УДК 519.876.5

# МОДЕЛИРОВАНИЕ ПРИОРИТЕЗАЦИИ НАГРУЗКИ В ПУЛЕ ПОТОКОВ ДЛЯ СЕРВЕРА БАЗ ДАННЫХ

**Труб И.И. (Москва)**

## Введение

В работе [5] рассмотрена имитационная модель пула потоков (трэдпула) для СУБД-сервера MySQL. Она хорошо описывает функционирование трэдпула в случае однородного потока запросов со своими параметрами (распределение длительности обслуживания, частота и длительность обращения к диску и др.). Однако в случае смешанных нагрузок трэдпул работает неэффективно, и никакой подбор доступных параметров в рамках существующей модели не позволяет исправить ситуацию и повысить Quality-of-Service (QoS). Явление иллюстрируется рисунком на основе экспериментальных данных (рис. 1).

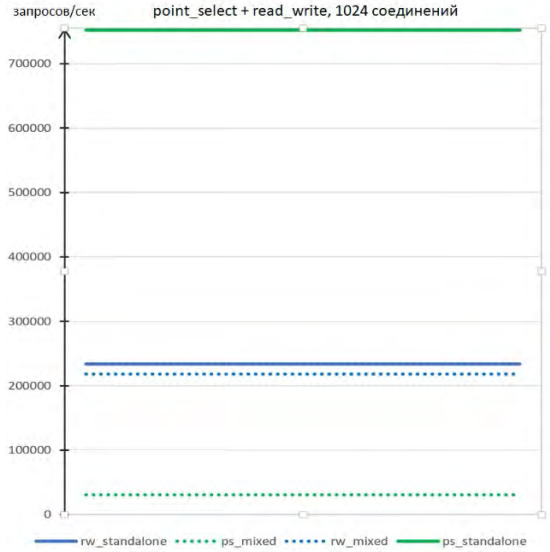


Рис. 1. Производительность трэдпула при смешанной нагрузке

Сравним результаты трех экспериментов – однородного потока *point_select_standalone* (поток простых запросов типа *select <имя_поля> from <имя_таблицы> where id=?*, где *id* – первичный ключ); однородного потока *read_write_standalone*; смешанного потока *ps+rw*. Видно, что производительность для запросов *rw* при смешанной нагрузке снизилась незначительно (7%). Причиной является существующая дисциплина обслуживания, когда запросы, являющиеся частью активной транзакции, обслуживаются с более высоким приоритетом. Поэтому их QoS почти не страдает от одновременной *ps*-нагрузки. В то же время деградация QoS для *ps*-нагрузки чрезвычайно велика (96%). Такая ситуация совершенно нетерпима, но в рамках существующей модели трэдпула исправить ее нельзя.

Решением является приоритезация запросов по их типу, т.е. выделение запросов одного типа в отдельную очередь. Тогда, управляя приоритетами очередей, мы выделим каждому типу гарантированную «полосу пропускания» и улучшим трэдпул в целом. При этом важно понимать, что же мы оптимизируем, т.е. что должны наблюдать для вывода, что реализация трэдпула как очередей с приоритетами дала улучшение? Например, общая средняя задержка уменьшилась, но средняя задержка для какого-то низкоприоритетного типа запроса возросла. Будем ли мы рассматривать это как успех

или нет? Трэдпул с приоритетами – малоизученная область. В [13] описано одно из первых технических решений, где в результате разделения OLTP и OLAP-нагрузок были получены перспективные результаты. Текущая же промышленная реализация, типовая в том числе и для MySQL [14], реализует приоритезацию из двух очередей, которая никак не решает описанную выше проблему. Трэдпул с приоритетами по типам запросов – достаточно сложная и пока еще технически неочевидная система, закономерности поведения которой необходимо предварительно изучить, прежде чем выбирать способ реализации. Здесь на помощь приходит такой испытанный инструмент как имитационное моделирование. В [3] сказано: «Все попытки создания реально применимых методик, учитывающих находящиеся в каналах и в очередях количества заявок каждого вида, заведомо обречены на неудачу в связи с непомерным разрастанием пространства состояний. Не всегда помогают и имитационные системы: GPSS World не позволяет моделировать многоканальные устройства с приоритетными прерываниями – более того, даже одноканальные с кратными прерываниями.» И хотя приоритетных прерываний у нас нет – мы рассматриваем относительные приоритеты, а не абсолютные – алгоритмическая сложность трэдпула делает эти слова справедливыми и в нашем случае. Среди современных работ следует отметить [4], где предложен интересный подход к моделированию СМО с динамическим приоритетом, реализованный в виде модели на C#, а также [2, гл. 4], где рассмотрено моделирование очередей с приоритетами в системе *AnyLogic*.

Работа имеет следующую структуру. Раздел II содержит краткий теоретический обзор классических и современных дисциплин обслуживания очередей с приоритетами. В разделе III описана имитационная модель трэдпула с приоритетами, точнее, те дополнения реализации, которые отличают ее от модели [5]. В разделе IV представлены результаты апробации модели и их обсуждение, раздел V завершает работу краткими выводами.

## Таксономия дисциплин обслуживания очередей с приоритетами

Под дисциплиной обслуживания будем понимать набор правил, по которым заявка выбирается на обслуживание из множества очередей с различными приоритетами. Мы будем рассматривать только те дисциплины, которые так или иначе применимы к трэдпулу, поэтому дисциплины с абсолютным приоритетом или иные дисциплины с прерыванием обслуживания не рассматриваются. Мы будем следовать как классическому труду [1] и более позднему обзору [9], так и современным результатам, порожденным активным развитием информационных технологий последних десятилетий.

Итак, обратимся к рис. 2. Решение о выборе заявки на обслуживание может зависеть только от ее приоритета – такие дисциплины в терминологии [1] называются *экзогенными*. С другой стороны, алгоритм выбора может зависеть и от текущего состояния СМО, например, от приоритета последней обслуженной заявки или времени ожидания заявки в очереди, что характерно для *эндогенных* дисциплин.

*Чистые приоритеты:* выбирается заявка из непустой очереди с наибольшим приоритетом. C++ STL-контейнер *priority_queue* реализует именно эту дисциплину. Основной недостаток: если очередь с более высоким приоритетом очень редко бывает пустой, заявкам с более низким приоритетом придется очень долго ждать, или даже они вообще никогда не будут выбраны.

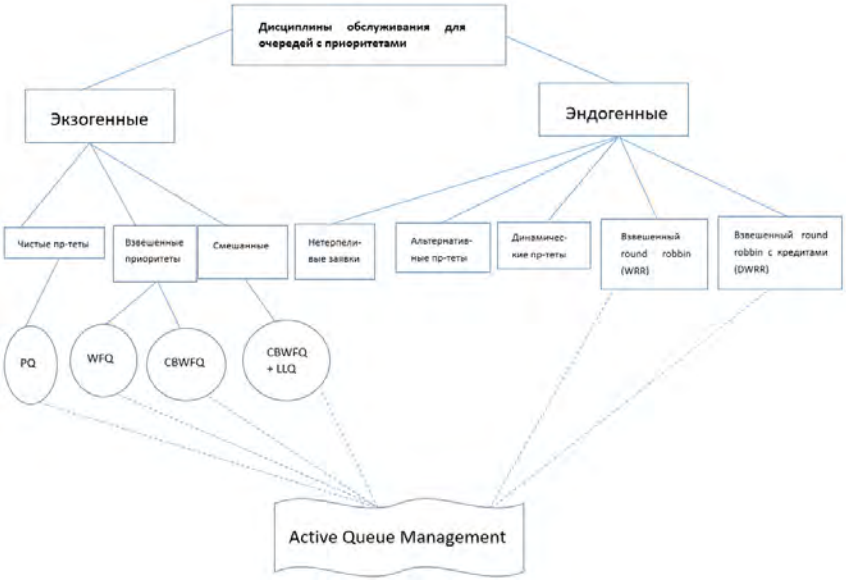


Рис. 2. Таксономия дисциплин обслуживания с приоритетами

Подробный пример имитационного моделирования такой СМО приведен в [6, гл. 10]. Наиболее известное свойство: высокий приоритет для коротких заявок повышает производительность СМО и снижает среднее число заявок, ожидающих в очереди.

*Взвешенные приоритеты:* приоритет – измеримая величина, где мера (вес) – число из [0;1], сумма весов равна 1. Вес очереди – это вероятность того, что следующая заявка будет выбрана именно из этой очереди, если она непуста. Именно такой подход был применен в [13]. *CBWFQ (Class Based WFQ)* является модификацией взвешенных приоритетов, использующейся в технологии *Active Queue Management*, возникшей в соответствии с требованиями современных телекоммуникационных систем. Обзор этой технологии можно найти в [7] и [19]. *CBWFQ+LLQ* (Low Latency Queue) – еще одна модификация, где одна из очередей становится т.н. LLQ – очередью с малой задержкой. Остальные очереди обслуживаются как CBWFQ, в то время как на уровне «LLQ – другие очереди» работают чистые приоритеты. Иными словами, остальные очереди ожидают, пока LLQ не пуста.

*Нетерпеливые заявки*: иногда обслуживание заявки должно быть завершено до окончания времени ее жизни (*deadline*), т.е. для каждой очереди назначается предельное время ожидания в ней. Если оно превышено, заявка повышает приоритет на единицу и переходит в соседнюю очередь. Это предотвращает, наподобие взвешенным приоритетам, бесконечное ожидание в низкоприоритетной очереди. Такая модель обслуживания исследована в [8].

*Чередующийся приоритет:* если сервер завершил обслуживание заявки *i*-го приоритета, следующая заявка выбирается того же приоритета, если соответствующая очередь не пуста. В противном случае выбирается заявка с наибольшим приоритетом. Заметим, что это именно эндогенная дисциплина, т.к. решение принимается на основе знания того, какого приоритета была последняя обслуженная заявка. Представим, что заявка *C1* (рис. 3) прибыла первой. За время ее обслуживания прибыли остальные заявки. На рисунке показано, в каком порядке они будут обслуживаться. Дисциплина исследована в [10].

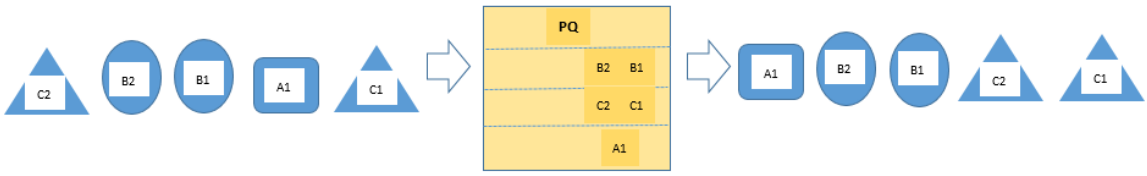


Рис. 3. Иллюстративная схема к дисциплине «чередующийся приоритет»

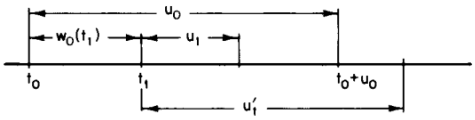


Рис. 4. Иллюстративная схема к дисциплине «динамический приоритет»

*Динамический приоритет:* дисциплина эндогенна, т. к. порядок обслуживания новой заявки по отношению к уже ожидающим определяется не только приоритетом заявок, а также тем, сколько каждая заявка уже ожидает. Пусть заявка прибывает в момент $t$ и имеет индекс срочности (*urgency number*) $u$ (можно интерпретировать как предельное время ожидания). Для каждой заявки в очереди мы вычисляем накопленное к моменту $t$ время ожидания $\widehat{W}_i(t)$ и индекс срочности $u_i$. Тогда новая заявка имеет преимущество над теми заявками $j$, для которых $(u_j - u) \geq \widehat{W}_j(t)$. Рис.4, взятый из [1], иллюстрирует это. Заявка прибыла в момент $t_0$ и имеет индекс срочности $u_0$, а другая заявка – в момент $t_1$. Если индекс срочности второй заявки равен $u_1$, она имеет преимущество над первой, если же $u_1'$ - не имеет. Таким образом, динамический приоритет преобразует несколько очередей в одну, но не FIFO очередь.

*Взвешенный round robbin:* циклически просматривается каждая очередь, но на обслуживание из нее берется не одна заявки, а количество, пропорциональное весу очереди. Таким образом, чем выше приоритет очереди, тем больше из нее берется заявок.

*Взвешенный round robbin с кредитами:* рассмотрена в [12]. Для каждой очереди назначается квант времени $Q_i$. Когда доходим до очереди $i$, выбирается количество заявок с суммарной длиной, не превышающей $Q_i$. Как это работает? Когда выборка происходит из очереди $i$, ее кредит увеличивается на размер кванта. Если кредит больше времени обслуживания первой заявки, она выбирается, и кредит уменьшается на это время. Затем кредит сравнивается с длиной следующей заявки и т.д. Как только кредита перестает хватать, алгоритм переходит к следующей очереди. Если очередь пуста, кредит для нее полагается равным нулю.

Среди программных реализаций очередей с приоритетами в открытом доступе, отметим:

– [15] – реализация чистых приоритетов с их динамическим повышением;
– [16] – реализация дисциплины CBWFQ+LLQ;
– [17] – полная реализация AQM-дисциплин применительно к управлению сетевым трафиком;
– [18] – реализация т.н. «израильской» очереди [11], где заявка выбирается из той очереди, где текущее время ожидания головного элемента максимально.
–

**Описание имитационной модели**

Перечислим особенности моделируемой системы (трэдпула), которые отличают ее от системы, описанной в [5]:

– *каждая группа потоков (Threadgroup) содержит произвольное, но одинаковое для всех групп, число очередей, каждая из которых характеризуется своим*

*приоритетом*. Запросы (заявки – в терминах СМО) выбираются из очередей в соответствии с их приоритетами и дисциплиной обслуживания. Заметим, что в текущей реализации каждая группа содержит ровно две очереди с приоритетом, основанном на том, является ли запрос частью транзакции. В новой реализации это становится просто одной из дисциплин;

– *распределение заявок по очередям (приоритетам) группе потоков*. В реальном трэдпуле приоритет запроса может вычисляться на основе многих факторов. В первую очередь, это тип запроса (упрощенно, *select/insert/update/delete*, хотя SQL-выражений существуют десятки), но также может учитываться, с какими таблицами работает этот запрос, наложены ли на эти таблицы блокировки, привилегии пользователя, отправившего запрос и др. Модель же просто генерирует потоки заявок каждого типа со своими характеристиками, которые моделью учитываются;

– *дисциплина обслуживания*. Задает порядок выбора заявок из очередей. Поиск оптимальной дисциплины и есть главное предназначение модели. Обзор возможных дисциплин был представлен в предыдущем разделе данной работы.

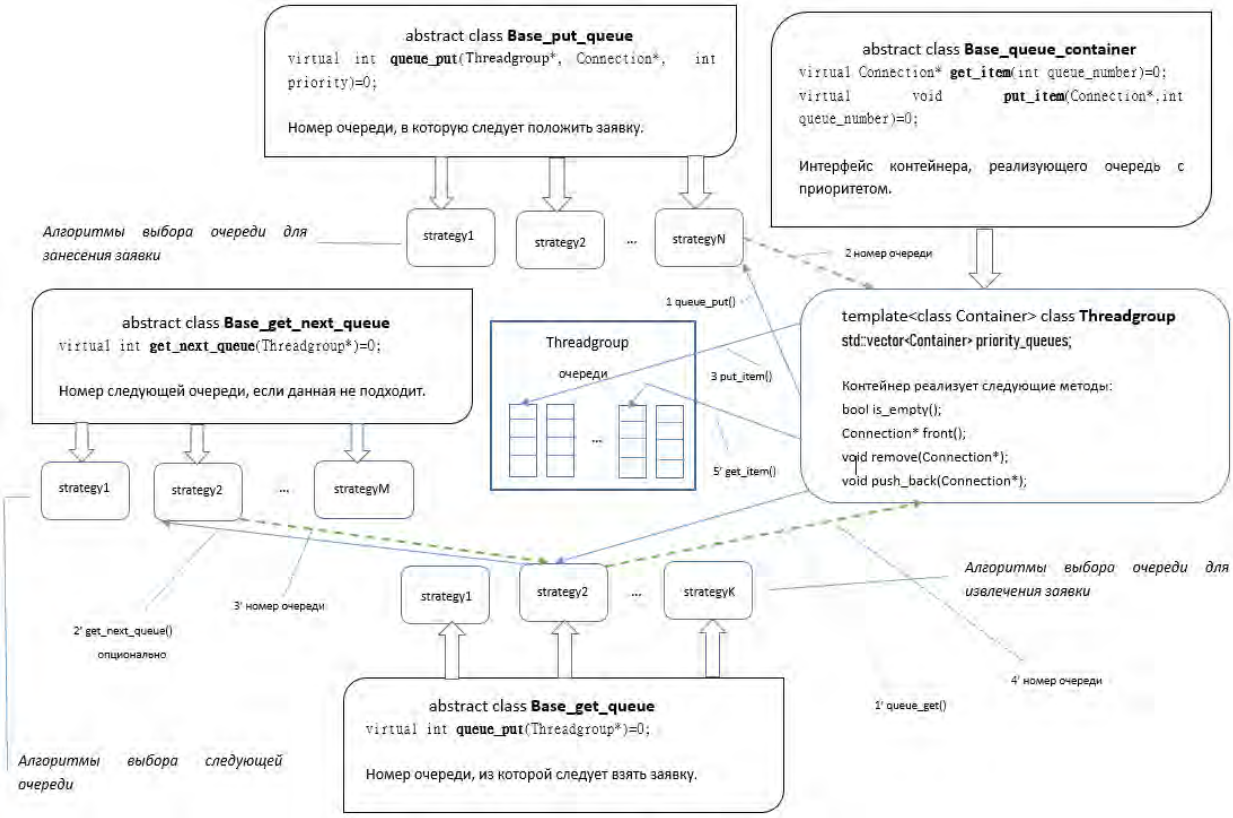


Рис. 5. Объектный дизайн имитационной модели

Прокомментируем некоторые особенности дизайна имитационной модели (рис. 5):

– объектно-ориентированная архитектура, использующая наследование, полиморфизм, шаблоны и «умные» указатели;

– произвольное количество очередей в группе потоков;

– возможность назначить любую из реализованных дисциплин следующих видов: занесение заявок в очередь; выборка заявок из очереди; выбор другой очереди, если выбранная очередь пуста;

– возможность выбора контейнера для хранения очереди при условии, что он поддерживает предопределенный интерфейс;

– по сравнению с моделью [5] в классе *Threadgroup* появляется новое свойство – массив очередей с приоритетами, а методы *Threadgroup::queue_get()* и *Threadgroup::queue_put()* становятся методами классов соответствующих дисциплин.

Например, класс *weighted_queues*, реализующий взвешенные приоритеты, работает следующим образом. Свойством этого класса является массив весов очередей, которые в сумме дают единицу. Например, если веса равны (1/9; 1/3; 5/9), метод *queue_get()* этого класса возвращает значения 1, 2 или 3 с соответствующими вероятностями. Вес очереди может быть равен и нулю. В этом случае *queue_get()* возвращает ее номер тогда и только тогда, когда все очереди с ненулевыми весами пусты. Глобальная переменная, задающая дисциплину обслуживания определяется так:

*std::unique_ptr<Base_get_queue> base_get_queue;*

Пример ее инициализации:

*base_get_queue = std::unique_ptr<Base_get_queue>(new weighted_queues());*

Класс, наследующий *Base_get_next_queue*, отвечает на вопрос: из какой очереди следует выбрать заявку, если очередь, возвращенная классом – наследником *Base_get_queue*, пуста. Экземпляр такого класса является свойством класса-наследника *Base_get_queue*. В общем случае, любой наследник *Base_get_queue* может использовать любого наследника *Base_get_next_queue*. Примеры реализаций:

– *class first_non_empty_queue* – возвращает непустую очередь с наибольшим приоритетом;

– *class non_empty_queue_with_max_weight* - возвращает непустую очередь с наибольшим весом.

Это свойство класса *Base_get_queue* объявляется так:

*std::unique_ptr<Base_get_next_queue> base_get_next_queue ;*

Пример инициализации для класса *weighted_queues*:

*base_get_next_queue = std::unique_ptr<Base_get_next_queue>(new non_empty_queues_with_max_weights(weights));*

где *weights* – свойство-вектор класса *weighted_queues*, задающее веса очередей.

## Апробация модели и результаты

Опишем результаты тестирования модели на смешанной нагрузке, т.к. цель трэдпула с приоритетами – оптимизация именно таких случаев. Сравниваются результаты двух моделей – старой и новой – для каждого вида нагрузки в отдельности и суммарной, число соединений для каждого вида нагрузки – 1024. По результатам делается вывод, как соотносятся улучшение и ухудшение производительности для обоих видов нагрузки.

Тестируется дисциплина «Взвешенные приоритеты», т.к. для двух нагрузок она характеризуется простой вариативностью – приоритетом одной из них. Для каждой рассмотренной пары было проведено 14 тестов:

1. *workload1_standalone*: результат прогона только первой нагрузки на старой модели;
2. *workload2_standalone*: результат прогона только второй нагрузки на старой модели;
3. *common_original = workload1_original + workload2_original*: обе нагрузки одновременно запускаются на старой модели.
4. *common_with_priority, workload1_with_priority, workload2_with_priority*: 11 тестов на новой модели, в которых вес одной из нагрузок меняется от 0 до 1 с шагом 0.1. Вес откладывается на оси абсцисс. Этот параметр по смыслу влияет только на три кривые на рисунках и безразличен для констант, полученных из экспериментов на

старой модели, где нет приоритетов. Например, значение веса 0.7 для заявок второго типа означает следующее:

– если обе очереди непусты, мы выбираем заявку из второй очереди с вероятностью 0.7 и из первой очереди с вероятностью 0.3;

– если непуста только одна очередь, мы выбираем заявку из нее с вероятностью 1.

Исходные данные для моделирования были взяты из запусков на сервере MySQL 8.0.27 для следующей конфигурации:

– *Таблиц*: 40.

– *Размер таблицы*: 20 миллионов записей.

– *Размер базы данных*: 192 Gb.

– *Размер буфера (buffer pool)*: 40 Gb.

– *Бенчмарк*: sysbench, продолжительность запуска 200 секунд.

IV.1. point_select + read_write (рис. 6.)

Прежде всего, вернемся к тому, с чего начали – к рис.1, дополненному результатами прогонов модели с приоритетами. Рассмотрим вес *rw*-нагрузки, равный 0.8 и вес *ps*-нагрузки, равный 0.2 соответственно. Это означает, что мы даем, хотя и небольшие, но какие-то права *ps*-запросам в сравнении со старым трэдпулом. Однако, даже такой небольшой вес, как видно, дал следующий результат: производительность *ps*-нагрузки выросла на 67% (!) (*ps_with_priority* и *ps_original*), с 30800 до 51300 запросов в секунду. Суммарная производительность при этом не уменьшилась, а для нагрузки *rw* уменьшилась, но незначительно - на 9% (*rw_with_priority* и *rw_original*), с 218K до 198K. Таким образом, назначением приоритета мы можем управлять желаемым балансом.

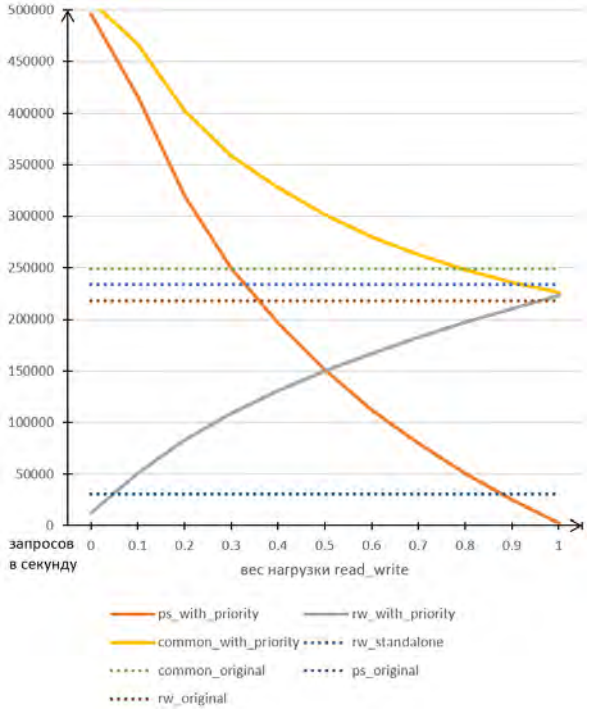


Рис. 6. Результаты для смешанной нагрузки *point_select+read_write*

IV.2. read_only + write_only (рис. 7.)

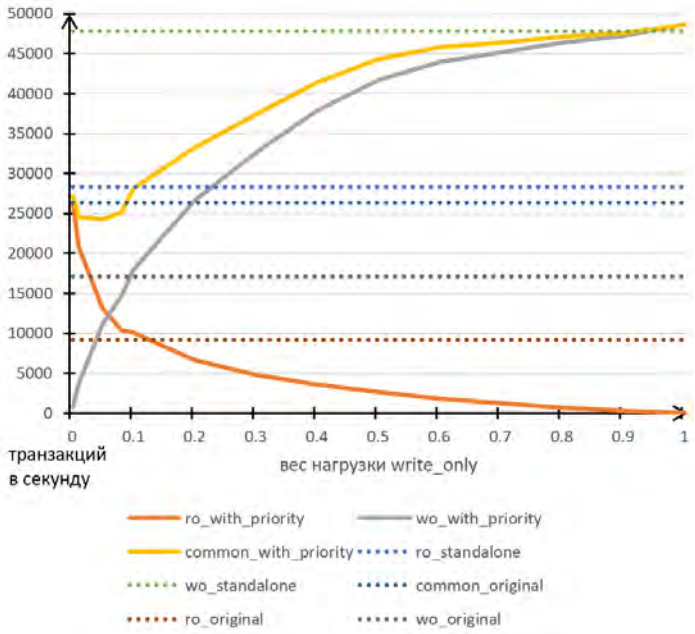


Рис. 7. Результаты для смешанной нагрузки *read_only+write_only*

Для нагрузки *read_only* характерны более сложные запросы, чем для *point_select* – с аггрегирующими функциями, ключевыми словами *between…and, order by, distinct*. *ro*-нагрузка сильно деградирует с ростом приоритета более тяжелой *wo*. Тем не менее, при весе *wo* около 0.1 *ro* лучше на 10%, суммарная производительность лучше на 10%, и даже *wo* лучше на 5%. Таким образом, 0.1–0.11 – это тот оптимальный вес, для которого трэдпул с приоритетами имеет неоспоримое преимущество.

IV.3. point_select + read_only (рис. 8)

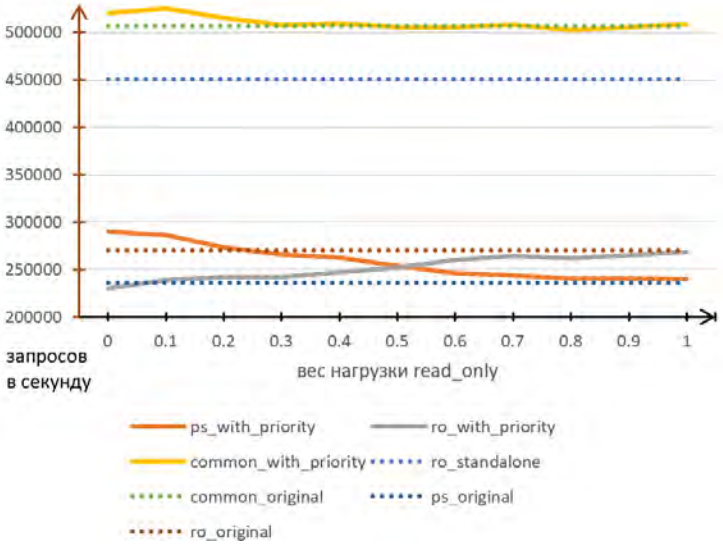


Рис. 8. Результаты для смешанной нагрузки *point_select+read_only*

Это пример случая, когда трэдпул с приоритетами не дает улучшения. Создается впечатление, что тест “original” подобен тесту на новом трэдпуле с приоритетом 1 для *ro*. В самом деле, мы видим, что *ps_with_priority* стремится к *ps_original*, *ro_with_priority* - к *ro_original* и *common_with_priority* - к *common_original*. Новый трэдпул опережает старый только для *ro*-весов 0 или 0.1, но очень незначительно (около 3%).

**Выводы**

По результатам апробации модели и сравнения результатов ее работы с поведением реального трэдпула на сервере СУБД можно сделать следующие выводы:

1) приоритезация очередей в трэдпуле по типу нагрузки существенным образом влияет на качество его работы и позволяет ставить новые задачи оптимизации;

2) главной из этих задач является выбор дисциплины извлечения заявок из очереди и численных параметров этой дисциплины. Ценность имитационной модели заключается в возможности быстрого сравнения альтернатив с учетом характеристик конкретного сервера СУБД и нахождения решения, являющегося если и не оптимальным, то достаточно хорошим приближением к нему;

3) использование модели позволяет установить способ устранения узких мест в работе сервера СУБД, когда часть запросов в течение длительного времени не получают обслуживания, ухудшая тем самым показатели Quality-of-Service;

4) в то же время модель позволяет выявить типы нагрузок, где очереди с приоритетами не дают преимущества.

**Литература**

1. Джейсуол Н. Очереди с приоритетами. М.: Мир, 1973. 275 с.

2. Осипов Г.С. Математическое и имитационное моделирование систем массового обслуживания. Академия естествознания, 2017. URL: https://monographies.ru/docs/2017/06/file_5937aed7c4f3f.pdf

3. Рыжиков Ю.И. Имитационное моделирование в обосновании методик расчета многоканальных приоритетных систем // Имитационное моделирование. Теория и практика: сборник докладов первой всероссийской научно-практической конференции ИММОД-2003. СПб.: ЦНИИТС. 2003. Т.1. С. 161–165.

4. Савинов Ю.Г., Подгорнов М.Д. Математическая модель многоканальной СМО с динамическим приоритетом // Ученые записки УлГУ. Серия: Математика и информационные технологии. 2022. Вып. 1. С. 56–64.

5. Труб И.И., Копытов А.А., Строганов А.А. Имитационная модель пула потоков для сервера баз данных // Имитационное моделирование. Теория и практика: десятая Всеросс. научно-практ. конференция (ИММОД-2021): труды конференции. СПб: ЦТСС, 2021. С. 412–420.

6. Труб И. Объектно-ориентированное моделирование на C++. –Питер, 2005. 416 с.

7. Adams R. Active Queue Management: A Survey. IEEE Communication Surveys & Tutorials, vol. 15, no. 3, third quarter, 2013.

8. Ahmadi M. et al. Processor Sharing Queues With Impatient Customers and State-Dependent Rates. IEEE // ACM Transactions on Networking, vol.29, no.6, December, 2021.

9. Gail H.R., Hantler S.L., Taylor B. Analysis of a non-preemptive priority multiserver queue // Advances in applied probability, 1988, vol. 20. pp. 852–871.

10. Groenevelt R., Altman E. Analysis of Alternating-Priority Queueing Models // Queueing Systems, 51, 2005, pp.199–247.

11. Perel N., Yechiali U. The Israeli Queue with Priorities // Stochastic Models, no. 29, 2013, pp.353–379.

12. Shreedhar M., Varghese G. Efficient Fair Queueing using Deficit Round Robbin // Report number: WUCS-94-17 (1994). URL: https://openscholarship.wustl.edu/cse_research/339

13. Sinvani E. Workload Prioritization: Running OLTP and OLAP Traffic on the Same Superhighway. URL: https://www.scylladb.com/2019/05/23/workload-prioritization-running-oltp-and-olap-traffic-on-the-same-superhighway

14. https://alibabacloud.com/help/en/polardb-for-mysql/latest/thread-pool (April, 2023)
15. https://github.com/addisalemtafere/E-banking
16. https://github.com/CumulusNetworks/iproute2/blob/master/man/man8/tc-mqprio.8
17. https://github.com/idosch/mlxsw-1/wiki/Queues_Management
18. https://github.com/Sealights/Israeli-queue
19. https://intronetworks.cs.luc.edu/current/html/dynamicsB.html#active-queue-management