектно-реляционная СУБД, разработанная Postgres Professional в рамках проекта Postgres Pro на основе PostgreSQL [16]. В отличие от внешнего, встроенный пул не требует дополнительного обслуживания и не налагает на клиента жестких ограничений (размер не должен превышать 32767).

Работа встроенного пула аналогична работе внешнего. Число обслуживающих процессов, которые могут использоваться для отдельно взятой БД ограничивается размером пула. При достижении этого значения, процесс-демон *postmaster* перестает запускать новые процессы, а передает последующее подключение запущенному процессу. Так как один процесс может работать только с одной БД, возникает необходимость поддержки отдельного пула соединения для каждой БД. При появлении подключения к новой БД, добавляется новый пул. Пулы функционируют только на уровне транзакций, т.е. процесс может переключиться на обслуживание нового соединения только после завершения транзакции [16].

На рисунках 1.5 – 1.6 представлен цикл соединения с БД без пула и с его использованием соответственно [17].

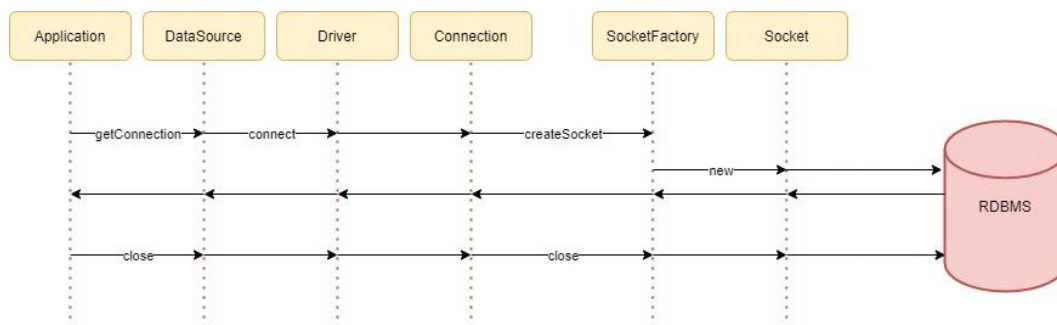


Рисунок 1.5 — Цикл соединения с БД.

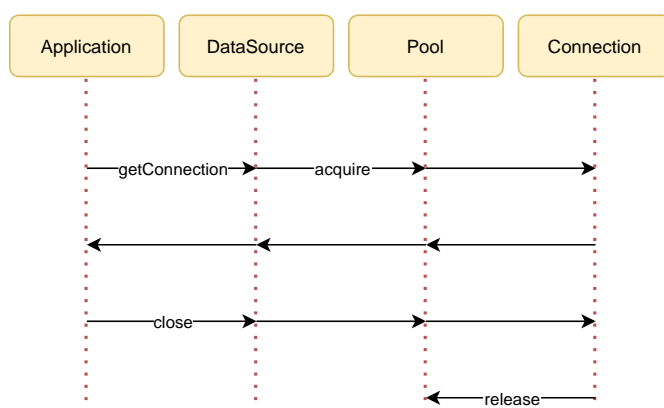


Рисунок 1.6 — Цикл соединения с БД с использованием пула.

### 1.5.2 Распараллеливание запросов

Методы, описанные выше, позволяют повысить скорость выполнения запросов путем сокращения числа соединений.

Принципиально другим подходом повышения производительности выполнения запроса — его оптимизация в рамках одного потока. Такой оптимизацией может стать распараллеливание запросов.

Распараллеливание — это возможность построения таких планов запросов, которые будут задействовать несколько ядер. Наибольший выигрыш при использовании данного метода можно получить в случае работы с запросами, обрабатывающими большой объем данных, но возвращающих пользователю всего несколько строк [9].



### 1.5.2.1 Планировщик

Сервер, получив запрос от прикладной программы, выполняет его разбор. Разбор подразумевает синтаксическую проверку запроса и построение дерева запроса (рисунок 1.7).

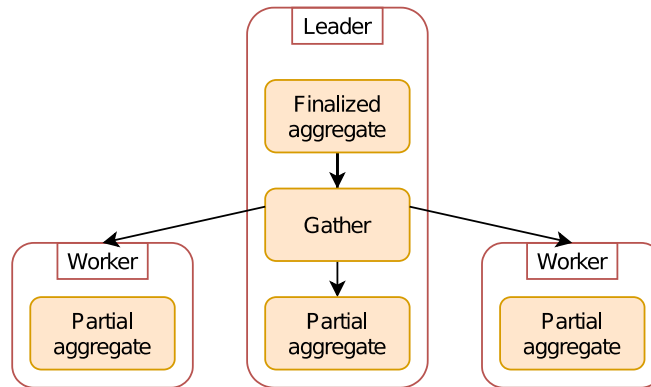


Рисунок 1.7 — Дерево частично параллельного запроса.

Задача планировщика заключается в создании плана запроса на основании полученного дерева. Выбор плана состоит из двух этапов: рассмотрение всевозможных вариантов для получения одного и того же результата и оценка каждого для выбора самого дешевого.

В случае, когда планировщик принимает решение о том, что параллельное выполнение запроса будет наилучшим вариантом, он создает план запроса. Данный план будет состоять из узла *Gather*(сбор) или *Gather Merge*(сбор со слиянием) [9]. Если один из описанных узлов будет располагаться в вершине дерева, то весь запрос будет выполняться параллельно. Иначе – частично.

### 1.5.2.2 Алгоритм работы параллельно выполняемых процессов

При достижении одного из узлов *Gather* или *Gather Merge* процесс, обслуживающий текущее подключение, выполняет запрос фоновых рабочих процессов. Каждый успешно запущенный фоновый процесс будет выполнять параллельную часть плана. При этом ведущий процесс помимо выполнения части плана будет выполнять контроль фоновых процессов. Его задача состоит в чтении кортежей, полученных от фоновых процессов, и выполнении других

шагов обработки, связанных с узлами плана, расположенных выше *Gather* или *Gather Merge*.

На рисунке 1.7 представлена схема реализации параллельного запроса.

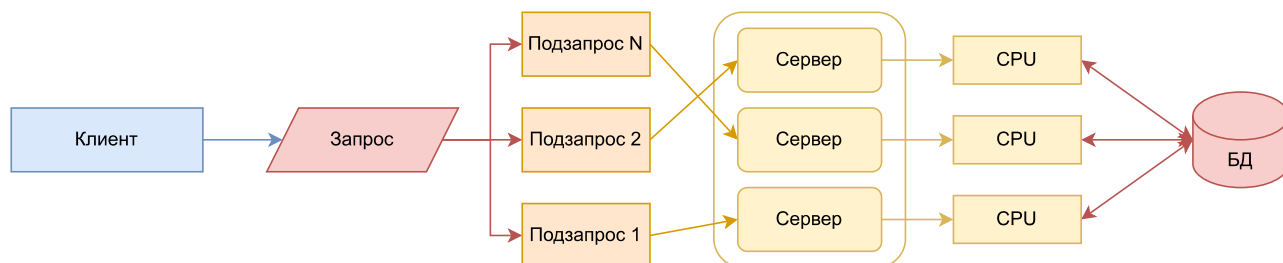


Рисунок 1.7 – Схема реализации параллельного запроса.

### 1.5.2.3 Безопасность метода

Планировщик классифицирует запросы следующим образом:

- Безопасно распараллеливаемая. Не мешает выполнению запроса.
- Ограниченно распараллеливаемая. Не может выполняться в параллельном рабочем процессе, но может выполняться в ведущем.
- небезопасно распараллеливаемая. Не может выполняться параллельно даже в ведущем процессе. Для такого запроса параллельное выполнение полностью исключается.

К ограниченно распараллеливаемым запросам относятся такие запросы, как сканирование временных, сторонних таблиц, общих табличных выражений [9].

### 1.5.2.4 Недостатки метода

Для большинства запросов параллельное выполнение не дает никаких преимуществ во времени. Это может быть связано с тем, что последовательный план быстрее параллельного или с ограничением текущей транзакции. Также число рабочих процессов может не соответствовать ожидаемому, поскольку число исполнителей и число фоновых процессов ограничено. А нехватка рабочих процессов ведет к снижению производительности.

Описанный метод применим только к ограниченному числу запросов (большие данные, короткий ответ).

### **1.5.3 Сравнительный анализ методов**

Пул соединений и распараллеливание запросов являются принципиально разными подходами увеличения производительности.

Пул соединений позволяет оптимизировать процесс выполнения запроса путем сокращения создаваемых соединений к БД. Это, в свою очередь, сокращает число серверных процессов PostgreSQL, и, как следствие, снижает потребляемые ресурсы системы (память, время).

Распараллеливание запроса повышает производительность на этапе его выполнения. Планировщик путем рассмотрения и оценки всевозможных вариантов выполнения распределяет запрос между процессами. При таком подходе с уменьшением времени выполнения наблюдается увеличение потребляемых ресурсов.

В зависимости от условия задачи пользователь может выбрать тот или иной подход. Однако стоит отметить, что пул соединений является более универсальным методом, в то время как для распараллеливания требуется соблюдение многих ограничений.

Что касается сравнения внешнего и внутреннего пулов, то недостатками первого является ограничение максимального количества одновременных подключений к БД, сложность встраиваемости в код и настройка конфигурации. В то время как второй доступен только в коммерческой версии PostgreSQL Pro.

## **1.6 Вывод**

В данном разделе был выполнен анализ существующих СУБД и приведена аргументация выбора PostgreSQL, изучены основные положения архитектуры и инструменты реализации соединения в PostgreSQL. Также были рассмотрены

существующие методы и подходы выполнения запросов к СУБД и выполнен сравнительный анализ на основе рассмотренных методов.

В результате анализа было установлено, что использование пула соединений позволяет не только уменьшить время выполнения запроса, но и сократить потребляемые ресурсы системы, в то время как распараллеливание запроса увеличивает их потребление. Однако пул имеет ряд ограничений, не позволяя достичь требуемой эффективности в многопоточной программе.

Задача разрабатываемого метода – предложить принципиально другой подход повышения производительности при выполнении запроса.

## **2 Конструкторский раздел**

В данном разделе будет разработан метод параллельного выполнения запросов к СУБД PostgreSQL в пределах одного соединения. Также будут описаны основные особенности предлагаемого метода, сформулированы ограничения предметной области. Будут выделены функции и структуры исходного кода, используемые методом. Ключевые этапы будут изложены в виде диаграммы IDEF0 и схем алгоритма.

### **2.1 Формализация задачи**

В ходе сравнительного анализа существующих методов и подходов выполнения запросов к СУБД PostgreSQL, приведенного выше, было установлено, что главным ограничением повышения производительности в многопоточной программе является соединение с БД. Задача разрабатываемого метода заключается в сокращении числа открытых соединений путем реализации возможности работы с одним подключением в разных потоках.

### **2.1 Требования к разрабатываемому методу**

Метод должен работать с одним соединением и запросом. Работа метода должны быть асинхронной. Для формирования корректного результата работа метода должна включать механизм синхронизации. Метод должен обрабатывать запрос корректно за конечное время.

### **2.2 Ключевые этапы работы метода**

Разрабатываемый метод состоит из следующих этапов обработки запроса:

1. Отправка запроса серверу
  - а. Формирование очереди запроса
  - б. Отправка запроса из очереди серверу
2. Получение ответа

На рисунке 2.1 представлена IDEF0-диаграмма разрабатываемого метода.



Рисунок 2.1 – Диаграмма IDEF0.

### 3 Технологический раздел

В данном разделе будет выполнено обоснование программных средств реализации метода. Также будет разработано программное обеспечение, реализующее описанный метод, и выполнено тестирование путем выполнения запросов в пределах одного соединения в нескольких потоках. Будут описаны основные особенности реализации.

#### 3.1 Обоснование программных средств

В качестве языка программирования был выбран язык Си, так как исходный код PostgreSQL, в том числе библиотека `libpq`, написаны на нем. Языком программирования внешнего модуля был выбран C++ из-за наличия опыта написания на нем, а также предоставляемых возможностей использования объектно-ориентированного подхода.

Средой разработки был выбран текстовый редактор SublimeText из-за его высокой скорости работы и низких требований к ресурсам операционной системы.

#### 3.2 Структура разработанного ПО

Как было упомянуто выше, помимо разработанного метода, который является частью библиотеки `libpq` PostgreSQL также был реализован внешний модуль. Задача внешнего модуля – работа с реализованной функцией, т.е. создание заданного количества потоков, в каждом из которых происходит вызов функции *PQexecThread()*.

#### 3.3 Реализация основных этапов метода

Разрабатываемый метод состоит из двух этапов: отправка запроса серверу и получение результата (листинг 3.1). В случае возникновения ошибки при попытке отправки запроса, возвращается нулевой указатель. В случае успеха — результат, полученный от сервера.

Листинг 3.1 — Внешняя функция PQexecThread.

```
2419 PGresult *
2420 PQexecThread(PGconn *conn, const char *query)
2421 {
2422     if (!PQsendQueryThread(conn, query))
2423         return NULL;
2424     return PQgetResultThread(conn);
2425 }
```

Ниже будут рассмотрены более подробно каждый из описываемых этапов.

### 3.3.1 Отправка запроса серверу

Для отправки запроса была реализована функция *PqsendQueryThread()*. Асинхронность функции достигается путем отправки команд серверу без ожидания получения результата. Процесс отправки запроса подразумевает его постановку в очередь команд, а затем обработку сформированной очереди. Функция является оберткой над внутренней функцией *PqsendQueryInternalThread()*. В работе функции можно выделить несколько стадий подготовки отправки запроса.

#### 3.3.1.1 Блокировка мьютекса

Работа функции отправки запроса начинается с блокировки мьютекса (листинг 3.2). Прежде всего мьютекс необходим для корректного возвращения результата обработки запроса в соответствующий поток. Также мьютекс защищает параметры подключения, от которых зависит результат выполнения команды, для данного запроса от изменений другим потоком.

Листинг 3.2 – Начало работы функции *PqsendQueryInternalThread*.

```
1476 static int
1477 PqsendQueryInternalThread(PGconn *conn, const char *query, bool newQuery)
1478 {
1479     pthread_mutex_lock(&mutex);
```



После того, как была произведена защита от изменений текущих параметров подключения выполняется проверка их корректности и инициализация соответствующих полей.

### 3.3.1.2 Проверка соединения на безопасность

Прежде всего выполняется проверка наличия открытого соединения с БД. Выполняется очистка буфера ошибок и сброс состояния ошибки для подключения. Также запрос не может быть отправлен, если соединение уже занято или сокет не допускает прием сообщений.

Следует отметить, что при постановке команд в очередь состояние соединения меняется не сильно, поскольку оно уже используется для текущей команды. Состояние подключения будет обновлено, когда процесс перейдет к началу обработки поставленного в очередь сообщения. Задача функции PQsendQueryStartThread() убедиться, что запрос может быть безопасно поставлен в очередь с учетом текущего состояния соединения.

На листинге 3.3 представлен вызов функции, выполняющей описанные выше проверки, а также проверка корректности самого запроса. В случае, если проверка не была успешной, выполняется заполнение буфера ошибок соответствующим сообщением и выход из функции отправки.

Листинг 3.3 – Проверка параметров перед постановкой команды в очередь.

```
1482     if (!PQsendQueryStartThread(conn, newQuery))
1483         return 0;
1484
1485     if (!query)
1486     {
1487         appendPQExpBufferStr(&conn->errorMessage,
1488                             libpq_gettext("command string is a null pointer\n"));
1489         return 0;
1490     }
```

### 3.3.1.3 Создание очереди команд

Функция *pqAllocCmdQueueEntry()* выполняет определение точки входа очереди команд для текущего запроса. Для каждого соединения определена

очередь повторного использования. Она используется для того, чтобы сократить количество *malloc*. После выполнения команды вместо вызова *free*, память, выделенная для ее хранения в основном списке команд, сохраняется в список повторного использования. Поэтому вновь поступивший запрос будет сохранен в очереди повторного использования в начале (если очередь пуста), либо будет добавлен в конец. На листинге 3.4 представлен вызов функции *pqAllocCmdQueueEntry()*. В случае неудачи при выделении памяти для текущего запроса отправка не будет выполнена, функция отправки завершается.

Листинг 3.4 – Вызов функции определения точки входа очереди команд.

```
1492     entry = pqAllocCmdQueueEntry(conn);
1493     if (entry == NULL)
1494         return 0;
```

### 3.3.1.4 Формирование сообщения серверу

Идея заключается в том, чтобы записать сообщение в буфер текущего соединения (*conn->outBuffer*). Запись выполняется непрерывно, т.е. в конец буфера, сразу за данными, уже расположенными в буфере. Размер буфера увеличивается по мере необходимости. Помимо самого буфера в памяти хранятся размер выделенной для него памяти и текущий размер.

Сначала происходит запись типа сообщения (в случае запроса – ‘Q’, query), затем само сообщение (запрос). После происходит завершение записи. На рисунке 3.1 представлен формат сформированного сообщения.

Query



Рисунок 3.1 – Формат сообщения серверу.

Завершение записи заключается в проверке размера буфера и его возможной отправки серверу. Отправка происходит только после того, как был накоплен объем данных, более 8129 символов (стандартный размер буфера в Unix системах). Указанная проверка позволяет избежать отправки маленьких пакетов.

В случае, если какой-либо из этапов завершился с ошибкой, выполняется очистка выделенной памяти в очереди запросов и функция отправки завершается (листинг 3.5).

Листинг 3.4 – Этапы формирования сообщения серверу.

```
1496     if (conn->pipelineStatus == PQ_PIPELINE_OFF)
1497     {
1498         if (pqPutMsgStart('Q', conn) < 0 ||
1499             pqPuts(query, conn) < 0 ||
1500             pqPutMsgEnd(conn) < 0)
1501         {
1502             pqRecycleCmdQueueEntry(conn, entry);
1503             return 0;
1504         }
1505
1506         entry->queryclass = PGQUERY_SIMPLE;
1507         entry->query = strdup(query);
1508     }
```

Помимо формирования сообщения перед отправкой следует указать использование простого протокола запросов и сохранить сам SQL-запрос.

### 3.3.1.5 Отправка запроса

Отправка сообщения происходит, если был достигнут размер буфера, равный 8129, или, если была вызвана функция *pqFlush()*, которая выполняет запись содержимого буфера в очередь для отправки в сокет (листинг 3.5). В простом протоколе запроса продвижение очереди команд выполняется согласно протоколу fe-protocol3.

Листинг 3.5 – Вызов функции отправки запроса серверу.

```
1545     if (pqPipelineFlush(conn) < 0)
1546         goto sendFailed;
```

### 3.3.1.6 Завершение отправки

Перед завершением функции отправки запроса серверу выполняется добавление точки входа созданной очереди к очереди команд текущего соединения и обновление статуса подключения (листинг 3.6).

Листинг 3.6 – Вызов функции, завершающей отставку сообщения серверу.

```
1548     pqAppendCmdQueueEntryAsync(conn, entry);
1549     return 1;
```

### 3.3.2 Получение результата от сервера

Для получения результата следует выполнять прослушивание сервера. Пока сервер находится в состоянии «занят» выполняются попытки чтения данных из сокета до тех пор, пока не поступит полное сообщение от сервера или не обнаружится чтение EOF. Переход сервера в состояние «свободен» свидетельствует о завершении отправки результата, после чего начинается формирование ответа.

На листинге 3.4 представлена работа функции PQgetResultThread(). В бесконечном цикле выполняется чтение данных без блокировки пока PQgetResult готов к получению данных. Так как PQgetResult возвращает последний результат выполнения, а завершение работы соответствует параметру NULL, была введена дополнительная переменная lastResult. Ее содержимое соответствует результату перед тем, как был получен нулевой указатель. Выход из бесконечного цикла происходит в случае, если был получен корректный результат. Также выполняется проверка состояния соединения, и в случае потери соединения с сервером или получения некорректного результата происходит завершение работы цикла.

Поскольку взаимодействие с сервером в рамках выполнения запроса данного соединения завершается, перед возвращением из функции следует разблокировать мьютекс для успешного выполнения следующего запроса.

Листинг 3.4 – Функция получения результата обработки запроса от сервера.

```
2326 PGresult *
2327 PQgetResultThread(PGconn *conn)
2328 {
2329     PGresult *res = NULL;
2330     PGresult *lastRes = NULL;
2331
2332     for (;;)
2333     {
2334         while (PQisBusy(conn))
2335         {
2336             if (!PQconsumeInput(conn))
2337             {
2338                 lastRes = NULL;
2339                 break;
2340             }
2341         }
2342
2343         res = PQgetResult(conn);
2344         if (res == NULL)
2345             break;
2346
2347         PQclear(lastRes);
2348         lastRes = res;
2349
2350         if (PQresultStatus(lastRes) == PGRES_COPY_IN ||
2351             PQresultStatus(lastRes) == PGRES_COPY_OUT ||
2352             PQresultStatus(lastRes) == PGRES_COPY_BOTH ||
2353             PQstatus(conn) == CONNECTION_BAD)
2354             break;
2355     }
2356
2357     pthread_mutex_unlock(&mutex);
2358     return lastRes;
2359 }
```

Использование PQsendQueryThread и PQgetResultThread позволяют получить результат каждой команды индивидуально в случае, если строка команды содержит несколько SQL-команд. Также исключается блокировка пользователя в ожидании ввода от сервера баз данных. Однако в представленной реализации отсутствует конкатенация сообщений об ошибке: сообщение будет соответствовать только первому неудавшемуся выполнению.

### 3.4 Внешний модуль

Внешний модуль, используя интерфейс командной строки, предоставляет пользователю возможность выбора запускаемой реализации. Помимо разработанного метода, пользователю доступны следующие реализации:

- однопоточная;
- многопоточная;
- с использованием внешнего пула.

Также пользователю предоставляется возможность выполнения всех описанных реализаций с выводом результатов замера времени в таблицу для их последующего сравнения.

На рисунке 3.1 изображен пример работы внешнего модуля. В приведенном примере сначала выполняется запрос "SELECT \* FROM table100;" посредством описанного метода, а затем этот же запрос выполняется всеми доступными реализациями для выполнения сравнения.

```
MODE
-----
Single connection.....1
Multithreaded connection...2
Pool connection.....3
Custom connection.....4

Full comparison.....5
Exit.....0

>> 4
Input the number of connections (1-1000): 100
Time: 0.043219

>> 5
Input the number of connections (1-1000): 50

50 connects
Time
-----
single  0.031207
multi   0.039042
pool    0.012877
custom  0.012994

>> 0
```

Рисунок 3.1 — Демонстрация работы внешнего модуля, выполняющего вызов реализованной функции.

### 3.4.1 Внешний пул соединений

Внешний модуль, описанный выше, также включает в себя реализацию внешнего пула соединений, использующего интерфейс библиотеки `libpq`. Внешний пул был разработан с использованием умных указателей для предотвращения возможной утечки ресурсов. Сам пул был реализован в качестве очереди соединений: в конец добавлялись свободные соединения, работа с которыми была завершена.

На листингах 3.1-2 приведена реализация класса `PGBackend`, содержащего метод создания пула. Помимо умных указателей используется *mutex*, задачей которого является блокирование потока в ожидании возвращения соединения в пул. Данная реализация актуальна в случае, когда пул пустой, а все созданные подключения выполняют запросы к БД.

Листинг 3.1 — Класс `PGBackend`.

```
4  #include <memory>
5  #include <mutex>
6  #include <string>
7  #include <queue>
8  #include <condition_variable>
9  #include <libpq-fe.h>
10 #include "pgconnection.h"
11
12 class PGBackend
13 {
14 public:
15     PGBackend(const std::string host, const std::string port,
16              const std::string dbName, const std::string user, const std::string password);
17
18     std::shared_ptr<PGConnection> connection();
19     void freeConnection(std::shared_ptr<PGConnection>);
20
21 private:
22     void createPool(const std::string host, const std::string port,
23                   const std::string dbName, const std::string user, const std::string password);
24
25     std::mutex m_mutex;
26     std::condition_variable m_condition;
27     std::queue<std::shared_ptr<PGConnection>> m_pool;
28
29     const int POOL_SIZE = 10;
30 };;
```

### Листинг 3.2 — Реализация методов класса PGBackend.

```
1  #include <iostream>
2  #include <thread>
3  #include <fstream>
4  #include <sstream>
5  #include "pgbackend.h"
6
7  using namespace std;
8
9  PGBackend::PGBackend(const string host, const string port,
10                      const string dbName, const string user, const string password)
11  {
12      createPool(host, port, dbName, user, password);
13  }
14
15  void PGBackend::createPool(const string host, const string port,
16                            const string dbName, const string user, const string password)
17  {
18      lock_guard<mutex> locker_( m_mutex );
19
20      for (int i = 0; i < POOL_SIZE; i++) {
21          m_pool.emplace(make_shared<PGConnection>(host, port, dbName, user, password));
22      }
23  }
24
25  shared_ptr<PGConnection> PGBackend::connection()
26  {
27      unique_lock<mutex> lock_( m_mutex );
28
29      while (m_pool.empty()){
30          m_condition.wait(lock_);
31      }
32
33      auto conn = m_pool.front();
34      m_pool.pop();
35
36      return conn;
37  }
```

## 3.5 Тестирование программного продукта

Для выполнения тестирования была создана БД, состоящая из таблиц размером 10, 100, 200, 500, 1000, 5000 и 10.000 строк. При выполнении запроса SELECT для каждой из таблиц наблюдалась стабильная работа разработанного ПО. Также тестирование включало в себя выполнение четырех базовых запросов БД: SELECT, INSERT, UPDATE и DELETE. В таблице 3.1 приведены результаты замеров времени выполнения каждого запроса для стандартной функции PQexes и для разработанного метода PQexesThread.

Таблица 3.1 — Время выполнения базовых запросов БД для стандартной и реализованной функций.



	<b>PQexec</b>	<b>PQexecThread</b>
<b>SELECT</b>	1.029	1.0557
<b>INSERT</b>	1.204	1.6248
<b>UPDATE</b>	1.123	1.223
<b>DELETE</b>	1.356	1.911

### **3.6 Вывод**

В данном разделе было выполнено обоснование программных средств реализации метода. Также было выполнено тестирование разработанного программного обеспечения, реализующее описанный метод. Тестирование проводилось путем выполнения запросов в пределах одного соединения в нескольких потоках. Были описаны основные особенности реализации.

## 4 Исследовательский раздел

В данном разделе будет проведено исследование временной эффективности метода параллельного выполнения запросов в пределах одного соединения. Будет выполнен сравнительный анализ реализованного метода с однопоточной и многопоточной реализациями.

### 4.1 Выполнение простого запроса без нагрузки БД

В данном эксперименте выполнялось сравнение времени выполнения простого запроса для 4 реализаций:

1. Последовательная реализация.
2. Параллельная реализация.
3. Реализация с использованием внешнего пула соединений.
4. Реализация с использованием разработанного метода.

В качестве внешнего пула был выбран разработанный пул, работа которого основана на асинхронных функциях библиотеки `libpq`.

Для каждого опыта учитывалось время создания соединения, выполнения запроса, очистки результата выполнения и закрытия соединения. В случае многопоточности, время создания и ожидания синхронизации потоков также влияло на итоговый результат.

На рисунке 4.1 представлены результаты эксперимента — график зависимости времени выполнения запроса от количества создаваемых соединений для каждой реализации. Согласно результатам, наибольшее время выполнения было зафиксировано у однопоточной реализации: 0.42 секунды при 500 потоках. У многопоточной реализации наибольшее преимущество перед однопоточной достигается в случае 500 соединений (в 1.7 раз работает быстрее), однако при малом числе (10) соединений результаты работы отличаются незначительно (в 0.99 раз).

Время работы модуля с использованием внешнего пула и с использованием разработанного метода разнится незначительно в данном эксперименте. Но стоит отметить, что реализация с разработанным методом в 4.55 раза работает быстрее однопоточной на большом количестве создаваемых соединений.

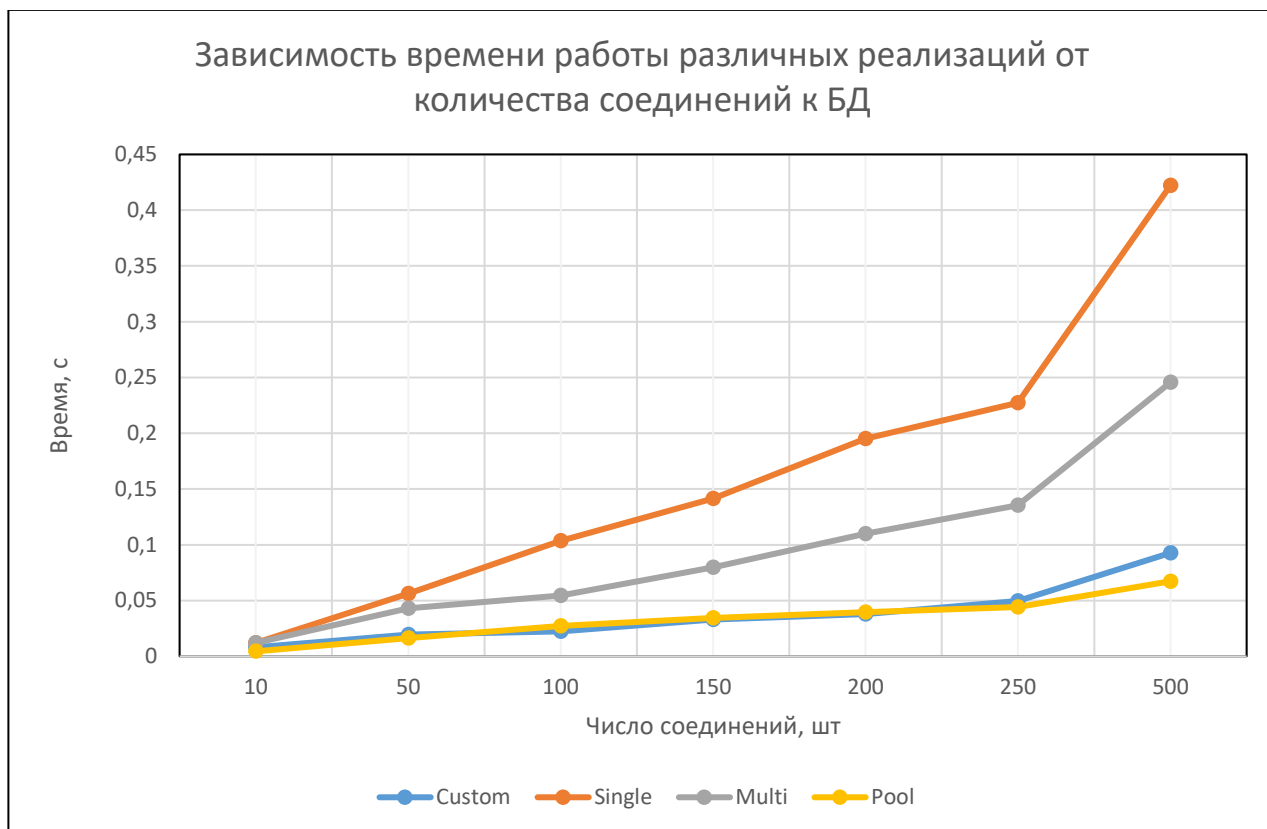


Рисунок 4.1 — График зависимости времени выполнения запроса от количества создаваемых соединений для каждой реализации.

#### 4.1.1 Сравнение разработанного метода с пулом соединений

На основе результатов опыта, описанного выше, было установлено, что конкурентоспособность по времени разработанному методу составляет только пул соединений.

В данном эксперименте проводилось сравнение времени работы пула, использующего библиотеку `librq` и пула, реализованного в качестве внешней службы (PGBouncer), с разработанным методом.

Был выбран PGBouncer, поскольку он потребляет меньше памяти, чем PGPool, а также предоставляет возможность настроить очередь соединений. Размер пула был вдвое меньше фактического количества создаваемых соединений.

Согласно результатам эксперимента (рисунок 4.2), при более 100 соединений PGBouncer теряет временную эффективность: в 2.74 раза работает дольше, чем внешний пул. Однако при 10 соединениях работает быстрее разработанного метода в 1.24 раза.

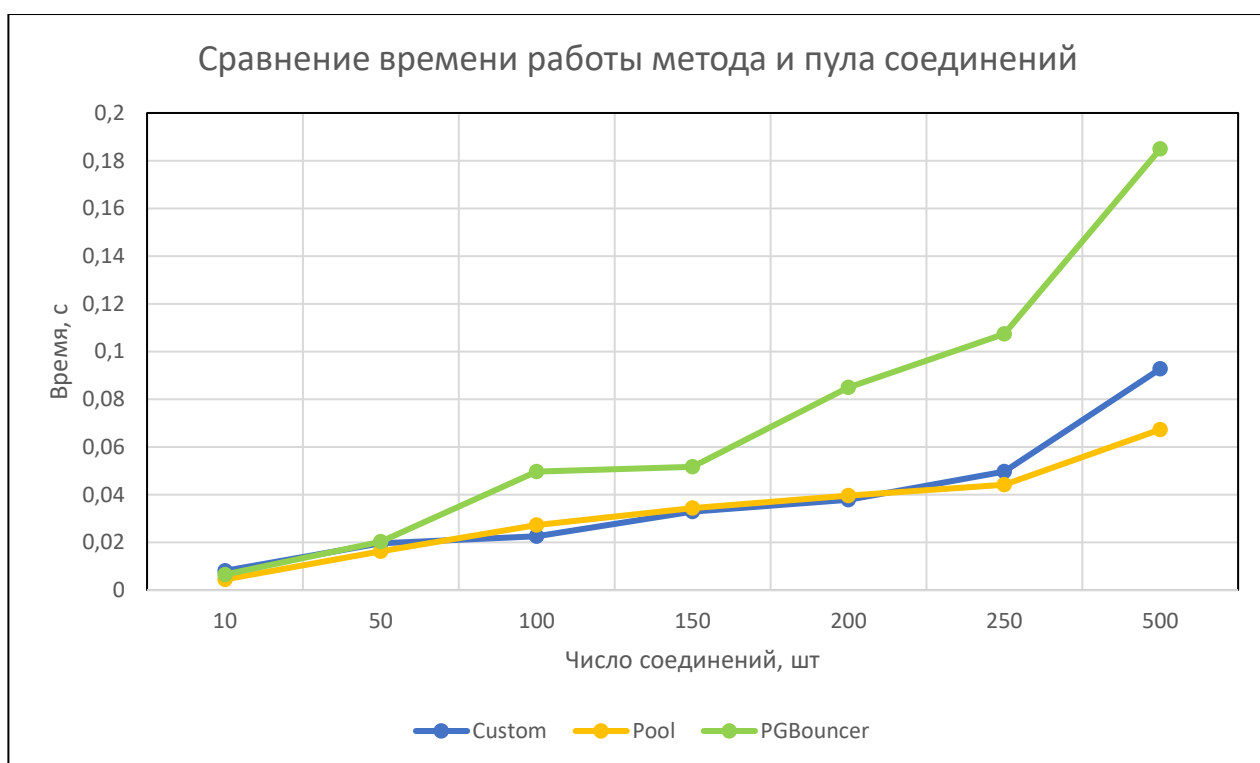


Рисунок 4.2 — Сравнение времени работы метода и пула соединений.

## 4.2 Выполнение простого запроса с нагрузкой БД

В данном эксперименте выполнялось сравнение времени выполнения простого запроса для разработанного метода и пула соединений при нагрузке на сервер БД. Для нагрузочного тестирования использовался инструмент Apache Jmeter. Так как эксперимент подразумевал большое число одновременных подключений к БД, параметр `max_connections` был увеличен до 1000 (рисунок 4.3).

```

postgres=# show max_connections;
max_connections
-----
1000
(1 row)

```

Рисунок 4.3 — Максимальное число одновременных подключений к серверу БД.

Для нагрузки сервера были созданы пул соединений размером 200 и 150 потоков (пользователей), каждый из которых выполнял один запрос к БД. Запрос подразумевал декартовое произведение (CROSS JOIN) таблиц размером 200 и 500 строк. Таким образом, результат состоял из 100.000 строк. Конфигурация, описанная выше, представлена на рисунках 4.4 – 4.6.

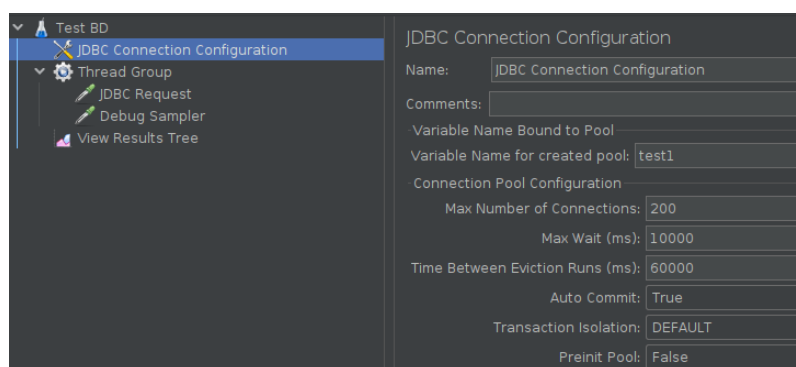


Рисунок 4.4 — Конфигурация пула соединений Jmeter.

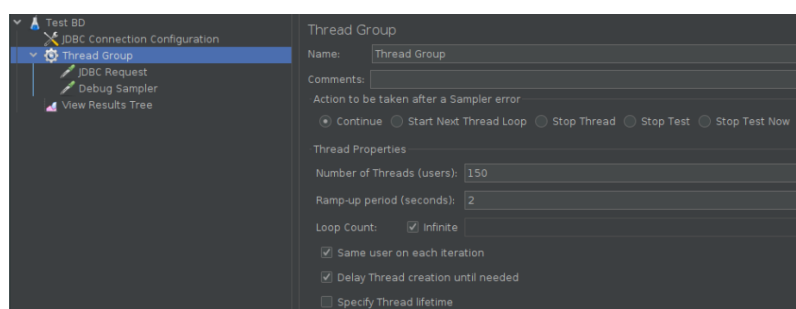


Рисунок 4.5 — Конфигурация потоков Jmeter.

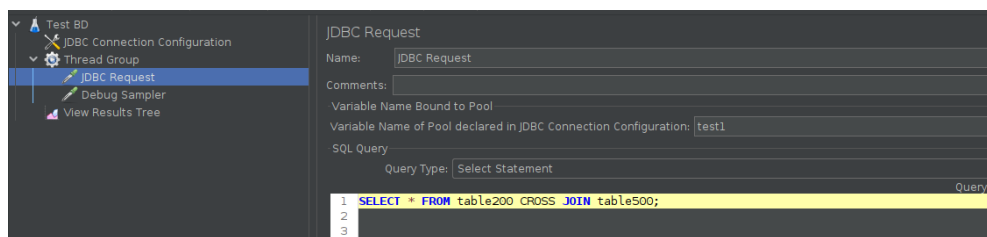


Рисунок 4.6 — Конфигурация запроса Jmeter к БД.

Результаты эксперимента приведены на рисунке 4.6. Следует отметить, что время работы реализованного метода сильно разнилось для каждого опыта. На графике представлены средние значения для 30 опытов.

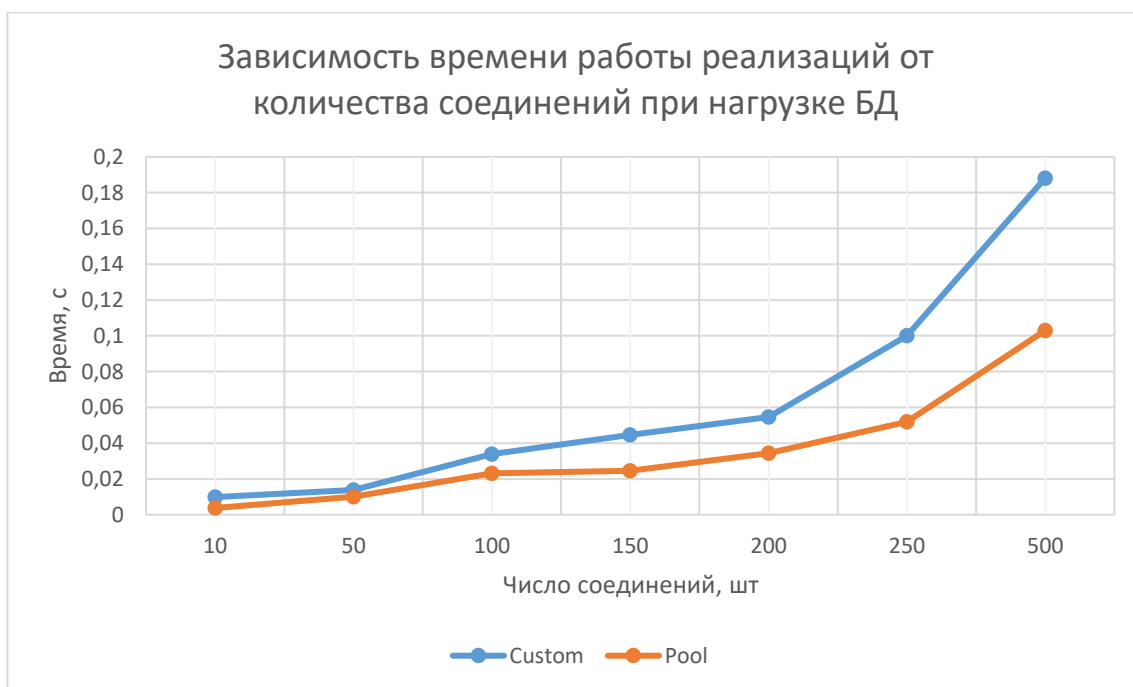


Рисунок 4.5 — График зависимости времени работы реализаций в зависимости от количества соединений при нагрузке БД.

Согласно графику, разработанный метод уступает по времени реализации, основанной на внешнем пуле соединений. Так, при 500 соединениях пул работает в 1.83 раза быстрее. Однако при 50 соединениях разница минимальна и составляет 3.79 мс.

### 4.3 Анализ памяти

Был выполнен эксперимент, фиксирующий затраты памяти для каждой реализации в случае создания 10 соединений и выполнения простого запроса. Результаты эксперимента представлены в таблице 4.1.

Таблица 4.1 — Затраты памяти каждой из реализаций.

Реализация	Число раз выделения памяти	Суммарный объем используемой памяти
Однопоточная	729	588,870 байт
Многopotочная	812	593,508 байт
Внешний пул	831	586,212 байт
Разработанный метод	182	180,794 байт

Согласно результатам, наибольшее потребление памяти (593 байта) у многопоточной реализации, что в 3.28 раза больше, чем память, потребляемая разработанным методом. Также, метод требует в 3.24 раза меньше памяти, чем внешний пул.

Сильное отклонение реализуемого метода от среднего значения объясняется тем, что при создании соединения, серверный процесс *postmaster* создает новый процесс для обслуживания данного клиента. Поскольку метод использует только 1 открытое соединение, то суммарный объем потребляемой памяти меньше, чем в других реализациях.

### 4.4 Вывод

В данном разделе было проведено исследование временной эффективности метода параллельного выполнения запросов в пределах одного соединения. Также был выполнен сравнительный анализ реализованного метода с однопоточной и многопоточной реализациями.

В ходе исследования было установлено, что в случае выполнения запроса без нагрузки БД разработанный метод работает в 4.55 раза быстрее однопоточной реализации и в 2.64 раза быстрее, чем многопоточная. Время работы внешнего пула и метода разнится незначительно.

В результате сравнения внешнего пула на основе `libpq`, пула в качестве внешней службы и разработанного метода было выявлено, что наибольшее время выполнения соответствует службе `PGBouncer`: в 2.74 раза дольше внешнего пула.

Во время эксперимента с нагрузкой БД было установлено, что метод в 1.83 раза работает дольше внешнего пула. Однако анализ затрат памяти помог установить, что для реализации разработанного метода требуется в 3.24 раза меньше памяти, по сравнению с внешним пулом.

Таким образом, использование той или иной реализации зависит от условия задачи. Применение метода оправдано в тех задачах, где объем предоставляемой памяти ограничен, а скорость выполнения запросов не критична. В случае, когда в задаче важно время выполнения, следует использовать внешний пул. Однако при использовании следует учитывать затраты на его разработку, тестирование, конфигурацию и встраиваемость в код. В то время, как использование разработанного метода подразумевает написание вызова одной функции.



## **Вывод**

## Список использованных источников

1. ГОСТ 34.320-96 Информационные технологии. Система стандартов по базам данных. Концепции и терминология для концептуальной схемы и информационной базы.
2. ГОСТ Р ИСО/МЭК ТО 10032-2007: Эталонная модель управления данными.
3. Основные классы современных параллельных компьютеров. [Электронный ресурс]. Режим доступа: <https://parallel.ru/computers/classes.html>.
4. Lyman P., Varian H.R. How much information Архивная копия от 19 февраля 2018 на Wayback Machine. Release of the University of California. Oct.27, 2003.
5. Копцев П. Ю., Аникьева Э. Н. Зарождение и история развития баз данных // Наука и образование. – 2018. – Том 1 №1. Режим доступа: <http://www.opusmgau.ru/index.php/see/article/view/288>.
6. Дикарев Н. И., Шабанов Б. М., Шмелев А. С. Векторный потоковый процессор: оценка производительности // Известия ЮФУ. Технические науки. 2014. №12 (161). Режим доступа: <https://cyberleninka.ru/article/n/vektornyuy-potokovyy-protssessor-otsenka-proizvoditelnosti>.
7. Knowledge Base of Relational and NoSQL Database Management Systems. [Электронный ресурс]. Режим доступа: <https://db-engines.com/en/ranking>.
8. Концепция архитектуры PostgreSQL. [Электронный ресурс]. Режим доступа: <http://www.dataved.ru/2014/09/postgresql.html>
9. PostgreSQL: Документация: 12.8. [Электронный ресурс]. Режим доступа: <https://postgrespro.ru/docs/postgresql/14>.
10. Тарасов, С. В. СУБД для программиста. Базы данных изнутри / С. В. Тарасов. — Москва: СОЛОН-Пресс, 2018. — 320 с. Режим доступа: <https://www.iprbookshop.ru/90409.html>.
11. Пан К. С., Цымблер М. Л. Разработка параллельной СУБД на основе последовательной СУБД PostgreSQL с открытым исходным кодом // Вестник ЮУрГУ. Сер. Математическое моделирование и программирование. – 2012. – №18. – С. 277. Режим доступа: <https://cyberleninka.ru/article/n/razrabotka->

parallelnoy-subd-na-osnove-posledovatelnoy-subd-postgresql-s-otkryтым-  
ishodnym-kodom.

12. Воронова, Н. М. Алгоритмы оценки производительности модуля работы с данными / Н. М. Воронова, А. С. Кованова, Н. С. Корж // Инновации. Наука. Образование. – 2021. – № 38. – С. 647-658.
13. Shetty N. Everything you need to know about Connection Pooling in Postgres. – 2019. [Электронный ресурс]. Режим доступа: <https://www.ashnik.com/everything-you-need-to-know-about-connection-pooling-in-postgres/>.
14. Object Pool. [Электронный ресурс]. Режим доступа: <https://www.oodeesign.com/object-pool-pattern.html>.
15. Шиндов Д. А. Разработка пула соединений для работы с СУБД MYSQL на языке программирования C++ // ББК 1 А28. – 2021. – С. 112-113.
16. Aboagye M. Improve database performance with connection pooling. – 2020. [Электронный ресурс]. Режим доступа: <https://stackoverflow.blog/2020/10/14/improve-database-performance-with-connection-pooling/>.
17. Документация к Postgres Pro Enterprise 14.8.1. [Электронный ресурс]. Режим доступа: <https://postgrespro.ru/docs/enterprise/14>.
18. Shaikh S. S., Pachghare V. K. A Comparative Study of Database Connection Pooling Strategy // International Research Journal of Engineering and Technology. – 2017.