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 ВКР _____ (Подпись, дата) М. В. Филиппов (И.О.Фамилия)

Консультант _____ (Подпись, дата) Ю. М. Гаврилова (И.О.Фамилия)

Нормоконтролер _____ (Подпись, дата) Д. Ю. Мальцева (И.О.Фамилия)

2022 г.

РЕФЕРАТ

Расчетно-пояснительная записка: 59 страниц, 3 таблицы, 22 рисунка, 14 листингов, 19 источников, 2 приложения.

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

Объектом разработки является метод параллельного выполнения запросов к системе управления базами данных PostgreSQL.

Цель работы — разработка метода параллельного выполнения запросов к системе управления базами данных PostgreSQL в пределах одного соединения.

В первой части работы выполняется анализ существующих подходов повышения эффективности выполнения запроса. Во второй части описываются разрабатываемые алгоритмы. В третьей части приведено описание примененных технологий и разработанных сущностей. В четвертой части выполнен сравнительный анализ времени работы метода и различных реализаций выполнения запросов.

СОДЕРЖАНИЕ

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 базами 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 конкурен-

тоспособности. 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:

- Выполнить анализ предметной области и существующих методов выполнения запросов в MPP системах;
- Разработать метод параллельного выполнения запросов к СУБД PostgreSQL в пределах одного соединения;
- Реализовать разработанный метод;
- Выполнить сравнительный анализ времени работы метода и различных реализаций выполнения запросов.

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 Формализация задачи

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.

Согласно исследованиям, 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 1.1 приведены результаты сравнения работы однопоточной и многопоточной программ на больших данных [7].

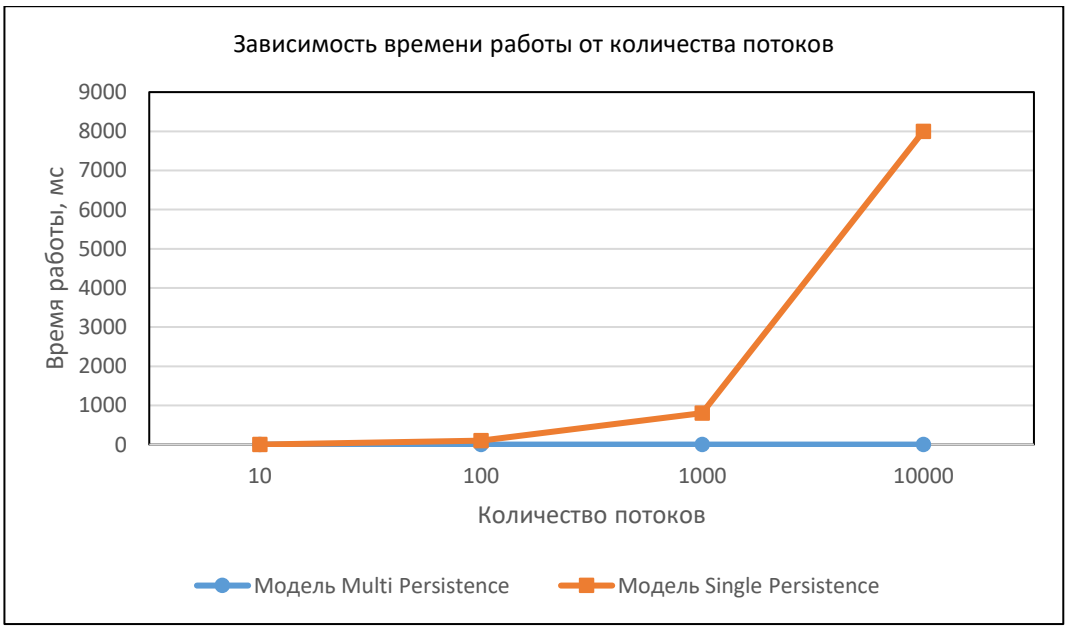

Рисунок 1.1 – Сравнение однопоточной и многопоточной программ, выполняющих запросы чтения к БД.

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 [8]), 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.

1.2 Анализ СУБД

В таблице 1.1 представлен рейтинг популярности СУБД, составленный компанией «DB-Engines» по состоянию на начало 2022 года [9].

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 [9]. Выбор аргументирован такими преимуществами как доступность исходного кода и кросс-платформенность.

1.3 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 1.2) [10].

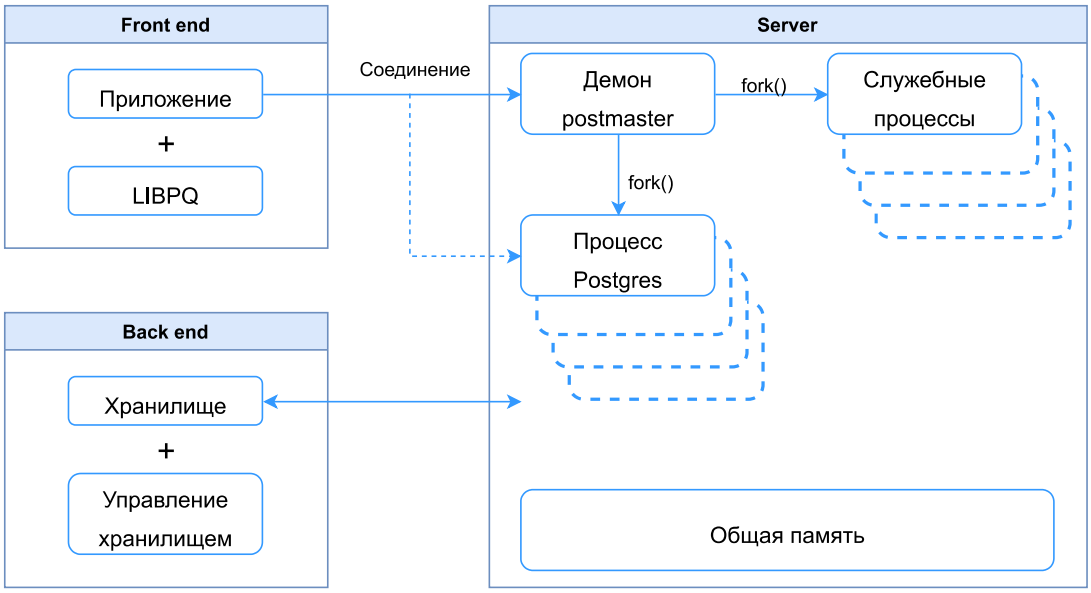

Рисунок 1.2 – «Клиент-серверная» архитектура PostgreSQL.

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.

1.3.1 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 [11].

1.3.2 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 [12].

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 [13]. Описанные механизмы позволяют обеспечить целостность данных при одновременном обращении к ним [11].

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

1.3.3 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 1.3 представлена организация хранилища [13].

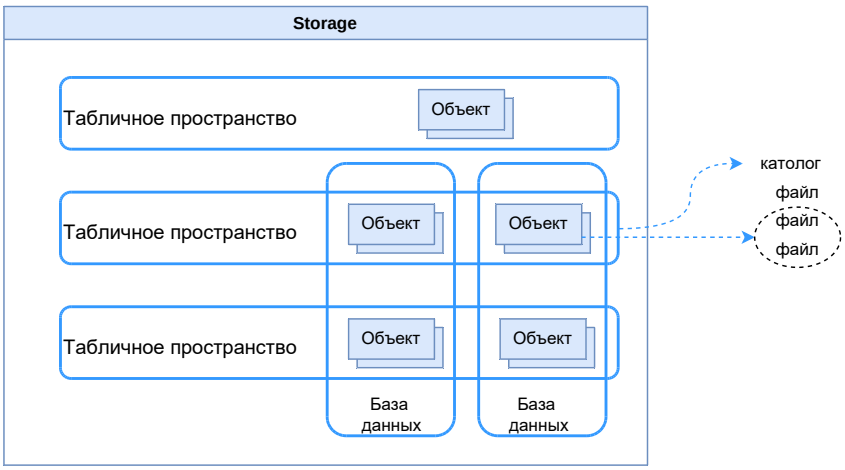

Рисунок 1.3 – Организация хранилища PostgreSQL.

1.4 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» [11].

1.5 Классификация существующих решений

На сегодняшний день существует несколько подходов повышения производительности при выполнении запроса. Ниже будут рассмотрены более подробно существующие решения.

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 [14].

В PostgreSQL отсутствует встроенный пул соединений [11], 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 создание необходимого количества соединений. Предельный размер пула 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 [15].

Преимущество данного метода наглядно демонстрирует следующий пример. Устанавливается соединение для 10 клиентов, каждый из которых выполняет 10000 запросов к 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 [16].

Главным недостатком внешнего пула являются затраты на его разработку. В том числе могут возникать трудности при его встраивании в крупный модуль, которые могут привести 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.

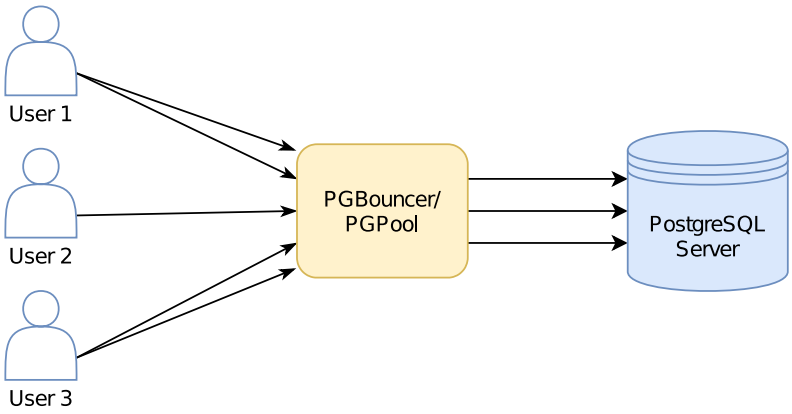

Рисунок 1.4 – Архитектура приложения, использующего внешнюю службу.

Вместо создания соединений с PostgreSQL, создается более «легкое» соединение с PGBouncer/PGPool, которые пользуются уже существующими соединениями с БД [13]. 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. Сообщение об ошибке 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 [11]. 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 значения указанных параметров зависит, насколько увеличится или уменьшится пропускная способность транзакции.

1.5.1.3 Server pool

Хотя PostgreSQL не имеет встроенного пула соединений, 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 новой пул. Пулы функционируют только на уровне транзакций, т.е. процесс может переключиться на обслуживание нового соединения только после завершения транзакции [17].

На рисунках 1.5 – 1.6 представлен цикл соединения с БД без пула и с его использованием соответственно [18].

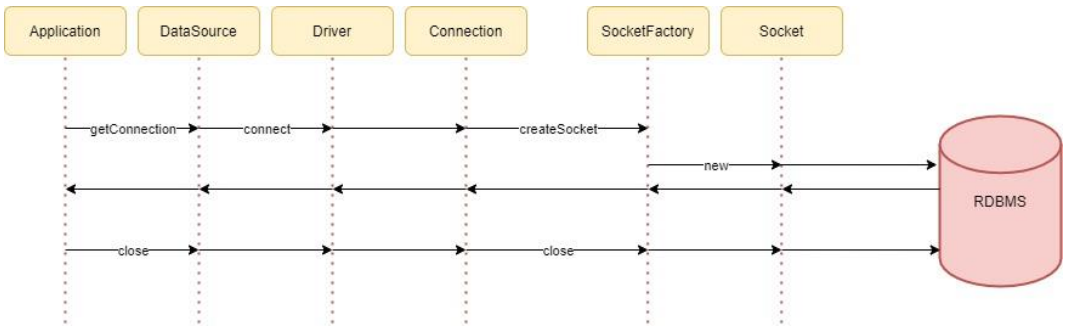

Рисунок 1.5 – Цикл соединения с БД без использования пула.

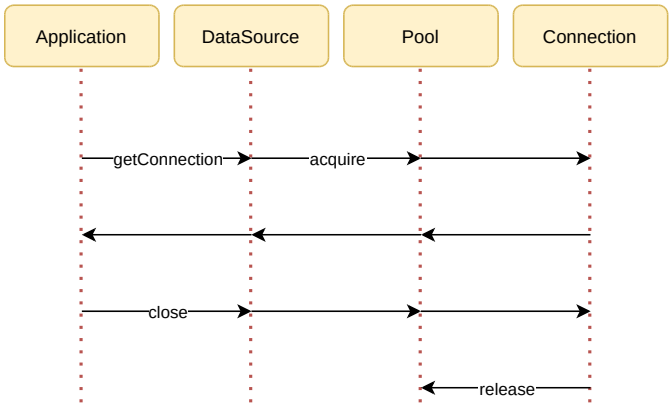

Рисунок 1.6 – Цикл соединения с БД с использованием пула.

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 запроса.

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 объемом данных, но возвращающих пользователю всего несколько строк [11].

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).

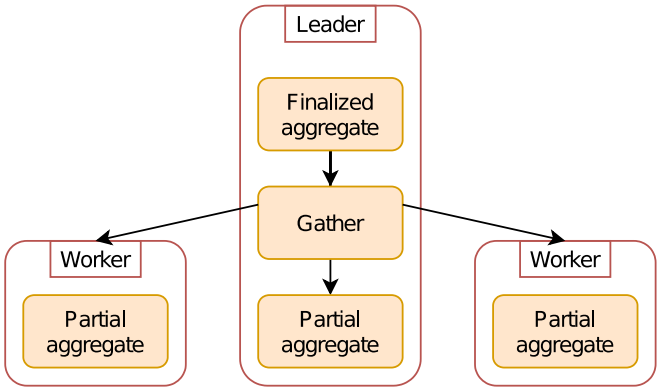

Рисунок 1.7 — Дерево частично параллельного запроса.

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) [11]. 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 1.8 представлена схема реализации параллельного запроса.

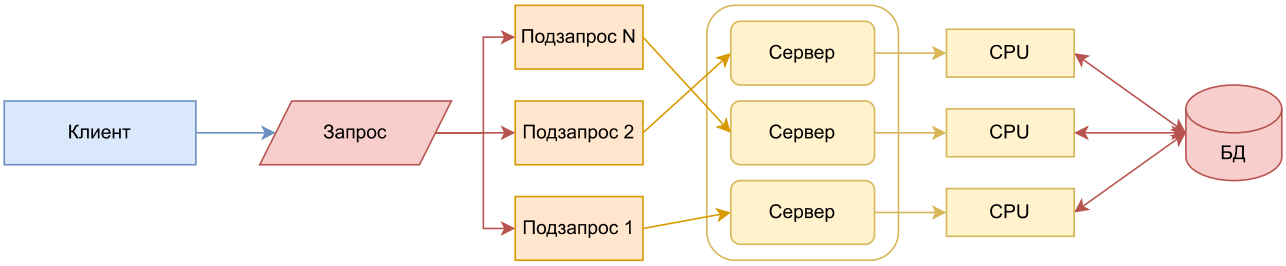

Рисунок 1.8 – Схема реализации параллельного запроса.

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 [11].

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 ис-

полнителей 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 числа создаваемых соединений с 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 большого числа 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, которые не позволяют достичь требуемой эффективности в многопоточной программе.

Задача разрабатываемого метода заключается в сокращении числа открытых соединений путем реализации возможности параллельной работы из разных потоков с одним объектом соединения.

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 Требования к разрабатываемому методу

Метод должен работать с одним соединением и строкой запроса. Работа метода должны быть асинхронной. Метод должен обрабатывать запрос корректно за конечное время. На рисунке 2.1 изображена ветка A0 диаграммы IDEF0.

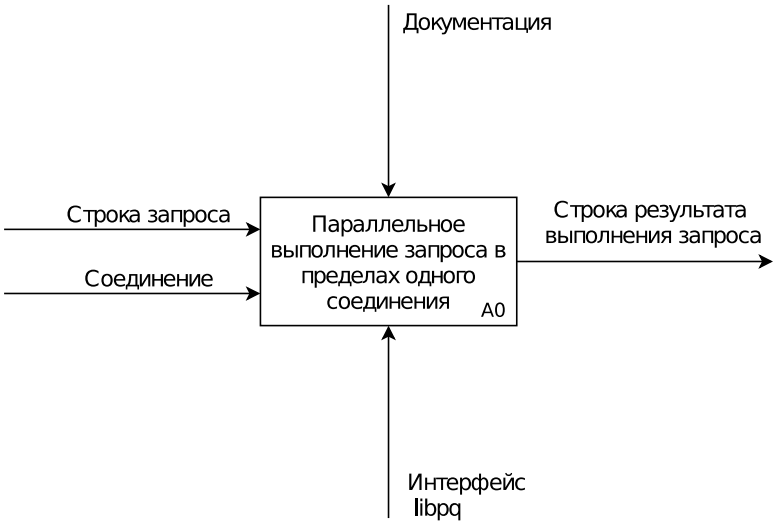

Рисунок 2.2 – IDEF0-диаграмма разрабатываемого метода.

2.2 Объект соединения PGconn

Объект PGconn является структурой (*pg_conn*), содержащей данные состояния текущего подключения к БД. Помимо базовых параметров, таких как порт, хост, имя БД, пароль, структура содержит большое число полей, недоступных

пользователю, но необходимых для взаимодействия с сервером. Ключевыми параметрами для разработки метода являются параметры, описывающие входной и выходной буферы, а также очередь команд.

Выходной буфер предназначен для хранения данных, еще не отправленных на сервер. Помимо самого буфера, структура хранит выделенный размер памяти и текущий размер данных в буфере (листинг 2.1) [11].

Листинг 2.1 – Поля структуры pg_conn, описывающие выходной буфер.

```
char    *outBuffer;     /* currently allocated buffer */
int     outBufSize;     /* allocated size of buffer */
int     outCount;       /* number of chars waiting in buffer */
```

Входной буфер необходим для получения данных от сервера, которые еще не были обработаны. В *pg_conn* помимо размера памяти, выделенной буферу, определены также указатели на начало и конец необработанных данных (листинг 2.2) [11].

Листинг 2.2 – Поля структуры pg_conn, описывающей входной буфер.

```
char    *inBuffer;      /* currently allocated buffer */
int     inBufSize;      /* allocated size of buffer */
int     inStart;        /* offset to first unconsumed data in buffer */
int     inCursor;       /* next byte to tentatively consume */
int     inEnd;          /* offset to first position after avail data */
```

Для каждого соединения определена очередь команд, реализованная в качестве односвязного списка (листинг 2.3) [11]. Помимо строки запроса, каждый элемент списка содержит тип протокола.

Листинг 2.3 – Структура, описывающая очередь команд.

```
typedef struct PGcmdQueueEntry
 {
   PGQueryClass queryclass;         /* Query type */
   char      *query;                /* SQL command, or NULL if none/unknown/OOM */
   struct PGcmdQueueEntry *next;  /* list link */
 } PGcmdQueueEntry;
```

2.3 Протокол запросов

Протокол запросов определяет взаимодействие клиента с сервером.

Простой протокол

Клиент передает сообщение серверу в виде текстовой строки. Допускается конкатенация нескольких SQL-запросов в одну строку. В таком случае, ошибка выполнения в одном из них приводит к откату изменений, сделанных предыдущими операторами. Для изменения описанного поведения, команда должна использовать операторы COMMIT или ROLLBACK.

В ответ сервер возвращает одно или несколько сообщений и завершает общение сообщением *ReadyForQuery*, которое означает, что клиент может выполнять безопасную передачу следующего сообщения. Поскольку требуется асинхронное выполнение отправки, реализованный метод не дожидается получения *ReadyForQuery*, а переходит к прослушиванию сервера. При взаимодействии клиента с сервером на основе данного протокола, результатом является текстовый формат.

Расширенный протокол

Данный протокол разбивает вышеописанный на несколько этапов: Bind, Parse, Describe, Execute и Close. Несмотря на то, что расширенный протокол сложнее, он является более безопасным.

Для разрабатываемого метода будет достаточным использование простого протокола.

2.4 Синхронизация потоков

Поскольку результат выполнения запроса напрямую зависит от текущего состояния параметров соединения, требуется механизм их защиты от изменений другим потоком. Таким механизмом был выбран мьютекс, так как в отличии от семафора, он обеспечивает более высокую степень параллелизма. Выбор также

аргументирован тем, что только владелец мьютекса имеет право его разблокировки [19].

2.5 Декомпозиция разрабатываемого метода

Разрабатываемый метод состоит из следующих этапов обработки запроса:

1. Отправка запроса серверу
    a. Формирование очереди запросов
    b. Отправка запроса из очереди серверу
2. Получение ответа

На рисунке 2.2 представлена IDEF0-диаграмма разрабатываемого метода.

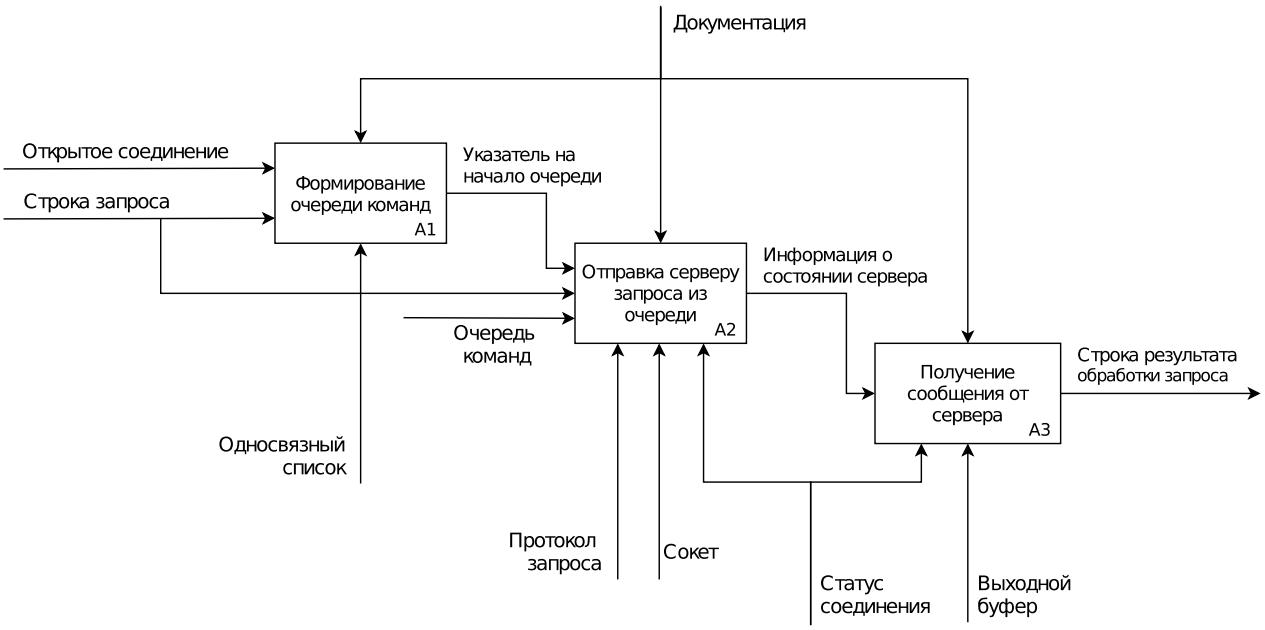

Рисунок 2.2 – IDEF0-диаграмма, ветка A0.

Далее будет подробно рассмотрен каждый из этапов.

2.5.1 Формирование очереди команд

Работа метода начинается с блокировки мьютекса. После защиты параметров от изменений выполняется их проверка, в том числе проверка безопасности текущего соединения.

В случае успешного прохождения всех проверок следует определить точку входа очереди команд. Этот этап подразумевает выделение памяти для начала очереди или для конца.

После определения точки входа, выполняется добавление команды в очередь и указание типа протокола запроса.

На рисунке 2.3 изображена схема формирования очереди команд.

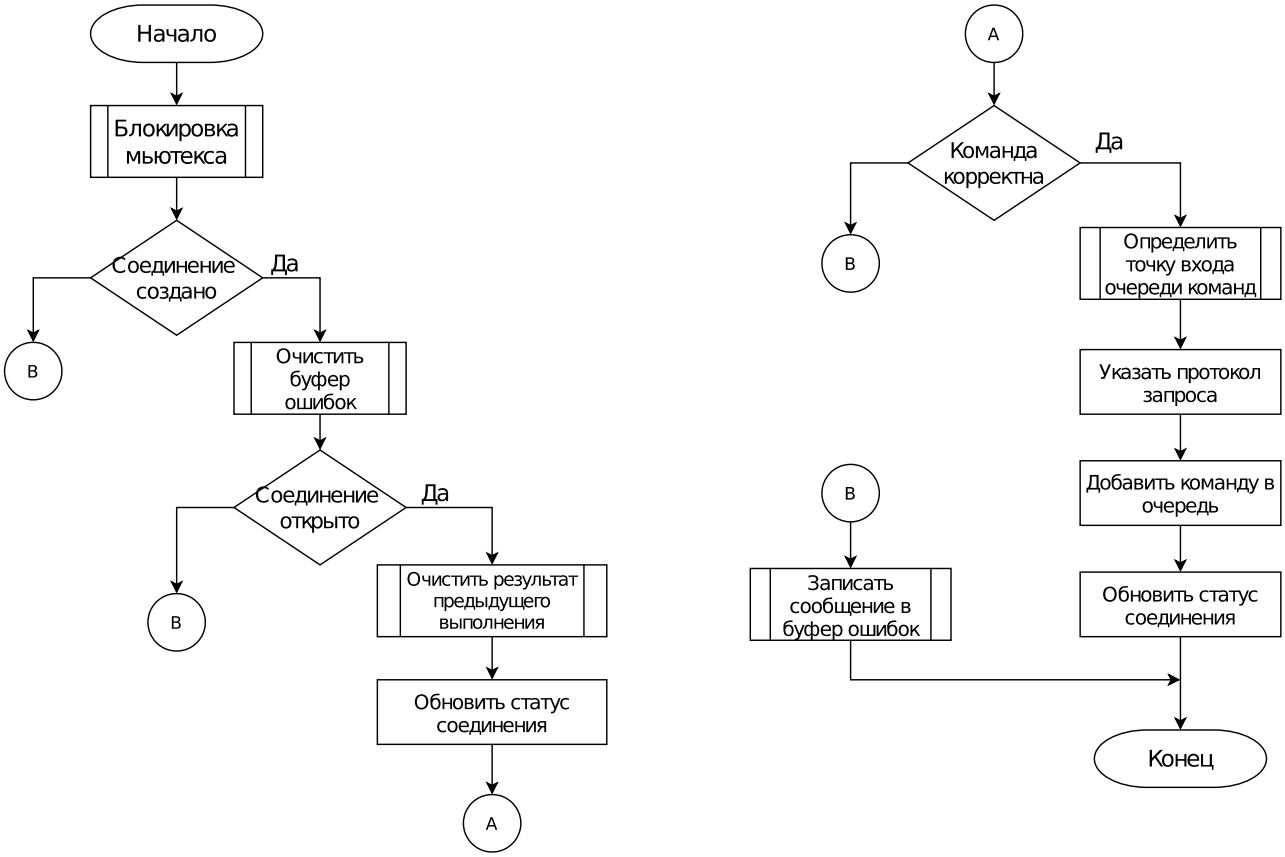

Рисунок 2.3 – Схема алгоритма формирования очереди команд.

2.5.2 Отправка серверу запроса из очереди

Помимо добавления запроса в очередь выполняется его запись в выходной буфер. Отправка сообщения серверу происходит в двух случаях:

1. размер выходного буфера превышает 8Кб (стандартный размер буфера в Unix-системах);
2. происходит вызов функции *fflush()* на стороне клиента.

Проверка размера буфера позволяет избегать отправки маленьких пакетов.

При этом запрос, расположенный в начале очереди является запросом, выполняющимся на сервере. Поэтому добавление запросов происходит в конец очереди.

2.5.3 Получение сообщения от сервера

Получение результата от сервера выполняется путем прослушивания его состояния. До тех пор, пока сервер готов передать запрос, выполняется чтение данных. Обработки и непосредственного чтения данных не происходит: полученные данные записываются во входной буфер. Как только сервер изменил свое состояние с «занят» на «свободен» его прослушивание завершается. Далее выполняется проверка полученного результата на корректность, а также изменение текущего состояния соединения.

Перед завершением обработки запроса следует выполнить разблокировку мьютекса для возможности корректной обработки следующего запроса.

На рисунке 2.4 изображена схема алгоритма получения результата выполнения запроса от сервера.

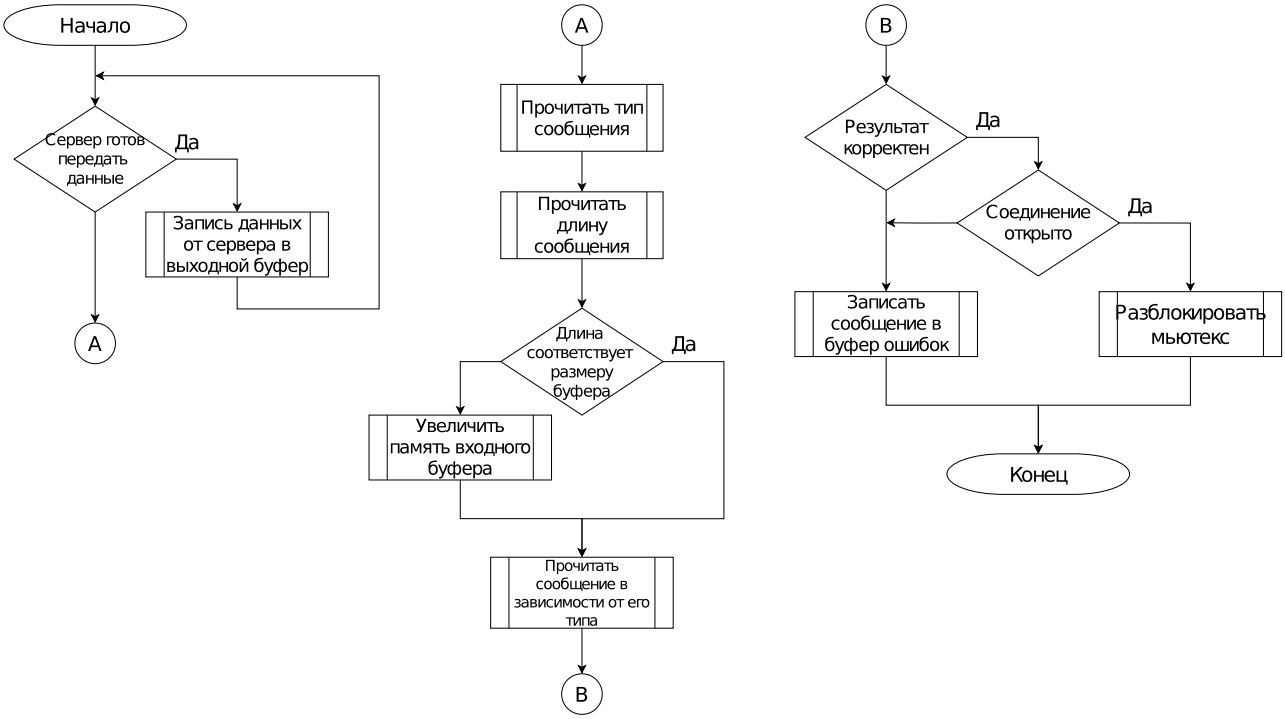

Рисунок 2.4 – Схема алгоритма получения сообщения от сервера.

2.6 Внешний модуль взаимодействия с разработанным методом

Внешний модуль предназначен для вызова метода, реализованного в качестве библиотечной функции. 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.

Структура внешнего модуля представлена на рисунке 2.5.

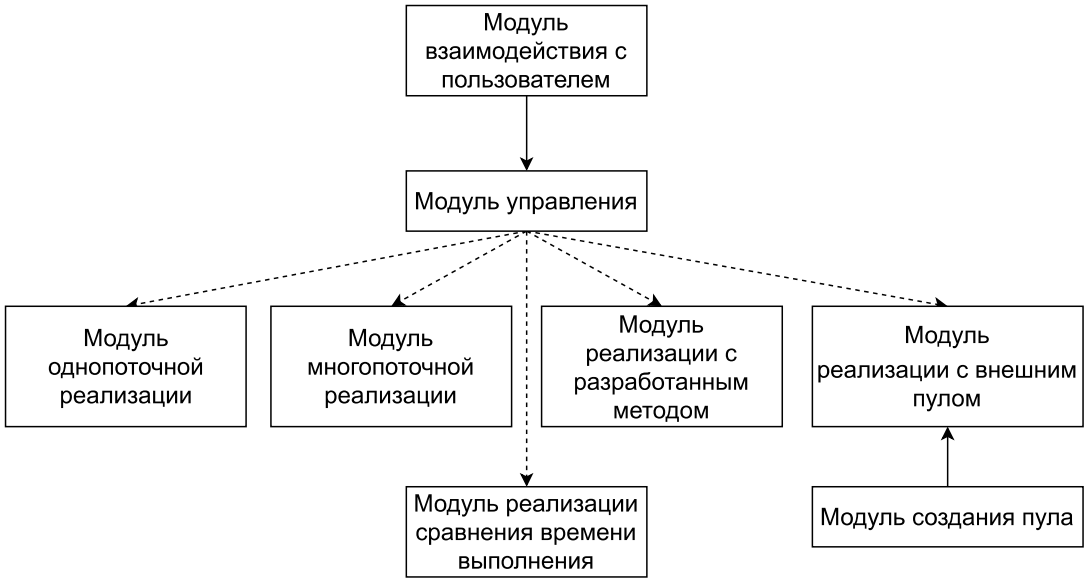

Рисунок 2.5 – Структура внешнего модуля.

2.7 Вывод

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

3 Технологический раздел

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

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

Разработанный метод является частью библиотеки libpq, причем пользовательским программам доступна только одна функция *PQexecThread()*.

Как было упомянуто выше, помимо разработанного метода был реализован внешний модуль. Задача внешнего модуля — взаимодействие пользователя с методом, т.е. создание заданного числа потоков, в каждом из которых происходит вызов функции *PQexecThread()*.

3.3 Реализация основных этапов метода

Реализация разрабатываемого метода может быть поделена на два этапа: отправка запроса серверу и получение результата (листинг 3.1). В случае возникновения ошибки при попытке отправить запроса, возвращается нулевой указатель. В случае успеха — результат, полученный от сервера.

Листинг 3.1 – Внешняя функция, реализующая разработанный метод.

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

Ниже будет более подробно рассмотрен 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. Функция является оберткой над внутренней функцией *PQsendQueryInternalThread()*. В ее работе можно выделить несколько стадий отправки запроса.

3.3.1.1 Блокировка мьютекса

Работа функции отправки команды начинается с блокировки мьютекса (листинг 3.2). Мьютекс защищает параметры подключения, от которых зависит результат выполнения команды, от изменений другими потоками.

Листинг 3.2 – Начало работы функции отправки запроса.

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

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

3.3.1.2 Проверка соединения на безопасность

Прежде всего проверяется наличие 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 запроса 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 какая-либо 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()* (листинг 3.4) выполняет определение точки входа очереди команд для текущего запроса. Для каждого соединения также определена очередь повторного использования. Она необходима для того, чтобы сократить количество *malloc*. После выполнения команды из основной очереди вместо вызова *free*, память, выделенная для ее хранения, сохраняется в

список повторного использования. Поэтому вновь поступивший запрос будет сохранен в очереди повторного использования.

Листинг 3.4 – Функция *pqAllocCmdQueueEntry()* определения точки входа очереди команд.

```
1219  static PGcmdQueueEntry *
1220  pqAllocCmdQueueEntry(PGconn *conn)
1221  {
1222      PGcmdQueueEntry *entry;
1223
1224      if (conn->cmd_queue_recycle == NULL)
1225      {
1226          entry = (PGcmdQueueEntry *) malloc(sizeof(PGcmdQueueEntry));
1227          if (entry == NULL)
1228          {
1229              appendPQExpBufferStr(&conn->errorMessage,
1230                                   libpq_gettext("out of memory\n"));
1231              return NULL;
1232          }
1233      }
1234      else
1235      {
1236          entry = conn->cmd_queue_recycle;
1237          conn->cmd_queue_recycle = entry->next;
1238      }
1239      entry->next = NULL;
1240      entry->query = NULL;
1241
1242      return entry;
1243  }
```

На листинге 3.5 представлен вызов функции *pqAllocCmdQueueEntry()*. В случае неудачи при выделении памяти для текущего запроса отправка не будет выполнена, функция отправки завершается.

Листинг 3.5 – 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

Помимо очереди запросов, выполняется заполнение выходного буфера. Запись выполняется непрерывно, т.е. в конец буфера, сразу за данными, уже расположенными в буфере. Размер буфера увеличивается по мере необходимости.

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.

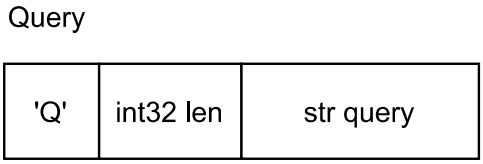

Рисунок 3.1 — Формат сообщения серверу.

Завершение записи заключается в проверке размера буфера и его возможной отправки серверу.

В случае, если какой-либо из этапов завершился с ошибкой, выполняется очистка выделенной памяти в очереди запросов и функция отправки завершается (листинг 3.6).

Листинг 3.6 – 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 определенный размер буфера или, если была вызвана функция *pqFlush()*, которая выполняет запись содержимого буфера в очередь для отправки в сокет (листинг 3.7). В простом протоколе запроса продвижение очереди команд выполняется согласно протоколу fe-protocol3 (протокол третьей версии).

Листинг 3.7 – 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 3.8 – 3.9).

Листинг 3.8 – Вызов функции, завершающей отправку сообщения серверу.

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

Листинг 3.9 – Добавление команды в очередь команд и обновление статуса соединения.

```
1253  static void
1254  pqAppendCmdQueueEntry(PGconn *conn, PGcmdQueueEntry *entry)
1255  {
1256      Assert(entry->next == NULL);
1257
1258      if (conn->cmd_queue_head == NULL)
1259          conn->cmd_queue_head = entry;
1260      else
1261          conn->cmd_queue_tail->next = entry;
1262
1263      conn->cmd_queue_tail = entry;
1264
1265      switch (conn->pipelineStatus)
1266      {
1267          case PQ_PIPELINE_OFF:
1268          case PQ_PIPELINE_ON:
1269              if (conn->asyncStatus == PGASYNC_IDLE)
1270                  conn->asyncStatus = PGASYNC_BUSY;
1271              break;
1272
1273          case PQ_PIPELINE_ABORTED:
1274              if (conn->asyncStatus == PGASYNC_IDLE)
1275              {
1276                  resetPQExpBuffer(&conn->errorMessage);
1277                  pqPipelineProcessQueue(conn);
1278              }
1279              break;
1280      }
1281  }
```

3.3.2 Получение результата от сервера

На листинге 3.10 представлена реализация ожидания поступления результата от сервера. Ожидание реализовано с помощью цикла, на каждой итерации которого выполняется проверка состояния сервера. Эта проверка осуществляется путем вызова функции *PQisBusy()*. Возвращаемое значение 0 показывает, что функция *PQgetResult()* гарантированно не будет заблокирована при ее вызове. Поскольку сама функция *PQisBusy()* не будет пытаться прочитать данные с сервера, то для выхода из занятого состояния необходим вызов функции *PQconsumeInput()*. Функция выполняет чтение данных и их запись в буфер.

Листинг 3.10 – Ожидание поступления результата от сервера.

```
2334            while (PQisBusy(conn))
2335            {
2336                if (!PQconsumeInput(conn))
2337                {
2338                    lastRes = NULL;
2339                    break;
2340                }
2341            }
```

Получение самого результата выполняется вызовом функции *PQgetResult()* (листинг 3.11). В случае, когда команда завершена и больше результатов не будет, возвращается нулевой указатель. Данная функция вызывается повторно в бесконечном цикле, работа которого завершается, как только был получен нулевой указатель. Если вызвать данную функцию, когда нет ни одного активного запроса, также вернется нулевой указатель сразу.

Для сохранения результата, отличного от *NULL,* была введена дополнительная переменная *PGresult *lastRes*. Ее значение соответствует предпоследнему вызову функции *PQgetResult()*.

Листинг 3.11 – Получение результата от сервера.

```
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        }
```

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 3.12).

Листинг 3.12 – Разблокирование мьютекса и завершение работы функции получения результата.

```
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 базы 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.

На рисунке 3.2 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.

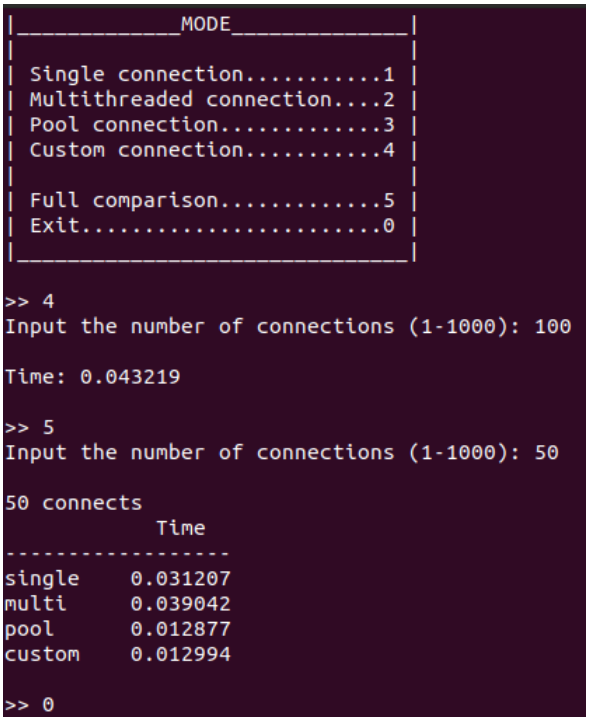

Рисунок 3.2 — Демонстрация работы внешнего модуля, выполняющего вызов реализованной функции.

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 3.13 – 3.14 приведена реализация класса *PGBackend*, содержащего метод создания пула. Помимо умных указателей используется мьютекс, 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 3.13 – Класс PGBackend.

```
#include <memory>
#include <mutex>
#include <string>
#include <queue>
#include <condition_variable>
#include <libpq-fe.h>
#include "pgconnection.h"

class PGBackend
{
public:
    PGBackend(const std::string host, const std::string port,
              const std::string dbName, const std::string user, const std::string password);

    std::shared_ptr<PGConnection> connection();
    void freeConnection(std::shared_ptr<PGConnection>);

private:
    void createPool(const std::string host, const std::string port,
                    const std::string dbName, const std::string user, const std::string password);

    std::mutex m_mutex;
    std::condition_variable m_condition;
    std::queue<std::shared_ptr<PGConnection>> m_pool;

    const int POOL_SIZE = 10;
};
```

Листинг 3.14 – Реализация методов класса PGBackend.

```
#include <iostream>
#include <thread>
#include <fstream>
#include <sstream>
#include "pgbackend.h"

using namespace std;

PGBackend::PGBackend(const string host, const string port,
                     const string dbName, const string user, const string password)
{
    createPool(host, port, dbName, user, password);
}

void PGBackend::createPool(const string host, const string port,
                           const string dbName, const string user, const string password)
{
    lock_guard<mutex> locker_( m_mutex );

    for (int i = 0; i < POOL_SIZE; i++) {
        m_pool.emplace(make_shared<PGConnection>(host, port, dbName, user, password));
    }
}

shared_ptr<PGConnection> PGBackend::connection()
{
    unique_lock<mutex> lock_( m_mutex );

    while (m_pool.empty()){
        m_condition.wait(lock_);
    }

    auto conn = m_pool.front();
    m_pool.pop();

    return conn;
}
```

3.5 Тестирование программного модуля

Для выполнения тестирования была создана БД, состоящая из таблиц размером 10, 100, 200, 500, 1000, 5000 и 10.000 строк. Тестирование включало в себя выполнение четырех базовых запросов БД: SELECT, INSERT, UPDATE и

DELETE. При выполнении каждого запроса отмечалась стабильная и корректная работа разработанного модуля.

В таблице 3.1 приведены результаты замеров времени выполнения каждого запроса для стандартной функции *PQexec()* и для разработанного метода *PQexecThread()*.

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

	PQexec	PQexecThread
SELECT	1.029	1.0557
INSERT	1.204	1.6248
UPDATE	1.123	1.223
DELETE	1.356	1.911

3.6 Вывод

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

В качестве языка программирования разработанного метода был выбран язык Си, языком программирования внешнего модуля — C++. Средой разработки был выбран SublimeText. Были подробно описаны этапы формирования и отправки запроса, а также получение ответа результата от сервера. Тестирование показало работоспособность реализованного ПО.

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;
2. параллельная;
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 при большом количестве создаваемых соединений.

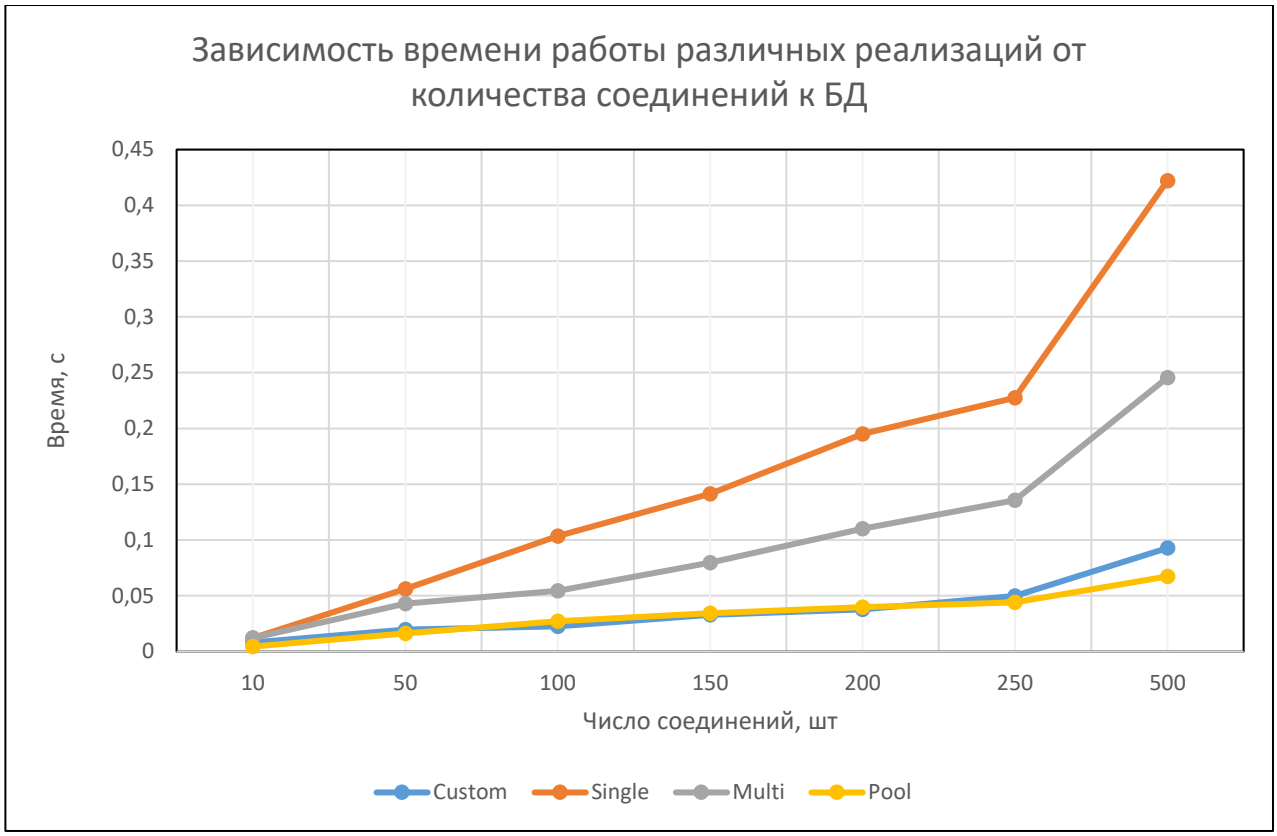

Рисунок 4.1 — График зависимости времени выполнения запроса от количества создаваемых соединений для каждой реализации.

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 libpq 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.

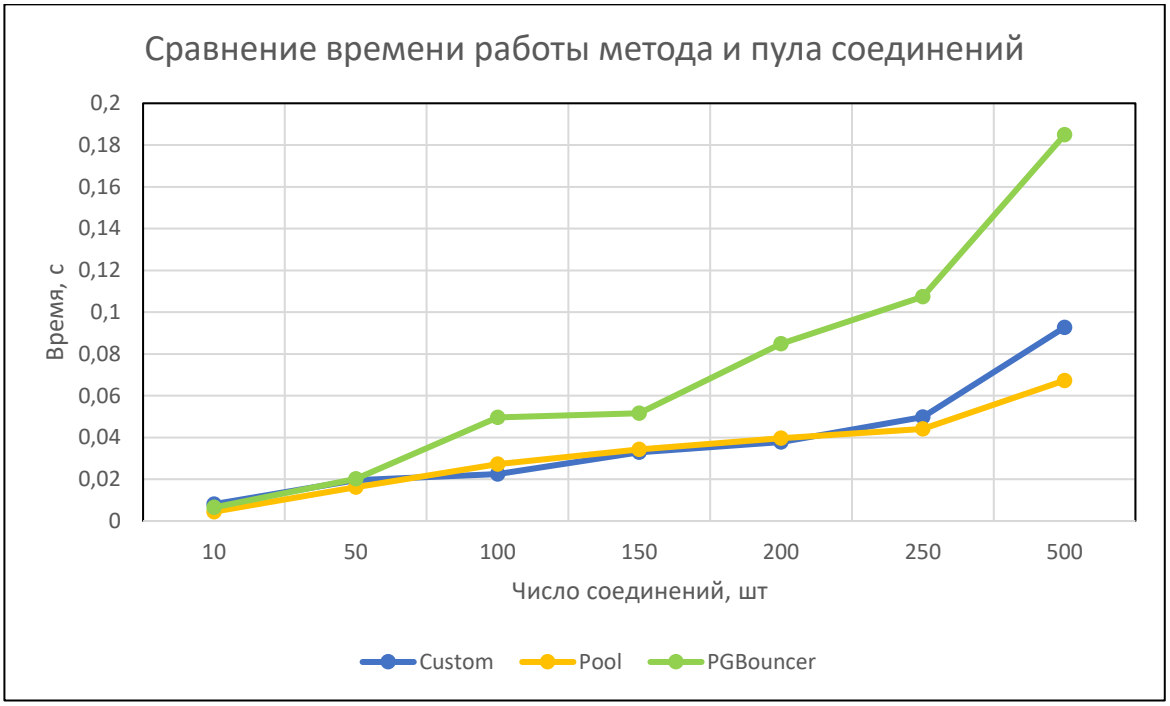

Рисунок 4.2 — Сравнение времени работы метода и пула соединений.

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 100000 строк. Конфигурация, описанная выше, представлена на рисунках 4.4 – 4.6.

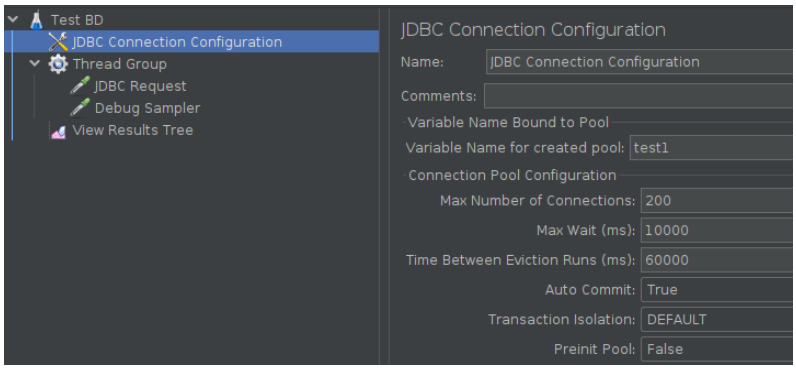

Рисунок 4.4 — Конфигурация пула соединений Jmeter.

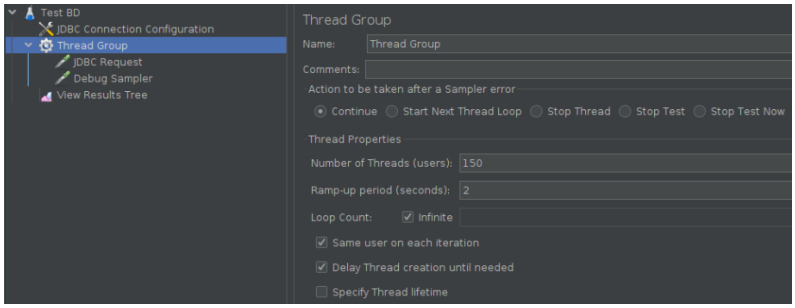

Рисунок 4.5 — Конфигурация потоков Jmeter.

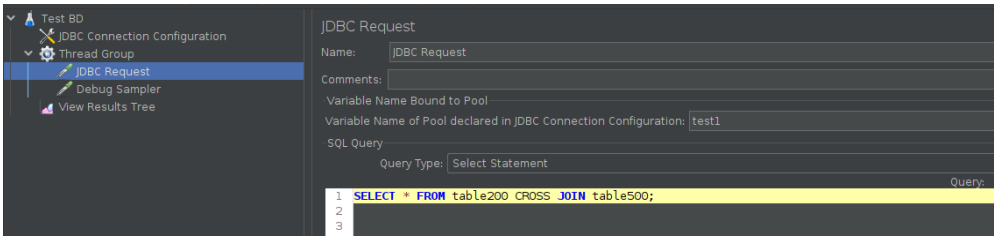

Рисунок 4.6 — Конфигурация запроса Jmeter к БД.

Результаты эксперимента приведены на рисунке 4.7. Следует отметить, что время работы реализованного метода сильно разнилось для каждого опыта. На графике представлены средние значения для 30 опытов.

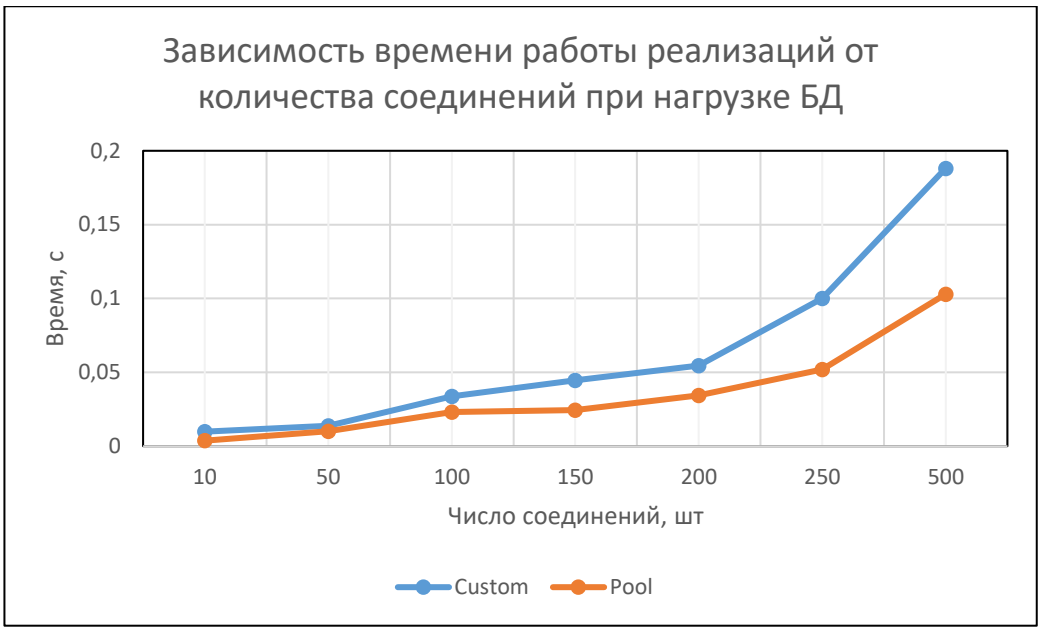

Рисунок 4.7 — График зависимости времени работы реализаций в зависимости от количества соединений при нагрузке БД.

Согласно графику, разработанный метод уступает по времени реализации, основанной на внешнем пуле соединении. 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 требуемых ресурсов

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
Многопоточная	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 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 вызов одной функции.

ЗАКЛЮЧЕНИЕ

В рамках выпускной квалификационной работы был разработан и реализован метод параллельного выполнения запросов к системе управления базами данных PostgreSQL в пределах одного соединения.

В результате проделанной работы были решены все поставленные задачи:

- Выполнен анализ предметной области и существующих методов выполнения запросов в MPP системах;
- Разработан метод параллельного выполнения запросов к СУБД PostgreSQL в пределах одного соединения;
- Реализован разработанный метод;
- Выполнен сравнительный анализ времени работы метода и различных реализаций выполнения запросов.

Разработанный метод имеет перспективу дальнейшего развития и улучшения. Так, может быть выполнена реализация пользовательского вывода информации об ошибках в случае конкатенации нескольких запросов в одну команду. Другим направлением развития может стать разработка метода управления ресурсным пулом в случае потери потоком соединения с базой данных.

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/vektornyy-potokovyy-protsessor-otsenka-proizvoditelnosti.
7. 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-otkrytym-ishodnym-kodom.
8. 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.
9. Концепция архитектуры PostgreSQL. [Электронный ресурс]. Режим доступа: http://www.dataved.ru/2014/09/postgresql.html
10. Knowledge Base of Relational and NoSQL Database Management Systems. [Электронный ресурс]. Режим доступа: https://db-engines.com/en/ranking.
11. PostgreSQL: Документация: 14.2. [Электронный ресурс]. Режим доступа: https://postgrespro.ru/docs/postgresql/14.

12. Джуба С., Волков А. Изучаем PostgreSQL 10. – Litres, 2022.
13. 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.
14. 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/.
15. Object Pool. [Электронный ресурс]. Режим доступа: https://www.oodesign.com/object-pool-pattern.html.
16. Шиндов Д. А. Разработка пула соединений для работы с СУБД MYSQL на языке программирования C++ // ББК 1 А28. – 2021. – С. 112-113.
17. 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/.
18. Документация к Postgres Pro Enterprise 14.8.1. [Электронный ресурс]. Режим доступа: https://postgrespro.ru/docs/enterprise/14.
19. Олег Цилюрик. Инструменты программирования в ядре: Часть 73. Параллелизм и синхронизация. Блокировки. Часть 1. [Электронный ресурс]. Режим доступа: https://developer.ibm.com/technologies/linux/

ПРИЛОЖЕНИЕ А

Внутренний метод библиотеки libpq

Листинг А.1 – Внешняя функция отправки запроса и получения результата.

```
PGresult *
PQexecThread(PGconn *conn, const char *query)
{
	if (!PQsendQueryThread(conn, query))
		return NULL;
	return PQgetResultThread(conn);
}
```

Листинг А.2 – Функция отправки запроса.

```
static int
PQsendQueryInternalThread(PGconn *conn, const char *query, bool newQuery)
{
	pthread_mutex_lock(&mutex);
	PGcmdQueueEntry *entry = NULL;

	if (!PQsendQueryStartThread(conn, newQuery))
		return 0;

	if (!query)
	{
		appendPQExpBufferStr(&conn->errorMessage,
					libpq_gettext("command string is a null pointer\n"));
		return 0;
	}

	entry = pqAllocCmdQueueEntry(conn);
	if (entry == NULL)
		return 0;

	if (conn->pipelineStatus == PQ_PIPELINE_OFF)
	{
		if (pqPutMsgStart('Q', conn) < 0 ||
			pqPuts(query, conn) < 0 ||
			pqPutMsgEnd(conn) < 0)
		{
			pqRecycleCmdQueueEntry(conn, entry);
			return 0;
		}

		entry->queryclass = PGQUERY_SIMPLE;
		entry->query = strdup(query);
	}
```

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

	pqAppendCmdQueueEntryAsync(conn, entry);
	return 1;

sendFailed:
	pqRecycleCmdQueueEntry(conn, entry);
	return 0;
}
```

Листинг А.3 – Функция создания очереди запросов.

```
static bool
PQsendQueryStartThread(PGconn *conn, bool newQuery)
{
	if (!conn)
		return false;

	if (newQuery)
		resetPQExpBuffer(&conn->errorMessage);

	if (conn->status != CONNECTION_OK)
	{
		appendPQExpBufferStr(&conn->errorMessage,
							libpq_gettext("no connection to the server\n"));
		return false;
	}

	if (conn->asyncStatus != PGASYNC_IDLE &&
		conn->pipelineStatus == PQ_PIPELINE_OFF)
	{
		appendPQExpBufferStr(&conn->errorMessage,
					libpq_gettext("another command is already in progress\n"));
		return false;
	}

	if (conn->pipelineStatus != PQ_PIPELINE_OFF)
	{
		switch (conn->asyncStatus)
		{
			case PGASYNC_IDLE:
			case PGASYNC_READY:
			case PGASYNC_READY_MORE:
			case PGASYNC_BUSY:
				break;
			case PGASYNC_COPY_IN:
			case PGASYNC_COPY_OUT:
			case PGASYNC_COPY_BOTH:
				appendPQExpBufferStr(&conn->errorMessage,
```

```
                                        libpq_gettext("cannot queue commands during COPY\n"));
                                return false;
                        }
                }
                else
                {
                        pqClearAsyncResult(conn);
                        conn->singleRowMode = false;

                }
                conn->asyncStatus = PGASYNC_IDLE;
                return true;
}
```

Листинг А.4 – Функция получения результатов от сервера.

```
PGresult *
PQgetResultThread(PGconn *conn)
{
        PGresult *res = NULL;
        PGresult *lastRes = NULL;

        for (;;)
        {
                while (PQisBusy(conn))
                {
                        if (!PQconsumeInput(conn))
                        {
                                lastRes = NULL;
                                break;
                        }
                }

                res = PQgetResult(conn);
                if (res == NULL)
                        break;

                PQclear(lastRes);
                lastRes = res;

                if (PQresultStatus(lastRes) == PGRES_COPY_IN ||
                        PQresultStatus(lastRes) == PGRES_COPY_OUT ||
                        PQresultStatus(lastRes) == PGRES_COPY_BOTH ||
                        PQstatus(conn) == CONNECTION_BAD)
                        break;
        }

        pthread_mutex_unlock(&mutex);
        return lastRes;
}
```

ПРИЛОЖЕНИЕ Б

Внешний модуль для вызова реализаций

Листинг Б.1 – Модуль с использованием разработанного метода.

```
const string CONN_FILE = "connection_data/connection.data";
static const char *QUERY = "SELECT * FROM table100;";

void connectFunction(PGconn *conn, const int idx)
{
	if (PQstatus(conn) != CONNECTION_OK) {
		cout << "Can't open database: " << PQerrorMessage(conn) << endl;
		return;
	}

	PGresult *res = PQexecThread(conn, QUERY);
	if (PQresultStatus(res) != PGRES_TUPLES_OK) {
		PQclear(res);
		cout << "Database query completed with an error: " << PQresultStatus(res) << endl;
		return;
	}

	PQclear(res);
}
```

```
double customConn(const int numThreads)
{
	string host, port, dbName, user, password;
	readParamsFromFile(CONN_FILE, host, port, dbName, user, password);

	try {
		thread thr[numThreads];

		clock_t begin = clock();

			PGconn *conn = PQsetdbLogin(host.c_str(), port.c_str(), nullptr, nullptr,
                                                                dbName.c_str(),
user.c_str(), password.c_str());

			for (int i = 0; i < numThreads; i++) {
				thr[i] = thread(connectFunction, conn, i);
			}

			for (int i = 0; i < numThreads; i++) {
				thr[i].join();
			}

			PQfinish(conn);

		clock_t end = clock();
```

```
            double seconds = (double)(end - begin) / CLOCKS_PER_SEC;
            return seconds;
        }

        catch (const std::exception &er) {
            cerr << er.what() << endl;
            return 1;
        }
}
```

Листинг Б.2 – Модуль с использованием пула соединения.

```
class PGBackend
{
public:
   PGBackend(const std::string host, const std::string port,
         const std::string dbName, const std::string user, const std::string password);

   std::shared_ptr<PGConnection> connection();
   void freeConnection(std::shared_ptr<PGConnection>);

private:
   void createPool(const std::string host, const std::string port,
               const std::string dbName, const std::string user, const std::string password);

   std::mutex m_mutex;
   std::condition_variable m_condition;
   std::queue<std::shared_ptr<PGConnection>> m_pool;

   const int POOL_SIZE = 10;
};
```

```
PGBackend::PGBackend(const string host, const string port,
                        const string dbName, const string user, const string password)
{
        createPool(host, port, dbName, user, password);
}
```

```
void PGBackend::createPool(const string host, const string port,
                                    const string dbName, const string user, const string password)
{
        lock_guard<mutex> locker_( m_mutex );

        for (int i = 0; i < POOL_SIZE; i++) {
            m_pool.emplace(make_shared<PGConnection>(host, port, dbName, user, pass-
word));
        }
}
```

```
shared_ptr<PGConnection> PGBackend::connection()
{
	unique_lock<mutex> lock_( m_mutex );

	while (m_pool.empty()){
		m_condition.wait(lock_);
	}

	auto conn = m_pool.front();
	m_pool.pop();

	return conn;
}
```

```
void PGBackend::freeConnection(shared_ptr<PGConnection> conn)
{
	unique_lock<mutex> lock_(m_mutex);
	m_pool.push(conn);
	lock_.unlock();
	m_condition.notify_one();
}
```