СЛАЙД 3

С ростом популярности СУБД возникает вопрос об оптимизации ее работы. Так как один из самых распространенных способов увеличения производительности — параллельное выполнение, следует рассмотреть оптимизацию многопоточной программы.

Согласно исследованию, операция доступа (без какой-либо оптимизации) к базе данных объемом 100.000 записей в случае многопоточной программы выполняется примерно в 1000 раз быстрее. В тоже время однопоточная программа показывает нестабильную работу на больших данных.

Несмотря на указанное преимущество многопоточности, узким горлышком все равно остается операция подключения, одна из самых дорогостоящих (т.к. процесс подключения занимает от 2 до 3 МБ памяти).

СЛАЙД 4.А

Данная работа будет основываться на объектно-реляционной СУБД PostgreSQL 14-ой версии. Выбор аргументирован такими преимуществами как доступность исходного кода и кроссплатформенность.

PostgreSQL содержит инструменты для реализации многопоточности. Однако, обратившись к документации, можно выделить следующее ограничение: «параллельное выполнение команд из разных потоков через один объект соединения недопустимо»

СЛАЙД 4.Б

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

Клиентская часть состоит из пользовательского приложения, инициирующего соединение с сервером.

Соединение принимается процессом-демоном *postmaster*, который в дальнейшем с помощью системного вызова *fork()* создаст новый процесс Postgres для обслуживания данного соединения, а также ряд служебных процессов.

Третья часть сформирована из хранилища данных и средств его управления.

СЛАЙД 5

На сегодняшний день можно выделить несколько подходов повышения производительности при выполнении запросов. Один из них – сокращение числа соединений путем использования пула. Пул соединений представляет из себя набор открытых и готовых к использованию соединений с БД.

И хотя в PostgreSQL отсутствует встроенный пул соединений, он может быть реализован на основе библиотеки libpq или в качестве внешней службы.

СЛАЙД 6

Преимущество использование пула – это увеличение пропускной способности транзакции до 60%.

Однако каждый из описанных пулов имеет свои ограничения. Так, главным недостатком пула на основе библиотеки – затраты на его разработку. В том числе необходимость изменения архитектуры приложения. Недостаток пула в качестве внешней службы – однопоточная реализация самой службы. Встроенный же пул доступен только в коммерческой версии.

СЛАЙД 7

Задача разрабатываемого метода заключается в сокращении числа открытых соединений путем реализации параллельной работы из разных потоков с одним объектом соединения. Он состоит из двух этапов обработки запроса.

Он состоит из двух этапов обработки запроса: 1ый – отправка запроса серверу, включающий а) формирование очереди запросов и б) отправку запроса из очереди; 2й – получение ответа от сервера. Рассмотрим более подробно каждый из этапов.

СЛАЙД 8

1 этап подразумевает отправку запроса серверу. Работа функции отправки начинается с блокировки мьютекса. Прежде всего мьютекс необходим для защиты параметров подключения, от которых зависит результат выполнения, от изменений другими потоками.

После того как была произведена блокировка, выполняется проверка корректности параметров и инициализация соответствующих полей. В том числе, определяется точка входа очереди команд. После указания протокола запросов, команда может быть добавлена в очередь.

СЛАЙД 9

Ожидание поступления результата от сервера реализовано с помощью бесконечного цикла, на каждой итерации которого выполняется проверка состояния сервера. Пока сервер готов вернуть данные, выполняется их чтение и запись во входной буфер. Как только состояние сервера меняется на «свободен», выполняется обработка полученных данных. Обработка включает в себя чтение типа сообщения, его длины, и непосредственно сообщение результата.

Перед выходом из функции выполняется проверка корректности полученных данных и разблокировка мьютекса для обработки следующего запроса.

СЛАЙД 10

Помимо встроенного модуля, был реализован внешний, выполняющий вызов разработанного метода. Кроме того, пользователю доступны однопоточная, многопоточная реализации, реализация с внешним пулом и сравнение времени выполнения каждой.

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.

СЛАЙД 11

В данной работе было проведено исследование времени работы реализованного метода. Выполнялось сравнение времени выполнения простого запроса для 4 реализаций.

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).

Из опыта видно, что конкурентоспособность по времени разработанному методу составляет только пул соединений.

СЛАЙД 12

Поэтому было проведено другое исследование, где выполнялось сравнение времени работы пула, использующего библиотеку libpq и пула, реализованного в качестве внешней службы (PGBouncer), с разработанным методом.

Согласно результатам эксперимента, при более 100 соединений PGBouncer теряет временную эффективность: в 2 раза работает дольше, чем внешний пул. Однако при 10 соединениях работает быстрее разработанного метода в 1.24 раза.

СЛАЙД 13

Также было выполнено исследование требуемых ресурсов. Был проведен эксперимент, фиксирующий затраты памяти для каждой реализации в случае создания 10 соединений и выполнения простого запроса.

Согласно результатам, наибольшее потребление памяти (593 байта) у многопоточной реализации, что в 3 раза больше, чем память, потребляемая разработанным методом. Также, метод требует в 3 раза меньше памяти, чем внешний пул.