Федеральное государственное автономное образовательное
учреждение высшего образования
«Национальный исследовательский университет
«Высшая школа экономики»

Факультет компьютерных наук
Основная образовательная программа
Прикладная математика и информатика

КУРСОВАЯ РАБОТА

Программный проект на тему «Cache-словари на SSD в ClickHouse»

Выполнил студент группы 175, 3 курса,
Васильев Никита Сергеевич

Руководитель КР:
Руководитель группы разработки ClickHouse,
Миловидов Алексей Николаевич

Москва 2020

Аннотация

При работе с СУБД ClickHouse [1] иногда возникают запросы, в которых требуется при помощи словарей использовать данные из внешних источников. Производительность является одной из главных причин выбора ClickHouse пользователями, а запросы во внешние источники могут быть довольно долгой операцией, поэтому в ClickHouse предусмотрены cache-словари, которые кэшируют данные из внешних источников. На данный момент в ClickHouse существуют только cache-словари, хранящие свои кэши в оперативной памяти, однако оперативная память хоть и очень быстра, но её размер ограничен, а цена высока. SSD же имеет больший объем, меньшую цену за байт, но и меньшую скорость, а также сильно отличающийся от оперативной памяти и жестких дисков принцип работы, который требует других подходов для эффективного использования. В данной работе предложены и реализованы два cache-словаря для СУБД ClickHouse, располагающие данные на SSD, использующие особенности работы ClickHouse и SSD для получения максимальной производительности. Они хранят в оперативной памяти $16\frac{1}{16}$ байт на ключ для ключей-чисел и $18\frac{1}{16} + k$ байт на ключ для других типов ключей, где k — размер ключа в байтах. Такие словари обеспечивают достаточно высокую для практических задач производительность и позволяют увеличить объем кэша, при этом не используя большого объема оперативной памяти.

Ключевые слова: системы управления базами данных, кэширование, SSD, хранилище ключ-значение, кэширование на SSD.

Abstract

While working with ClickHouse DBMS [1] you can face some queries, which require the use of dictionaries to access data from external sources. Performance is one of the main reasons why users choose ClickHouse. However, queries to external sources can take a lot of time for execution. That's why cache-dictionaries exist in ClickHouse. They cache data from external sources. In current time there are only cache-dictionaries which use RAM as storage for cached data. RAM is fast, but its size is limited and its price is high. SSD has a larger size, a lower price per byte, but poorer performance. Also, it has many differences from RAM and hard drives and requires another approach for effective use. In this work cache-dictionaries for ClickHouse, which store data on SSD and take maximum performance from features of ClickHouse and SSD, are proposed and implemented. They store in RAM only $16\frac{1}{16}$ bytes per key for numeric keys and $18\frac{1}{16} + k$ bytes per key for other types of keys, where k is the size of the key in bytes These dictionaries provide enough performance for use in practice and allow to increase cache size without using too much RAM.

Содержание

1 Введение

СУБД ClickHouse предназначена для выполнения аналитических запросов на больших объемах данных (может хранить петабайты данных на кластере). Часто возникают запросы, в которых требуется использовать данные из внешних источников, например, по хранящемуся в таблице действий пользователей на сайте идентификатору пользователя получить город, указанный у него в профиле, из другой СУБД. Для таких действий в ClickHouse существуют словари. Cache-словарь — это один из видов словарей, представляющий собой кэш элементов из внешнего источника, который хранит ограниченное число элементов в оперативной памяти не более определенного количества времени. Ключ словаря представляет собой кортеж из числовых типов и строк, каждому ключу сопоставляется кортеж значений, также состоящий из числовых типов и строк.

1.1 Актуальность и значимость

Локальное кэширование необходимо для ускорения работы ClickHouse и снижения нагрузки на внешние источники, так как ClickHouse может обрабатывать огромное число строк за короткий промежуток времени, и запросы каждого элемента из внешнего источника могут отрицательно сказаться на производительности системы и стоимости её обслуживания. В данный момент в ClickHouse cache-словари поддерживают кэш в оперативной памяти, но её объем сильно ограничен, поскольку она используется и для множества других задач.

SSD имеют более высокие задержки по сравнению с оперативной памятью, однако стоят дешевле. При правильном использовании SSD для хранения данных, ранее хранимых в оперативной памяти, во многих случаях можно получить достаточную для работы системы производительность, при этом увеличив объем хранимых данных и/или снизив затраты на используемую память.

Используя SSD в качестве хранилища кэша в ClickHouse, можно будет при

тех же затратах на оборудование увеличить объем кэша, тем самым уменьшив нагрузку на сеть и на внешние источники, ускорив запросы (из-за большей доли попадающих в кэш запросов) и освободив часть оперативной памяти для других данных, что положительно скажется на производительности всей СУБД.

1.2 Цели и задачи

У СУБД ClickHouse есть особенности, которые обуславливают требования к словарям и их интерфейсу. Во-первых, ClickHouse — аналитическая СУБД. Это означает, что во время запроса она максимально использует доступные ресурсы, чтобы его выполнить, поэтому нагрузка неравномерная, в том числе и на словарь, и есть время и ресурсы для выполнения фоновых операций, однако словари не являются самой важной частью СУБД и не должны требовать много ресурсов даже для фоновых задач. Во-вторых, СУБД ClickHouse активно использует векторизацию, поэтому строки обрабатываются пачками, в которых обычно несколько тысяч строк, соответственно в словарь запрос приходит тоже пачкой строк.

Сложность реализации кэша увеличивает еще и внутреннее устройство SSD. Оно подробно описано в статье [2]. Минимальным записываемым/читаемым элементом SSD является страница, которая имеет размер в несколько килобайт (например, 4Кбайт), страницы объединены в блоки (например, по 64 страницы в одном блоке). Удалять данные с SSD можно только целыми блоками. Эта операция сильно медленнее чтения/записи, поэтому освобождение места на SSD происходит в фоновом режиме. Каждый блок в SSD может быть очищен только ограниченное число раз. Итого небольшие случайные записи на SSD неэффективны и уменьшают его срок службы по сравнению с последовательной записью большого объема данных. Также операции чтения и записи могут выполняться параллельно. SSD делятся по интерфейсу подключения и протоколу работы на SATA SSD, которые используют то же подключение, что и жесткие диски, и NVMe SSD, которые используют подключение к PCIe и дру-

гой протокол, эффективнее использующий возможности SSD. В рамках данной работы разница между NVMe и SATA SSD не рассматривается.

В статье [3] выдвигается гипотеза «The RUM Conjecture», что выбрав структуру данных для хранения, которая для накладных расходов на чтение, запись и занимаемую память ограничивает сверху две характеристики из трех, мы ограничиваем снизу оставшуюся. В случае кэша в ClickHouse предполагается, что большая доля запросов попадает в кэш, а промахов, за которыми следуют записи, мало, поэтому интересуют нас в первую очередь чтение и объем занимаемой памяти.

В целью данной работы является реализация cache-словарей, которые для хранения значений элементов используют SSD вместо оперативной памяти, тем самым уменьшая её расход и позволяя увеличить число элементов в кэше.

Задачи работы состоят в выборе оптимальных структур данных для словарей с учетом особенностей работы СУБД ClickHouse и SSD для увеличения скорости обработки запросов, срока службы SSD и нагрузки на внешние источники, а также в реализации двух cache-словарей, один из которых работает с любыми ключами, а второй работает только с ключами, которые являются целыми неотрицательными 64-битными числами, при этом эффективнее используя оперативную память и работая быстрее.

1.3 Некоторые определения

Для удобства дальнейшего описания приведу некоторые используемые определения.

1. Словарь с простым ключом — словарь, ключ которого является 64-битным целым неотрицательным числом.

2. Словарь со сложным ключом — словарь, ключ которого является кортежем из чисел и строк.

3. Сериализация — перевод структуры в памяти в последовательность байт.

4. Десериализация — перевод последовательности байт в структуру в памяти.

1.4 Структура работы

Работа состоит из четырех частей. Сначала идет краткий обзор существующих решений для кэширования данных на SSD, затем следуют описание внутреннего устройства, использования и реализации cache-словарей на SSD, после чего приведены измерения производительности реализованных словарей с анализом результатов.

2 Существующие решения

При выборе структуры данных для кэша на SSD возникает множество различных вопросов, относительно ответов на которые можно классифицировать разные решения, например, как искать значения, как ограничивать размер кэша.

С некоторыми вопросами определиться легко, например, запись везде происходит большими блоками, но некоторые части реализуются разными способами. Среди таких частей системы поиск и структура хранения данных на SSD. В рассмотренных статьях представлены 3 варианта:

1. Хранить несколько отсортированных файлов с ключами и значениями, а искать значение бинарным поиском по ключу.

2. Хранить на SSD только значения, а поиск производить по индексу в ОЗУ, который отображает ключи в адреса на SSD.

3. Хранить на SSD ключи и значения, а искать по индексу в ОЗУ, который отображает хеши от ключей в адреса на SSD.

Для рассмотрения сгруппируем работы по методам поиска и хранения, так как эти отличия наиболее существенны. Также можно классифицировать кэши

по алгоритмам вытеснения (что делать, если элементов слишком много), алгоритмам сборки мусора (что делать, если на SSD занято слишком много места) и способам работы с SSD (через файловую систему или напрямую).

2.1 Существующие структуры данных для кэша на SSD

2.1.1 RocksDB

Сначала рассмотрим вариант, минимально использующий оперативную память, — хранение файлов с отсортированными строками ключ-значение. Такой вариант хранения реализуется в RocksDB [4] [5] и называется Log Structured Merge Tree. Файлы с отсортированными записями, разбитыми на блоки, называются Sorted Sequence Table (SST). Помимо блоков с элементами в файле хранится блок с индексом для поиска нужного блока по ключу бинарным поиском и дополнительные блоки, например, со статистикой или фильтрами Блума. Элементы записываются сначала в оперативную память в структуру под названием memtable, а по достижении нужного размера выгружаются на диск (есть еще write ahead log, но здесь он не рассматривается, так как для кэшей не требуется устойчивость к отказам). Такой способ записи эффективен для SSD, поскольку лучшим образом происходят именно большие последовательные записи. Аналогично работает запись и во всех остальных рассматриваемых подходах. При чтении RocksDB просматривает memtable и файлы и ищет в них нужный ключ, сначала производя бинарный поиск блока в индексе, а потом поиск ключа в блоке. Также RocksDB имеет множество оптимизаций для поиска, например, фильтры Блума для определения наличия ключа в SST. Чтобы SST не становилось слишком много и для удаления с диска устаревших, обновленных и удаленных данных в фоновом режиме происходят слияния SST. Каждая SST имеет свой уровень (например, L0 для самых новых). Несколько SST одного уровня сливаются в SST следующего уровня. RocksDB позволяет установить время жизни для записей, чтобы при слияниях записи с истекшим сроком жизни удалялись. Такое устройство хранения хорошо работает при интенсивной записи,

но в случае кэша записей мало, но много чтений, а для чтения в данном случае может понадобиться просмотреть несколько файлов. Также поиск выполняется за $O(\log N)$, где N — количество элементов, что при большом числе элементов может быть медленно по сравнению с хеш-таблицей. Стоит отметить, что из-за слияний элементы могут перезаписываться несколько раз, что негативно скажется на сроке службы SSD, а также слияния могут увеличить расход оперативной памяти и загрузку центрального процессора.

2.1.2 CaSSanDra

Другой вариант подразумевает хранение в оперативной памяти индекса для быстрого поиска (например, хеш-таблицы), отображающего ключи в адреса значений на SSD. Такой способ используется в статье [6]. В ней описывается кэш второго уровня (первый в оперативной памяти) на SSD для Cassandra [7], представленный в виде набора файлов фиксированного размера, в которых хранятся значения, и хеш-таблицы в оперативной памяти, которая хранит отображение из ключей в адреса. Адрес представляет собой 64-битное число, где первые 32 бита — номер файла, а последние — смещение в файле. Значения записываются в конец последнего созданного файла блоками по 16 КБ, до этого они хранятся в буфере в оперативной памяти. Удаление ключа из кэша просто удаляет его из хеш-таблицы, для удаления элементов с SSD используется фоновая очистка мусора, которая выбирает один из файлов при помощи хранящейся статистики о количестве мусора в файле и убирает из него удаленные ключи. Из кэша при превышении лимита размера индекса удаляется неиспользованный дольше всех ключ, то есть используется алгоритм LRU. Такой кэш эффективно пишет/читает с SSD и хорошо обеспечивает долгий срок службы за счет записей большими блоками и хранения обновляемого индекса в оперативной памяти, поскольку отсутствуют лишние записей на SSD, но фоновые задачи очистки мусора могут перезаписывать элементы, что снижает срок службы SSD. Также, если ключи имеют большой размер, то индекс может занимать большой объем оперативной памяти. Он хорошо работает в случаях, когда запрос нового эле-

мента — дорогая операция, и, когда у элементов кэша большое время жизни, поскольку он сохраняет элементы максимально долго.

2.1.3 Fatcache

Большинство решений не хранят ключи в оперативной памяти. Вместо них хранится хеш от ключа, а сам ключ хранится на диске. Все решения рассмотренные далее также, как и пример выше, пишут на SSD данные пачками и имеют сборку мусора.

Одним из таких примеров является система Fatcache [8], которая разработана в Twitter. Вместо ключа в оперативной памяти используется его SHA-1 хеш, а сам ключ хранится на SSD и сверяется уже при чтении. Соответственно, если у двух ключей хеш совпал, то при записи они будут перезаписывать друг друга в индексе. Сборка мусора запускается при записи данных на диск. Блоки записи в Fatcache называются плитами (slab), если свободного места нет. При сборке мусора просто удаляется самая старая плита (алгоритм FIFO), а новая пишется вместо нее. Такой способ хранения уменьшает требования к объему оперативной памяти, когда ключи имеют большой размер, поскольку в оперативной памяти хранится только хеш, однако способ будет потреблять больше памяти, если ключи имеют маленький размер, примером таких ключей могут быть простые ключи. Удаление самой старой плиты выглядит хорошей идеей, поскольку в нем предполагается наличие наибольшего числа неиспользуемых элементов и элементов, срок жизни которых скоро выйдет.

2.1.4 FlashStore

FlashStore [9] ставит одной из целей минимизацию использования оперативной памяти. Система работает с SSD и HDD, используя последний как медленное хранилище. Рассматривать работу с HDD в рамках данного обзора я не буду. Система использует в качестве индекса хеш-таблицу с модифицированным хешированием кукушки, то есть с несколькими хеш-функциями. Для запи-

си выбирается первая пустая ячейка, среди соответствующих хеш-функциям от ключа. Если пустого места нет, то система пытается сделать несколько перемещений ключей, а если ячейка не освободилась, то ключ из нее перекладывается в дополнительную структуру данных малого размера. В самой хеш-таблице вместе с указателем хранится небольшой хеш от ключа, чтобы уменьшить количество чтений с SSD. При большом количестве ключей или занимаемой кэшем памяти на SSD запускается сборка мусора, которая перемещает последние данные на HDD, однако для каждой ячейки хеш-таблицы в индексе хранится еще бит обозначающий недавнее использование этой ячейки. Если этот бит установлен в 1, то элемент возвращается из удаляемых данных в буфер в оперативной памяти. То есть в данном случае используется модификация алгоритма FIFO. Этот подход позволяет поддерживать индекс одновременно быстрый, использующий мало памяти и не страдающий от переписывания элементов при совпадающих хешах, а возвращение недавно использованных элементов может увеличить долю попаданий в кэш, что улучшит его производительность. Однако данные подходы добавляют сложности реализации, что может привести к увеличению числа ошибок в программе. Также в статье описывается, что для обслуживания клиентов, записи значений на SSD и сборки мусора используются разные отдельные потоки. Такое устройство облегчает написание блокировок в многопоточном коде, что уменьшает количество ошибок.

2.1.5 BlueCache

Похожий подход представлен в системе BlueCache [10]. Здесь индекс состоит из n корзин, в каждой из которых хранится до 4-х элементов. Корзина элемента определяется хеш-функцией. Если приходит новый ключ, но корзина заполнена, то из нее просто удаляется запись самого старого ключа. Таким образом для вытеснения элементов используется LRU внутри корзины. Ключи хранятся на SSD вместе со значением, а в оперативной памяти хранится только хеш ключа, поэтому при положительных результатах проверки наличия ключа в индексе надо проверить совпадение ключа с ключом на SSD. Очистка мусо-

ра на SSD при недостатке места очень простая — удаление самого старого записанного блока. Этот кэш так же экономит оперативную память, Интересным отличием BlueCache также является работа с SSD напрямую без flash translation layer, симулирующего работу с жестким диском, что позволяет эффективно использовать параллелизм, но создает дополнительные сложности с управлением чтением и записью, например, системе приходится следить за износом блоков. Помимо самого алгоритма в статье есть сравнение производительности кэша в оперативной памяти и на SSD при разных размерах пар ключ-значение. Пропускная способность кэша на SSD при чтении на порядок меньше, чем в оперативной памяти. Также указано, что для чтения с SSD задержки больше, чем у чтения из оперативной памяти из-за заметных задержек SSD и задержек передачи данных. Для записи таких сильных различий не наблюдалось, хотя запись на SSD все-равно происходит медленнее при больших размерах пар, однако для пар размером 64 байта запись на SSD происходила быстрее, чем в оперативную память. Такой эффект авторы объясняют тем, что в случае с SSD паттерн записи более последовательный.

2.1.6 Flashield

Недавно был представлен Flashield [11]. Индекс в данном случае является хеш-таблицей в оперативной памяти с несколькими хеш-функциями. Если ключ не найден по одной хеш-функции, то берется следующая. Сами ключи хранятся на SSD, поэтому при нахождении адреса элемента в индексе надо сверять значение ключа. Для определения адреса элемента на SSD также используется хеш-функция, в индексе хранится номер записанного блока и номер хеш-функции из набора, которая определяет адрес элемента внутри блока. Помимо прочего в данном кэше представлена интересная идея фильтрации элементов в оперативной памяти перед записью на SSD, при помощи алгоритма машинного обучения SVM кэш определяет, будут ли еще происходить обращения к данному элементу, и не пишет на SSD элементы, к которым обратятся с низкой вероятностью, тем самым увеличивая срок службы SSD. Для выбора элемен-

тов для вытеснения из кэша используется вариант алгоритма CLOCK. На каждый элемент хранятся 2 бита, при удалении элементы обходятся, их значения при обходе уменьшаются на 1, первый элемент с нулевым значением удаляется, для приближения LRU биты устанавливаются в 1 при обращении, а для MFU (самый часто используемый) — увеличиваются на 1. Элементы из оперативной памяти удаляются, а с SSD помечаются «призраками», которые перестают таковыми являться, если к ним будет произведено обращение. В случае превышения размеров кэша на SSD удаляется самый старый записанный блок, а его элементы, не являющиеся «призраками», возвращаются в оперативную память. Интересной особенностью такого подхода является хранение в оперативной памяти всего 4х байтов на элемент. Такой кэш увеличивает срок службы SSD, однако он предполагает несколько чтений с SSD для каждого ключа до нахождения требуемого, как и в прошлом случае наличие достаточно большого количества ключей с одних хешем будет вызывать лишние записи на SSD, предлагаемая адресация приводит к появлению неиспользуемого пространства, а фильтрация требует дополнительных ресурсов.

2.2 Чтение и запись на SSD

Помимо самих алгоритмов нужно еще выбрать интерфейс взаимодействия с SSD. Как отмечалось выше, BlueCache работает с SSD напрямую, что создает дополнительные сложности в системе. Различные способы чтения и записи файлов рассматриваются в [12], где авторы описывают, почему в Scylla [13] используется асинхронный ввод/вывод. Такой интерфейс хорошо подходит для работы с SSD, поскольку он избегает лишних копирований данных и блокировок потоков, позволяет выравнивать запись по меньшим блокам, а не по используемым операционной системой, что может быть важным при чтении. Также добавлю, что такой интерфейс позволяет эффективнее использовать параллелизм SSD, при этом управляя задачами из одного потока.

Существует еще интерфейс io_uring [14], который может быть более эф-

фективным в данной задаче, чем обычный асинхронный ввод/вывод, однако он появился совсем недавно и может не везде поддерживаться.

2.3 Выводы из сравнения

Таблица 2.1. Сравнение систем для хранения кэша на SSD

	вытеснение из кэша/ сборка мусора	поиск	запись	структура на SSD
RocksDB	compaction (настраиваемые слияния)	бинарный поиск	большими объемами с буфером в ОЗУ	SST
Fatcache	FIFO	хеш-таблица	большими объемами с буфером в ОЗУ	ключ-значение
CaSSanDra	LRU/ compaction (выбор лучшего файла)	хеш-таблица	большими объемами с буфером в ОЗУ	значения
FlashStore	FIFO с возвращениями	хеш-таблица	большими объемами с буфером в ОЗУ	ключ-значение
BlueCache	LRU внутри корзины/ FIFO	n корзин по 4 элемента	большими объемами с буфером в ОЗУ	ключ-значение
Flashield	CLOCK/ FIFO с возвращениями	хеш-таблица и хеш-функция для адресации внутри блока	большими объемами с буфером в ОЗУ	ключ-значение внутри блока, адресуемые хеш-функцией

Итоги сравнения подходов к реализации структуры данных кэша кратко представлены в таблице 2.1. В столбцах указаны их основные отличия.

Как видно, большинство систем используют в качестве индекса различные варианты хеш-таблиц, что позволяет быстро находить требуемый ключ на диске. Без хеш-таблицы работает только RocksDB, но эта встраиваемая база данных не оптимизировалась изначально для работы в качестве кэша. В основном используется хранение на SSD и ключа, и значения, а в индексе хранится хеш от ключа. Это позволяет уменьшать расход оперативной памяти, жертвуя большим количеством обращений к SSD во время чтения. Однако, для реализации кэша в ClickHouse выбран индекс состоящий из корзин, определяемых по хеш-функции от ключа, в оперативной памяти. Ключи хранятся в оперативной памяти целиком. Размер ключей предполагается небольшим, следовательно много памяти они не займут.

Разными системами используются разные алгоритмы вытеснения кэша, но

все они довольно близки к LRU. Для вытеснения элементов из кэширующего словаря в ClickHouse используется алгоритм FIFO внутри корзин. Элементы кэша разбиты на небольшие корзины, которые просматриваются при поиске, а в случае конфликтов при вставке вытесняется самый старый элемент. Такой подход легок в реализации и хорош тем, что удаляется элемент, срок жизни которого истечет быстрее, чем у остальных элементов в корзине.

Для сборки мусора на SSD почти везде используется FIFO, т.е. удаление самых старых элементов. Такой вариант выбран и для кэша ClickHouse, поскольку элементы имеют определенное время жизни, соответственно старые элементы все-равно нужно будет скоро удалить. Другой используемый вариант — фоновые операции compaction, однако они увеличивают сложность реализации, требуют хранения и обновления дополнительных статистик, а также потребляют оперативную память и занимают процессор, что может негативно сказываться на работе других частей системы.

Запись новых значений везде одинакова — запись большими объемами из буфера в оперативной памяти. Такая запись оптимально использует SSD, а в тестах производительности BlueCache видно, что такая запись даже не сильно медленнее записи в оперативную память (напомню, что BlueCache работает с SSD напрямую и максимально использует возможности для параллельной записи). Запись в cache-словарях в ClickHouse производится большими блоками данных, так как это эффективно позволяет использовать возможности SSD и увеличивает его срок службы. Хранятся на SSD и ключи, и значения, поскольку при сборке мусора потребуется удалять элементы из индекса.

Работа с SSD осуществляется через интерфейс асинхронного ввода/вывода. Поскольку в случае словарей ClickHouse запросы приходят большими пачками, то требуемые записи будут распределены по разным блокам записи на SSD. Асинхронный ввод/вывод позволяет читать несколько блоков из запроса одновременно, используя параллелизм SSD, при этом не создавая дополнительных потоков в самом ClickHouse.

3 Описание cache-словарей на SSD

3.1 Архитектура

Система состоит из файла со значениями на SSD, а также индекса и буфера записи в оперативной памяти.

При запросе обрабатывается пачка ключей, для которых требуется получить значения. Во время запроса сначала ключам по индексу сопоставляются адреса, по которым значения ищутся в буфере в оперативной памяти и на SSD и пишутся в ответ. Затем, ключи, которые не были найдены в индексе, и ключи с истекшим сроком жизни, запрашиваются во внешнем источнике и записываются в ответ и в буфер, который при заполнении сбрасывается на SSD.

На весь кэш используется одна блокировка, позволяющая читать значения нескольким потокам одновременно, а писать только одному. Такой вариант хорошо работает, поскольку в кэше записи должны происходить сильно реже чтений, однако архитектура данного кэша позволяет реализовать гранулярные блокировки, если производительности текущего варианта недостаточно.

3.2 Хранение данных

Данные хранятся на SSD блоками, размер которых задает пользователь. Рекомендуется размер равный размеру страницы SSD, чтобы для одного блока читалась с SSD только одна страница. Блок является минимальной единицей чтения с SSD. Блок имеет следующую структуру: 64 бита для контрольной суммы блока, 32 бита для количества ключей в блоке, далее идут записи, которые состоят из ключа, метаданных и значений. Метаданные всегда представляют собой 64 бита, в которых старший бит является флагом, не найден ли был ключ в источнике для словаря. В конце блока могут оставаться неиспользованные байты, поскольку каждая запись должна попадать в один блок, чтобы чтения значений затрагивали минимальное число страниц SSD для увеличения

пропускной способности кэша. Схематически структура блока изображена на рисунке 3.1.

Рис. 3.1. Структура одного блока данных

Значения сериализуются стандартным в ClickHouse способом: для значений, имеющих фиксированный размер (числа), используется простое копирование байт, для строк же сначала пишется размер строки, а за ним уже идет сама строка. Если ключи — целые неотрицательные 64-битные числа, то есть простые ключи, они сериализуются простым копированием, в противном случае ключи рассматриваются как не интерпретируемая последовательность байт переменной длины. Получается этот набор байт стандартной в ClickHouse сериализацией. При сериализации сложного ключа сначала пишется его размер в двух байтах, а потом уже сам ключ.

Заданное в настройках количество блоков составляют гранулу. Гранула — минимальная единица записи на SSD. Файл на SSD состоит из нескольких гранул, буфер записи в оперативной памяти — из одной гранулы. Место для хранения файла резервируется заранее при первом обращении к словарю. Структура файла изображена на рисунке 3.2.

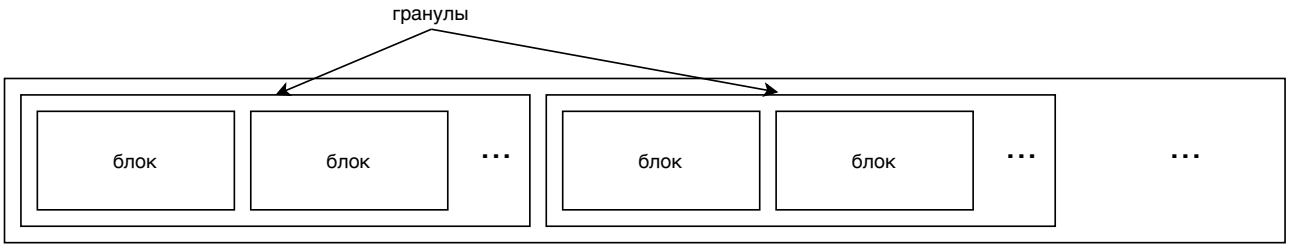

Рис. 3.2. Структура файла с данными

При записи элементов в кэш, после их получения из источника, они пишутся сначала в буфер, во время записи добавляясь в индекс. Элементы пишутся в последний незаполненный блок в буфере. Если запись текущего элемента требует больше памяти, чем есть свободного места в текущем блоке, то запись

блока завершается, то есть заполняются поля с количеством элементов в блоке и контрольной суммой, и запись продолжается в следующий блок.

Если в процессе записи буфер заполнился, то он сбрасывается в файл на SSD на место пустой гранулы, а, если таких нет, то буфер записывается на место самой старой гранулы (используется алгоритм вытеснения FIFO), но перед этим из гранулы, на место которой происходит запись, читаются все ключи, затем ключи, для которых элементы адресуются индексом в эту гранулу, удаляются из индекса. Во время сброса также меняется адрес в индексе для сбрасываемых на SSD ключей.

При чтении используются адреса, элементов, полученные из индекса. В адресе хранится информация о номере блока, смещении начала элемента в блоке и расположении блока, он в файле или буфере. Отдельно происходит чтение из буфера записи. Там требуемый элемент находится по адресу и читается.

Чтение с SSD устроено сложнее. Полученные адреса сортируются и группируются по блокам, в которые они указывают. Затем создается буфер для чтения размером в заданное пользователем число блоков. Пусть в буфере для чтения m блоков, а прочитать требуется n блоков, тогда iый блок из требуемых будет читаться в блок в буфере с номером $(i \mod m)$. Чтение происходит при помощи интерфейса асинхронного ввода/вывода. Для чтения поддерживаются два указателя to_push и to_pop, так, что $to\_pop \leqslant to\_push$ и $to\_push - to\_pop \leqslant m$. to_push указывает на последний блок, который еще не был отправлен на чтение, а to_pop — на последний блок, который еще не был прочитан. Цикл чтения выглядит следующим образом:

1. Если $to\_push = to\_pop = n$, тогда выходим, так как все прочитано.

2. Узнаем, какие задачи чтения уже завершились и записали результат в буфер чтения, помечаем их завершенными.

3. Двигаем указатель to_pop до первой незавершенной задачи (или до конца). При каждом перемещении читаем из записанного блока требуемые элементы, используя сведения об их смещении из адреса.

4. Отправляем новые задачи чтения и передвигаем to_push. При этом добавляется не более, чем $\min(m - (to\_push - to\_pop), n - to\_push)$ задач, чтобы не получилось ситуации, когда две задачи читают данные в один блок буфера.

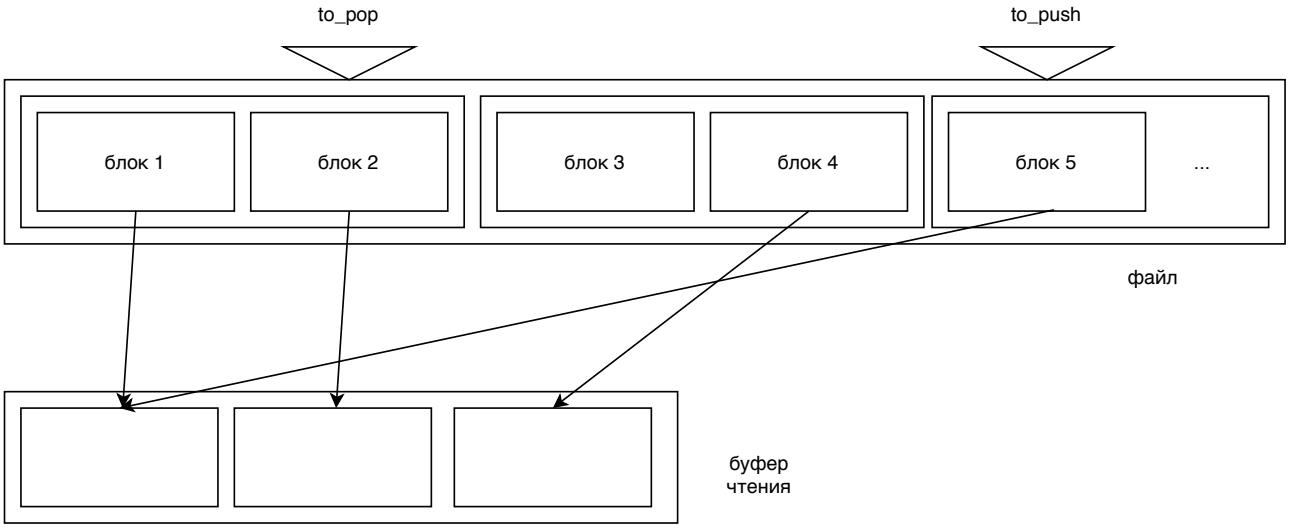

Рис. 3.3. Чтение блоков с SSD

На рисунке 3.3 изображен процесс чтения с SSD блоков данных. Стрелками указано, в какой блок буфера будет записан блок с SSD. В данной ситуации блок 1 уже был прочитан, а значения из него записаны в ответ, для блоков 2 и 4 есть запущенные задачи чтения, но их результаты пока еще не получены. Задача на чтение блока 5 еще не отправлена, так как на ней установлен указатель to_push.

Чтение элемента происходит следующим образом:

1. Читаются метаданные. Если срок жизни элемента истек, то возвращается, что элемент не найден.

2. Если в метаданных указано, что элемент не был найден в источнике, то в ответ записывается значение по-умолчанию.

3. В остальных случаях десериализуетя нужное значение элемента, которое находится пропуском предыдущих значений.

Такой способ хранения выбран, поскольку он позволяет использовать параллелизм SSD при чтении, так как одновременно выполняются несколько за-

дач на чтение данных. Запись в cache-словарях в ClickHouse производится большими блоками данных со удалением последней гранулы, так как в таком случае удаляются самые старые элементы, срок жизни которых быстрее остальных закончится, следовательно их и так придется удалить раньше остальных. Также такая запись эффективно использует пространство на SSD, заполняя страницы целиком. Хранятся на SSD и ключи, и значения, поскольку при сборке мусора потребуется удалять элементы из индекса. Такой способ позволяет получить удаленные ключи из очищенного сборкой мусора блока, просто прочитав его, то есть не потребуется поддерживать дополнительные структуры данных для этой задачи. Также, если несколько ключей расположены в одном блоке, то он прочитается только один раз, что положительно сказывается на производительности кэша.

3.3 Индекс

Индекс представляет собой массив пар 64-битных чисел, первое из которых является ключом в случае, если ключ простой, или ссылкой на ключ, который хранится в сериализованном виде в специальном пуле ключей, представляющим собой набор набор больших непрерывных блоков оперативной памяти, в противном случае, а второе — адрес записи. Массив разбит на корзины по 8 элементов. Корзина определяется по хэшу от ключа. Используется intHash64 из ClickHouse для ключей-чисел и CityHash для остальных видов ключей.

При поиске ключа производится проход по корзине, с поиском требуемого ключа и проверкой, не имеет ли адрес во втором числе пары значение несуществования. Если ключ с таким адресом находится, то возвращается этот адрес.

При вставке в кэш нового элемента, происходит поиск ключа, и, если ключ не найден, то новое значение записывается в корзину на первое пустое место или на самого старого элемента, если пустых мест не осталось. Здесь для вытеснения элементов используется алгоритм FIFO.

Для быстрого нахождения самого старого элемента хранится второй мас-

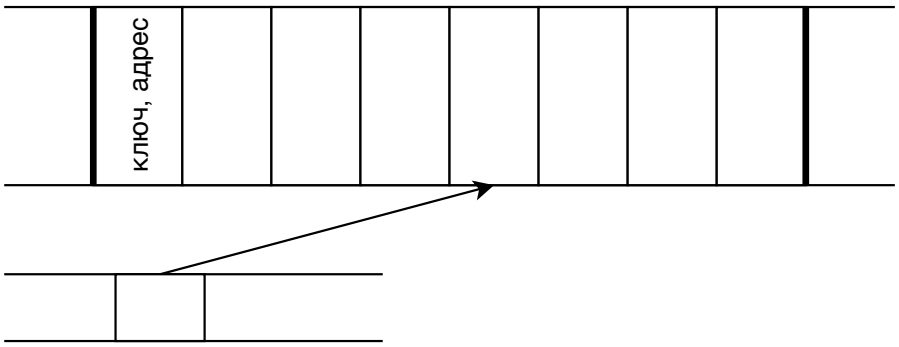

Рис. 3.4. Устройство корзины индекса

сив чисел, где каждой корзине сопоставлено 4 бита, значение которых является местом самого старого элемента в корзине, то есть местом, куда пойдет следующая запись. Тут хватило бы и 3 бита, но для удобства вычислений были взяты 4. После записи значение в этих 4х битах складывается с 1, и результат этой суммы берется по модулю 8 при помощи операции «или» с 111_2, тем самым обеспечивая движение по кольцу. При чтении же ячейки корзины просматриваются в порядке обратном этому движению, чтобы быстрее обнаруживать более новые элементы. На рисунке 3.4 сверху изображена одна корзина индекса, а снизу указатель из второго массива на самый старый элемент в корзине.

Такое устройство позволяет легко ограничивать количество хранимых в индексе элементов и обеспечивает удаление самого старого элемента в корзине, который, вероятно, уже не используется. Также такая структура быстро работает за счет просмотра рядом лежащих чисел, что обеспечивает хорошее попадание чтений в кэш процессора, и требует хранения всего $16\frac{1}{16}$ байт для простых ключей и $18\frac{1}{16} + k$ байт для остальных ключей, где k — размер ключа (дополнительные 2 байта возникают из-за того, что для ключа в пуле хранится еще его размер). То есть для хранения n ключей в индексе понадобится $const + 16\frac{1}{16} \cdot n$ байт для простых ключей и $const + 18\frac{1}{16} \cdot n + \sum_{i=1}^{n} k_i$ байт для сложных ключей, где k_i — размер i-ого ключа.

3.4 Адресация

Адрес элемента кэша представляет собой 64-битное число, где первый бит обозначает хранится ли данный элемент в буфере в оперативной памяти или на SSD, 32 бита с 17 по 48 хранят номер блока, в котором хранится элемент, а младшие 16 бит хранят смещение элемента от начала блока. Помимо вышесказанного индекс поддерживает специальное значение — все биты единицы, обозначающее, отсутствие такого элемента.

Такой способ представления адреса позволяет эффективно находить элемент, а также позволяет группировать адреса по блокам на SSD простой сортировкой.

4 Использование

Для использования словаря надо в качестве способа хранения словарей в памяти указать `SSD_CACHE` для простых ключей, а для сложных ключей — `COMPLEX_KEY_SSD_CACHE`. При создании словаря можно также указать следующие параметры:

1. `BLOCK_SIZE` — размер минимального читаемого блока. Рекомендуется выставлять равным размеру страницы SSD.

2. `WRITE_BUFFER_SIZE` — размер буфера для записи (он же размер гранулы). Должен быть кратен `BLOCK_SIZE`.

3. `FILE_SIZE` — размер файла на SSD в байтах. Должен быть кратен `WRITE_BUFFER_SIZE`.

4. `PATH` — путь к файлу словаря на SSD.

5. `MAX_STORED_KEYS` — максимальное число ключей в индексе. Оно же — число ячеек в индексе. Будет округлено вверх до ближайшей степени двойки.

Время хранения словаря указывается в отдельной части запроса, общей для всех словарей ClickHouse LIFETIME. Там выставляется максимальное и минимальное значение времени жизни элемента словаря в секундах. Каждому элементу будет назначено в качестве времени жизни случайное число из отрезка между этими двумя числами[1], при этом элемент может быть удален из кэша раньше, чем закончится его время жизни. Это сделано, чтобы много элементов не удалялись одновременно и не создавалось большой нагрузки на источник для словарей.

Пример запроса создания cache-словаря со сложным ключом на SSD:

```
CREATE DICTIONARY ssd_dict_complex_key
(
    `k1` String,
    `k2` Int32,
    `a` UInt64 DEFAULT 0,
    `b` Int32 DEFAULT -1,
    `c` String DEFAULT 'none'
)
PRIMARY KEY k1, k2
SOURCE(CLICKHOUSE(HOST 'localhost' PORT 9000
                  USER 'default' TABLE 'table_for_dict' PASSWORD ''
                  DB 'database_for_dict'))
LIFETIME(MIN 1000 MAX 2000)
LAYOUT(COMPLEX_KEY_SSD_CACHE(FILE_SIZE 1073741824
                             PATH '/var/lib/clickhouse/clickhouse_dicts/1d'
                             BLOCK_SIZE 4096
                             WRITE_BUFFER_SIZE 16384
                             MAX_STORED_KEYS 1048576))
```

В данном случае создается словарь с ключом, который является кортежем из строки и целого 32-битного числа, и тремя атрибутами. Времена жизни элементов этого словаря будут находиться в промежутке от 1000 до 2000 секунд. В качестве источника в поле SOURCE указана таблица с ClickHouse, но это поле общее для всех словарей и в данной работе никаких действий с ним не произ-

[1]Этот функционал взят из обычных cache-словарей ClickHouse.

водилось, поэтому не будем на нем останавливаться.

Также в ClickHouse возможно создание словаря через конфигурацию в формате xml, и cache-словари на SSD не исключение. Пример создания словаря с простым ключом через конфигурацию:

```
<dictionary>
    <name>ssd_dict</name>
    <structure>
        <id>
            <name>id</name>
        </id>
        <attribute>
            <name>attr</name>
            <type>String</type>
            <null_value>'none'</null_value>
         </attribute>
    </structure>
    <source>...</source>
    <layout>
        <ssd_cache>
            <file_size>1073741824</file_size>
            <block_size>4096</block_size>
            <write_buffer_size>16384</write_buffer_size>
            <max_stored_keys>2097152</max_stored_keys>
            <path>/home/nikita/ch_test/data/normal</path>
        </ssd_cache>
    </layout>
    <lifetime>
        <min>1000</min>
        <max>2000</max>
    </lifetime>
</dictionary>
```

Удаление происходит через стандартный для ClickHouse запрос удаления словаря:

```
DROP DICTIONARY ssd_dict_simple
```

Обращаться к словарям можно при помощи обычных функций `dictHas(dict_name, key)` и `dictGet*(dict_name, attr, key)`, где `dict_name` — наименование словаря, `attr` — наименование атрибута, `key` — ключ, по которому выполняется запрос, а `*` — тип требуемого элемента (`Int32`, `String`, ...). Примеры выполнения запросов к cache-словарям приведены на рисунке 4.5.

```
ThinkPad-E570 :)  SELECT dictGetUInt64('database_for_dict.ssd_dict_complex_key', 'a', tuple('test', toIn
t32(3))) AS one,  SELECT dictGetUInt64('database_for_dict.ssd_dict_complex_key', 'a', tuple('test', toIn
t32(3))) AS one, dictGetString('database_for_dict.ssd_dict_complex_key', 'c', tuple('test', toInt32(3)))
 AS two, dictHas('database_for_dict.ssd_dict_complex_key', tuple('test', toInt32(3))) AS three

SELECT
    dictGetUInt64('database_for_dict.ssd_dict_complex_key', 'a', ('test', toInt32(3))) AS one,
    dictGetString('database_for_dict.ssd_dict_complex_key', 'c', ('test', toInt32(3))) AS two,
    dictHas('database_for_dict.ssd_dict_complex_key', ('test', toInt32(3))) AS three

┌─one─┬─two─────────┬─three─┐
│ 100 │ clickhouse  │     1 │
└─────┴─────────────┴───────┘

1 rows in set. Elapsed: 0.010 sec.
```

Рис. 4.5. Выполнение запросов к словарю

5 Реализация

Работа реализована на языке C++ в формате пулл-реквеста[2] в репозиторий ClickHouse. Объем написанного кода составил более 5 тысяч строк. Оба словаря состоят из 3 классов:

1. `SSDCachePartition` / `SSDComplexKeyCachePartition` — классы, которые отвечают за хранение, чтение и запись данных словаря и поддерживают индекс.

2. `SSDCacheStorage` / `SSDComplexKeyCacheStorage` — классы, отвечающие за получение данных из внешних источников и работу с `SSDCachePartition` и `SSDComplexKeyCachePartition` соответственно.

3. `SSDCacheDictionary` / `SSDComplexKeyCacheDictionary` — классы, которые предоставляют стандартные для ClickHouse интерфейсы для рабо-

[2]Ссылка на пулл-реквест https://github.com/ClickHouse/ClickHouse/pull/8624.

ты со словарями, являются наследниками `IDictionaryBase` (класс, определяющий интерфейс словарей в ClickHouse). Являются оберткой для соответственно `SSDCacheStorage` и `SSDComplexKeyCacheStorage`.

Также реализован класс `BucketCacheIndex`, отвечающий за хранение индекса и предоставляющий интерфейс для работы с ним, и вспомогательные классы, такие как `Metadata` и `Index`, которые предназначены для хранения метаданных и адреса записи соответственно, а также шаблонный класс `ComplexKeysPoolImpl<Arena>`, который предназначен для хранения сложных ключей в пуле типа `Arena`.

Тестирование происходило при помощи написания стандартных функциональных тестов ClickHouse в виде набора SQL-запросов. Также производилось тестирование с дополнительным логгированием, чтобы проверить, что программа реализует ожидаемое поведение.

6 Эксперименты

6.1 Измерение производительности чтения и записи

Сравнивалась скорость чтения/записи в четырех словарях. Все словари имели простые ключи, и атрибуты, которые были строками длиной в 16 символов. В качестве источника данных все они использовали написанный на C++ исполняемый файл [3], который для любого ключа возвращал в качестве значения атрибута строку «testtesttesttest». Срок жизни элементов у всех словарей был выставлен в 100000 (и `MIN`, и `MAX`), чтобы он не истек до окончания тестирования.

Отличался только способ хранения словаря. Конфигурация способов хранения словарей представлена в таблице 6.2. Помимо cache-словарей на SSD в сравнении участвовал словарь со способом хранения `CACHE`, который хранит все данные в оперативной памяти, а для поиска использует хеш-таблицу с от-

[3] Код доступен по ссылке https://gist.github.com/nikvas0/36897a94fae5738b8f3a03ea886852ea.

крытой адресацией. У него есть параметр SIZE_IN_CELLS равный максимальному числу ячеек в хеш-таблице.

Таблица 6.2. Конфигурация тестируемых словарей

Словать	Тип	Конфигурация	Комментарий
ram_dict	CACHE	SIZE_IN_CELLS = 2097152	Хранит все в оперативной памяти.
ssd_dict_small	SSD_CACHE	FILE_SIZE = 1073741824 BLOCK_SIZE = 4096 WRITE_BUFFER_SIZE = 4096 MAX_STORED_KEYS = 2097152	Cache-словарь на SSD с маленьким буфером. Почти все ключи на SSD.
ssd_dict	SSD_CACHE	FILE_SIZE = 1073741824 BLOCK_SIZE = 4096 WRITE_BUFFER_SIZE = 16384 MAX_STORED_KEYS = 2097152	Cache-словарь на SSD с буфером размера 16КБайт.
ssd_dict1m	SSD_CACHE	FILE_SIZE = 1073741824 BLOCK_SIZE = 4096 WRITE_BUFFER_SIZE = 1048576 MAX_STORED_KEYS = 2097152	Cache-словарь на SSD с буфером размера 1Мбайт.
ssd_dict_large	SSD_CACHE	FILE_SIZE = 1073741824 BLOCK_SIZE = 4096 WRITE_BUFFER_SIZE = 1073741824 MAX_STORED_KEYS = 2097152	Cache-словарь на SSD с большим буфером. Все ключи попадают в оперативную память.

Далее для каждого словаря выполнялось два запроса. Сначала с пустым кэшем выполнялся запрос `SELECT sum(ignore(dictGetString('dict', 'str', number))) FROM numbers(300000)`, где вместо `dict` стояло название словаря, а `str` было названием атрибута. При помощи него измерялась скорость заполнения кэша стандартными средствами ClickHouse. `numbers(x)` — это системная таблица в ClickHouse, возвращающая числа от 0 до $x - 1$ при запросе. `ignore` — функция в ClickHouse, которая всегда возвращает 0, при этом вычисляя значение аргумента.

После уже с заполненным кэшем выполнялись запросы `SELECT sum(ignore(dictGetString('dict', 'str', toUInt64(intHash64(number + 123) % 300000)))) FROM numbers(count)`, для `count` равных 30000 и 100000 тысячам. При помощи этих запросов уже измерялась скорость чтения из заполненного кэша. Хеш от чисел в данном запросе нужен, чтобы равномерно распределить ключи по доступному пространству.

Тройка запросов для каждого словаря запускалась 3 раза, выбиралось наименьшее время, полученное за эти 3 запуска.

6.2 Анализ результатов

Таблица 6.3. Результаты тестирования производительности cache-словарей

Словарь	Запись, с.	Чтение ($count = 30000$), с.	Чтение ($count = 100000$), с.	Использовано оперативной памяти, байт
ram_dict	11.462	0.007	0.026	100659280
ssd_dict_small	103.015	0.173	0.346	34611185
ssd_dict	26.369	0.135	0.299	34623473
ssd_dict1m	1.207	0.11	0.261	35655665
ssd_dict_large	0.457	0.012	0.029	1108348913

Полученные результаты приведены в таблице 6.3. Видно, что чем больше буфер, тем быстрее происходит чтение и запись. Запись происходит быстрее, поскольку реже происходят сбросы данных на SSD. Можно предположить, что на время записи в данном случае сильно влияет задержка, при этом пропускная способность SSD достаточно велика. Чтение же работает быстрее, поскольку большая доля читаемых ключей попадает в блоки, находящиеся в оперативной памяти и не сброшенные на SSD. Отсюда можно сделать вывод, что буфер следует делать достаточно большим, чтобы сбросы на диск не происходили слишком часто.

Видно, что `ssd_dict_large`, у которого все данные помещаются в буфер и не сбрасываются на диск, показывает близкие к `ram_dict` результаты, что показывает, что десериализация значений при чтении не является медленной операцией, сильно влияющей на производительность.

Помимо вышесказанного можно наблюдать, что чтение с SSD примерно в 10-15 раз медленнее, чем из оперативной памяти (`ssd_dict_large`), однако для многих вариантов использования такой производительности может быть достаточно.

Стоит отметить, что при настоящей нагрузке кэша записи происходят нечасто, следовательно очень большой буфер сильных преимуществ в производительности записи давать не будет. Также можно предположить, что при настоящей нагрузке чтения будут распределены неравномерно, как в тесте, а в таком случае чаще несколько ключей будут попадать в один блок кэша, то есть

уменьшится объем передаваемых между SSD и оперативной памятью данных, что улучшит скорость чтения.

Виден также интересный эффект, что запись в SSD словарь с буфером более или равным 1 Мбайт работает быстрее записи в словаре ClickHouse, хранящемся полностью в оперативной памяти. Такая сильная разница скорее всего связана с разным устройством хранения элементов. Новые элементы в cache-словаре на SSD записываются в конец текущего блока, что обеспечивает лучшую локальность записи, чем запись в ячейку по хешу в обычном cache-словаре. Как было сказано в 2.1.5, подобный эффект наблюдался и в системе BlueCache [10].

7 Заключение

В итоге поставленные задачи выполнены. В данной работе реализованы два cache-словаря, хранящие элементы на SSD, а в оперативной памяти поддерживающие только индекс. Архитектура данных словарей учитывает и эффективно использует особенности работы с SSD, читая с SSD данные блоками и записывая сразу большой объем данных, предварительно сохраняя их в буфере, и особенности словарей ClickHouse, обрабатывая запросы большими наборами ключей.

Тесты производительности показали, что можно достигнуть достаточно высокой производительности кэша, выбрав буфер достаточно большого размера, при этом используя меньше оперативной памяти, чем обычные cache-словари ClickHouse. В работе cache-словари на SSD используют $const + 16\frac{1}{16} \cdot n + b$ байт для простых ключей, где b — размер буфера, а n — максимальное количество хранимых ключей, и $const + 18\frac{1}{16} \cdot n + b + \sum_{i=1}^{n} k_i$ байт для сложных ключей, где k_i — размер i-ого ключа. Стоит также отметить, что буфера размером в 1 МБайт оказалось достаточно для того, чтобы запись из внешнего источника в cache-словарь на SSD работала эффективнее, чем в cache-словарь в оперативной памяти.

Дальнейшие возможные улучшения связаны в основном с увеличением производительности. Во-первых, можно улучшить работу словаря, используя гранулярные блокировки потоков. Блоки в файле можно блокировать по гранулам, которыми файл записывается, а индекс можно блокировать по нескольким корзинам. Во-вторых, видно, что чтение с SSD происходит медленнее, чем из оперативной памяти. Можно попробовать сжимать блоки при записи и разжимать при чтении. Тогда увеличится вероятность того, что два ключа из запроса попадают в один блок, а следовательно придется считывать меньше блоков с SSD. В-третьих, можно использовать более сложные алгоритмы вытеснения элементов из кэша и сборки мусора. Например, можно для каждой гранулы отслеживать количество обращений и перезаписывать не самую старую гранулу, а ту, из которой дольше остальных не было чтений. В-четвертых, переход на io_uring интерфейс для работы с SSD имеет шанс улучшить скорость чтения и записи за счет уменьшения количества системных вызовов, однако такой интерфейс может не поддерживаться в старых версиях операционной системы. В-пятых, следует сделать 2 буфера записи, писать в один, а сброс на диск сделать асинхронным. Такая оптимизация поможет замаскировать задержки при работе с SSD. Кроме производительности можно улучшать еще и объем занимаемой памяти, например, для cache-словарей со сложными ключами перенести хранение ключей на SSD, а в оперативной памяти хранить только хеш от ключа, как, например, делается в BlueCache. Стоит также отметить возможность оптимизаций, увеличивающих время жизни SSD, которые определяют нужно ли записывать элемент в кэш, как это делается во Fashield.

Также интерес представляют дальнейшие исследования зависимости скорости чтения и записи от размера блока и буфера.

Список литературы

1. ClickHouse. — Режим доступа: `https://clickhouse.tech` (дата обращения: 03.05.2020).

2. Design Tradeoffs for SSD Performance / Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber et al. // Proceedings of the 2008 USENIX Technical Conference (USENIX'08). — USENIX, 2008. — Июн.

3. Designing Access Methods: The RUM Conjecture. / Manos Athanassoulis, Michael S Kester, Lukas M Maas et al. // EDBT. — Vol. 2016. — 2016. — P. 461–466.

4. RocksDB. — Режим доступа: `https://github.com/facebook/rocksdb/` (дата обращения: 03.05.2020).

5. Optimizing Space Amplification in RocksDB. / Siying Dong, Mark Callaghan, Leonidas Galanis et al. // CIDR. — Vol. 3. — 2017. — P. 3.

6. CaSSanDra: An SSD boosted key-value store / Prashanth Menon, Tilmann Rabl, Mohammad Sadoghi, Hans-Arno Jacobsen // 2014 IEEE 30th International Conference on Data Engineering. — 2014. — P. 1162–1167.

7. Lakshman Avinash, Malik Prashant. Cassandra: A Decentralized Structured Storage System // ACM SIGOPS Operating Systems Review. — 2010. — Vol. 44, no. 2. — P. 35–40.

8. Fatcache. — Режим доступа: `https://github.com/twitter/fatcache` (дата обращения: 03.05.2020).

9. Debnath Biplob, Sengupta Sudipta, Li Jin. FlashStore: high throughput persistent key-value store // Proceedings of the VLDB Endowment. — 2010. — Vol. 3, no. 1-2. — P. 1414–1425.

10. Bluecache: a scalable distributed flash-based key-value store / Shuotao Xu, Sungjin Lee, Sang-Woo Jun et al. // Proceedings of the VLDB Endowment. — 2016. — Ноя. — Vol. 10. — P. 301–312.

11. Flashield: a Hybrid Key-value Cache that Controls Flash Write Amplification / Assaf Eisenman, Asaf Cidon, Evgenya Pergament et al. // 16th USENIX

Symposium on Networked Systems Design and Implementation (NSDI 19). — Boston, MA : USENIX Association, 2019. — Фев. — P. 65–78.

12. Kivity Avi. Different I/O Access Methods for Linux, What We Chose for Scylla, and Why. — 2017. — Окт. — Режим доступа: https://www.scylladb.com/2017/10/05/io-access-methods-scylla/.

13. Scylla. — Режим доступа: https://www.scylladb.com/ (дата обращения: 03.05.2020).

14. Corbet Jonathan. Ringing in a new asynchronous I/O API. — 2019. — Янв. — Режим доступа: https://lwn.net/Articles/776703/.