



Московский Государственный Университет им. М.В. Ломоносова

Факультет Вычислительной Математики и Кибернетики

Кафедра Автоматизации Систем Вычислительных Комплексов

Шпилевой Владислав Дмитриевич

**Разработка и реализация алгоритма
отложенного обновления вторичных индексов
на LSM-деревьях**

Магистерская диссертация

Научный руководитель:

к.ф.-м.н., ассистент

Д.Ю. Волканов

Москва, 2018

Аннотация

В настоящее время растет популярность баз данных, хранящих данные на диске не в виде традиционных B-деревьев и их производных, а в виде LSM деревьев. Главное преимущество LSM деревьев в том, что их обновление всегда приводит только к последовательной записи на диск, в отличие от B-деревьев. Это возможно благодаря тому, что LSM дерево способно хранить множество версий одной и той же записи - за счет этого при обновлении дерева не нужно точечно читать и удалять старые данные - это происходит позже во время слияния уровней LSM дерева. Это работает, когда в таблице только один индекс - первичный. При наличии вторичных индексов LSM деревья лишаются преимуществ версионности, так как при обновлении дерева нужно явно читать и удалять старые данные из всех вторичных индексов. В настоящей работе представлен обзор существующих способов решения этой проблемы, а также разработанная модификация LSM дерева, которая позволяет не делать явных чтений и удалений старых данных из вторичных индексов. Проведенное экспериментальное исследование нового LSM дерева показало прирост скорости на порядок при наличии нескольких вторичных индексов.

Содержание

1 Введение	4
1.1 Актуальность	4
1.2 Структура работы	5
2 Постановка задачи	6
3 Описание LSM дерева	7
3.1 Слияние уровней	8
3.2 Индексы на LSM деревьях	9
3.3 Проблема обновления индексов	9
4 Обзор существующих способов ускорения записи	11
4.1 Отложенные обновления индексов на B-деревьях	11
4.2 Второй уровень индексов на B-деревьях	12
4.3 Тонкая настройка LSM дерева	13
5 Отложенное обновление вторичных индексов на LSM деревьях	15
5.1 Вставка и удаление	16
5.2 Слияние уровней	17
5.2.1 Первичный индекс	17
5.2.2 Вторичный индекс	18
5.3 Чтения	19
5.4 Реализация	20
5.5 Оценка сложности алгоритма	22
6 Экспериментальное исследование	26
6.1 Микробенчмарк	26
6.2 LinkBench	27
7 Заключение	29
Список литературы	30

1 Введение

1.1 Актуальность

LSM (Log-Structured Merge) деревья были разработаны в 1990-х годах для задач с интенсивной записью, и использовались в файловых системах и для резервного копирования [1]. Но их применение в СУБД (Система Управления Базой Данных) было ограничено из-за *скрытых чтений*. Скрытыми называют чтения, которые выполняются СУБД при обновлении данных, чтобы либо найти старые данные и удалить их, либо чтобы проверить ограничения уникальности при вставке новых данных. Значительная часть чтений в СУБД - скрытая, так как почти любое обновление данных требует проверки различных ограничений и удаления старых данных, и это почти ничего не стоит в В-деревьях, где для обновления данных в любом случае нужно читать [2]. Но на LSM деревьях скрытые чтения значительно снижают производительность, поскольку лишают их преимуществ версионности данных, когда можно сохранять новые данные не читая и не удаляя старые явно.

С появлением SSD (Solid-State Drive) дисков абсолютная скорость любых чтений и записи возросла на порядки по сравнению со старыми механическими дисками, но существенно увеличился разрыв между скоростями последовательной записи и последовательного чтения [3]. Благодаря тому, что LSM дерево всегда выполняет запись на диск последовательно, а чтения из него на SSD по скорости мало отличаются от В-дерева, LSM дерево стало одной из стандартных структур данных для СУБД. Например, на момент написания работы LSM деревья уже используются в LevelDB [4], RocksDB [5], Cassandra [6], Tarantool [7], BigTable [6], HBase [6], Riak [8], MySQL [9].

Однако SSD хоть и делает LSM дерево более конкурентоспособным, но не решает проблему существования скрытых чтений на любое обновление при наличии вторичных индексов, что не позволяет использовать все возможности LSM деревьев, когда в таблице в БД (База Данных) больше одного индекса.

Когда в таблице только один индекс на LSM дереве, то любые изменения, не требующие знания старых данных (такие как *REPLACE*, *DELETE*), возможны без скрытых чтений. Например, в случае *REPLACE* запись просто попадает в дерево с новой версией. Тоже самое при *DELETE* - ключ (набор индексируемых колонок и их значений), по которому производится удаление, попадает в дерево с новой версией и пометкой, что это именно удаление, а не вставка. Это называется *удаление через вставку*. Скрытых чтений не выполняется. Но при появлении вторичного индекса даже *REPLACE* и *DELETE* вынуждены читать старые данные из первичного индекса, чтобы узнать, какой у старой записи был вторичный ключ, и вставить его *DELETE* в LSM дерево вторичного индекса.

Это обычная процедура для индексов на В-деревьях, где нет версионности данных, и на классических LSM деревьях ее тоже нельзя избежать. Это происходит из-за того, что удаление старых версий данных в LSM дереве работает так, что записи считают-

ся разными версиями одних и тех же данных, только если они равны по ключу, по которому сортируется дерево. И если некоторый запрос меняет этот ключ в уже существующей записи, не читая и не удаляя ее явно, то новая запись становится никак не связанной со старой, и старая не удалится никогда - LSM дерево видит их как имеющие разные ключи.

Таким образом, задача борьбы со скрытыми чтениями в таблицах с индексами на LSM деревьях становится актуальной.

1.2 Структура работы

Во третьей главе для более тонкого понимания актуальности проблемы скрытых чтений именно на LSM деревьях раскрываются основные детали устройства LSM дерева, кратко изложены алгоритмы его обновления (вставки, удаления, слияния уровней), устройство индекса, хранящего данные в LSM дереве. В той же главе на примере таблицы с множеством индексов на LSM деревьях разобрана проблема ее обновления.

Во четвертой главе приведен обзор существующих способов уменьшения влияния скрытых чтений на запись в БД, рассказано о том, какие у них недостатки и преимущества, и почему их не достаточно для решения задачи.

В главе 5 описан предложенный алгоритм отложенного обновления индексов на LSM деревьях. В этой главе детально разобран каждый шаг алгоритма, и приведены математические доказательства ускорения записи. Здесь же разобрана реализация новых LSM деревьев на архитектуре БД Tarantool.

В шестой главе представлены результаты экспериментального измерения производительности новых таблиц на LSM деревьях в сравнении со старыми.

Последняя глава, седьмая, содержит описание результатов работы, и рассуждения о том, в какую сторону можно развивать новый алгоритм, и какие еще есть способы ускорить запись в БД.

2 Постановка задачи

Целью работы является увеличение скорости обновления базы данных, которая хранит индексы таблиц в LSM деревьях при помощи модификации процедур обновления и слияния уровней LSM дерева.

Для этого требуется разработать и реализовать алгоритм отложенного обновления вторичных индексов на LSM-деревьях. Его реализация может быть выполнена на основе базы данных Tarantool. Для достижения заданной цели необходимо решить следующие подзадачи:

1. исследовать существующие способы уменьшения влияния скрытых чтений и ускорения записи;
2. разработать алгоритм отложенного обновления вторичных индексов на LSM-деревьях;
3. реализовать алгоритм на основе архитектуры базы данных Tarantool;
4. провести экспериментальную апробацию реализации.

3 Описание LSM дерева

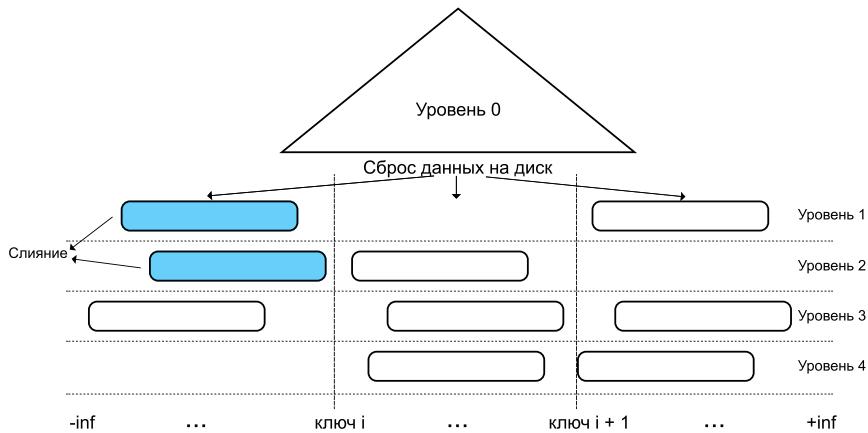


Рис. 1: Схематичное устройство LSM дерева с диапазонами

LSM дерево [1] - это структура данных, оптимизированная для задач с интенсивной записью. Оно состоит из множества уровней: нулевой уровень хранится в оперативной памяти, а остальные - на диске. В нулевой уровень записываются все новые данные, обновления и удаления старых данных. Удаление в LSM дереве тоже реализовано через вставку особой записи в нулевой уровень.

Когда нулевой уровень становится слишком велик, то он записывается на диск, и удаляется из памяти. При этом запись нулевого уровня на диск выполняется последовательно независимо от того, какие ключи, какие операции и в каком порядке он содержит. Записанный уровень становится первым, и в памяти создается новый пустой нулевой уровень для последующих обновлений.

При достижении числом уровней определенного порога LSM дерево выполняет процедуру слияния уровней, во время которой некоторые соседние уровни объединяются в один. LSM дерево может содержать разные версии одних и тех же данных на разных уровнях, например, если запись с некоторым ключом сначала вставили в нулевой уровень, затем он был записан на диск, и в новый нулевой уровень было вставлено обновление или удаление по тому же ключу. Во время слияния уровней старые версии записей обнаруживаются и удаляются - они просто не попадают в новый уровень. То есть слияние в том числе выполняет функцию сборки мусора. После окончания объединения старые уровни удаляются и заменяются одним новым.

Способ организации нулевого уровня и уровней на диске не специфицируется явно, и зависит от реализации. Например, нулевой уровень может быть организован как B+ или красно-черное дерево. На диске уровни могут храниться в виде B деревьев, или как просто массивы записей, отсортированные по ключу.

В данной работе рассматривается такая архитектура, при которой данные на диске хранятся в виде отсортированных массивов, а в памяти в виде B+ дерева. Кроме того, в рассматриваемом LSM дереве значения всех ключей, которые оно хранит, разбиваются на диапазоны, каждый из которых содержит подуровни с данными его ключей.

Внутри диапазона подуровни могут объединяться независимо от других диапазонов - это позволяет сделать процесс слияния уровней LSM дерева более гранулярным. На рисунке 1 изображена схема дерева, устроенного таким образом.

Описанный вариант LSM дерева не ограничивает общности алгоритма отложенного обновления, вводимого в следующих главах, но позволяет рассматривать алгоритмы работы с LSM деревом более конкретно в настоящей работе, и может быть применим к другим вариациям LSM дерева.

3.1 Слияние уровней

Слияние уровней служит для уменьшения высоты дерева, и для удаления старых версий записей. Оно выполняется на некотором подмножестве верхних уровней (при этом возможна ситуация, когда объединяются все уровни дерева). Результатом слияния является новый уровень LSM дерева, который по размеру не превышает суммарного размера объединенных уровней, и заменяет их в дереве. Старые уровни удаляются.

То, как именно выполняется слияние, зависит от реализации LSM дерева. Например, на рассматриваемой здесь архитектуре, слияние выполняется не на целых уровнях, а на подуровнях внутри каждого диапазона (на рисунке 1 изображен пример).

Некоторое множество верхних подуровней, представленных массивами записей, отсортированных по ключу дерева, при помощи сортировки слияниями объединяется в новый отсортированный массив. Сортировка слияниями гарантирует, что если в разных источниках есть разные версии одной записи (то есть они равны по ключу), то на некотором шаге сортировки можно будет увидеть их все разом, и в этот момент понять, какая запись самая новая и попадет в новый массив, а какие будут пропущены. Возможна ситуация, что ни одна из версий не попадет в новый массив, если самая новая версия - это запись об удалении ключа.

Частота выполнения слияний уровней определяется несколькими параметрами дерева:

- коэффициент размера уровней - это такое число, что в каждой паре соседних уровней i и $i + 1$ размер уровня $i + 1$ должен быть как минимум в это число раз больше, чем размер уровня i ;
- максимальная глубина дерева - это ограничение на максимальное число уровней.

Если нарушено хотя бы одно из этих условий, то процедура слияния выполняется на таком числе уровней, чтобы удовлетворить оба условия.

Слияние уровней на практике является дорогой процедурой, поскольку выполняются чтение и запись диска (все уровни, кроме нулевого, хранятся на диске). Но так как в LSM дереве данные не меняются на месте (вместо этого создаются новые версии), то гарантируется, что уже записанные на диск уровни не изменятся, и это позволяет выполнять процедуру слияния уровней в фоне и одновременно с их чтением различными пользовательскими запросами. Архитектура с разбиением значений ключей дерева

на диапазоны открывает возможность выполнения одновременных слияний в разных диапазонах.

3.2 Индексы на LSM деревьях

LSM деревья, помимо прочих применений, используются для хранения индексов в базах данных [4, 5, 7]. Индекс в БД - это некая структура, не обязательно LSM дерево, хранящая данные таблицы в отсортированном виде [20]. Одна таблица может иметь множество индексов, и каждый сортирует одни и те же данные по своим ключевым полям. Индексы используются для ускорения поиска по значениям индексируемых полей. У любой таблицы всегда есть уникальный первичный индекс (даже если он скрыт от пользователя [11]), и сколько угодно вторичных индексов, которые могут быть не уникальны.

Записи первичного индекса хранят каждую запись целиком в том виде, в котором она попала в таблицу. Вторичный индекс хранит свой ключ объединенный с первичным ключом, и не хранит всю остальную часть записи. Это позволяет экономить память, храня не влияющую на сортировку какого-либо индекса часть записи только один раз - в первичном индексе. Первичный ключ в записях вторичного индекса используется, чтобы по любой записи во вторичном индексе можно было найти ее полную версию в первичном, когда запрос выполняет поиск по вторичному индексу.

Описанная архитектура индексов не является уникальной для LSM деревьев. Индексы на B и B+ деревьях могут храниться таким же образом [10, 11]: полная запись в первичном индексе, и только ключевые части во вторичных.

Поскольку вторичные индексы связаны с первичным, СУБД должна поддерживать консистентность индексов - если записи нет в первичном, то ее не должно быть ни в одном из вторичных. И наоборот - если запись есть в первичном, то она должна существовать в каждом вторичном индексе. Именно по этой причине существует необходимость в скрытых чтениях - если обновление таблицы изменяет вторичный ключ некоторой записи, или вовсе ее удаляет, то это нужно отразить во всех вторичных индексах, для чего выполняется чтение полной записи из первичного индекса и удаление старых ключей из других индексов. Иначе нарушается консистентность.

3.3 Проблема обновления индексов

Способность LSM дерева хранить множество версий одного ключа позволяет избегать скрытых чтений на такие операции как *REPLACE* и *DELETE*, которые не могут нарушить консистентность на таблице с единственным индексом и без внешних ключей (*FOREIGN KEY*). Эти операции называются *слепой записью* [12]. Слепая запись всегда значительно быстрее, чем операции вроде *INSERT* или *UPDATE*, которые в худшем случае обращаются к диску, чтобы проверить ограничения уникальности, или чтобы прочитать полную запись для выполнения обновления.

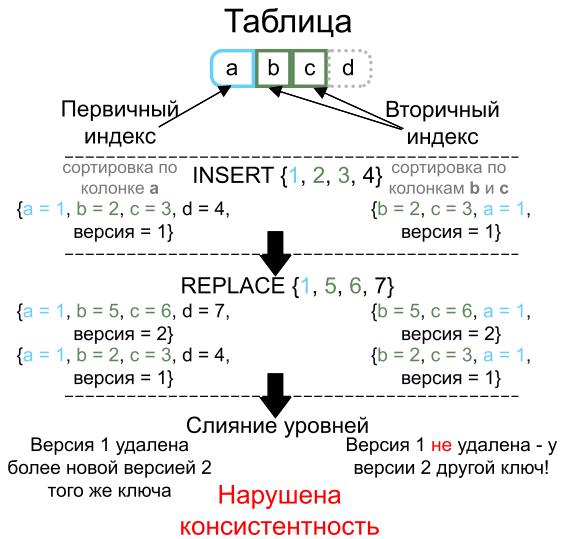


Рис. 2: Пример нарушения консистентности в LSM индексах

Проблема обновления индексов таблицы на LSM деревьях состоит в том, что при наличии вторичных индексов слепая запись становится невозможна ни для каких операций. На рисунке 2 изображен пример нарушения консистентности, к которому может привести слепой *REPLACE* во вторичный индекс. В примере у таблицы 4 колонки: первая - ключ первичного индекса, вторая и третья - ключ вторичного индекса, и четвертая не используется в индексах. Перед выполнением *REPLACE*{1, 5, 6, 7} должен прочесть старую запись из первичного индекса по ключу {1}, извлечь вторичный ключ {2, 3}, удалить его из вторичного индекса, и только затем вставлять везде новую запись. Если старую запись из вторичного индекса не удалить явно, то во время слияния уровней она становится мусором, у которого больше нет ссылки на полную запись в первичном индексе. Но и удалить этот мусор нельзя, так как более новая запись с тем же первичным ключом имеет другой вторичный ключ. Сравнить их по первичному ключу в общем случае невозможно, так как индекс отсортирован в первую очередь по вторичному ключу, и во время слияний эти записи могут никогда не встретиться.

Поскольку доступ к диску всегда намного медленнее, чем доступ к памяти, то наличие вторичных индексов делает версионность LSM дерева бесполезной - неявное удаление старых версий ключа более новыми версиями не работает.

4 Обзор существующих способов ускорения записи

Цели обзора:

- показать, что проблема скрытых чтений появилась давно, действительно актуальна до сих пор, и не только на LSM деревьях;
- вычленить из существующих алгоритмов идеи, которые были бы применимы для поставленной задачи;
- учесть ошибки обозреваемых решений.

4.1 Отложенные обновления индексов на В-деревьях

Необходимость ускорения обновления индексов стала актуальна еще до изобретения LSM дерева, когда повсеместно для хранения использовались В-деревья и механические жесткие диски. В работе [16] (Analysis of a deferred and incremental update strategy for secondary indexes) 1991 года предлагается способ откладывания обновлений индексов до выполнения запроса по ключам, по которым выполнялось обновление.

Идея решения в том, чтобы избавиться не от скрытых чтений, а от случайной записи в В-дерево. Дело в том, что обновление В-дерева всегда приводит к точечной записи на диск, что в худшем случае вызывает перебалансировку дерева.

Решение состоит в том, что после прочтения старой записи с диска и применения операций обновления в памяти, новая запись попадает на сразу в В-дерево, а в некоторый специальный файл изменений (differential file), который создается для каждого вторичного индекса. В файле изменений записано, какой вторичный ключ до какого нового значения был обновлен, и по какому первичному ключу искать полную запись в первичном индексе.

Когда выполняется поиск по вторичному индексу, то сперва нужный ключ ищется в файле изменений. Если там нашлась запись об этом ключе, и пользователь запрашивал только колонки вторичного индекса, то они извлекаются из этой записи, и на этом поиск окончен. Если же нужны другие столбцы, то из записи берется первичный ключ, и по нему выполняется поиск полной записи. В системе есть некоторый фоновый процесс, который занимается применением файлов изменений в В-деревья индексов.

Работа [16] - это не первое место, где появляются файлы изменений. Первое их упоминание находится в одноименной работе Differential Files [17]. Файлы изменений представляются как некоторый способ версионирования, например, В-дерева, при котором в файлах индексов могут оставаться старые данные, а новые уже лежат в файлах изменений.

Но такая архитектура не позволяет хранить множество версий одной и той же записи, не дает избавиться от скрытых чтений, не уменьшает издержек на запись новых данных и удаления старых данных из В-деревьев. В противовес В-деревьям можно сказать, что LSM деревья полностью состоят из файлов изменений.

Однако идея откладывать обновления, чтобы избавиться от самой дорогой операции В-дерева, может быть перенесена на LSM дерево, чтобы избавиться от его самой дорогой операции - скрытого чтения.

4.2 Второй уровень индексов на В-деревьях

В 2013 году вновь была поднята проблема обновления вторичных индексов в работе [19] (Making Updates Disk-I/O Friendly Using SSDs). Авторы рассуждают о том, что все более актуальной становится проблема выполнения аналитических запросов по мультиверсионной БД, находящейся под большой нагрузкой на запись.

Мультиверсионные БД хранят множество версий одних и тех же записей, подобно LSM дереву, но только в отличие от LSM в них версионность диктуется требованиями приложения, а не устройством структуры хранения данных - данные не удаляются никогда, или удаляются крайне редко. Это типичный сценарий финансовых приложений. Устаревшие версии должны быть доступны для чтения.

Чем более изощренные и разнообразные требуются запросы на чтения, тем больше нужно поддерживать индексов на одну и ту же таблицу, включая старые версии, а обилие индексов сильно ударяет по скорости вставки. SSD диски хотя и могут помочь с ускорением обновлений индексов, будучи на порядок быстрее HDD во всем, все еще значительно дороже магнитных жестких дисков, и иногда не могут использоваться для хранения большого объема данных.

Авторы [19] предлагают использовать для хранения индексов сразу два диска - HDD для индексов и данных, и SSD для хранения дополнительного "индекса по индексам" который хранит ссылку на самую актуальную версию каждой записи (ее LID - Logical record identifier) в паре с RID (Physical row identifier), по которому можно быстро найти данные на HDD. В результате на SSD каждая запись занимает лишь несколько байт для LID, что позволяет не тратить дорогой SSD на сами записи. LID вычисляется как хеш от всей записи.

Индексы на HDD в такой архитектуре отделены от данных, и хранят только LID. За счет этого они значительно меньше, и не индексируют устаревшие данные. Когда происходит вставка новой записи, она дописывается в конец некоторого списка на HDD, от нее вычисляются LID и RID. Затем LID попадает в индексы на том же HDD и на SSD в таблицу соответствия LID и RID.

При этом скорость обновления таблицы не зависит от количества индексов - LID в новые индексы вставляется отложенно. Как результат, скорость обновления не меняется с увеличением числа индексов, как изображено на рисунке 3, а скорость чтений замедляется незначительно, на единицы процентов согласно графику на рисунке 4.

Рассмотренное решение помогает решить проблему обновления множества индексов, но никуда не исчезают ни скрытые чтения, которые теперь дополнительно читают LID индекс на SSD диске, ни случайная запись на диск. Кроме того, такой подход работает только на В-деревьях и не позволяет избавиться от записи на диск даже при помощи

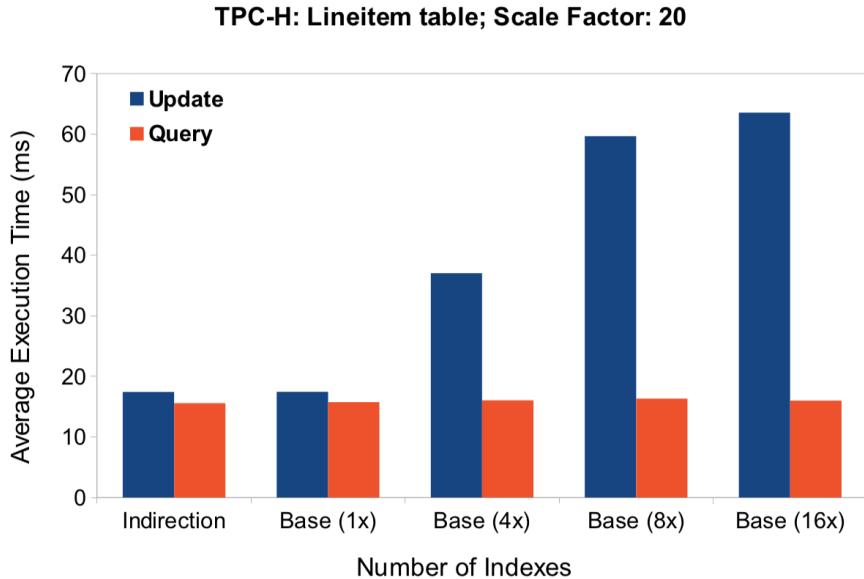


Рис. 3: Скорость обновления обычных индексов, и с дополнительным индексом на SSD

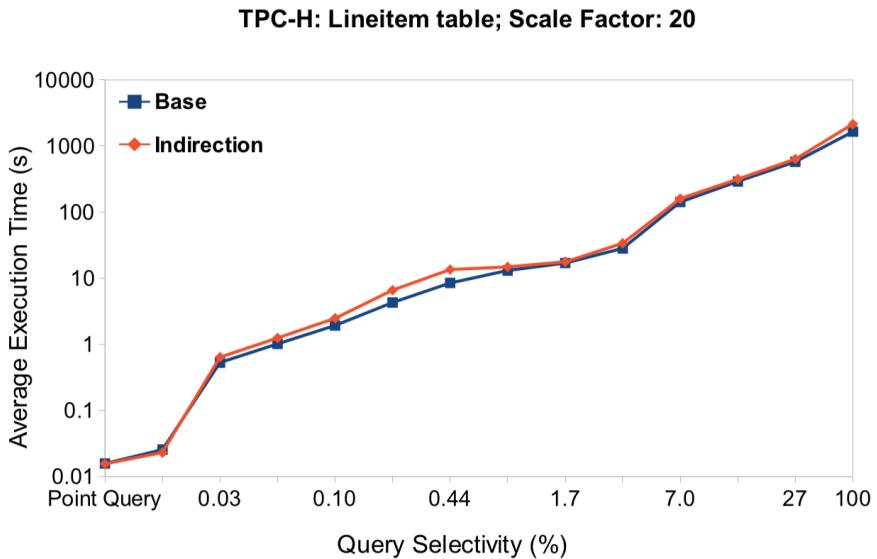


Рис. 4: Скорость чтения обычных индексов, и с дополнительным индексом на SSD

LSM деревьев из-за того, что при любом изменении данных нужно узнать RID новой записи, который можно получить только выполнив запись на диск.

4.3 Тонкая настройка LSM дерева

Кроме изобретения новых алгоритмов или модификации уже разработанных существует другой способ ускорить запись, который введен в работе [18] (Towards Accurate and Fast Evaluation of Multi-Stage Log-structured Designs, 2016). Он заключается в более точном подборе системных параметров. Конкретно в [18] рассматриваются различные параметры для оптимизации LSM дерева и его реализаций таких как LevelDB и RocksDB.

За счет подбора более оптимального соотношения размеров уровней LSM дерева, ме-

тода выбора диапазонов для выполнения слияния уровней, более точных оценок read и write amplification авторам удалось уменьшить стоимость вставок в LevelDB и RocksDB на 9.4-26.2% и 32% соответственно.

В работе утверждается, что разработанный метод оценок системных параметров может применяться к любым MSLS (Multi-Stage Log-Structured) структурам, а не только к LSM деревьях в различных их реализациях.

Подбор параметров в реализациях LSM дерева, или можно сказать, его настройка, это действительно важно для высоконагруженных систем, но это не решает проблему скрытых чтений, хотя возможно и делает ее менее заметной. Действительно, уменьшение read amplification может ускорить скрытые чтения, что косвенно ускоряет обновления дерева, но это не решает проблему самого существования скрытых чтений.

5 Отложенное обновление вторичных индексов на LSM деревьях

В данной главе изложено детальное описание нового алгоритма отложенного обновления LSM индексов, разделенное на три секции, каждая из которых фокусируется на одной из самостоятельных частей алгоритма.

Ключевая идея алгоритма в том, что удаление старых данных из вторичных индексов может быть отложено до слияния уровней первичного индекса. Это позволяет избежать скрытых чтений на операции *REPLACE* и *DELETE* по первичному ключу, поскольку они используют скрытые чтения исключительно для поиска и явного удаления старых данных, а не для поддержания консистентности, если у таблицы все вторичные индексы не уникальны и нет внешних ключей.

Из этого вытекает, что область применимости алгоритма - таблицы с индексами на LSM деревьях, где все вторичные индексы не уникальны, и нет внешних ключей. В случае уникальных вторичных индексов алгоритм не применим, так как можно нарушить ограничение уникальности, если не читать старую версию записи. Например, пусть есть таблица с колонками $\{a, b\}$, уникальный первичный индекс на колонку $\{a\}$ и уникальный вторичный индекс на колонку $\{b\}$. Пусть в таблице содержится две записи: $\{1, 1\}$ и $\{2, 2\}$. Теперь выполнить операцию $REPLACE\{1, 2\}$ невозможно, так как это приводит к появлению дубликата во вторичном индексе - об этом можно узнать только прочитав старую запись.

REPLACE не всегда реализован таким образом, что он может либо вставить новую запись, либо заменить одну старую. Например, в MySQL 5.7 *REPLACE* в описанном выше примере удалит обе старые записи и вставит одну новую - то есть конфликты невозможны. Но чтения при таком поведении не могут быть отложены, поскольку при слиянии уровней первичного индекса будет невозможно понять, что запись с одним первичным ключом должна удалить запись с другим первичным ключом. Такие операции мульти-вставки и мульти-удаления не рассматриваются в настоящей работе.

Алгоритм отложенного обновления состоит из следующих самостоятельных частей:

1. Обновление нулевого уровня всех индексов. Это то, что происходит в самом начале при вставке или удалении, и из-за чего индексы рассинхронизируются до момента слияния уровней;
2. Слияние уровней, алгоритм которого теперь отличается у первичного и вторичного индексов. Первичный индекс во время формирования нового уровня LSM дерева определяет записи, которые подлежат удалению, извлекает из них вторичные ключи, и вставляет их удаления как новый уровень в LSM дерево каждого вторичного индекса. Вторичный индекс учитывает этот новый уровень с удалениями при слиянии своих уровней;
3. Чтение вторичного индекса, при котором теперь возможно прочтение записей,

уже удаленных в первичном индексе, но пока что не удаленных из вторичного. Существование каждой записи, прочитанной из вторичного индекса, проверяется через первичный. Это выполнялось бы в любом случае, поскольку вторичный индекс хранит неполные записи, которые пользователю в общем случае нельзя вернуть.

5.1 Вставка и удаление

Сначала рассматривается алгоритм отложенного *REPLACE*, и затем очень схожий с ним *DELETE*. Откладывание операции *INSERT* невозможно, так как необходимо чтение первичного индекса для проверки уникальности. *UPDATE* нельзя отложить, так как в общем случае требуются части старой записи для построения новой.

Пусть есть таблица с одним первичным индексом и несколькими вторичными, и над ней выполняется операция *REPLACE*. Запись в *REPLACE* вставляется в нулевой уровень каждого индекса как есть, без скрытых чтений и удалений чего-либо откуда-либо.

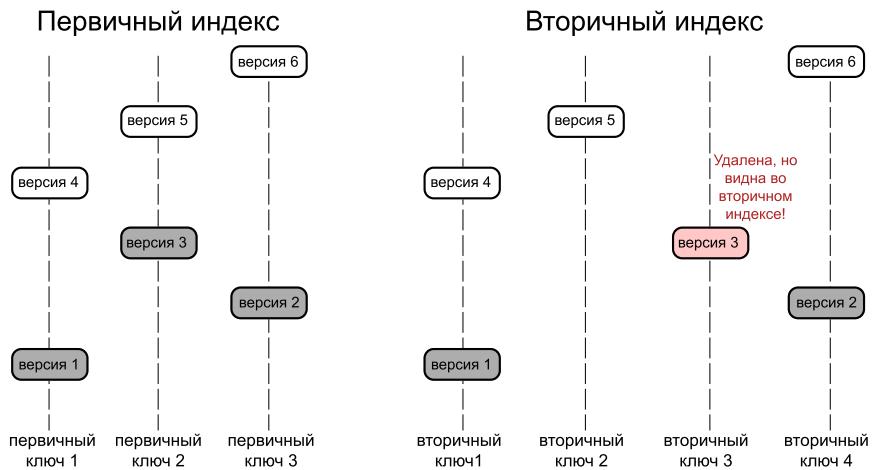


Рис. 5: Отложенное обновление LSM индексов

При этом возможно возникновение такой ситуации, что некоторые записи уже перекрыты более новыми версиями в первичном индексе, но все еще актуальны в некоторых вторичных индексах, как изображено на рисунке 5. В изображенном на нем примере в таблице есть запись с версией 3, первичный ключ которой равен записи с более новой версией 5, но вторичные ключи у них отличаются. Из-за этого версия 3 считается устаревшей в первичном индексе, но все еще актуальна во вторичном. Такие записи называются **грязными**.

В следующей секции представлена часть алгоритма отложенного обновления, которая отвечает за удаление грязных записей при помощи слияния уровней.

5.2 Слияние уровней

5.2.1 Первичный индекс

Слияние уровней запускается по тем же признакам, что и в классическом LSM дереве - когда высота дерева становится слишком высока, или когда нарушено ограничение на соотношение размеров соседних уровней.

Алгоритм слияния почти в точности повторяет классическое дерево. Уровни объединяются в новый при помощи сортировки слияниями, во время которой удаляются записи со старыми версиями. Данные каждого уровня считываются с диска, чтобы можно было выполнить сортировку. И именно эти чтения во время слияния уровней заменяют скрытые чтения при обновлениях и удалениях. Фактически, скрытые чтения **откладываются** до этого момента, откуда и название алгоритма. Выигрыш в том, что во время слияния уровней чтения выполняются в любом случае, и потому можно это использовать вместо скрытых чтений. Более того, здесь чтения последовательно считывают большими пачками все данные каждого из объединяемых уровней, а не ищут в них конкретные ключи, как скрытые чтения. Последовательные чтения и на SSD, и на HDD гораздо быстрее случайных.

Из старых записей, которые при слиянии подлежат удалению, извлекаются все вторичные ключи. Затем для каждого вторичного индекса они сортируются в его порядке и записываются на диск как новый первый уровень его LSM дерева. Вторичные ключи попадают в этот уровень с пометкой, что это удаление, с сохранением версии оригинальной записи. Таким образом, новая процедура слияния уровней первичного индекса создает по одному новому уровню для каждого из индексов.

Шаг с сортировкой удаляемых записей в порядок каждого вторичного индекса необходим для того, чтобы соблюсти инвариант LSM дерева, согласно которому его уровни на диске должны быть отсортированы по его ключу. Иначе использовать такой уровень будет невозможно ни для поиска, ни для следующих слияний.

Метод сортировки таких "суррогатных" уровней зависит от реализации, которая может быть, например, одной из следующих:

1. "Быстрая сортировка" в оперативной памяти всех подлежащих удалению записей.

Минус очевиден - если удаляется много записей, то памяти может не хватить, и кроме того эту сортировку придется выполнить столько раз, сколько существует вторичных индексов. Но вариант отличается больше простотой, чем оптимальностью;

2. Сортировка деревом в оперативной памяти. Перед началом чтения объединяемых уровней создаются пустые деревья поиска в памяти для каждого вторичного индекса, сортируемые по их ключам. Дерево может быть бинарным, красно-черным или B+. По мере выполнения основной сортировки слияниями, каждая запись, определенная как устаревшая, кладется в каждое из созданных деревьев. Когда

слияние уровней первичного индекса окончено, деревья вторичных индексов с удаляемыми записями уже заполнены, и могут быть сброшены на диск в виде отсортированных массивов. Для такой задачи хорошо подойдет B+ дерево, у которого все его значения связаны в список - это позволяет за линейное времябросить его на диск как массив. Минус у этого варианта сортировки такой же, как у предыдущего - в память все деревья вторичных индексов могут не влезть;

3. Многошаговая сортировка слияниями на диске. Такая сортировка может быть использована при любых размерах объединяемых уровней, и при любом количестве записей, подлежащих удалению. Идея в том, чтобы использовать один из двух описанных ранее подходов для подмножеств удаляемых записей, а не для всех сразу. По мере слияния уровней первичного индекса удаляемые записи накапливаются в памяти пока не будет достигнут некоторый лимит по памяти или количеству записей. Накопленные записи в отсортированном виде записываются во временные файлы - по одному для каждого вторичного индекса. Эта процедура повторяется, пока не кончится слияние уровней первичного индекса. В результате у каждого вторичного индекса есть список временных файлов, в которых лежат отсортированные в его порядке удаляемые записи. Они сортировкой слияниями объединяются в один результирующий файл, который попадает в первый уровень LSM дерева своего индекса.

5.2.2 Вторичный индекс

Алгоритм слияния уровней вторичного индекса модифицирован, чтобы корректно обрабатывать удаления, полученные от первичного индекса. Особенность записи, помеченной как *DELETE* и полученной из первичного индекса в том, что она имеет ту же версию, что и грязная запись, которую надо удалить. В классическом LSM дереве такая ситуация невозможна - для каждой пары ключ-версия существует только одна запись.



Рис. 6: Слияние уровней вторичного LSM индекса

В новом алгоритме *DELETE*, встретив запись с такими же версией и ключом, должен при слиянии уровней заменить ее собой в новом уровне, поскольку это означает, что эта запись и все более старые в первичном индексе уже не актуальны. На рисунке 6 изображен пример слияния уровней вторичного индекса, один из которых получен

от первичного индекса. В этом примере первичный и вторичный индексы хранят две записи: $\{pk1, sk1\}$ и $\{pk1, sk2\}$. В первичном индексе эти записи хранятся на разных уровнях как разные версии одного ключа - $\{pk1\}$. Когда уровни объединяются, запись $\{pk1, sk1\}$ удаляется из первичного индекса, и попадает как $DELETE\{pk1, sk1, \text{версия} = 1\}$ в новый уровень вторичного с сохранением версии. Когда объединяются уровни вторичного индекса, $DELETE$ с версией 1 встречается с грязной записью с такими же версией и ключом, и заменяет ее собой.

5.3 Чтения

Чтение первичного индекса выполняется ровно так же, как в классическом LSM дереве, поскольку первичный индекс не содержит грязных записей. С точки зрения хранения данных первичный индекс не меняется никак.

Однако при чтении вторичного индекса необходимо принимать во внимание возможность найти грязную запись, которой уже нет в первичном индексе. Это проверяется чтением первичного индекса на каждую запись, найденную во вторичном. Если в первичном запись не найдена, то это означает, что она грязная, и пока что не удалена слиянием уровней вторичного индекса. Такие записи пропускаются, пока не будет найдена запись, существующая в обоих индексах.

Чтение первичного индекса на каждую запись вторичного индекса является недостатком алгоритма, однако в общем случае это же происходит как в классических LSM деревьях, так и в B-деревьях, так как записи вторичного индекса неполные, и не могут удовлетворить любой пользовательский запрос.

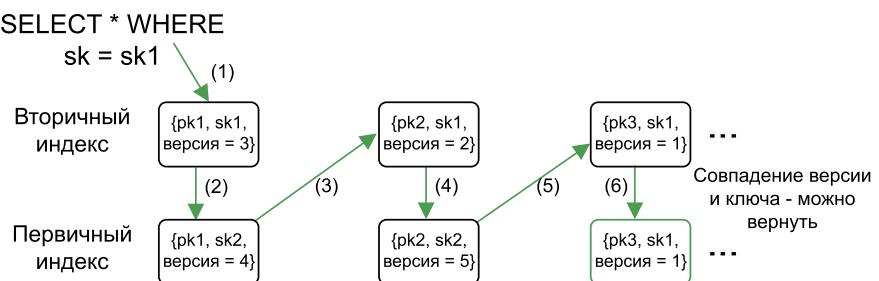


Рис. 7: Пример чтения вторичного индекса

На рисунке 7 чтение по вторичному ключу $\{sk1\}$ находит искомую запись с третьей попытки. Сначала прочитана запись $\{pk1, sk1\}$, как имеющая самую новую версию искомого ключа. При проверке в первичном индексе оказалось, что запись уже удалена, так как в первичном индексе по ключу $\{pk1\}$ была найдена запись с другой версией и другим вторичным ключом. Следующей проверяется запись $\{pk2, sk1\}$, которая при проверке в первичном индексе оказывается уже удаленной более новой записью $\{pk2, sk2\}$. Следующей нашлась запись вторичного индекса $\{pk3, sk1\}$, и оказалось, что в первичном индексе она все еще есть - ее можно возвращать пользователю.

5.4 Реализация

Реализация алгоритма выполнена на основе реализации LSM деревьев в СУБД Tarantool, где она называется Vinyl.

Vinyl хранит данные в индексах на LSM деревьях. Каждый индекс разбит на диапазоны значений своего ключа. Все уровни LSM дерева, кроме нулевого, состоят из подуровней, каждый из которых представлен одним файлом. Слияние выполняется на подуровнях отдельно взятого диапазона ключей. Запись нулевого уровня на диск и слияния уровней выполняются в Tarantool параллельно, в рабочих потоках, в то время как обработка транзакций выполняется в одном главном потоке, который не делает никаких долгих операций вроде работы с диском или сетью - все эти действия выполняются в отдельных потоках.

Каждый подуровень, представленный файлом, состоит из страниц. Для любой страницы в памяти хранится ее метаинформация: минимальный и максимальный ключи, минимальная и максимальная версии среди всех ключей. При выполнении поиска по ключу в памяти можно быстро узнать конкретную страницу, где его искать, вместо сканирования всего подуровня. Размер страницы - это компромисс между скоростью поиска и размером метаданных о страницах.

Для каждого подуровня LSM дерева в Tarantool хранится bloom-фильтр с несколькими улучшениями: уменьшение количества хеширований [13] и блочная структура [14].

Поток обработки транзакций состоит из сопрограмм (coroutine) называемых файберами, реализованных на языках C и Assembler. Когда главному потоку требуется выполнение долгой операции, такой как сброс нулевого уровня LSM дерева на диск, или доступ к сети, он отправляет запрос одному из рабочих потоков, и переключает выполнение на другой файбер, в то время как старый остается в режиме ожидания ответа от рабочего потока. Преимущества разбиения одного главного потока на файбера вместо множества параллельных потоков следующие:

- Практически не требуется синхронизировать доступ к критическим секциям блокировками поскольку их нет, кроме очередей задач между главным потоком и рабочими;
- Поток использует процессорное время по максимуму, не тратя его на ожидание конца блокировки.

Реализация алгоритма отложенного обновления состоит из трех частей, так же как и сам алгоритм: откладывание чтений на *REPLACE* и *DELETE*, слияние уровней, явные чтения.

Откладывание *REPLACE* и *DELETE* самая простая часть реализации, и состоит лишь в удалении кода, отвечающего за скрытые чтения. *REPLACE* сразу помещается в нулевые уровни всех индексов, а *DELETE* только в нулевой уровень первичного индекса.

Слияние уровней первичного индекса - самая сложная часть реализации. В Vinyl до откладывания обновлений оно было реализовано так:

1. Главный поток определяет, что необходимо выполнить слияние (например, нарушено соотношение размеров уровней, или их стало слишком много), и отправляет запрос рабочему потоку с информацией о том, какие уровни надо объединить, и в каких файлах они хранятся;
2. Рабочий поток, получив запрос, начинает выполнять задачу при помощи сортировки слияниями файлов уровней. Использованные уровни и их файлы не удаляются и даже не изменяются в рабочем потоке, так как они все еще могут параллельно использоваться чтениями в главном потоке. Закончив работу, рабочий поток имеет один новый уровень в одном новом файле. Он уведомляет главный поток о том, что слияние завершено;
3. Спустя некоторое время один из файберов главного потока обрабатывает результаты, полученные рабочим потоком. Новый уровень кладется вместо старых уровней в LSM дерево индекса, а старые удаляются вместе со своими файлами.

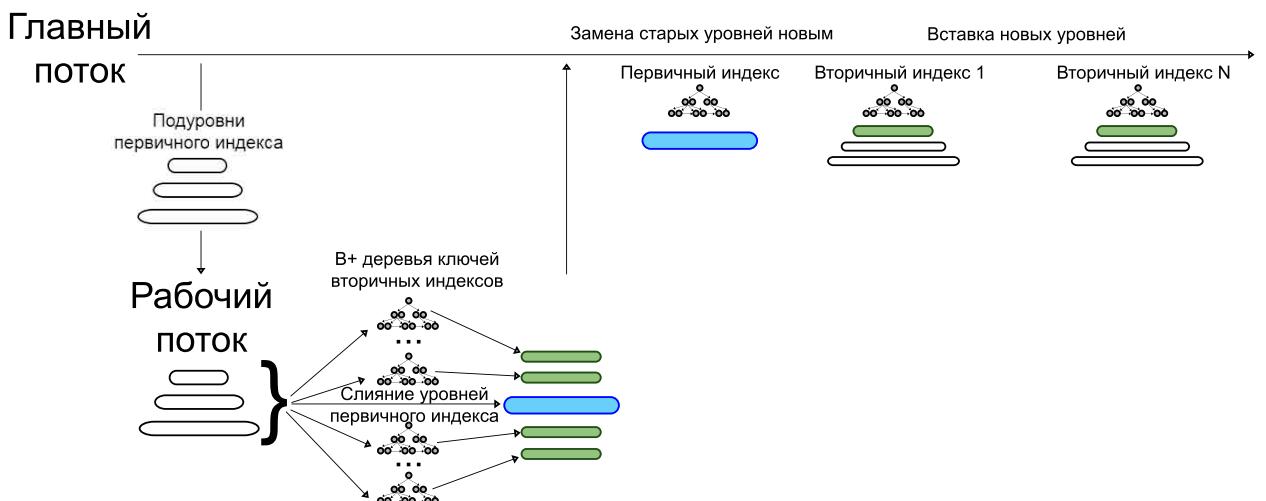


Рис. 8: Реализация слияния уровней первичного индекса

После реализации отложенного обновления второй и третий шаги слияния уровней становятся совершенно другими. На рисунке 8 изображена схема новой процедуры слияния. Когда первичный индекс определяется как нуждающийся в слиянии уровней, главный поток отправляет рабочему потоку не только информацию о том, что нужно объединять, но и данные вторичных индексах - сколько их, какие у каждого ключевые поля - это требуется, чтобы во время слияния из устаревших записей можно было извлечь вторичные ключи.

Когда уровни первичного индекса сортируются, устаревшие записи с номерами своих версий попадают в B+ деревья, созданные для каждого вторичного индекса. По завершении слияния все B+ деревья сбрасываются в виде отсортированных массивов

в отдельные файлы. Сброс выполняется за линейное время за счет того, что значения в B+ деревьях хранятся в листовых вершинах, которые связаны в список. На этом рабочий поток уведомляет главный поток о завершении задачи.

Один из файлов главного потока, получив уведомление от рабочего, заменяет в первичном индексе объединенные уровни на один новый, а в каждый вторичный индекс помещается соответствующий ему уровень, содержащий удаления старых данных. Новый уровень вторичного индекса становится именно первым, продвигая текущий первый уровень на второе место, так как он может содержать удаления в том числе тех данных, которые находятся на текущем первом уровне.

Реализация чтений изменена таким образом, чтобы учитывать существование грязных записей. Во-первых, обрабатывается ситуация, когда для одной версии и одного ключа находятся сразу две записи - удаление и вставка. В этом случае согласно алгоритму они обе и все более старые пропускаются, так как это означает, что в первичном индексе они уже устарели. Во-вторых, для прочих записей при поиске их полных версий в первичном индексе учитывается то, что в первичном запись может быть не найдена - это означает, что она грязная, и ее удаление еще не было получено от первичного индекса. Такая запись пропускается, а итерирование продолжается дальше. Одна из особенностей реализации, не связанная с алгоритмом, в том, что уровни вторичных индексов, содержащие удаления, неотличимы от прочих уровней того же индекса. За счет этого при чтении можно увидеть, что некоторая запись уже удалена в первичном индексе, без ее поиска в нем. Однако это же снижает скорость чтения вторичного индекса, так как требуется читать больше уровней.

Существует другой способ реализации чтения вторичного индекса, который имеет свои особые преимущества и недостатки. Ключевая идея в том, чтобы отличать друг от друга уровни с данными, и уровни с удалениями, полученные от первичного индекса. И при явных чтениях не пользоваться вторыми. Это ускоряет чтения тех ключей, которые не устарели, поскольку нужно сканировать меньше уровней. Кроме того, это значительно упрощает реализацию чтений, так как ситуация встречи двух записей с одной версией и одним ключом становится невозможна. Но чтение ключей, которые часто меняются, становится медленнее, так как будут происходить дополнительные проверки их существования в первичном индексе даже если их удаление уже получено вторичным.

5.5 Оценка сложности алгоритма

В данной секции оценивается математическая сложность нового алгоритма в сравнении с оригинальным LSM деревом, без отложенных обновлений. В оценках использу-

зуются следующие обозначения:

- N – общее количество записей на всех уровнях,
- b – фактор ветвления B+ дерева нулевого уровня,
- r – коэффициент размеров уровней,
- K – общее количество всех индексов таблицы.

Согласно определению коэффициента r , уровень i не менее чем в r раз больше, чем уровень $i - 1$, так что высота дерева оказывается равна $O(\log_r N)$ [12]. Пусть это значение будет обозначаться через lc (level count).

Сложность вставки в нулевой уровень зависит от его размера, который может быть вычислен по N и lc . Определив его как m , можно вычислить значение используя формулу суммы геометрической прогрессии:

$$\begin{aligned} N &= m + m * r + m * r^2 + \dots + m * r^{lc}, \\ N &= \frac{m(1 - r^{lc+1})}{1 - r}, \\ m &= \frac{N(1 - r)}{1 - r^{lc+1}}. \end{aligned}$$

В рамках настоящей работы, нулевой уровень - это B+ дерево, сложность поиска по которому - $O(\log_b m)$. С использованием полученных значений получена следующая оценка сложности для операций обновления таблицы:

Сложность REPLACE:

$$O(\log_b m) + O(\log_r(N - m)) + O(\log_b m) * (2K - 1)$$

Скрытое чтение старой записи - это ее поиск в памяти ($O(\log_b m)$) и на диске ($O(\log_r(N - m))$). В худшем случае, найдена старая запись с таким же первичным ключом, что и в новой записи: тогда происходит вставка двух записей ($2(K - 1)$) в нулевой уровень каждого вторичного индекса ($O(\log_b m)$). В первичном индексе новая запись сама удаляет старую, так как будет вставлена с таким же первичным ключом, но более новой версией, и потому туда не вставляется явное удаление. Всего записей: $2(K - 1) + 1 = 2K - 1$.

Сложность DELETE:

$$O(\log_b m) + O(\log_r(N - m)) + O(\log_b m) * K$$

Скрытое чтение старой записи - это поиск в памяти и на диске в первичном индексе ($O(\log_b m) + O(\log_r(N - m))$). В худшем случае она будет найдена, и удаление вставляется в нулевые уровни всех индексов ($O(\log_b m) * K$).

После реализации отложенного обновления все скрытые чтения исчезают.

Сложность отложенного REPLACE:

$$O(\log_b m) * K$$

Новая запись вставляется в нулевые уровни всех индексов - нет удалений, нет чтений: все это отложено до слияния уровней. Отложенный *REPLACE* минимум в два раза быстрее, чем классический, согласно следующим вычислениям:

$$\begin{aligned} \frac{O(\log_b m) + O(\log_r(N - m)) + O(\log_b m) * (2K - 1)}{O(\log_b m) * K} &= \\ \frac{O(\log_b m * 2K) + O(\log_r(N - m))}{O(\log_b m) * K} &= \\ 2 + \frac{O(\log_r(N - m))}{O(\log_b m) * K}. \end{aligned}$$

Размер нулевого уровня ограничен размером оперативной памяти. Индексы и их количество фиксируются в начале работы программы, и если и меняются, то очень редко. Фактор ветвления нулевого уровня LSM дерева - константа. То есть знаменатель $O(\log_b m) * K$ можно рассматривать как константу. Числитель содержит N - это значение может быть огромно в интенсивных по записи задачах, и чем оно больше, тем значительнее разница в скоростях вставок классического алгоритма обновления, и нового с отложенными удалениями. Например, при такой оценке параметров:

$$\begin{aligned} r &= 3, \\ b &= 10, \\ m &= 10^5, \\ N &= 10^8, \\ K &= 5. \end{aligned}$$

Отложенный *REPLACE* уже в 2.55 раз быстрее классического даже в теории. На практике, рост скорости еще больше, как это видно в экспериментальном исследовании, за счет того, что большая часть N хранится на диске, а доступ к нему всегда сильно медленнее, чем к памяти. То есть числитель $O(\log_r(N - m))$ зависит от скорости работы с диском. В этом причина того, что даже на небольших таблицах возможен прирост скорости до 10 раз.

Сложность отложенного DELETE:

$$O(\log_b m)$$

Удаление вставляется в нулевой уровень первичного индекса. Поскольку задействован только один индекс, откладывание *DELETE* ускоряет его минимум в $K + 1$ раз, и чем

большая часть LSM дерева находится на диске, тем больше относительный выигрыш:

$$\frac{O(\log_b m) + O(\log_r(N - m)) + O(\log_b m) * K}{O(\log_b m)} =$$
$$\frac{O(\log_b m) * (K + 1) + O(\log_r(N - m))}{O(\log_b m)} =$$
$$K + 1 + \frac{O(\log_r(N - m))}{O(\log_b m)}$$

6 Экспериментальное исследование

Реализация алгоритма протестирована двумя бенчмарками: микробенчмарком, на котором разница между обычными и отложенными обновлениями видна особенно сильно, и Linkbench [15], на котором видно замедление чтений. Оба бенчмарка выполнены на одной машине с Apple SSD SM0512L, Intel i7 2.7Ghz, 4 ядра.

6.1 Микробенчмарк

Микробенчмарк сравнивает стоковый Tarantool 1.7 и модифицированный отложенными обновлениями. Бенчмарк получает наибольшую разницу скорости, нагружая запущенную БД запросами *DELETE* и *REPLACE* по таблице с неуникальными вторичными индексами. Схема БД состоит из одной таблицы на Vinyl движке, одного первичного индекса по первой колонке, и четырех вторичных индексов, по одному на оставшиеся поля. В SQL синтаксисе определение выглядело бы так:

```
create table test
(field1 unsigned integer primary key,
 field2 unsigned integer, field3 unsigned integer,
 field4 unsigned integer, field5 unsigned integer);
create not unique index on test(field2);
create not unique index on test(field3);
create not unique index on test(field4);
create not unique index on test(field5);
```

В процессе работы проводилось измерение количества обработанных запросов в секунду - RPS (requests per second), количество записей на диске, и количество сбросов нулевого уровня первичного LSM индекса на диск.

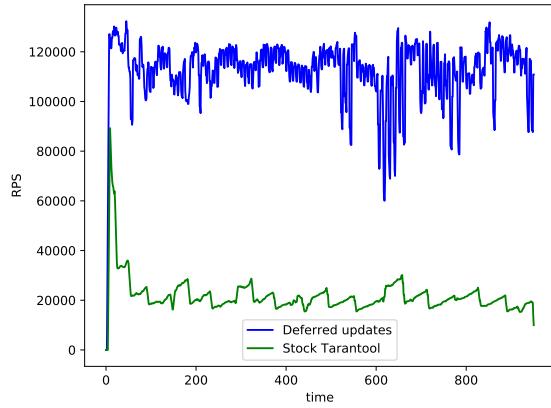


Рис. 9: Микробенчмарк, RPS

Нулевой уровень LSM дерева ограничен 128 мегабайтами. Нагрузка генерируется 4 клиентскими соединениями на той же машине, что и запущенная СУБД. Каждое

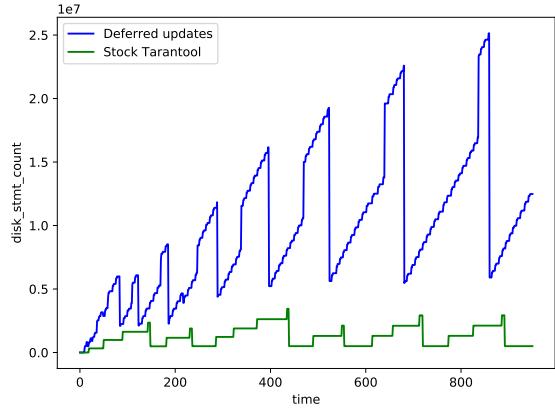


Рис. 10: Микробенчмарк, количество записей на диске

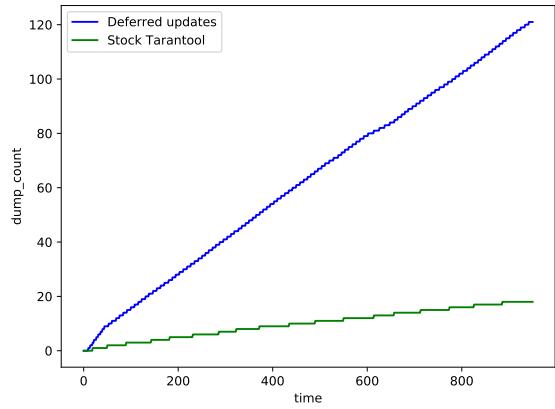


Рис. 11: Микробенчмарк, количество сбросов нулевого уровня

соединение генерирует *DELETE* по первичному ключу и *REPLACE* запросы пакетами по 1-500 обновлений. Соотношение удалений и вставок - 1 к 1. Поскольку сперва таблица была заполнена миллионом записей, и затем примерно одинаковое их число удаляется и вставляется, то все время нагрузки суммарно в таблице остается миллион записей, не считая устаревших версий еще не удаленных слияниями уровней.

Два рабочих потока занимаются сбросом нулевых уровней LSM деревьев на диск и слияниями уровней. Минимальное соотношение размеров соседних уровней - 3.5, максимальное число подуровней в одном уровне - 2. Блюм-фильтр установлен в 5 ложноположительных срабатываний. Размер страницы - 8 килобайт, в которые помещается около 160 записей.

Результаты измерений скорости (RPS) представлены на рисунке 9.

В таблице 1 содержатся агрегированные показания RPS, которые показывают, что на небольшой таблице с миллионом записей RPS вырос в 6 раз в среднем.

6.2 LinkBench

Откладывание обновлений замедляет чтения, так как существование ключа, прочитанного из вторичного индекса, должно быть проверено через поиск его записи в

Таблица 1: Микробенчмарк, агрегированный RPS

	Отложенные обновления	Классические обновления
Среднее	112343 з/с	21681 з/с
Максимум	132316 з/с	89292 з/с
Медиана	114772 з/с	20097 з/с

первичном, чтобы исключить грязные записи. Степень замедления зависит от типа нагрузки - чем больше обновлений вторичных ключей вместо вставки новых данных или изменения неиндексируемых данных, тем больше записей становятся грязными, и тем медленнее чтения. На рисунке 12 показаны результаты работы Linkbench. Большинство запросов в Linkbench - это чтения (более 70%), а обновления в основном меняют индексируемые данные. При таком типе нагрузки чтения замедлились более чем в два раза. На том же графике изображены результаты Linkbench СУБД MySQL с использованием реализации LSM деревьев под названием MyRocks.

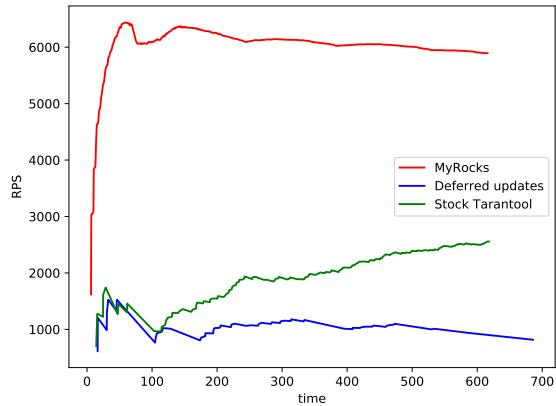


Рис. 12: LinkBench RPS

7 Заключение

В рамках выполнения настоящей работы выполнены следующие задачи:

- Разработан алгоритм отложенного обновления LSM дерева, который позволяет не делать скрытых чтений на операции *REPLACE* и *DELETE* даже при наличии связности с другими LSM деревьями. Это типичный сценарий хранения индексов таблицы в LSM деревьях;
- Выполнена реализация алгоритма на основе СУБД Tarantool, которая доступна на github: <https://github.com/tarantool/tarantool>;
- Выполнены математические и экспериментальные оценки увеличения скорости выполнения отложенных обновлений в сравнении с обычными. Откладывание обновлений согласно математическим выкладкам ускоряет операцию *REPLACE* в более чем два раза, а операцию *DELETE* в число равное количеству индексов. Причем при увеличении размера таблицы относительное ускорение только растет. Однако в теоретических оценках не учитывалась разница в скоростях доступа к диску и памяти, из-за чего практические результаты оказались значительно лучше - на таблице из миллиона записей скорость *REPLACE* и *DELETE* увеличивалась до 10 раз.

Остались открытыми для исследования вариации разработанного алгоритма, а именно:

- При выполнении слияния уровней первичного индекса существует множество способов сортировки удаляемых записей по ключам вторичных индексов. Самой масштабируемой, хоть и не самой быстрой, представляется сортировка слияниями, при которой по мере чтения первичного индекса данные сортируются по вторичным ключам в памяти до некоторого предела, после чего сбрасываются в файлы в виде отсортированных массивов. Так происходит до конца слияния уровней первичного индекса, после чего для каждого вторичного индекса остается провести сортировку слияниями накопившихся для него файлов с удаляемыми записями;
- При выполнении чтений вторичных индексов существует возможность игнорировать уровни, полученные от первичного индекса для удаления мусора. Это может существенно ускорить чтения, когда основная часть обновлений - это вставка новых записей, что и является самой сильной стороной LSM дерева как структуры данных.

Список литературы

- [1] O’Neil P. et al. The log-structured merge-tree (LSM-tree) //Acta Informatica. – 1996. – Т. 33. – №. 4. – С. 351-385.
- [2] Comer D. Ubiquitous B-tree //ACM Computing Surveys (CSUR). – 1979. – Т. 11. – №. 2. – С. 121-137.
- [3] Agrawal N. et al. Design Tradeoffs for SSD Performance //USENIX Annual Technical Conference. – 2008. – Т. 8. – С. 57-70.
- [4] Wang P. et al. An efficient design and implementation of LSM-tree based key- value store on open-channel SSD //Proceedings of the Ninth European Conference on Computer Systems. – ACM, 2014. – С. 16.
- [5] Yang F. et al. Optimizing NoSQL DB on flash: A case study of RocksDB //Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC- ScalCom), 2015 IEEE 12th Intl Conf on. – IEEE, 2015. – С. 1062-1069.
- [6] Wang P. et al. An efficient design and implementation of LSM-tree based key- value store on open-channel SSD //Proceedings of the Ninth European Conference on Computer Systems. – ACM, 2014. – С. 16.
- [7] Сайт базы данных Tarantool [Электронный ресурс] : сайт содержит документацию всех выпущенных версий Tarantool. – Режим доступа: <https://tarantool.io>. - Загл. с экрана.
- [8] Lersch L. et al. An analysis of LSM caching in NVRAM //Proceedings of the 13th International Workshop on Data Management on New Hardware. – ACM, 2017. – С. 9.
- [9] Dong S. et al. Optimizing Space Amplification in RocksDB //CIDR. – 2017.
- [10] Liu L. C. H., Yoneda K. Secondary index search : пат. 6266660 СИЛА. – 2001.
- [11] Owens M., Allen G. SQLite. – Apress LP, 2010.
- [12] Ren K. et al. SlimDB: a space-efficient key-value storage engine for semi-sorted data //Proceedings of the VLDB Endowment. – 2017. – Т. 10. – №. 13. – С. 2037-2048.
- [13] Kirsch A., Mitzenmacher M. Less hashing, same performance: building a better bloom filter //European Symposium on Algorithms. – Springer, Berlin, Heidelberg, 2006. – С. 456-467.

- [14] Putze F., Sanders P., Singler J. Cache-, hash-and space-efficient bloom filters //International Workshop on Experimental and Efficient Algorithms. – Springer, Berlin, Heidelberg, 2007. – C. 108-121.
- [15] Armstrong T. G. et al. LinkBench: a database benchmark based on the Facebook social graph //Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data. – ACM, 2013. – C. 1185-1196.
- [16] Omiecinski E., Liu W., Akyildiz I. Analysis of a deferred and incremental update strategy for secondary indexes //Information Systems. – 1991. – T. 16. – №. 3. – C. 345-356.
- [17] Severance D. G., Lohman G. M. Differential files: their application to the maintenance of large databases //ACM Transactions on Database Systems (TODS). – 1976. – T. 1. – №. 3. – C. 256-267.
- [18] Lim H., Andersen D. G., Kaminsky M. Towards Accurate and Fast Evaluation of Multi-Stage Log-structured Designs //FAST. – 2016. – C. 149-166.
- [19] Sadoghi M. et al. Making updates disk-I/O friendly using SSDs //Proceedings of the VLDB Endowment. – 2013. – T. 6. – №. 11. – C. 997-1008.
- [20] Garcia-Molina H. Database systems: the complete book. – Pearson Education India, 2008.