УДК 004.651.4
DOI: 10.25559/SITITO.18.202202.337-352

Научная статья

LSM-индекс с общими компонентами для индексации хронологических данных

А. М. Неганов
ФГАОУ ВО «Московский физико-технический институт (национальный исследовательский университет)», г. Долгопрудный, Российская Федерация
Адрес: 141701, Российская Федерация, Московская область, г. Долгопрудный, Институтский переулок, д. 9
neganovalexey@gmail.com

Аннотация

Современная эпоха характеризуется взрывным ростом объема хранимых и обрабатываемых данных. В таких условиях особенно важными становятся производительность хранения и индексации данных во внешней памяти. Существует множество приложений, требующих доступа к истории изменения данных, таких как приложения резервного копирования, информационные системы в банковской сфере, медицине и т. д. Данные с историей, т. е. объекты, имеющие время жизни в определенной шкале времени, называются хронологическими. Предлагается новый алгоритм индексации хронологических данных, LSM с общими компонентами, который сочетает в себе возможность хранения части индекса во внешней памяти, эффективность операций записи во внешнюю память (запись всегда выполняется в последовательном режиме), не зависящее от количества версий объектов время выполнения запросов по диапазону ключей при фиксированном времени, а также возможность разделения «исторических» («холодных») данных и их хранения на отдельных носителях. Алгоритм был теоретически проанализирован, реализован и протестирован. Его поведение было изучено в сравнении с известными подходами для нескольких вариантов использования.

Ключевые слова: хронологические данные, резервное копирование, снапшот, LSM-дерево, вычисления с интенсивным использованием данных

Автор заявляет об отсутствии конфликта интересов.

Для цитирования: Неганов А. М. LSM-индекс с общими компонентами для индексации хронологических данных // Современные информационные технологии и ИТ-образование. 2022. Т. 18, № 2. С. 337-352. doi: https://doi.org/10.25559/SITITO.18.202202.337-352

© Неганов А. М., 2022

Контент доступен под лицензией Creative Commons Attribution 4.0 License.
The content is available under Creative Commons Attribution 4.0 License.

Original article

LSM Index with Common Components for Historical Data Indexing

A. M. Neganov
Moscow Institute of Physics and Technology (National Research University), Dolgoprudny, Russian Federation
Address: 9 Institutskiy per., Dolgoprudny 141701, Moscow Region, Russian Federation
neganovalexey@gmail.com

Abstract

The modern era is characterized by an explosive growth in the amount of data stored and processed. In such circumstances, the performance of storing and indexing of data in external memory become especially important. There are a variety of applications that require access to the history of data changes, such as backup applications, information systems in banking, medicine, etc. The data with history, i.e., objects that have a lifetime on a certain timescale, is called chronological data. A new algorithm for indexing chronological data is proposed, LSM with shared components, which combines the ability to store a part of the index in external memory, the efficiency of write operations to external memory (writes are always performed in sequential mode due to out-of-place update design), the complexity of key-range time-slice queries that does not depend on the number of object versions and the possibility of separating "historical" ("cold") data and storing it on separate media. The algorithm was analyzed theoretically, implemented and benchmarked. Its behavior was studied in comparison with known approaches for several use cases.

Keywords: historical data, backup, snapshot, LSM tree, data-intensive computing

The author declares no conflict of interest.

For citation: Neganov A.M. LSM Index with Common Components for Historical Data Indexing. *Sovremennye informacionnye tehnologii i IT-obrazovanie = Modern Information Technologies and IT-Education*. 2022; 18(2):337-352. doi: https://doi.org/10.25559/SITITO.18.202202.337-352

Введение

Существуют разнообразные приложения, требующие доступа к истории изменения данных в режиме реального времени, такие, как информационные системы в банковском деле, медицине и т. д. Это позволило ещё в 1977 году заявить, что будущее баз данных заключается в хранении полных журналов событий. С начала 1980-х годов проводились исследования по хранению и индексации данных с историей; тогда же появилось определение хронологической базы данных.

Современная эпоха характеризуется взрывным ростом числа хранимых и обрабатываемых данных [1], что привело к распространению новых парадигм в области хранения и индексации данных, получивших обобщённое название NoSQL [2]. В силу активных разработок в области NoSQL-хранилищ множество направлений исследования остаются незакрытыми. В частности, остаётся слабоизученной проблема индексации неструктурированных данных с историей с требованием высокой скорости записи.

Данная работа посвящена индексации данных с историей, т. е. объектов, обладающих не только именем (ключом), но и временем жизни на некоторой временной шкале, что, таким образом, представляет собой как открытую область теоретических исследований, так и практическую задачу высокой значимости.

В работе проанализированы существующие подходы к индексированию данных с историей в контексте их применимости к решению поставленной задачи.

Предложен новый алгоритм индексации данных с историей — LSM с общими компонентами, сочетающий эффективность операций записи во внешнюю память с не зависящим от количества версий записей временем выполнения запросов по диапазону ключей при фиксированном времени.

Проведено теоретическое исследование свойств алгоритма, создана экспериментальная реализация и исследовано его поведение в сравнении с известными подходами для некоторых сценариев использования.

Постановка задачи

Следуя принятому для хронологических баз данных подходу определим время как дискретное и представляющее собой строго возрастающую последовательность неотрицательных целых чисел $(t_n) = \{t_0, t_1, ...\}$. Элемент такой последовательности будем называть моментом времени. Числовые значения моментов времени могут быть как идущими подряд (например, 1, 2, 3, 4, 5 и т. д.), так и с пропусками (например, 3, 74, 1056, 9808 и т. д.). В связи с тем, что последовательность моментов времени монотонно возрастает по определению, множество моментов времени является линейно упорядоченным. Для каждого момента времени все прочие назовём предшествующими, если они имеют меньшее числовое значение, либо следующими, если они имеют большее числовое значение.

В некоторых приложениях моменты времени, определенные как элементы числовой последовательности, могут быть связаны с упорядоченными в том же порядке значениями физического или астрономического времени, либо показаниями системных часов компьютера. Также в некоторых приложениях указанные временные метки могут быть связаны с временными метками транзакций СУБД.

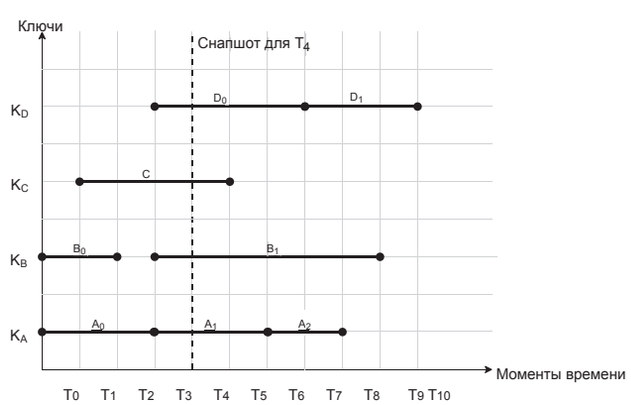
Рассмотрим набор именованных объектов (т. е. объектов обладающих уникальными ключами), так что для любого момента времени t для каждого объекта A определено, существует ли он в этот момент времени, а также его состояние в этот момент времени. Для каждого объекта A, идентифицируемого ключом k_A, назовём версией объекта триплет объекта и двух моментов времени (A, t_{As}, t_{Ae}), $t_{As} \le t_{Ae}$, так что для каждого момента времени t, такого что $t_{As} \le t \le t_{Ae}$, состояние объекта A остаётся прежним и для каждого объекта A множество его версий задано однозначно, т. е. интервалы $(t_{As}0, t_{Ae}0), (t_{As}1, t_{Ae}1), ...$ не пересекаются. Момент времени t_{As} здесь и далее именуется начальным, а t_{Ae} — конечным для версии (A, t_{As}, t_{Ae}).

Если перенумеровать транзакции добавления данных в систему, то в качестве задающих версию объекта моментов времени можно взять соответствующие номера транзакций. Будем говорить, что версия файла (A,i,j) соответствует моменту времени k, если $i \le k \le j$.

Если определить особый вид версий для представления удалённых файлов (подобно маркерам удаления в LSM-деревьях [3] или в Time-Split B-tree [4]), то можно принять эквивалентное определение версии объекта, как неизменяемого объекта (A,i), заданного парой ключ — момент времени, представляющего собой состояние файла A от заданного момента времени i до следующего по счёту j, для которого существует версия (A,j), $i < j$, или до конца отсчёта времени T.

Используя предложенную в [5] классификацию, примем, что входные данные являются ступенчато постоянными (*step-wise constant)*. Такой подход естественен, например, для приложений резервного копирования, когда появление новых данных чётко подразделяется на логические единицы создания очередных резервных копий, соответствующих состояниям обслуживаемой внешней системы.

Снапшотом номер k будем называть совокупность файловых версий, соответствующих моменту времени k, для всех возможных значений ключа. Такое определение вводится по аналогии с употреблением слова «снапшот» в теории хронологических баз данных [6], [7].



Р и с. 1. Частный случай совокупности версий объектов и снапшота

F i g. 1. A special case of a combination of object versions and a snapshot

Рисунок 1 иллюстрирует частный случай совокупности версий объектов и снапшота. Объект A с ключом K_A имеет три версии — A_0 с начальным моментом времени T_0 и конечным моментом времени T_3, A_1 с начальным моментом времени T_3 и конечным T_6, а также A_2 с начальным моментом T_6 и конечным T_8. Объект B с ключом, равным K_B, имеет две версии. Начальный момент времени первой версии B_0 равен T_0, а конечный момент равен T_2. В момент времени T_2 объект удаляется и в T_3 объект с тем же ключом повторно добавляется в систему. Начальный момент времени второй версии B_1 равен T_3, а конечный момент равен T_9. Объект C с ключом, равным K_C, имеет единственную версию. Его начальная точка во времени равна T_1, а конечная точка равна T_5. Объект D с ключом, равным K_D, имеет две версии. Начальная точка времени первой версии D_0 равна T_3, а конечная точка равна T_4. Начальный момент времени второй версии D_1 равен T_4, а конечный момент равен T_{10}. Снапшот для момента времени T_4 включает в себя A_1, C и D_1.

Из существующей в литературе[1] [7], [8] классификации запросов на чтение для хронологических хранилищ выделим следующие.

1. Запросы по диапазону ключей [*A*,*B*] при фиксированном моменте времени *i (keyrange time-slice query)*. Такой запрос можно определить также как реконструкцию части снапшота номер *i*. В приложении резервного копирования к такого рода запросам относится, например, листинг файлов в заданной директории заданной резервной копии.
2. Запросы по диапазону моментов времени [*i*,*j*] при фиксированном ключе *A (time-range key-slice query)*. Такой запрос можно определить как выдачу версий заданного файла *A*.
3. Запрос по заданным ключу *A* и моменту времени *i (point query)*, т. е. выдача версии файла *A*, соответствующей моменту времени *i*.

Целью данной работы является описание метода индексации системы снапшотов, обладающего следующими свойствами.

1. Возможность хранения части индекса во внешней памяти. Это требование вытекает из необходимости индексации данных большого размера и относительной дороговизны оперативной памяти.
2. Высокая скорость записи во внешнюю память. Для магнитных дисков скорость последовательной записи примерно в 1000 раз выше скорости произвольного доступа [9]. Для твёрдотельных дисков запись в режиме произвольного доступа приводит к деградации производительности примерно на порядок [10], [11], и даже несмотря на то, что контроллеры большинства современных Flash-носителей осуществляют группировку данных перед записью, всё же последовательная запись примерно в 3 раза быстрее записи в режиме произвольного доступа [12]. Это значит, что алгоритм должен поддерживать группировку операций записи и писать в режиме добавления *(log, append-only)*. Вместе с тем, чтение в режиме произвольного доступа допустимо.
3. Отсутствие модификации метаданных на месте в силу тех же причин, структуры индекса по возможности должны быть близки к неизменяемым.
4. Эффективное исполнение запросов на чтение по диапазону ключей при фиксированном моменте времени — вычислительная сложность операции не должна зависеть от количества снапшотов в индексе.
5. Возможность разделения данных на «горячие» и «холодные», по определению [3], и хранения данных различной температуры на носителях различного типа.

Обзор известных способов индексации данных с историей

B-деревья [13] и их разнообразные вариации стали промышленным стандартом индексирования персистентных данных десятилетия назад [14]. Высокая популярность B-деревьев обеспечена тем, что они обеспечивают высокую производительность как точечного поиска, так и выдачу данных по диапазону ключа при хранении индекса во внешней памяти, а также тем, что такой индекс позволяет относительно легко поддержать транзакции с ACIDсвойствами, так как блокировка на диапазон ключей может быть непосредственно привязана к поддереву [15]. Отсюда известно большое количество алгоритмов индексации хронологических данных, основанных на B-деревьях [4], [7], [16]-[21]. Основным недостатком B-деревьев и их модификаций является то, что вставка в B-дерево порождает запись во внешнюю память в режиме произвольного доступа, что сильно ограничивает производительность записи в хранилище с таким индексом. Исходя из этого, можно заключить, что индексы на основе Вдеревьев не годятся для решения поставленной задачи.

LSM-дерево [3], [22] — сложный, многоуровневый индекс, решающий проблему производительности записи во внешнюю память путём группировки изменений и записи в последовательном режиме. Оно состоит из нескольких сортированных компонентов *(sorted runs)*, организованных таким образом, что возможен эффективный обход элементов компонента в порядке ключей; к примеру, компоненты могут быть отсортированными массивами, деревьями (в частности B-деревьями и B+-деревьями), списками с пропусками (skiplist). Размер компонентов растёт в геометрической прогрессии, как предлагает оригинальная работа [3], или по иному закону [23]; при этом изменения всегда поступают во входную (нулевую) компоненту, которая всегда находится в оперативной памяти, и, по накоплении достаточного их количества, переходят последовательно в другие компоненты, лежащие во внешней памяти, путём исполнения процедуры слияния (*merge)* компонент. При этом важно, что в силу эффективного обхода исходных компонентов в порядке сортировки результат их слияния можно выписывать во внешнюю память строго последовательно.

Различные компоненты могут содержать элементы с одинаковыми ключами; при этом информация в дереве меньшего по номеру уровня считается приоритетной. Удаление элементов производится путём вставки особого элемента — маркера уда-

[1] Nascimento M. A., Eich M. H. An Introductory Survey to Indexing Techniques for Temporal Databases. Technical Report 95-CSE-1. Dallas: Southern Methodist University, 1995. 39 p.

ления — с требуемым ключом; физическое удаление элемента из индекса происходит при слиянии, образующем компоненту последнего уровня. Гарантируется, что компоненты (уровни) LSM-дерева, за исключением нулевой, неизменяемы, то есть они обновляются исключительно за счёт создания новых компонент путём выполнения процедуры слияния и удаления старых; это позволяет осуществлять чтение из индекса во время выполнения процедуры.

Эффективность использования места под метаданные относительно B-дерева зависит от ряда параметров. С одной стороны, возможное дублирование ключей на различных уровнях увеличивает размер индекса. С другой стороны, типичный коэффициент заполнения вершин в обычном B-дереве колеблется от 1/2 до 2/3, в то время как в случае LSM-дерева данные компактны, без промежутков пустого места; к тому же метаданные LSM-дерева лучше поддаются компрессии [24].

LSM-деревья не лишены недостатков, тормозящих их внедрение в системы управления базами данных. Так, для поиска элемента в индексе может оказаться необходимым осуществить поиск последовательно на каждом уровне дерева; частично эта проблема может быть решена с помощью фильтров Блума [25]. Производительность записи ограничивается феноменом амплификации записи *(write amplification)*, то есть необходимостью записать каждый конкретный элемент во внешнюю память не о одному разу в рамках слияний компонентов. Известные схемы, позволяющие уменьшить эффект амплификации записи, ведут к увеличению занимаемого места и худшей производительности чтений [22]. Многокомпонентность осложняет реализацию ACID-транзакций. Производительность записи в условиях реляционной базы данных снижается за счёт наличия так называемых скрытых чтений, возникающих при проверках различных ограничений и условий, наложенных на таблицу. Нетривиальную задачу представляет собой создание эффективных вторичных индексов при использовании LSM-дерева для первичного [26]. Тем не менее, в наши дни LSM-деревья стали весьма популярным выбором для использования в качестве индекса; так, например, их используют LevelDB, Cassandra, BigTable, HBase, RocksDB, Riak [27]-[29], во многом потому, что современные твёрдотельные накопители снижают остроту проблемы относительно медленных операций чтения.

Естественной идеей для применения LSM-дерева к задаче индексирования данных с историей является подход с составными ключами, полученными путём конкатенации ключа (имени) объекта и номера снапшота.

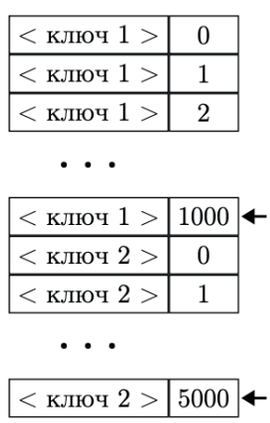
ключ объекта	момент времени

Р и с. 2. Составной ключ
F i g. 2. Composite key

При этом элемент считается принадлежащим снапшоту S, если его номер снапшота k меньше или равен S, k больше или равен номеру последнего полного снапшота F (без ограничения общности можно принять $F = 0$), не существует элемента с тем же именем и номером снапшота k', таким что $k < k' 6 S$, а также элемент не является маркером удаления.

Такой индекс позволяет весьма эффективно выполнять запросы на получение значений по диапазону спапшотов при фиксированном имени файла (time-range query, выдача версий заданного файла), так как такой запрос использует непрерывный диапазон значений ключа. Если принять общее количество элементов (файловых версий) в индексе за N и количество версий во временном диапазоне за s, то такой запрос исполнится за время $O(s\log(N))$.

В то же время такой индекс крайне неэффективно исполняет запросы на получение значений по диапазону ключей при фиксированном времени (key-range query, выдача файлов из заданной директории в заданном снапшоте). В диапазоне значений составного ключа оказываются значения, не принадлежащие заданному снапшоту, и их нужно отфильтровывать. В случае, когда спапшотов много и различия между ними велики (много новых или удалённых элементов в каждом снапшоте, иными словами, в индексе много элементов с различными номерами снапшотов в составном ключе), диапазон значений составного ключа может быть многократно больше диапазона имён. Если общее количество элементов (версий) в индексе равно N, количество имён в диапазоне имён равно n, а общее количество снапшотов — S, то время исполнения такого запроса в худшем случае (малая общность снапшотов) есть $O(n \cdot S \log(N))$, т. е. имеет линейную зависимость от числа снапшотов в индексе. Кроме того, элементы из одного снапшота оказываются в разных секторах жёсткого диска и разных кластерах файловой системы, а в случае хранения индекса в объектном хранилище — в разных объектах, что также существенно замедляет логически последовательное чтение.



Р и с. 3. Получение значений по диапазону ключей объектов в индексе с составным ключом для снапшота № 5000
F i g. 3. Obtaining values by object key range in index with composite key for snapshot 5000

Кроме того, для индекса с составными ключами неэффективно составление списка отличий (отличающихся элементов / файловых версий) между снапшотами. Для составления такого списка, даже для соседних снапшотов, необходимо обойти весь индекс, который может быть очень велик; сложность такой операции составляет $O(N)$.

Log-Structured History Data Access Method (LHAM) [30] использует составные ключи для индексации компонент и к тому же

группирует данные по времени в масштабе компонент LSM-дерева. Действительно, новые данные, имеющие большие значения времени своего порождения, естественным образом оказываются в верхних компонентах LSM-дерева; если наложить на дерево некоторые формальные ограничения, то это свойство можно использовать при поиске.

Предлагаются два способа разделения данных между компонентами:

1. при разделении без избыточности (*non-redundant partitioning)* компонента C_i содержит файловые версии со временем создания между соответствующими нижним пределом low_i и верхним high_i;
2. при разделении с избыточностью (*redundant partitioning)* компонента C_i содержит файловые версии, относящиеся к времени между low_i и high_i (валидные между low_i и high_i).

Для границ low_i, high_i, $i = 1...L$, вводится отдельная структура — общая директория (*global directory)*, которая для заданного снапшота позволяет выделить те компоненты LSM-дерева, в которых следует производить поиск.

Ясно, что разделение с избыточностью позволяет добиться существенно лучшей производительности запросов по диапазону ключа при фиксированном времени, так как файловые версии из одного снапшота оказываются в одной компоненте, а также, что увеличение производительности запросов достигается за счёт добавления дубликатов в индекс и увеличения его общего размера. Есть, однако, менее тривиальный недостаток разделения с избыточностью, вытекающий из того, что компонента C_0 должна входить в оперативную память, а предельные размеры прочих компонент также фиксированы и подчиняются определённому закону. При разделении с избыточностью часть места в каждой компоненте оказывается занята дубликатами для долгоживущих элементов, которые, таким образом, не могут полностью перейти на нижележащие уровни индекса, и если долгоживущих элементов много, то в компоненте заканчивается свободное место для вставки новых.

При разделении без избыточности чтение элементы из одного снапшота оказываются разбросаны по разным компонентам дерева (недавно созданные — в верхних, долгоживущие – в нижних), и для запросу по диапазону ключей при фиксированном времени необходимо обходить несколько компонент. Этот негативный эффект смягчается тем, что в большинстве приложений большая часть элементов последних снапшотов оказывается созданной в этих снапшотах [30], а также кэшированием обращений к диску на уровне операционной системы.

Хотя группировка версий на различных уровнях улучшает производительность запросов, ясно, что когда количество версий файла в среднем весьма велико, запрос по диапазону ключей при фиксированном времени всё так же вырождается в линейное сканирование.

LSM-индекс с общими компонентами

Предложим способ индексации данных с историей, который сочетает высокую скорость записи и эффективное выполнение запросов по диапазону ключа при фиксированном времени, основанный на LSM-дереве.

Пусть создан LSM-индекс первого, полного, спапшота с ключами индекса, равными ключам (именам) объектов. Для создания следующего, инкрементального, снапшота вычитаем в память дерево C_0 из первого снапшота и начнём использовать копию в памяти C'_0 для индексации. В случае, если размер дерева C'_0 превысит пороговое значение, проведём процедуру слияния с деревом C_1 из предыдущего снапшота и запишем во внешнюю память результат слияния — дерево C'_1, удалять дерево C_1 при этом не будем — на него ссылается первый снапшот. При необходимости проведём аналогичную процедуру слияния для деревьев $C_2 ... C_M$. В итоге индекс для инкрементального снапшота будет состоять из новых деревьев $C'_0 \ldots C'_M$ и старых, общих с индексом первого снапшота, деревьев $C_{M+1} ... C_N$. Следующий инкрементальный снапшот может иметь общие деревья как с первым снапшотом, так и со вторым, в зависимости от количества изменений. Таким образом, LSM-индекс каждого снапшота состоит как из новых деревьев верхних уровней, так и из ссылок на деревья нижних уровней из предыдущих снапшотов. Ясно, что в индексе нужна глобальная структура с дескрипторами снапшотов. Для поиска в индексе нужно найти дескриптор нужного снапшота, а затем произвести поиск в LSMдереве, образованного компонентами, на которые ссылается данный дескриптор.

Говоря более формально, пусть элемент это некая пара ключ-значение в индексе. Компонентом (*sorted run)* назовём набор элементов в основной или внешней памяти, такой, что возможен обход всех элементов компонента по порядку их ключей за линейное время, а поиск элемента по ключу — за логарифмическое. Компонент может быть представлен списком, массивом, деревом поиска, списком с пропусками и т. д. Компонент может быть создан путем создания пустого компонента и вставки новых элементов в него, по одному или в большом количестве, с сохранением неизменного порядка сортировки элементов, или же путем слияния двух или более ранее существовавших компонентов. Каждый компонент сам по себе может делиться на множество частей (например, в рамках техники разделения (partitioning) [22]).

LSM-индекс снапшота тогда представляет собой набор пронумерованных компонентов, с определенными условиями на необхожимость слияния компонентов при добавлении новых элементов в индекс, алгоритмом слияния компонентов, и, возможно, с некоторыми вспомогательными структурами в основной и внешней памяти. Последнее может включать, например, каталог компонентов, фильтры Блума, журналы предзаписи (write-ahead log) или их комбинацию. Компоненты индекса могут храниться на множестве различных внешних запоминающих устройств.

Скажем, что LSM-индекс снапшота включает элемент, если один из его компонентов включает этот элемент. Компоненты индекса нумеруются от последнего сформированного до первого, так что «самый новый» имеет наименьший номер, а «самый старый» — наибольший. Если несколько компонентов включают элементы с одним и тем же ключом, то LSMиндекс снапшота включает элемент со значением, равным значению элемента из компонента с наименьшим номером. Некоторые, наиболее «старые» компоненты могут быть унаследованы от индекса снапшота меньшего номера (то есть индекса предшествующего снапшота). Например, некоторые «старые» элемен-

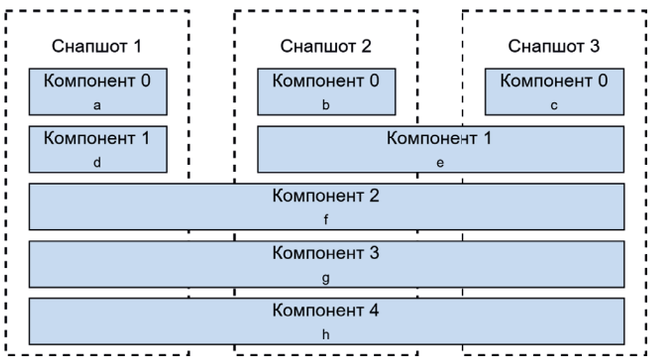
ты индекса могут быть объединены в относительно большой компонент, поддерживаемый во внешней памяти, а «новые» элементы могут находиться в относительно меньшем компоненте, поддерживаемом в основной памяти. Если нет обновлений для элементов из «старого» компонента, то таких ключей нет в «новом» компоненте и актуальными являются значения из «старого». В противном случае значения из «старого» компонента устаревают, а значения из «нового» являются актуальными. Если компонент включает элемент со специальным значением, называемым маркером удаления, то считаем, что в индексе снапшота нет элемента с таким ключом.

Поиск элемента в индексе снапшота подразумевает поиск элемента в его компонентах, начиная от компонентов с меньшим номером и при необходимости, если элемент не найден, переходя к компонентам с большим номером. Для выполнения поиска по диапазону ключа необходимо выполнять соответствующие запросы во всех компонентах и объединять результаты вместе (предпочитая более новый элемент более старому, если их ключи совпадают).

Индексы всех снапшотов, кроме текущего (формирующегося) являются неизменяемыми (иммутабельными). Чтобы вставить элемент в индекс текущего снапшота, необходимо вставить его в избранный компонент (с нулевым номером), что хранится в оперативной памяти компьютера. Если при вставке элемента выполняются заданные условия на необходимость слияния компонент, заданный алгоритм слияния преобразует существующий набор компонентов в новый набор, где указанные наборы включают в себя один и тот же набор ключей в вышеописанном смысле. Если коллекция компонентов содержит несколько элементов с одним и тем же ключом, алгоритм слияния может удалить элемент, который был добавлен в индекс раньше, заменив его элементом, который был добавлен позже, но не наоборот. После преобразования некоторые из компонентов могут быть равны ранее существовавшим (и их можно использовать повторно без копирования), часть компонентов может быть сформирована заново, а часть может быть отброшена. Для любых двух компонентов, включающих элементы с одним и тем же ключом, компонент с меньшим номером будет включать элемент, вставленный позднее. Запись данных компонент во внешнюю память должна производиться строго последовательно, что важно для производительности процедуры вставки в целом. Хранящиеся во внешней памяти компоненты неизменяемы, так что их можно создавать или удалять целиком, но не изменять частично. Эта неизменность позволяет использовать устройства типа «однократная запись, многократное чтение» (write once, read many; WORM).

Удаление элемента выполняется путем вставки элемента с тем же ключом и специальным значением, так называемым маркером удаления. При формировании компонента с наибольшим номером алгоритм слияния компонент должен исключить соответствующий ключ.

Термином *LSM-индекс с общими компонентами* обозначим множество LSM-индексов снапшотов, где некоторые компоненты могут быть общими для нескольких индексов снапшотов, т. е. использоваться ими совместно, наряду со структурой глобального каталога со списком ссылок на компоненты индекса каждого снапшота, так что для каждого номера снапшота t возможно перечислить компоненты, составляющие его LSM-индекс. Глобальный каталог может быть реализован в виде списка списков, дерева списков, списка деревьев, дерева деревьев, хэштаблицы списков, хэш-таблицы деревьев и т. д., и т. п.



Р и с. 4. Пример LSM-индекса с общими деревьями для трёх снапшотов

F i g. 4. An example of an LSM index with shared trees for three snapshots

Рисунок 4 иллюстрирует пример LSM-индекса с общими деревьями для трёх снапшотов. Индекс снапшота 1 состоит из компонентов № 0 (a), № 1 (d), № 2 (f), № 3 (g) и № 4 (h). Индекс снапшота 2 состоит из компонентов № 0 (b), № 1 (e), № 2 (f), № 3 (g) и № 4 (h). Компоненты с номерами больше 1 являются общими со снапшотом 1, компоненты с номерами 0 и 1 отличаются от компонент с соответствующими номерами в индексе снапшота 1. Индекс снапшота 3 состоит из компонентов № 0 (c), № 1 (e), № 2 (f), № 3 (f) и № 4 (h). Здесь компоненты с номерами больше 0 совместно используются с индексом снапшота 2, а компонент номер 0 отличается как от компонента с номером 0 из индекса снапшота 1, так и от компонента с номером 0 из индекса снапшота 2.

В силу того, что запись в индекс во всём аналогична записи в LSM-дерево, за исключением удаления старых компонент после слияния, индекс эффективен для операций записи во внешнюю память; запись ведётся строго последовательно, модификации метаданных «на месте» не происходит.

Для того, чтобы в LSM-индексе с общими деревьями найти версию объекта A с ключом k_A, соответствующую моменту времени t, необходимо выбрать в глобальном каталоге индекс для снапшота t и осуществить в нём поиск по ключу k_A. Для того, чтобы совершить запрос по диапазону ключа, от k_A до k_B, нужно также выбрать в глобальном каталоге индекс для снапшота t и осуществить такой запрос в нём.

Процедура удаления снапшота очень проста: нужно всего лишь удалить дескриптор этого снапшота из глобального индекса и уменьшить счётчики ссылок для компонент, на которые он ссылался, помечая соответствующие данные как удалённые при достижении счётчиком нуля.

Теорема 4.1. Пусть общее количество ключей в снапшоте t есть K_t, *количество элементов на странице, использующейся при операциях ввода-вывода —* b, *а размеры компонент LSM-дерева* в индексе снапшота растут по геометрическому закону с показателем r. *При предположении, что выбор дескриптора* снапшота осуществляется за $O(1)$ *(что достижимо, например, при реализации глобального индекса* снапшотов через хэш-та-

блицу) операция получения версии объекта (A,t) для некоторого объекта A исполняется за $O(\log_r K_t \log_b K_t)$.

Доказательство. Пусть n есть количество элементов в некотором компоненте снапшота. По определению компонента, поиск элемента в нём займёт $O(\log n)$, как в дереве поиска или в отсортированном массиве.

Если речь идёт о компоненте во внешней памяти (а в силу ограниченности объёма оперативной памяти достаточно большие компоненты с неизбежностью должны храниться во внешней памяти), а размер страницы ввода-вывода есть b, то можно сказать, что поиск элемента в компоненте займёт $O(\log_b n)$ [13].

Используя то, что логарифмы членов геометрической прогрессии образуют арифметическую прогрессию, получим, что Ясно, что сумма размеров компонент будет наибольшей, если наибольшая компонента содержит все ключи, то есть $K_t = r^{L-1}k$. Тогда

$$L = \log_r \left(\frac{K_t}{k}\right) + 1 \tag{4.2}$$

и время поиска элемента в наихудшем случае займёт

$$O\left(\log_r \left(\frac{K_t}{k}\right) \log_b K_t\right) = O(\log_r K_t \log_b K_t) \tag{4.3}$$

Заметим, что благодаря использованию Блум-фильтров, позволяющих пропускать компоненты, не содержащие нужного ключа, время поиска можно существенно улучшить, приблизив его сложность к $O(\log_b K_t)$.

Теорема 4.2. Пусть общее количество ключей в снапшоте t есть K_t, *количество элементов на странице, использующейся при операциях ввода-вывода —* b, *а размеры компонент LSM-дерева в индексе* снапшота растут по геометрическому закону с показателем r. *При предположении, что выбор дескриптора* снапшота осуществляется за $O(1)$, *операция чтения по диапазону ключей* $[k_A, k_B]$ при фиксированном моменте времени t исполняется за $O\left(\frac{q}{b} \log\log_r K_t + \log_r K_t \log_b K_t\right)$

где q *— количество ключей в заданном диапазоне.*

Доказательство. Действительно, поиск по диапазону ключей в снапшоте представляет собой последовательно несколько операций поиска в LSM-компонентах, принадлежащих снапшоту, и слияние полученных списков из всех компонент, так что если ключ присутствует в двух компонентах сразу, то приоритет признаётся за компонентой более высокого уровня.

Если компонента i содержит n_i элементов, то сложность поиска начального ключа k_A или следующего за ним есть $O(\log_b n_i)$. Если элемент с ключом, большим либо равным k_A и меньшим либо равным k_B, нашёлся, то следующие за ним элементы в порядке обхода вплоть до элемента, большего k_B или конца обхода можно рассматривать, как единый отсортированный список. Для ряда реализаций компонента, таких как отсортированный список, список с пропусками, B+дерево, это и будет буквально отсортированный список, но важнее, что для любой реализации компонента по предложенному выше определению не нужно фильтровать элементы, отсеивая элементы других снапшотов, как при использовании индексов с составными ключами, так что сложность получения этих элементов один за другим та же, что и у отсортированного списка.

Известно, что сложность слияния L списков длины, ограниченной q есть $O(q \log L)$[2], что с поправкой на размер страницы для чтения превращается в $O\left(\frac{q}{b} \log L\right)$.

Таком образом, если k — размер наименьшего списка, то общее время получения списков и их слияния, с учётом равенств (4.1), (4.2), (4.3), составит

$$O\left(\frac{q}{b} \log L + \sum_{i=0}^{L-1} \log_b (r^i k)\right) = O\left(\frac{q}{b} \log\log_r K_t + \log_r K_t \log_b K_t\right) \tag{4.4}$$

что доказывает теорему.

если размеры компонентов растут по геометрическому закону, то есть каждый следующий компонент больше предыдущего в r раз, размер наименьшего компонента есть k, а общее число компонентов (уровней, по терминологии [3]) есть L, то операция поиска, что в наихудшем случае осуществляет поиск во всех компонентах от наименьшего до наибольшего, займёт

$$O\left(\log k + \log_b(rk) + \log_b(r^2 k) + \cdots + \log_b(r^{L-1}k)\right)$$

$$= O\left(\frac{L}{2}(\log_b k + \log_b(r^{L-1}k))\right)$$

$$= O\left(L \log_b(r^{L-1}k)\right) \tag{4.1}$$

Важным преимуществом описываемого метода является то, что для получения списка изменений между снапшотами не нужно обходить весь индекс и сравнивать все компоненты заданных снапшотов: достаточно сравнения только тех компонентов, что не разделяются этими снапшотами. Для соседних снапшотов количество элементов для обхода может оказаться весьма невелико, например, может оказаться достаточным сравнить компоненты нулевого уровня C_0.

Ясно, что так как компоненты с наибольшими номерами, составляющие большую часть индекса, читаются реже, так как элемент может быть найден при чтении маленьких компонент с меньшими номерами, и, более того, элементы, соответствующие активно меняющимся объектам, будут найдены в «новых» компонентах с меньшими номерами с высокой вероятностью, то компоненты с большими номерами можно хранить на носителях с более низкой скоростью чтения, разделяя таким образом данные компонент на «горячие» и «холодные».

Так как компоненты LSM-дерева записываются раз и навсегда, не подвергаясь изменению и удалению (если не удаляются снапшоты), то их можно записывать на носители без возможности перезаписи (WORM), такие как оптические диски. При этом нет ресурсоёмкой процедуры выделения «исторических» данных и их перемещения.

Важное ограничение метода заключается в том, что весьма неэффективны становятся запросы на получение значений по диапазону моментов времени при фиксированном ключе (time-range query, выдача версий заданного объекта), так как

[2] Aho A. V., Hopcroft J. E., Ullman J. D. Data Structures and Algorithms. Addison-Wesley Series in Computer Science and Information Processing. Addison-Wesley Publishing Company, 1985.

такой запрос разбивается на серию независимых запросов во многих LSM-деревьях. При необходимости поддержки таких запросов возможно применение комбинированного подхода: объекты могут индексироваться дважды — в LSM-индекс с общими компонентами для эффективных запросов по диапазону ключей и в LSM-индекс с составными ключами для эффективных запросов по диапазону времени.

Кроме того, так как каждый ключ может содержаться сразу в нескольких компонентах, размер LSM-индекса с общими компонентами может существенно превышать сумму размеров содержащихся в нём элементов. Причина этого заключается в том, что при создании очередного снапшота количество новых элементов в индексе есть количество элементов в новых компонентов, созданных процедурой слияния, а это количество может быть существенно больше числа новых элементов на входе. Явление превышения количества новых элементов количеством записанных в память элементов в литературе оно носит название амплификации записи (*write amplification)* [22]. Это явление приводит как к ограничению скорости вставки новых элементов, так и к увеличению занимаемого места во внешней памяти, причём известные техники борьбы с ним позволяют либо уменьшить количество операций записи за счёт увеличения занимаемого места, либо наоборот уменьшить занимаемое место за счёт роста операций записи [22]. В обозначениях, принятых ранее, сложность записи в LSM-индекс с размерами компонент, растущими в геометрической прогрессии, будет равняться $O\left(L\frac{r}{b}\right)$, и такой индекс займёт в$O\left(\frac{r+1}{r}\right)$ больше места, чем простая совокупность его элементов [22].

Экспериментальное исследование

Общие замечания

Для исследования предложенного в разд. 4 алгоритма и сравнения его с иными подходами, такими как LSM-дерево с составными ключами и LHAM, был написан прототип на языке Go.

Был исследован сценарий, типичный для ряда приложений [31], когда в индексе создаётся первый (полный) снапшот с большим количеством элементов N_0, затем добавляются инкрементальные снапшоты с количеством новых элементов в каждом, принадлежащем распределению Пуассона с математическим ожиданием λ, где $\lambda \ll N_0$.

Этот сценарий разбивается на три варианта:

1. Элементы только добавляются в индекс, но не удаляются, т. е. каждый объект имеет лишь одну версию. В каждом снапшоте, начиная со второго, добавляется 1% от размера первого снапшота. Такая модель может соответствовать, например, резервному копированию неизменяемых и неудаляемых данных.
2. В каждом новом снапшоте удаляется столько же элементов, сколько и добавляется — также 1% от первого снапшота.
3. Все снапшоты после первого производят обновление данных по одним и тем же ключам (модель с наличием «горячих» и «холодных» данных). Долю «горячих» объектов возьмём также равной 1%.

Далее об этих вариантах будет говориться как о «1 сценарии создания снапшотов», «2 сценарии создания снапшотов» и «3 сценарии создания снапшотов» соответственно. Размер первого снапшота везде возьмём в 1000 элементов.

Для каждого из этих трёх вариантов были изучена производительность таких операций, как:

1. Создание инкрементального снапшота
2. Запрос по диапазону ключей при фиксированном моменте времени (листинг директории)
3. Запрос по диапазону моментов времени при фиксированном ключе (выдача версий файла)
4. Запрос по заданным ключу и моменту времени (выдача версии файла, принадлежащей снапшоту),

путём измерения среднего времени выполнения запроса (усреднение по 3000 запросам), а также исследована зависимость количества записей в индексе от количества снапшотов.

Таблица 1. **Аппаратные и программные характеристики платформы**
Table 1. **Hardware and software characteristics of the platform**

Тип накопителя	HDD
Модель накопителя	Toshiba MQ01ABD1
Архитектура процессора	x86-64
Частота процессора	2.30 ГГц
Количество ядер	4
Оперативная память	16 Гб
L2 кэш	3072 Кб
Операционная система	Linux 4.17.19
Компилятор	Go 1.11.1

Таблица 2. **Настройки индекса**
Table 2. **Index settings**

Предельный размер нулевого уровня	256 байт
Соотношение размеров соседних уровней	4
Размер записи	168 байт
Порог для начала сброса состояния процедуры слияния на диск	1 Кб

Запрос по диапазону ключей при фиксированном моменте времени

Сравним производительность поиска для LSM с составным ключом, LHAM и LSM с общими компонентами. В случае LSM с составным ключом поиск осуществляется путём нахождения начальной пары «ключ — номер снапшота» и дальнейшей фильтрации записей; случай LHAM аналогичен во всём, за исключением того, что поиск производится не во всех компонентах LSM, а только в тех, что могут содержать записи из данного снапшота, согласно информации из «глобальной директории». В случае же LSM с общими компонентами сначала находится дескриптор корня, соответствующего данному снапшоту, а затем производится обычный запрос в LSM-дереве, начиная от этого корня.

Обозначим для краткости τ_{KR} среднее время поиска 20 записей начиная от заданного ключа при фиксированном времени.

При первом сценарии создания снапшотов файлы только добавляются, т. е. каждый файл имеет лишь одну версию. Логично предположить, что все три алгоритма будут характеризоваться примерно константным временем поиска, так как фильтровать нечего. Это предположение подтверждается данными измерений, где также заметно, сколько времени отнимает поиск необходимого корня в случае LSM с общими компонентами.

Таблица 3. **τ_{KR} при 1 сценарии создания снапшотов, нс**
Table 3. **τ_{KR} with 1 snapshot creation scenario, ns**

К-во снапшотов	LSM с составным ключом	LSM с общими компонентами	LHAM
10	40697	49358	41139
100	41625	92196	43149
200	41675	90746	44175
300	42685	92942	42051
400	42903	98491	42227
500	43584	99975	43312

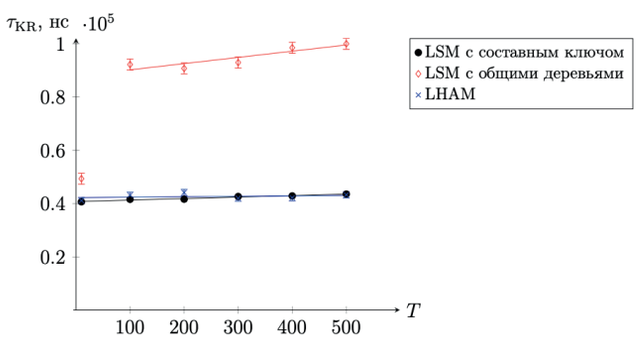


Рис. 5. Зависимость τ_{KR} от количества снапшотов T при 1 сценарии создания снапшотов
Fig. 5. Dependence of τ_{KR} on the number of snapshots T for 1 snapshot creation scenario

Во втором сценарии в каждом инкрементальном снапшоте удаляются элементы предыдущего. Задача становится сложнее, так как теперь нужно удалять из выдачи элементы, удалённые до заданного момента времени. Видно, как для такого случая хорошо работает относительно простая оптимизация LHAM.

Таблица 4. **τ_{KR} при 2 сценарии создания снапшотов, нс**
Table 4. **τ_{KR} with 2 snapshot creation scenario, ns**

К-во снапшотов	LSM с составным ключом	LSM с общими компонентами	LHAM
10	44891	52551	39436
100	47803	91869	56591
200	85738	97818	63504
300	82617	93840	56921
400	95432	99985	59290
500	136643	109049	65087

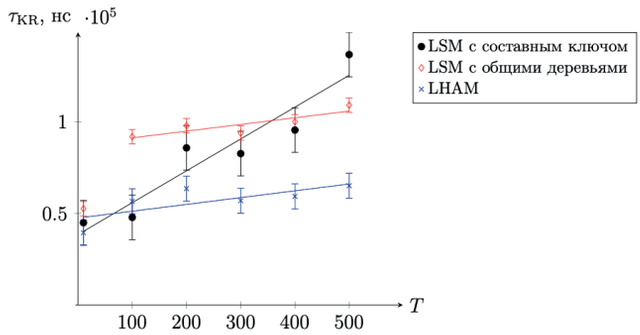


Рис. 6. Зависимость τ_{KR} от количества снапшотов T при 2 сценарии создания снапшотов
Fig. 6. Dependence of τ_{KR} on the number of snapshots T for 2 snapshot creation scenario

Третий сценарий — наиболее интересный, здесь существует некоторое количество файлов (всего 1%), имеющих громадное количество версий — столько же, сколько и снапшотов. Хотя доля «горячих» файлов и невелика, в эксперименте хорошо заметен линейный рост времени поиска для LSM с составным ключом как в с оптимизацией LHAM, так и без, а также почти константное поведение в случае LSM с общими компонентами.

Таблица 5. **τ_{KR} при 3 сценарии создания снапшотов, нс**
Table 5. **τ_{KR} with 3 snapshot creation scenario, ns**

К-во снапшотов	LSM с составным ключом	LSM с общими компонентами	LHAM
10	41350	48292	40938
100	51739	88239	49220
200	68582	90026	56849
300	90908	90212	73067
400	112633	92916	89402
500	127381	94011	123248

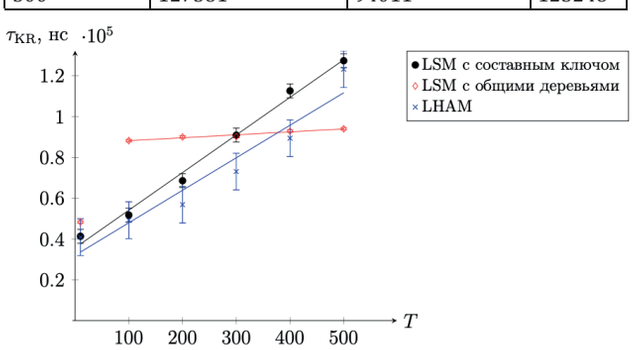


Рис. 7. Зависимость τ_{KR} от количества снапшотов T при 3 сценарии создания снапшотов
Fig. 7. Dependence of τ_{KR} on the number of snapshots T for 3 snapshot creation scenario

Запрос по диапазону моментов времени при фиксированном ключе

Так как оптимизация «глобальной директории» здесь не применима, то рассматривать LHAM отдельно от простого LSM с

составным ключом бессмысленно и сравниваться будет только LSM с составным ключом и LSM с общими компонентами. В случае LSM с составным ключом поиск является обычным поиском в LSM-дереве, так как записи с одинаковым файловым ключом идут подряд по возрастанию номера снапшота. В случае LSM с общими компонентами производится поиск записи от корня каждого снапшота. Ясно, что во всех трёх сценариях такая стратегия даст ощутимый проигрыш, что подтверждается и экспериментальными данными. Следует, тем не менее, заметить, что время поиска остаётся всё же линейным по числу снапшотов.

Обозначим для краткости τ_{TR} среднее время выдачи 20 версий по заданному ключу.

Таблица 6. **τ_{TR} при 1 сценарии создания снапшотов, нс**
Table 6. **τ_{TR} with 1 snapshot creation scenario, ns**

К-во снапшотов	LSM с составным ключом	LSM с общими компонентами
10	29426	675371
100	29492	8550541
200	28873	14269690
300	28970	18974112
400	28901	25192749
500	29438	32167701

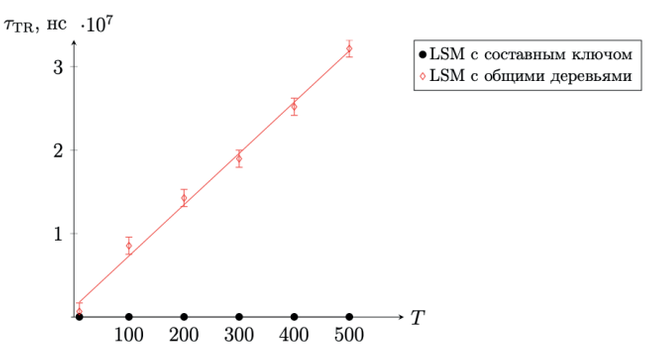


Рис. 8. Зависимость τ_{TR} от количества снапшотов T при 1 сценарии создания снапшотов
Fig. 8. Dependence of τ_{TR} on the number of snapshots T for 1 snapshot creation scenario

Таблица 7. **τ_{TR} при 2 сценарии создания снапшотов, нс**
Table 7. **τ_{TR} with 2 snapshot creation scenario, ns**

К-во снапшотов	LSM с составным ключом	LSM с общими компонентами
10	27526	988864
100	27428	1825173
200	44858	1712018
300	39279	1757728
400	40530	2174894
500	40865	1958441

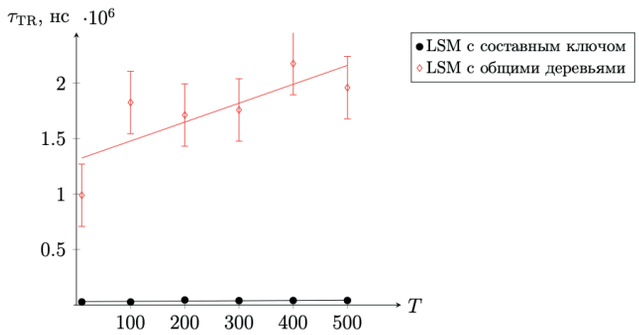


Рис. 9. Зависимость τ_{TR} от количества снапшотов T при 2 сценарии создания снапшотов
Fig. 9. Dependence of τ_{TR} on the number of snapshots T for 2 snapshot creation scenario

Таблица 8. **τ_{TR} при 3 сценарии создания снапшотов, нс**
Table 8. **τ_{TR} with 3 snapshot creation scenario, ns**

К-во снапшотов	LSM с составным ключом	LSM с общими компонентами
10	28884	690501
100	28841	1680690
200	28952	1560906
300	30391	1618204
400	29902	1806697
500	29769	1835833

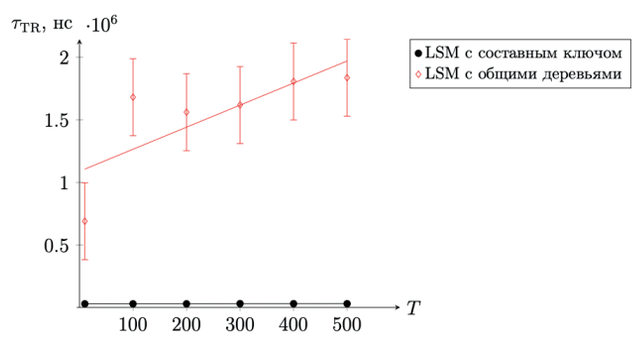


Рис. 10. Зависимость τ_{TR} от количества снапшотов T при 3 сценарии создания снапшотов
Fig. 10. Dependence of τ_{TR} on the number of snapshots T for 3 snapshot creation scenario

Запрос по заданным ключу и моменту времени

Как и в разд. 5.3, будем сравнивать только LSM с составным ключом и LSM с общими компонентами; в первом случае поиск является обычным поиском в LSM-дереве, во втором — сначала происходит поиск нужного корня. Ожидается, что время поиска будет примерно одинаковым с небольшим штрафом за поиск корня в случае LSM с общими компонентами, что и подтверждается экспериментом.

Обозначим для краткости τ_p среднее время выдачи 20 версий по заданному ключу.

Таблица 9. **τ_p при 1 сценарии создания снапшотов, нс**
Table 9. **τ_p with 1 snapshot creation scenario, ns**

К-во снапшотов	LSM с составным ключом	LSM с общими компонентами
10	28483	33325
100	28835	38541
200	28776	44905
300	28954	41398
400	29047	38513
500	29498	43823

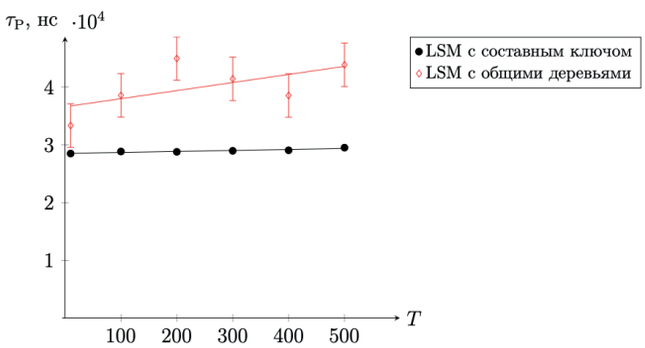


Рис. 11. Зависимость τ_p от количества снапшотов T при 1 сценарии создания снапшотов
Fig. 11. Dependence of τ_p on the number of snapshots T for 1 snapshot creation scenario

Таблица 10. **τ_p при 2 сценарии создания снапшотов, нс**
Table 10. **τ_p with 2 snapshot creation scenario, ns**

К-во снапшотов	LSM с составным ключом	LSM с общими компонентами
10	26147	33866
100	27552	42519
200	45530	44571
300	52876	37435
400	27348	40287
500	44874	43873

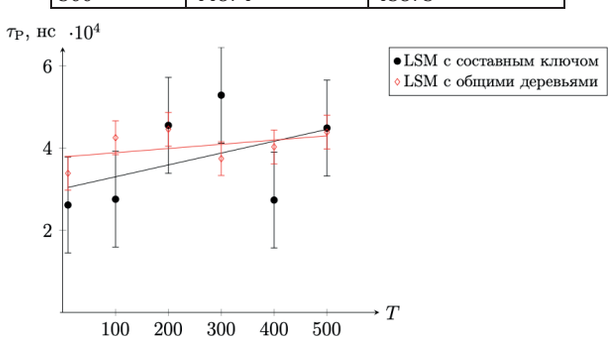


Рис. 12. Зависимость τ_p от количества снапшотов T при 2 сценарии создания снапшотов
Fig. 12. Dependence of τ_p on the number of snapshots T for 2 snapshot creation scenario

Таблица 11. **τ_p при 3 сценарии создания снапшотов, нс**
Table 11. **τ_p with 3 snapshot creation scenario, ns**

К-во снапшотов	LSM с составным ключом	LSM с общими компонентами
10	29015	31213
100	28769	30959
200	28662	30932
300	29253	30772
400	29256	31013
500	29083	30829

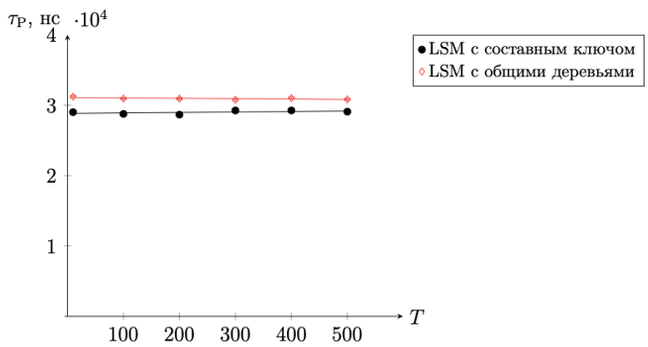


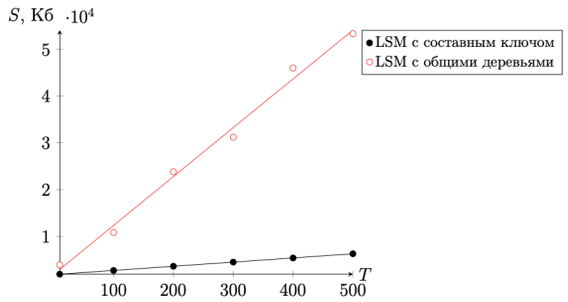
Рис. 13. Зависимость τ_p от количества снапшотов T при 3 сценарии создания снапшотов
Fig. 13. Dependence of τ_p on the number of snapshots T for 3 snapshot creation scenario

Размер метаданных индекса и время создания снапшотов

Как было отмечено в разд. 4, во всех случаях размер индекса для LSM с общими компонентами должен превышать размер обычного LSM с составными ключами. Это предположение подтверждается экспериментально, причём зависимость размера метаданных от количества данных во всех исследованных случаях носит строго линейный характер.

Таблица 12. **Размер индекса при 1 сценарии создания снапшотов, Кб**
Table 12. **Index size for 1 snapshot creation scenario, KB**

К-во снапшотов	LSM с составным ключом	LSM с общими компонентами
10	1924	3896
100	2716	10820
200	3612	23804
300	4500	31180
400	5380	45988
500	6284	53340



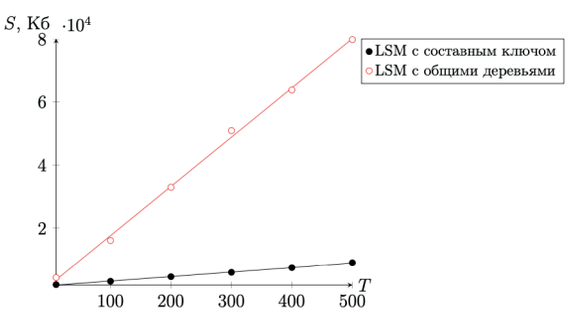
Р и с. 14. Зависимость размера индекса *S* от количества снапшотов *T* при 1 сценарии создания снапшотов

F i g. 14. Dependence of index size *S* on the number of snapshots *T* with 1 snapshot creation scenario

Т а б л и ц а 13. **Размер индекса при 2 сценарии создания снапшотов, Кб**

T a b l e 13. **Index size for 2 snapshot creation scenario, KB**

К-во снапшотов	LSM с составным ключом	LSM с общими компонентами
10	1984	4232
100	3060	16044
200	4520	32912
300	5912	50896
400	7412	63776
500	8964	79748



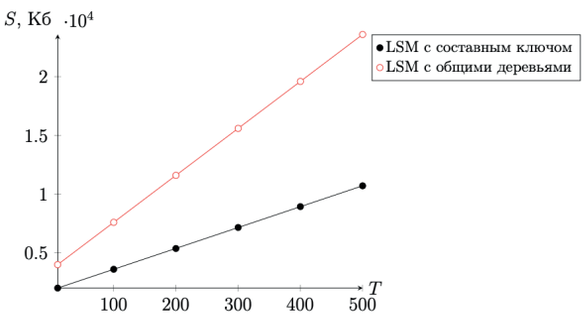
Р и с. 15. Зависимость размера индекса *S* от количества снапшотов *T* при 2 сценарии создания снапшотов

F i g. 15. Dependence of index size *S* on the number of snapshots *T* with 2 snapshot creation scenario

Т а б л и ц а 14. **Размер индекса при 3 сценарии создания снапшотов, Кб**

T a b l e 14. **Index size for 3 snapshot creation scenario, KB**

К-во снапшотов	LSM с составным ключом	LSM с общими компонентами
10	2008	4004
100	3608	7604
200	5376	11604
300	7168	15604
400	8944	19604
500	10716	23604



Р и с. 16. Зависимость размера индекса *S* от количества снапшотов *T* при 3 сценарии создания снапшотов

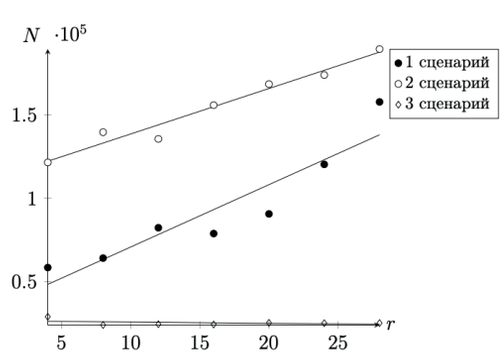
F i g. 16. Dependence of index size *S* on the number of snapshots *T* with 3 snapshot creation scenario

Рассмотрим теперь зависимость размера индекса от соотношения между размерами компонент. В таблице 15 колонки «1 сценарий», «2 сценарий» и «3 сценарий» содержат экспериментально полученное количество записей в индексе для соответствующих сценариев создания снапшотов — добавления неизменяемых данных, временных файлов и 1% «горячих» документов.

Т а б л и ц а 15. **Количество записей в индексе для 500 снапшотов в зависимости от множителя размера компонент *r***

T a b l e 15. **The entries number in the index for 500 snapshots, depending on the multiplier of the components size r**

r	1 сценарий	2 сценарий	3 сценарий
4	58452	121402	28985
8	64112	139506	24031
12	82282	135481	24528
16	78810	155702	24358
20	90576	168304	25348
24	120236	173728	25155
28	157628	189274	25215



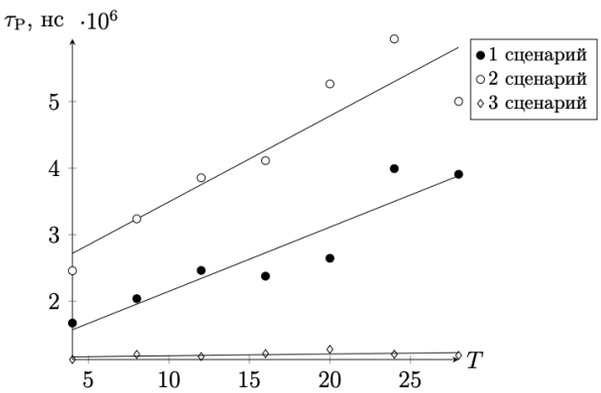
Р и с. 17. Зависимость количества записей в индексе *N* от множителя *r*

F i g. 17. Dependence of the number of records in the index *N* on the multiplier r

Ясно, что большему размеру индекса соответствует и большее время создания снапшотов, что подтверждается экспериментом. Это значит, что хотя, как показано в [3], для производительного чтения параметр r не должен быть слишком мал, чтобы не образовывалось слишком много уровней LSM-дерева, в целях производительной записи и экономии места разумно взять r не слишком большим.

Таблица 16. **Время создания инкрементального снапшота в зависимости от множителя размера компонент *r*, мс**
Table 16. **Incremental snapshot creation time depending on component size multiplier r, ms**

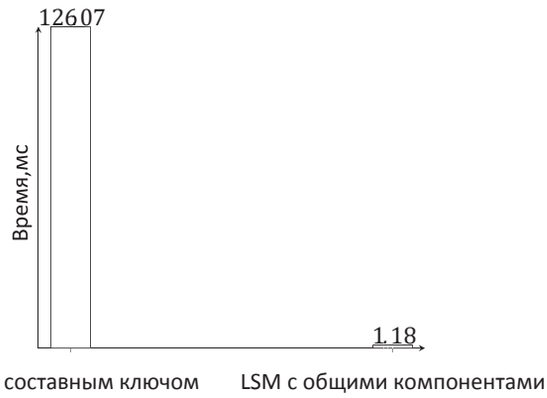
r	1 сценарий	2 сценарий	3 сценарий
4	1.673607	2.458984	1.125463
8	2.040070	3.237352	1.203038
12	2.463415	3.853969	1.171092
16	2.377378	4.111759	1.214444
20	2.645849	5.263210	1.278668
24	3.991571	5.939266	1.206070
28	3.907223	5.000783	1.188524



Р и с. 18. Зависимость времени создания инкрементального снапшота от множителя *r*
F i g. 18. Dependence of the time of creating an incremental snapshot on the multiplier r

Заметим также, что размер компонент, относящихся к заданному снапшоту (в том числе — последнему) в случае LSM с общими компонентами зависит только от количества ключей, но не от количества версий. Из этого следует, что в индексе будет меньше уровней, чем в LSM с составным ключом, и при вставке не будет происходить меньше дорогих операций слияния нижних уровней, что приведёт к увеличению скорости создания снапшотов.

Для 3 сценария создания снапшотов среднее время создания снапшота (для 3000 снапшотов) в LSM с составным ключом составило 126.072015 мс, а в LSM с общими компонентами — всего 1.175635 мс.



Р и с. 19. Среднее время создания инкрементального снапшота для 3 сценария, мс
F i g. 19. Average incremental snapshot creation time for scenario 3, ms

Выводы

Алгоритм «LSM с общими компонентами» оправдал теоретические ожидания.

Индекс эффективен для операций записи во внешнюю память; запись ведётся строго последовательно, модификации данных «на месте» не происходит. Компоненты индекса можно записывать на носители с различной производительностью доступа и на носители без возможности перезаписи.

Для получения списка изменений между соседними снапшотами достаточно сравнить лишь малую часть их компонентов. Время поиска по диапазону ключей при фиксированном моменте времени (листинга директории) практически не растёт с количеством снапшотов, как и предсказано в теореме 4.2, в то время как при наивном подходе наблюдается линейный рост даже при небольшой доле файлов с многими версиями. Кроме того, наблюдается кратный выигрыш в скорости создания снапшотов на некоторых сценариях.

В то же время предлагаемый метод не лишён важных недостатков. Хотя индекс, построенный по этому принципу, и позволяет производить поиск по диапазону моментов времени (выдачу версий файла), разрыв с более простым подходом столь велик, что первый следует признать слабо пригодным для данной задачи. Это значит, что в случае, когда важны оба типа запросов, необходимо комбинировать два индекса для различных целей.

Возможным направлением развития выбранной темы было бы изучение известных алгоритмов слияния для LSM применительно к хронологическим данных.

References

[1] Hilbert M., Lo´pez P. The World's Technological Capacity to Store, Communicate, and Compute Information. *Science.* 2011; 332(6025):60-65. (In Eng.) doi: https://doi.org/10.1126/science.1200970

[2] Leavitt N. Will NoSQL Databases Live Up to Their Promise? *Computer*. 2010; 43(2):12-14. (In Eng.) doi: https://doi.org/10.1109/MC.2010.58

[3] O'Neil P., Cheng E., Gawlick D., O'Neil E. The Log-structured Merge-tree (LSM-tree). *Acta Informatica*. 1996; 33(4):351-385. (In Eng.) doi: http://dx.doi.org/10.1007/s002360050048

[4] Lomet D., Salzberg B. Access Methods for Multiversion Data. *ACM SIGMOD Record*. 1989; 18(2):315-324. (In Eng.) doi: https://doi.org/10.1145/66926.66956

[5] Segev A., Shoshani A. Logical Modeling of Temporal Data. *ACM SIGMOD Record*. 1987; 16(3):454-466. (In Eng.) doi: https://doi.org/10.1145/38714.38760

[6] Jensen C.S., Clifford J., Gadia S.K., Segev A., Snodgrass R.T. A Glossary of Temporal Database Concepts. *ACM SIGMOD Record.* 1992; 21(3):35-43. (In Eng.) doi: https://doi.org/10.1145/140979.140996

[7] Tsotras V.J., Kangelaris N. The Snapshot Index: An I/O-optimal Access Method for Timeslice Queries. *Information Systems.* 1995; 20(3):237-260. (In Eng.) doi: https://doi.org/10.1016/0306-4379(95)00011-R

[8] Bertino E., et al. Indexing Techniques for Advanced Database Systems. *The Springer International Series on Advances in Database Systems.* Vol. 8. Springer, Boston, MA; 1997. 250 p. (In Eng.) doi: https://doi.org/10.1007/978-1-4615-6227-6

[9] Anderson D., Dykes J., Riedel E. More Than an Interface – SCSI vs. ATA*. Proceedings of FAST '03: 2nd USENIX Conference on File and Storage Technologies.* San Francisco, CA: USENIX Association; 2003. p. 245-257. Available at: https://www.usenix.org/legacy/events/fast03/tech/full_papers/anderson/anderson.pdf (accessed 17.05.2022). (In Eng.)

[10] Chen F., Koufaty D.A., Zhang X. Understanding Intrinsic Characteristics and System Implications of Flash Memory Based Solid State Drives. *ACM SIGMETRICS Performance Evaluation Review*. 2009; 37(1):181-192. (In Eng.) doi: https://doi.org/10.1145/2492101.1555371

[11] Picoli I.L., Pasco C.V., Jónsson B.Þ, Bouganim L., Bonnet P. uFLIP-OC: Understanding Flash I/O Patterns on Open-Channel Solid-State Drives. *Proceedings of the 8th Asia-Pacific Workshop on Systems (APSys'17)*. Mumbai, India: Association for Computing Machinery; 2017. Article number: 20. p. 1-7. (In Eng.) doi: https://doi.org/10.1145/3124680.3124741

[12] Hu X.-Y., Eleftheriou E., Haas R., Iliadis I., Pletka R. Write amplification analysis in flash-based solid state drives. *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference (SYSTOR'09).* Association for Computing Machinery, New York, NY, USA; 2009. Article number: 10. p. 1-9. (In Eng.) doi: https://doi.org/10.1145/1534530.1534544

[13] Bayer R., Mccreight E.M. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica.* 1972; 1(3):173-189. (In Eng.) doi: https://doi.org/10.1007/BF00288683

[14] Comer D. The Ubiquitous B-Tree. *ACM Computing Surveys.* 1979; 11(2):121-137. (In Eng.) doi: https://doi.org/10.1145/356770.356776

[15] Graefe G. Modern B-Tree Techniques. *Foundations and Trends® in Databases*. 2011; 3(4):203-402. (In Eng.) doi: http://dx.doi.org/10.1561/1900000028

[16] Ang C.-H., Tan K.-P. The Interval *B*-tree. *Information Processing Letters*. 1995; 53(2):85-89. (In Eng.) doi: https://doi.org/10.1016/0020-0190(94)00176-Y

[17] Becker B., Gschwind S., Ohler T., Seeger B., Widmayer P. An Asymptotically Optimal Multiversion B-tree. *The VLDB Journal.* 1996; 5(4):264-275. (In Eng.) doi: https://doi.org/10.1007/s007780050028

[18] Elmasri R., Wuu G.T.J., Kim Y.-J. The Time Index: An Access Structure for Temporal Data. *Proceedings of the 16th International Conference on Very Large Data Bases (VLDB'90).* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.; 1990. p. 1-12. Available at: https://www.vldb.org/dblp/db/conf/vldb/ElmasriWK90.html (accessed 17.05.2022). (In Eng.)

[19] Goh C.H., Lu H., Ooi B.-C., Tan K.-L. Indexing Temporal Data Using Existing B^+-trees. *Data & Knowledge Engineering.* 1996; 18(2):147-165. (In Eng.) doi: https://doi.org/10.1016/0169-023X(95)00034-P

[20] Gottstein R., Goyal R., Hardock S., Petrov I., Buchmann A. MV-IDX: indexing in multi-version databases. *Proceedings of the 18th International Database Engineering & Applications Symposium (IDEAS'14).* Association for Computing Machinery, New York, NY, USA; 2014. p. 142-148. (In Eng.) doi: https://doi.org/10.1145/2628194.2628911

[21] Shen H., Ooi B.C., Lu H. The TP-Index: A Dynamic and Efficient Indexing Mechanism for Temporal Databases. *Proceedings of 1994 IEEE 10th International Conference on Data Engineering.* Washington, DC, USA: IEEE Computer Society; 1994. p. 274-281. (In Eng.) doi: https://doi.org/10.1109/ICDE.1994.283041

[22] Luo C., Carey M.J. LSM-based storage techniques: a survey. *The VLDB Journal*. 2020; 29(1):393-418. (In Eng.) doi: https://doi.org/10.1007/s00778-019-00555-y

[23] Lim H., Andersen D.G., Kaminsky M. Towards Accurate and Fast Evaluation of Multi-stage Log-structured Designs. *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16).* Santa Clara, CA: USENIX Association; 2016. p. 149-166. Available at: https://www.usenix.org/system/files/conference/fast16/fast16-papers-lim.pdf (accessed 17.05.2022). (In Eng.)

[24] Dong S., Callaghan M.D., Galanis L., Borthakur D., Savor T., Strum M. Optimizing Space Amplification in RocksDB. *Proceedings of the 8th Biennial Conference on Innovative Data Systems Research (CIDR 2017).* Chaminade, CA, USA; 2017. p. 1-9. Available at: https://www.cidrdb.org/cidr2017/papers/p82-dong-cidr17.pdf (accessed 17.05.2022). (In Eng.)

[25] Petrov A. Algorithms Behind Modern Storage Systems. *Communications of the ACM*. 2018; 61(8):38-44. (In Eng.) doi: https://doi.org/10.1145/3209210

[26] Qader M.A., Cheng S., Hristidis V. A Comparative Study of Secondary Indexing Techniques in LSM-based NoSQL Databases. *Proceedings of the 2018 International Conference on Management of Data (SIGMOD'18).* Houston, TX, USA: ACM; 2018. p. 551-566. (In Eng.) doi: https://doi.org/10.1145/3183713.3196900

[27] Wang P., Sun G., Jiang S., Ouyang J., Lin S., Zhang C., Cong J. 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 (EuroSys'14).* Amsterdam, The Netherlands: ACM; 2014. Article number: 16. p. 1-14. (In Eng.) doi: https://doi.org/10.1145/2592798.2592804

[28] Yang F., Dou K., Chen S., Hou M., Kang J. -U., Cho S. Optimizing NoSQL DB on Flash: A Case Study of RocksDB. *2015 IEEE 12th Intl Conf on 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).* IEEE Computer Society; 2015. p. 1062-1069. (In Eng.) doi: https://doi.org/10.1109/UIC-ATC-ScalCom-CBDCom-IoP.2015.197

[29] Lersch L., Oukid I., Lehner W., Schreter I. An Analysis of LSM Caching in NVRAM. *Proceedings of the 13th International Workshop on Data Management on New Hardware (DAMON'17).* Association for Computing Machinery, New York, NY, USA; 2017. Article number: 9. p. 1-5. (In Eng.) doi: https://doi.org/10.1145/3076113.3076123

[30] Muth P., O'Neil P., Pick A., Weikum G. The LHAM Log-structured History Data Access Method. *The VLDB Journal*. 2000; 8(3-4):199-221. (In Eng.) doi: https://doi.org/10.1007/s007780050004

[31] Singhal M. Update transport: a new technique for update synchronization in replicated database systems. *IEEE Transactions on Software Engineering.* 1990; 16(12):1325-1336. (In Eng.) doi: https://doi.org/10.1109/32.62441

Поступила 17.05.2022; одобрена после рецензирования 29.06.2022; принята к публикации 11.07.2022.
Submitted 17.05.2022; approved after reviewing 29.06.2022; accepted for publication 11.07.2022.

Об авторе:

Неганов Алексей Михайлович, аспирант кафедры микропроцессорных технологий в интеллектуальных системах, факультет радиотехники и кибернетики, ФГАОУ ВО «Московский физико-технический институт (национальный исследовательский университет)» (141701, Российская Федерация, Московская область, г. Долгопрудный, Институтский переулок, д. 9), **ORCID: https://orcid.org/0000-0003-4451-5332,** neganovalexey@gmail.com

Автор прочитал и одобрил окончательный вариант рукописи.

About the author:

Aleksei M. Neganov, Postgraduate Student of the Chair of Microprocessor Technologies in Intelligent Systems, Department of Radio Engineering and Cybernetics, Moscow Institute of Physics and Technology (National Research University) (9 Institutskiy per., Dolgoprudny 141701, Moscow Region, Russian Federation), **ORCID: https://orcid.org/0000-0003-4451-5332,** neganovalexey@gmail.com

The author has read and approved the final manuscript.