Санкт-Петербургский государственный университет

Математическое обеспечение и администрирование информационных систем
Системное программирование

Толстопятов Всеволод Андреевич

Повышение производительности алгоритмов кэширования больших объёмов данных

Бакалаврская работа

Научный руководитель:
ст. преп. Сартасов С. Ю.

Рецензент:
Руководитель группы разработки Киракозов А. Х.

Санкт-Петербург
2016

SAINT-PETERSBURG STATE UNIVERSITY

Information Systems Administration and Mathematical Support
Software Engineering

Tolstopytaov Vsevolod

Efficiency increase of large data volume caching

Graduation Thesis

Scientific supervisor:
Senior assistant professor Stanislav Sartasov

Reviewer:
Team lead Alexander Kirakozov

Saint-Petersburg
2016

Оглавление

Введение

Кэширование — один из самых базовых и эффективных методов значительного улучшения производительности современных систем и приложений в различных областях. Улучшение достигается за счет хранения небольшой части данных в памяти приложения, которая быстрее или ближе к приложению, чем первичный источник данных, в ситуации, когда весь домен данных не помещается в такую память. Причина, по которой такой подход работает заключается в том, что во многих областях для данных, с которыми работает система, выполняется *принцип локальности*. Формально, локальность — характеристика частоты доступа к данным, которая принимает во внимание тот факт, что в реальном мире распределение вероятности сильно перекошено [3], что означает гораздо большую вероятность обращения к объектам из небольшого подмножества данных. В большинстве случаев паттерн обращения к данным и распределение вероятности изменяется с течением времени, что известно как *временная локальность*, что не позволяет использовать только частоту встречаемости элемента как основной оцениваемый параметр при вытеснении.

С учетом того, что размер кэша в большинстве случаев сильно ограничен, основная сложность алгоритмов кэширования заключается в выборе элементов, которые будут храниться. В частности, когда память, зарезервированная под кэш закончилась, нужно принять решение, какой элемент нужно *вытеснить*. Очевидно, что решение о вытеснении должно приниматься эффективно, чтоб избежать ситуаций, когда время принятия решения и поддержания внутренних структур данных кэша (известных также как *мета-данные*) перекрывает преимущества использования кэша в целом. В большинстве алгоритмов кэширования временная сложность манипуляции с мета-данными прямо пропорциональна количеству элементов в кэше, более того, операции чтения всегда требуют записи, что не является приемлемым для ситуаций, когда размер кэша очень большой, а скорость записи в быстрый источник данных невелика. Более того, существующие решения для кэширования (как

локального, так и распределенного) либо не утилизируют диск, даже в условиях, когда чтение с диска быстрее, чем загрузка данных из основного хранилища, либо используют его неэффективно: только в качестве дополнительного хранилища, куда вытесняются холодные данные, при это в расчет не берется наличие у нижестоящей операционной системы своей собственной подсистемы кэширования страниц (известной также как page cache), которая пытается кэшировать доступ к страницам диска, в итоге интерферируя с основным алгоритмом кэширования и сильно ухудшая как результат, так и расход оперативной памяти. Также трудностью является то, что при использовании диска резко возрастает количество мета-данных, которые необходимо поддерживать, что негативно влияет как на размер потребляемой приложением памяти, так и на подсистему сборки мусора в языках с автоматическим управлением памяти.

1. Постановка задачи

Цель работы — исследовать производительность существующих алгоритмов кэширования (как использующих частотные, так и временные эвристики вытеснения) на приближенных к реальности данных и реализовать наиболее подходящий алгоритм кэширования для дискового массива поверх высокопроизводительной библиотеки one-nio [1], разработанной в компании Одноклассники. Реализация должна быть написана на языке Java, использовать разделяемую нативную память, которая недоступна для сборки мусора, а накладные расходы на метаданные не должны превышать 1 гигабайта на 5 терабайт данных, при условии, что одно значение занимает 256 килобайт. В качестве тестовых данных для сравнения производительности будут использоваться запросы, распределенные по hotspot-распределению.

Данную работу можно разбить на несколько подзадач:

- Исследовать и отобрать для сравнения существующие алгоритмы кэширования. Необходимо проанализировать как теоретические работы, так и существующие практические реализации, такие как популярные библиотеки и существующие высокопроизводительные решения в области распределенного кэширования (*In-Memory Data Grid*).

- Протестировать отобранные алгоритмы на реальных данных с различными паттернами доступа к данным и проанализировать эффективность алгоритмов вытеснения.

- На основе полученных данных реализовать эффективный алгоритм кэширования поверх массива дисков в рамках высокопроизводительной библиотеки с открытым исходным кодом one-nio [1].

- Улучшить производительность полученного решения в условиях жесткого конкурентного доступа, используя как различные инженерные подходы, такие как асинхронный интерфейс операционной

системы для доступа к дисковым данным, так и различные технихи параллельного программирования.

2. Обзор

2.1. Алгоритмы вытеснения

В данной секции будут рассмотрены наиболее распространенные политики вытеснения, а также рассмотрены их сильные и слабые стороны, применимость и сложность реализации.

2.1.1. Least recently used

Наиболее популярная политика вытеснения, которая в качестве кандидата выбирает элемент, обращение к которому было наиболее давно. Несмотря на простоту реализации (связный список элементов, при каждом обращении к элементу он передвигается в голову списка) и хороший "уровень попаданий" (процент случаев, в которых данные были взяты из кэша, а не основного хранилища), данный алгоритм обладает существенным недостаткам: неустойчивостью к всплескам в распределении запросов, известных как "полное сканирование", когда вместо популярных элементов на протяжении небольшого промежутка времени обращения идут к новым элементам, что делает данный алгоритм неприемлемым для большинства практических задач.

2.1.2. Least frequently used

Политка, в которой вытесняется элемент, к которому было меньше всего обращений. Является крайне неоптимальной как с алгоритмической точки зрения, так как нахождение такого элемента требует не менее $O(\log n)$ операций (остальные политики имеют ассимптотику $O(1)$), так и с практической, так как данная политика не берет в расчет временную локальность: ранее популярный элемент может оставаться в кэше неопределенно долго после прекращения обращений к нему.

2.1.3. Adaptive replacement cache

Адаптивный алгоритм, разработанный и опубликованный исследовательской лабораторией IBM [4]. Алгоритм пытается балансировать между LRU и LFU, используя три очереди: элементы, доступ к которым был однажды, элементы, доступ к которым был осуществлен несколько раз и *резидентные* элементы, которые когда-то были в кэше, но после были вытеснены, динамически определяя размер каждой из них в зависимости от паттерна нагрузки и эффективности предыдущей настройки. Данный алгоритм показывает очень высокий процент попаданий на различных распределениях [4], но требует удвоенное количество памяти для поддержания резидентных записей, также данный алгоритм официально запатентован и не может быть использован без соглашения с IBM, и поэтому не используется большинством продуктов.

2.1.4. Low inter-reference recency set

Разработанный в 2002 году алгоритм, использующий в качестве динамической метрики эвристическую оценку *переиспользования* элемента (reuse distance), которая оценивает количество элементов, доступ к которым был совершен между двумя последовательными доступами к элементу. Для этого политика поддерживает очереди LIR (low inter-reference recency) и HIR (high inter-reference recency), при этом во второй очереди элемент может быть *резидентным*. Данный алгоритм сложен в реализации и достигает максимальной эффективности только когда размер метаданных втрое превышает размер максимально возможных хранимых значений [6].

2.2. Существующие библиотеки кэширования

2.2.1. Infinispan

Infinispan является платформой для распределенных вычислений и кэширования (In-Memory Data Grid), используя для кэширования алго-

ритм LIRS. Мета-данные кэша и основного хранилища (хэш-таблицы) хранятся вместе, мета-данные являются небезопасными и поэтому для повышения параллельности используется техника *сегментирования*, когда хэш-таблица представляет из себя массив хэш-таблиц, каждая из которых защищена блокировкой и хранит свои собственные мета-данные, то есть фактически кэш состоит из нескольких несвязанных кэшей. Для повышения параллельности чтений используется алгоритм отложенного обновления, описанный в [8], более того, алгоритм обновления является lock-free, то есть не требует блокировки и может исполняться не блокируя обновления от других потоков в ту же хэш-таблицу. Дисковое хранилище не используется.

2.2.2. Redis

Redis является NoSQL базой данных, которая в качестве алгоритма кэширования предоставляет approximate LRU. Когда приходит время выбора кандидата на вытеснение, Redis осуществляет *сэмплирование*: выбирается 5 случайных элементов и для вытеснения используется наихудший по времени обращения. Такой подход хоть и является легким в реализации, но на практике дает очень плохую точность для большого количества ключей при фиксированном размере сэмплирования. Начиная с версии 3.0 Redis использует *кэширующее сэмплирование*, когда после каждой итерации сэмплирования в массив длины 16 складываются наиболее подходящие кандидаты, а потом из них выбирается подходящий. Такой подход улучшает качество алгоритма и приближает его к идеальному LRU, но только на ограниченном размере данных[1]. При этом массив не синхронизирован с основным хранилищем и является *согласованным в конечном счете* и обновляется только при вытеснении. Для сохранения данных на диск Redis использует системный вызов *fork*, после которого записывает все содержимое памяти на диск как есть, без каких либо изменений.

[1]http://redis.io/topics/lru-cache

2.2.3. Guava

Guava является одной из самых популярных библиотек для языка Java. Предоставляет алгоритм кэширования LRU, используя технику сегментирования. Чтения являются отложенными (до восьми чтений), процесс обновления метаданных является блокирующим. В качестве дополнительных преимуществ Guava предоставляет возможности асинхронного обновления, вытеснения по времени с помощью использования дополнительной очереди (очередь на истечение (expire queue)) и хранению значений и ключей в виде soft и weak references (специфичная особенность платформы Java).

2.2.4. Hazelcast

Hazelcast является платформой для распределенных вычислений и кэширования (In-Memory Data Grid), используя в качестве алгоритма кэширования политику LRU. Однако на практике заявленные требования не выполняются, так как внутри реализован алгоритм *сэмплирующего* LRU на 20 итерациях, при этом сначала происходит 20 итераций, потом из набора удаляются устаревшие по времени элементы, и только потом происходит вытеснение наихудшего из оставшихся, что делает данный подход крайней неоптимальным. Hazelcast позволяет использовать диск в качестве вторичного хранилища при вытеснении записи, чтоб при последующем запросе совершить загрузку не из основного источника, а прочитать с диска. Для работы с диском используется обыкновенные буферизованные операции ввода-вывода, что является неоптимальным с точки зрения утилизации оперативной памяти из-за интерференции с кэшом операционной системы и из-за дополнительного потребления памяти на буферизацию.

3. Описание решения

3.1. Алгоритм кэширования

Прежде чем выбирать алгоритм кэширования, стоит заметить, что наиболее устойчивые политики кэширования основаны на тех же идеях, что и современные сборщики мусора, которые разделяют существующие в исполняемой среде объекты на несколько (два или более) поколений, которые обычно называют *молодым (eden)* и *старшим (tenuring)*. Объекты в молодом поколении существуют в программе недолго и быстро становятся недостижимыми, тогда как объекты из старшего поколения используются приложением долго и редко становятся недостижимыми. Для определения поколения с каждым объектом ассоциируется счетчик *эпохи*: количество минорных сборок мусора, которые пережил объект. Когда это значение превышает некоторую фиксированную величину, объект становится частью старшего поколения. Данный алгоритм используется как в сборщиках мусора языка Java (Parallel scavenge, Concurrent mark and sweep, Garbage first), так и в платформе .NET (client GC, server GC).
Можно считать, что LIRS, ARC и Segmented LRU используют несколько очередей как раз в качестве разделителя поколений, а эвристики перемещения – счетчиком объекта. Представленный алгоритм будет совмещением идеи сборки мусора, а также идеи об условном продвижении, взятой из политики W-TinyLFU [2].

3.1.1. Фильтр Блума с подсчетом

Для алгоритма кэширования, который берет в учет частоту элемента критически важно предоставить механизм для "забывания" предыдущих результатов, чтоб избавиться от проблемы, присущей алгоритму LFU, когда ранее популярные элементы считаются популярными до тех пор, пока не будут вытеснены еще более популярными элементами. Для решения той проблемы необходим механизм затухания, который и будет описан в данной секции.

Затухающий фильтр Блума – это фильтр Блума, в котором в каждом слове вектора бит присутствия заменен на число, а при добавлении элемента соответствующие хэш-функциям ячейки вектора увеличивают свое значение. Посчитать частоту элемента можно взяз минимум из всех значений соответствующих хэш-функций. *Ограниченный фильтра Блума с подсчетом* является улучшением предыдущей структуры данных, которая ограничивает сверху максимальное значение, которое может принимать значение вектора фильтра. Такой подход позволяет избавиться от лишних инкрементов, а также улучшить точность для элементов с большой частотой, так как менее вероятно, что их значения будут преувеличины большим количеством редких элементов[5]. Схема затухающего фильтра изображена на рисунке 1.

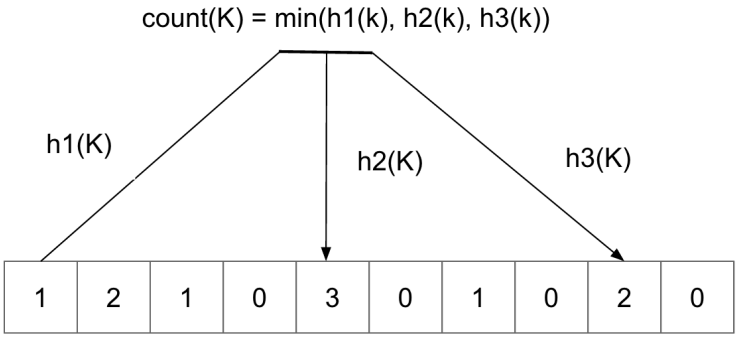

Рис. 1: Схема политики вытеснения с условным продвижением

3.1.2. Затухание

Операция затухания является одной из самых неочевидных как с точки зрения эффективной реализации, так и с точки зрения точности. Большинство подходов, описанных и исследованных в [7] требуют поддерживать несколько (m) фильтров Блума с одинаковой информацией, чтоб реализовать алгоритм *скользящего окна*. Такой подход повышает расходование памяти в m раз, а также увеличивает в более чем m раз константу для операций *add* и *count*, так как нужно не только обновлять

m фильтров, но и платить за непредсказуемый доступ к памяти, что негативно сказывается на подсистеме памяти процессора, замусоривая кэш процессора и игнорируя возможность подгружать память заранее при известном паттерне доступа (*prefetch*).
Подход, описанный в [2] является эффективным не только с точки зрения реализации и паттерна обращения к памяти, но и с точки зрения потребляемой памяти и точности при ограниченной памяти.
Вместо поддержания нескольких фильтров, с изначальным фильтром Блума связывается один счетчик и его значение увеличивается на единицу каждым вызовом операции *add*, которая успешно увеличила хотя бы одну ячейку вектора. Когда счетчик достигает некоторой константы W, значение каждой ячейки вектора и счетчика уменьшается вдвое. Такой подход не требует дополнительной памяти, прост в реализации, а его свойства (предсказуемый доступ к памяти и операция деления на два) оптимизируется современными процессорами. Корректность и точность такой операции доказана и описана в [2]. В рамках данной работы было выяснено, что наиболее оптимальная точность достигается при W, превышающим максимальный размер кэша в 8-10 раз.

3.1.3. Вытеснение при условном продвижении

Основное отличие политик кэширования от алгоритмов сборки мусора заключается в алгоритме удаления элементов из поколения. Так как для алгоритма кэширования отсутствует понятие достижимости, остро встает вопрос о том, какой именно элемент вытеснять, когда оба поколения оказались заполненными.
Используя затухающий фильтр Блума с подсчетом была реализована следующая схема: сначала элемент добавляется в молодое поколение, при этом вытесненный из молодого поколения элемент сначала не удаляется, а происходит попытка продвижения в старшее поколение. Если обращения к кандидату на вытеснение из старшего поколения случались чаще, то изначальный элемент удаляется, иначе "молодой" элемент продвигается в старшее поколения, а кандидат оттуда удаляется. В качестве фильтра для определения частоты встречаемости используется

затухающий фильтр Блума с подсчетом, который может гарантировать высокую точность, так как максимальное количество ключей в фильтре заранее известно, следовательно можно оптимально выбрать количество хэш-функций и размер. Такой подход позволяет обеспечить высокую точность для распределений, в которых случаются выбросы из последовательных "непопулярных" элементов, что гарантирует хорошую временную локальность. Также данный алгоритм не отбрасывает непопулярные элементы, а на протяжении небольшого промежутка времени хранит их в молодом поколении, что позволяет повысить точность для таких выбросов, не подвергая при этом вымыванию устоявшиеся элементы. В качестве политики для молодого поколения был выбрал алгоритм LRU, как наиболее простой и показывающий хорошее качество для небольшого молодого поколения с низко-частотным паттерном доступа. Для старшего поколения был выбрал сегментированный (Segmented LRU), чтоб "разделить" старшее поколения на сегмент апробации и основной сегмент, чтоб защищаться от серий запросов, который являются достаточно популярными, чтоб пройти молодое поколение, но недостаточно популярны, чтоб вымыть более старые записи. Конечная схема алгоритма представлена на рисунке 2.

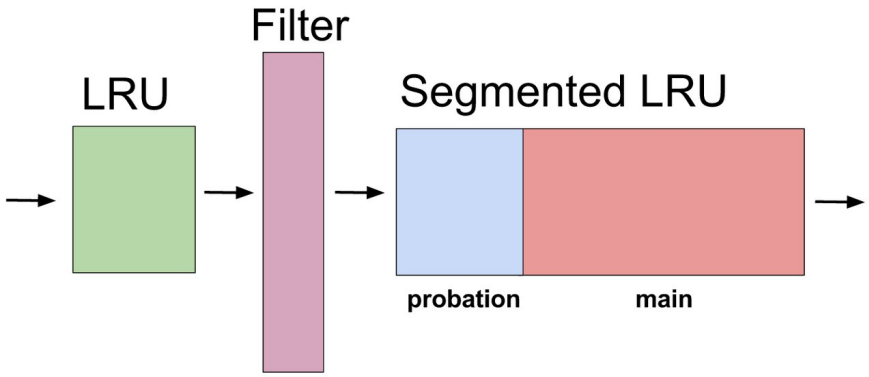

Рис. 2: Схема политики вытеснения с условным продвижением

3.2. Реализация

3.2.1. Direct IO

В качестве механизма обхода кэша страниц файлов операционной системы было решено использовать флаг O_DIRECT системного вызова *open*. Прямая работа с устройством ввода-вывода накладывает следующие ограничения: все операции записи и чтения должны оперировать блоками, размер которых кратен размеру блока файловой системы(обычно 4 или 8 килобайт), операции должы быть выровнены на размер страницы операционной памяти (4 килобайта, если не включены большие страницы), а также не пересекать частично границы блока файловой системы. Данные ограничения являются существенными как для управления памятью, так и для многопоточного доступа, однако в силу того, что в поставленной задаче размер одной записи составляет 256 килобайт, всеми этими ограничениями можно пренебречь, так как они автоматически выполняются: 256 килобайт всегда выровнены и на размер страницы, и на размер блока. Так как язык Java не поддерживает работу с системными вызовами, использовалась библиотека на основе API для системных вызовов (JNA[2]), которая автоматически проверяет корректность всех адресов при дисковых операциях и вызывает метод open напрямую.

3.2.2. Управление памятью

В качестве инструмента для работы с нативной памятью был выбран специфичный для виртуальной машины hotspot sun.misc.Unsafe API, так как в отличие от публичного стандартизованного API он предоставляет гораздо больше возможностей и не накладывает никаких дополнительных расходов на проверки безопасности, перекладывая эту обязанность на программиста. Так как все структуры данных оперируют блоками памяти одинакового размера, то вместо использования системной функции malloc() был написан lock-free аллокатор блоков

[2]https://github.com/java-native-access/jna

одинакового размера: это дает существенный прирост производительности благодаря отсутствию блокировок, позволяет экономить память, так как в заголовке выделенной памяти теперь не нужно хранить ее размер, а ABA-проблема решается tagged-указателями, где в качестве хранилища версии используются первые 16 бит адреса, так как разрядность шины памяти составляет 48 бит, а не 64. Для организации персистентности между перезапусками приложения была использована разделяемая (shared) память, которая благодаря устройству операционной системы Linux может интерпретироваться как файл (/dev/shm), а такой файл может быть отображен в память с помощью системного вызова mmap, что и было сделано: весь механизм управления нативной памятью работает с сырыми адресами, не заботясь о дополнительных структурах данных, а операционная система гарантирует, что эта память не пропадет после перезапуска приложения.

3.2.3. Write-ahead logging

Обновление мета-данных кэша на каждую операцию чтения является очень дорогой операцией в многопоточноей среде, более того, одни и те же потоки вероятнее всего будут обновлять один и тот же участок памяти, что приведет к нерациональному использованию ресурсов. Для решения этой проблемы был использован подход из статьи " BP-Wrapper: A System Framework Making Any Replacement Algorithms (Almost) Lock Contention Free" [8], которая написана на основне реализации кэша запросов в базе данных PostgreSQL. Основная идея заключается в поддержании истории обращения к кэшу вместо немедленного применения изменений к метаданным, изменения применяются отложено, после достижения определенного количества обращений, (8, 16 или 32) одной операцией. Данный подход позволяет не только амортизировать стоимость обновления метаданных и взятия блокировки, но и позволяет сделать метод чтения данных из кэша *почти* неблокирующим, что положительно сказывается на производительности чтения. Для добавления записи в историю (буфер) был использован wait-free алгоритм ограниченного кольцового буфера с помощью инструкции lock xadd.

Для повышения линейной масштабируемости было решено терять события чтения, если какой-то из потоков начал применение истории событий к метаданным. Таким образом в конечной реализации операции чтения являются *почти* неблокирующими и показывают высокую производительность чтений практически не жертвуя корректностью.
Схематиечски данный алгоритм описан в листинге 1. Для hotspot-распределен
с параметрами 80/20 данный подход показывает не более 1% ухудшения количества попаданий в кэш, увеличивая общую пропускную способность более чем на 20%.

Listing 1: Операция чтения

```
function read(key)
    result = diskStorage.get(key)
    historyQueue.tryAdd(key)
    if (historyQueue.isFull() && cacheLock.tryLock()) {
        applyHistoricalChanges()
    }
    return result
```

3.2.4. Структуры данных

Основная сложность реализации библиотеки кэширования заключается в необходимости обновления метаданных на каждую операцию чтения или записи. В многопоточном окружении это приводит к тому, что необходмо поддерживать два уровня блокировок: блокировки основного хранилища и блокировки мета-данных. Так как на уровне реализации структура данных хранилища не связана со структурой хранения кэша, то получается, что на одну операцию чтения обычно необходимо взять как минимум несколько блокировок, а из-за несвязности начинает играть роль порядок, в котором эти блокировки берутся; при использовании lock-free техник остро встают такие проблемы как ABA и safe memory reclamation. В качестве решения данной проблемой было решено поддерживать две структуры данных отдельно,

что делает их *согласованными в конечном счете (eventually consistent)*. При данном подходе повышается потребление памяти из-за накладных расходов на поддержание двух структур данных, а также появляются временные промежутки, когда мета-данные кэша и основного хранилища данных рассинхронизованы, то есть возможны ситуации, когда ключ есть на диске, но нету в кэше. Такие ситуации легко обходятся дополнительными проверками и логикой для обхода таких ситуаций. Несмотря на описанные недостатки, данный подход является оптимальным, так как теперь блокировки становятся несвязанными, а структуры данных можно оптимизировать под их паттерн использования.

В качестве основного хранилища дисковых данных была реализована хэш-таблица в разделяемой памяти с разрешением коллизий методом цепочек на сжатых связных списках (unrolled linked-lists) с блокировками на уровне цепочек (fine-grained): вместо поддержания связного списка из записей, элементы связного списка сворачиваются в несколько массивов, по которым осуществляется линейный поиск. Такой массив хранит ключ записи и указатель на адрес диска, по которому можно считать значение. Плюс в использвании сжатых списков заключается в отсутствии необходимости добавлять к каждой записи указатель на следующий элемент (8 байт), а с точки зрения производительности такой подход является более привлекательным: теперь все доступы в память являются предсказуемыми и в рамках одной кэш-линии процессора. Уровень загруженности (load-factor) 15 был выбран для уменьшения количества возможных различных блокировок на одну таблицу и исходя из размера двух кэш-линий на современных процессорах, а также для уменьшения количества указателей на "следующий" элемент. Адрес на диске хранится в 4-байтном целом типе в виде *сжатого* указателя: вместо настоящего адреса хранится лишь смещение для 256 килобайт, что позволяет уменьшить количество потребляемой памяти и адресовывать до 500 терабайт значений.

Таким образом для 20 миллионов ключей (5 ТБ данных) будет выделено 1.2 миллиона сжатых списков, то есть конечный размер будет равен 1200000 * (16 * (sizeof(pointer) + sizeof(disk_pointer)) + sizeof(list_overhead))

= 1200000 * (16 * 12 + 8)) = 240 megabytes per 5 terabytes of values.

Для фильтра Блума было решено использовать 4 бита на один счетчик (максимальная частота встречаемости ключа – 16), что приводит к расходу памяти 10 мегабайт на 5 терабайт значений.
Для мета-данных кэша была реализована связная хэш-таблица в разделямой памяти с разрешением коллизий методом цепочек, в которой каждая запись хранит не только значение, но и указатели на следующий и предыдущий элементы в порядке добавления или обращения. Основное отличие такой структуры данных от классической хэш-таблицы в нативной памяти состоит в возвращении не значения, а адреса значения, чтоб избавиться от дополнительных копирований памяти. Такой подход является безопасным, потому что благодаря амортизированному обновлению данных в один момент времени только один поток будет работать с данной таблицей. Так как порядок обращений никак не связан с группировкой по корзинам хэштаблицы, то сжатие списков не предоставляется возможным, так как при обращении надо точно знать следующие и предыдущие элементы без дополнительного поиска.
Для LRU и Segmented LRU потребление памяти является одинаковым, так как для Segmented LRU факт принадлежности элемента в очереди может быть закодирван в последних битах адреса, указывающего на запись, потому что адреса всех записей выровнены как минимум на восемь. Таким образом результирующий размер структуры составляет 20000000 * (sizeof(entry_overhead) + sizeof(key) + sizeof(two pointers)) = 2000000 * 32 = 640 megabytes per 5 terabytes of values.

3.3. Апробация

Апробация данного алгоритма была произведена на различных распределениях с использованием доработанного симулятора различных алгоритмов кэширования от автора кэша библиотеки Google Guava.
Основные сравнения проводились на так называемом *hotspot*-распределении: когда x% обращений приходятся на y% данных. При апробации изменя-

лись только параметры распределения, но не отношение размера кэша к количеству различных значений запросов, так как зависимость между этими параметрами линейная, отношение всегда составляло 1 к 2. Ниже приведена точность различных алгоритмов для данного распределения (LRU, LFU, ARC, LIRS, описанный выше алгоритм и идеальный алгоритм, который вытесняет страницу, обращение к которой не будет происходить наиболее долго).

Алгоритм	Точность (70/20)	Точность (75/25)	Точность (80/35)
LRU	51.62%	52.60%	49.44%
FIFO	47.97%	47.94%	44.35%
ARC	56.67%	59.79%	55.98%
LIRS	60.14%	63.25%	59.27%
Generational	68.16%	72.26%	66.23%
Ideal	77.65%	77.76%	74.99%

Таблица 1: Точность различных алгоритмов кэширования

Как следует из результатов в таблице 1, предоставленный алгоритм в среднем показывает лучшую точность на данных распределениях.

Заключение

В ходе данной работы были достигнуты следующие результаты:

- Исследованы и проанализированы теоретические результаты в области алгоритмов кэширования, разработана адаптация одного из них
- Сравнена эффективность исследованных алгоритмов на различных данных и выбрана наиболее подходящая.
- Проанализированы существующие решения в области кэширования: как распределенные in-memory data grid'ы, так и отдельные библиотеки, рассмотрены их достоинства и недостатки.
- Разработано решение для высокопроизводительного дискового кэширования в рамках библиотеки one-nio с использованием прямого доступа на диск в обход кэша операционной системы, амортизированной стоимостью обновления метаданных и линейным уровнем параллелизма и переключаемыми алгоритмами кэширования
- Поддержана работа с разделяемой памятью в операционной системе Linux, что позволяет решению поддерживать свое состояние даже между перезапусками приложения
- Оптимизированы внутренние структуры данных для минимизации времени блокировок и потребления памяти на поддержание метаданных

Список литературы

[1] A.A. Pangin. One-nio. — https://github.com/odnoklassniki/one-nio.

[2] Gil Einziger Roy Friedman. TinyLFU: A Highly Efficient Cache Admission Policy // Computer Science Department of Technion. — 2015.

[3] Lee Breslau Pei Cao Li Fan Graham Phillips, Shenker Scott. Web caching and zipf-like distributions: Evidence and implications // IEEE INFOCOM. — 1999.

[4] Nimrod Megiddo Dharmendra S. Modha. Outperforming LRU with an Adaptive Replacement Cache Algorithm // CIEEE Computer. — 2004.

[5] Saar Cohen Yossi Matias. Spectral bloom filters // CM SIGMOD. — 2003.

[6] Song Jiang Xiaodong Zhang. LIRS: An Efficient Low Inter-Reference Recency Set Replacement Policy to Improve Buffer Cache Performance // ACM Sigmetrics. — 2002.

[7] Xenofontas Dimitropoulos Marc Stoecklin Paul Hurley, Kind Andreas. The eternal sunshine of the sketch data structure // Comput. Netw. — 2008.

[8] Xiaoning Ding Song Jiang Xiaodong Zhang. BP-Wrapper: A System Framework Making Any Replacement Algorithms (Almost) Lock Contention Free // IEEE 25th International Conference on Data Engineering. — 2009.