УДК 004.657 DOI: 10.26102/2310-6018/2021.32.1.014

Технология выполнения поисковых запросов к базе данных на основе метода индексации данных CW-tree

В.С. Шевский, Ю.А. Шичкина

ФГАОУ ВО Санкт-Петербургский государственный электротехнический университет «ЛЭТИ» им. В.И. Ульянова (Ленина)
Санкт-Петербург, Российская Федерация

Резюме. На сегодняшний день в информационных технологиях наблюдается тенденция к многократному увеличению объемов хранимых данных. Наращивание объемов данных обусловлено глобальной цифровизацией различных сфер жизнедеятельности человека, распространения использования датчиков мониторинга, диагностирования и управления различными объектами. Несмотря на растущие объемы, по-прежнему требуется обработка данных. Методы обработки включают важный этап поиска, скорость выполнения которого сказывается на эффективности всей обработки. Поэтому разработки в области ускоренного поиска необходимых данных для интеллектуального анализа в различных базах данных являются актуальными. В данной статье предлагается разработанный авторами алгоритм на основе структуры данных CW-tree, который позволяет индексировать данные, максимально используя возможности вычислительной системы в условиях многопоточной обработки запросов. Структура данных CW-tree, также предложенная авторами, содержит два уровня: уровень ветвей, который предназначен для поиска вершины по заданному в запросе пользователем условию, и уровень листьев, который предназначен для хранения данных. В настоящей работе описан метод прохода по уровню листьев CW-tree при выполнении поискового запроса к базе данных. Также приведены результаты тестирования предложенного метода на тестовой базе данных, результаты сравнительного анализа выполнения поисковых запросов к базе данных, основанной на структуре CW-tree, и базе данных под управлением СУБД MySQL.

Ключевые слова: системы управления базами данных, алгоритмы деревьев, индексирование данных, многопоточность, оптимизация запросов к базам данных, CW-tree.

Для цитирования: Шевский В.С., Шичкина Ю.А. Технология выполнения поисковых запросов к базе данных на основе метода индексации данных CW-tree. *Моделирование, оптимизация и информационные технологии.* 2021;9(1). Доступно по: https://moitvivt.ru/ru/journal/pdf?id=913 DOI: 10.26102/2310-6018/2021.32.1.014

Technology for executing retrieval queries to a database based on the CW-tree data indexing method

V.S. Shevskiy, Y.A. Shichkina

FSAEI OF HE Saint Petersburg State Electrotechnical University "LETI" named after V.I. Ulyanov (Lenin),
Saint-Petersburg, Russian Federation

Abstract: There is a tendency to multiplying the volume of stored data in information technology today. The increase in data volumes is due to the global digitalization of various spheres of human life, the expansion of using sensors for monitoring, diagnosing, and controlling diverse objects. Despite the growing volumes, data still needs to be processed. Processing methods include an important retrieval step which speed affects the efficiency of the entire processing. Therefore, developments in the field of accelerated retrieval for the necessary data for mining in various databases are relevant. This article proposes an algorithm developed by the authors based on the CW-tree data structure. It allows data to be indexed by maximizing the capabilities of a computing system in conditions of multithreaded query

processing. The CW-tree data structure, also proposed by the authors, contains two levels. The branch level is designed to retrieve a vertex according to a user-specified query. The leaf level is used to store data. This paper describes a method for traversing the CW-tree leaf-level when executing a retrieval query to the database. Test results of the proposed method on a test database are also given, the results of a comparative analysis of the execution of retrieval queries to a database based on the CW-tree structure and a database controlled by the MySQL DBMS are presented.

Keywords: database management systems, tree algorithms, data indexing, multithreading, database query optimization, CW-tree.

For citation: V. S. Shevskiy, Y. A. Shichkina. Technology for executing retrieval queries to a database based on the CW-tree data indexing method. *Modeling, Optimization and Information Technology*. 2021;9(1). Available from: https://moitvivt.ru/ru/journal/pdf?id=913 DOI: 10.26102/2310-6018/2021.32.1.014 (In Russ).

Введение

Аналитическое агентство по исследованию рынка в области мировой информации Research and Markets в своем отчете от 02 марта 2020 года оценили глобальный рынок аналитики больших данных в 37,34 млрд долларов США в 2018 году и 105,08 млрд долларов США к 2027 году при среднегодовом росте в 12,3% в течение прогнозируемого периода с 2019 по 2027 год [1].

Большинство экспертов по большим данным согласны с тем, что объем генерируемых данных будет расти в геометрической прогрессии в будущем. В своем отчете Data Age 2025 международная исследовательская и консалтинговая компания International Data Corporation (IDC), занимающаяся изучением мирового рынка информационных технологий и телекоммуникаций, прогнозирует, что глобальная база данных достигнет 175 зетабайт к 2025 году [2].

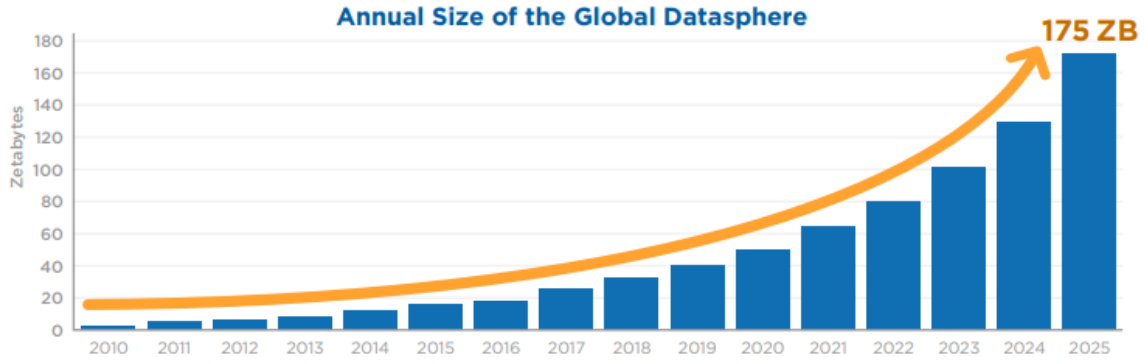

Рисунок 1 – Ежегодный рост данных [2]
Figure 1 – Annual data growth

Однако, производительность вычислительных систем растет медленнее и в основном только за счет увеличения числа вычислительных узлов в многопроцессорных системах. Поэтому решение проблемы повышения скорости обработки запросов в базах данных на основе имеющегося оборудования становится все более актуальной и будет сохранять свою актуальность еще на протяжении 5-10 лет.

С учетом высокой степени актуальности решения проблемы ускорения обработки данных большого объема программным способом в мире проводится достаточно большое число исследований по данной проблематике. В основном эти исследования касаются модификации и адаптации существующих типов деревьев к обработке

пространственных, многомерных данных и выполнения запросов в базах данных типа in-memory.

Однако существующие методы индексирования данных в постоянной памяти, хорошо зарекомендовавшие себя для последовательного выполнения запросов, не во всех случаях организации параллельной обработки запросов являются такими же успешными. Например, методы индексирования, основанные на B или B+-деревьях позволяют значительно ускорить выполнение нескольких параллельно исполняемых запросов, но время выполнения одного сложного запроса, включающего в себя подзапросы, с применением этих индексов будет примерно одинаковым на одноядерной и многоядерной вычислительной системе. Таким образом, очевидна неэффективность задействования возможностей многопоточности современных вычислительных систем.

В статье [3] было приведено описание CW-tree и алгоритма построения такого дерева. В данной статье предлагается к рассмотрению описание метода обхода CW-tree на уровне листьев, а также результаты сравнительного анализа выполнения поисковых SQL запросов в базе данных с применением индексирования CW-tree и в СУБД MySQL.

Статья построена следующим образом: рассмотрены основные тенденции в исследованиях, связанных с ускоренной обработкой данных в базах данных различного типа; приведено описание метода обхода CW-tree на уровне листьев и рассмотрен пример, демонстрирующий работу метода; представлены результаты тестирования поисковых запросов к базе данных с применением CW-tree и к базе данных MySQL, в заключении подведены итоги и намечены планы на дальнейшие исследования.

Деревья давно зарекомендовали себя как наиболее эффективные инструменты для оптимизации доступа к данным в хранилищах различного типа. Анализ публикаций в области применения деревьев в базах данных позволяет выделить следующие основные направления развития теории оптимизации доступа к данным на основе деревьев.

1. Применение деревьев для организации многомерных данных и пространственных данных. Применительно к многомерным данным первенство удерживают разновидности R-деревьев, X-деревьев и TV-деревьев [4], [5]. Применительно к пространственным данным часто используются различные модификации Quadtree [6].
2. Применение деревьев для доступа к данным в оперативной памяти. Во многих базах данных для случаев, когда востребованные («горячие») данные полностью хранятся в оперативной памяти используются T-деревья [7]. К таким базам данных относятся In-memory базы данных: Datablitz, EXtremeDB, MySQL Cluster, Oracle TimesTen и MobileLite. Усовершенствованный вариант T-деревьев, так называемое ART, адаптивное основополагающее дерево (trie), для индексации данных в оперативной памяти представлено [8]. Еще одним интересным подходом к компоновке дерева индексов, чувствительным к архитектуре вычислительной системы, является FAST. Это двоичное дерево, логически оптимизированное по таким параметрам как размеры страницы и кэша. FAST позволяет перестраивать деревья индексов менее чем за 0,1 секунды для наборов данных размером до 64Mkeys и применяет методы сжатия [9].
3. Индексирование данных на основе деревьев в специализированных базах данных. Например, в базах данных изображений для поиска похожих изображений применяются GC-tree (or the grid cell tree) [10, 11], нечеткие S-деревья [12], в мультимедиа базах данных применяются M-деревья, A-деревья, NB-деревья [13].
4. Деревья для организации данных в постоянной памяти. К наиболее часто используемым типам деревьев для индексации данных в постоянной памяти относятся B, B+, LSM-деревья. Но, в литературе можно также встретить,

алгоритмы, основанные, например, на VT-деревьях [14], которые являются расширением LSM-деревьев. В статье [15] рассматривается параллельная система B+-деревьев на основе фреймворка Hadoop. Новый подход к хранению данных и обработке запросов с использованием дерева взвешенных предикатов предлагается в [16].

Проанализировав эти и другие публикации за последние 4 года, можно сделать вывод о том, что технология индексирования на основе деревьев является достаточно устоявшейся технологий с набором типов деревьев для пространственных, многомерных, бинарных, мультимедийных баз данных с учетом их расположения в постоянной или оперативной памяти. Алгоритмы доступа к данным на основе этих деревьев не учитывают свойства SMP архитектур вычислительной системы.

В настоящей работе предлагается подход к индексированию данных на основе CW-tree, которое представляет собой гибкую архитектуру дерева, настраиваемую под число ядер вычислительной системы, а также алгоритмы обхода CW-tree, которые максимально ориентированы на принцип многопоточности вычислительной системы, что позволяет задействовать все ресурсы системы для ускорения процесса поиска необходимых данных.

Материалы и методы

Постановка задачи. Пусть имеется вычислительная система с SMP архитектурой. В данном случае под SMP (Symmetric MultiProcessing) архитектурой понимается архитектура многопроцессорных компьютеров, в которой два или более одинаковых процессора сравнимой производительности подключаются единообразно к общей памяти (и периферийным устройствам). Процессоры тесно связаны друг с другом через общую шину и имеют равный доступ ко всем ресурсам вычислительной системы (памяти и устройствам ввода-вывода) и управляются все одной копией операционной системы [17].

Необходимо выполнить поисковый запрос к базе данных на данной вычислительной системе. Тип базы данных может быть любой от реляционной базы данных до NewSQL. Основной задачей являлся поиск подхода к индексированию данных на основе дерева таким образом, чтобы можно было максимально задействовать все доступные вычислительные узлы системы.

Данная задача была решена авторами с помощью построения дерева специальной архитектуры, которая подробно описана в [3]. Там же описаны методы построения и модификации данного дерева.

В данной статье описывается алгоритм обхода дерева для поиска необходимых данных. Все примеры для тестирования запросов представлены на языке SQL, но данный подход может применяться не только к реляционным базам данных, но и к NoSQL базам данных документного типа или семейства столбцов. Особенно удачно данный подход к индексированию данных будет работать для баз данных типа ключ-значение, т.к. сам принцип построения данных основан на понятиях ключа и значения. Реляционная модель выбрана для тестирования как наиболее сложная в части организации взаимосвязей между объектами и частого использования сложных многоуровневых запросов. Это сделано с целью получения оценки эффективности предлагаемого подхода к наиболее сложно-структурированным данным.

Рассмотрим метод обхода CW-поддерева на уровне листьев.

Алгоритм индексирования данных CW-tree группирует данные в формате поддеревьев CW-tree. Каждое CW-tree содержит уровень ветвей и уровень листьев. Уровень ветвей содержит вершины, хранящие данные и индексы, а также информацию о соседних вершинах. Уровень листьев содержит только данные. Уровень ветвей

используется для того, чтобы осуществлять быстрый поиск необходимых данных по индексу. Также каждая вершина в ветви хранит такие же данные, которые хранятся в вершине на уровне листьев с тем же индексом. Такая организация хранения данных обусловлена тем, что при условии поиска данных по одному индексу отпадает необходимость спускаться на уровень листьев. В этом случае необходимые данные на этапе нахождения вершины на уровне ветвей будут уже получены. Уровень листьев предназначен для поиска множества данных при запросе с применением более чем одного ключа. Поскольку в алгоритме CW-tree подразумевается использование нескольких стартовых вершин на уровне ветвей, эти стартовые вершины могут быть использованы при проходе через уровень листьев. В данной работе описывается CW-tree с одним поддеревом.

Пример CW-tree с двумя уровнями представлен на Рисунке 2.

Представленное на Рисунке 2 дерево состоит из уровня ветвей, отображенного в верхней части рисунка 1, и уровня листьев, отображенного в нижней части. В уровне ветвей хранится 31 вершина. Хотя каждый узел может хранить произвольное количество вершин, в данном примере каждый узел хранит одну вершину. Номер вершины отображает ее индекс. Красным цветом помечены стартовые вершины. Уровень листьев содержит тоже 31 вершину и представлен в формате списка.

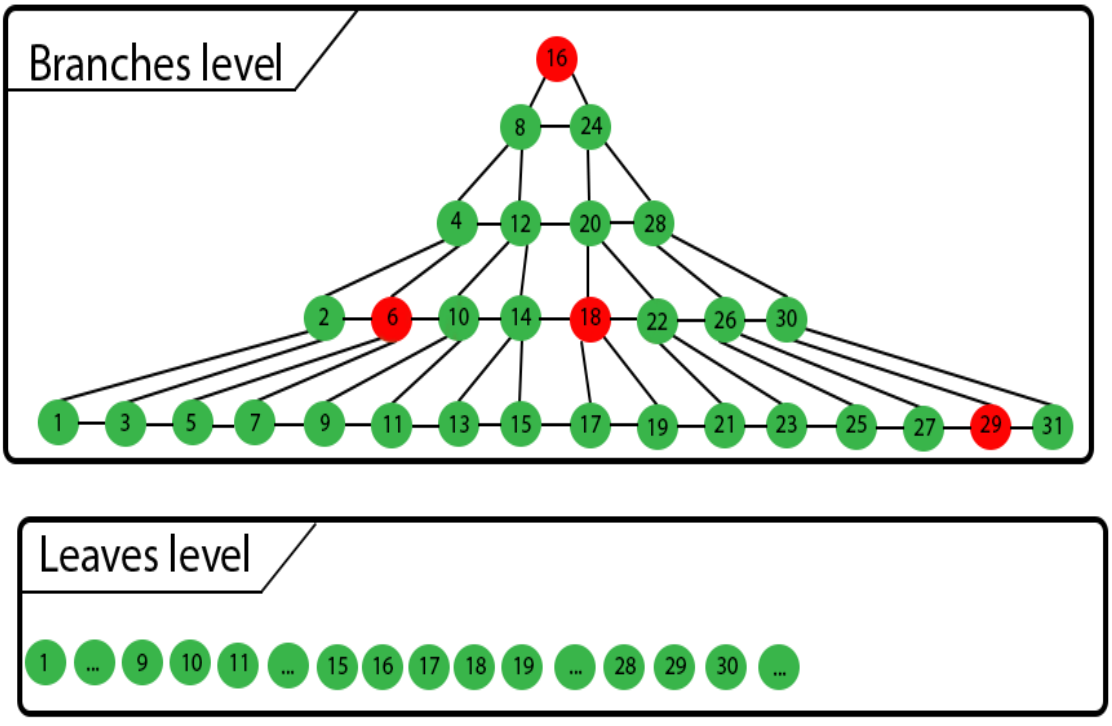

Рисунок 2 – Представление CW-tree с двумя уровнями
Figure 2 – CW-tree view with two levels

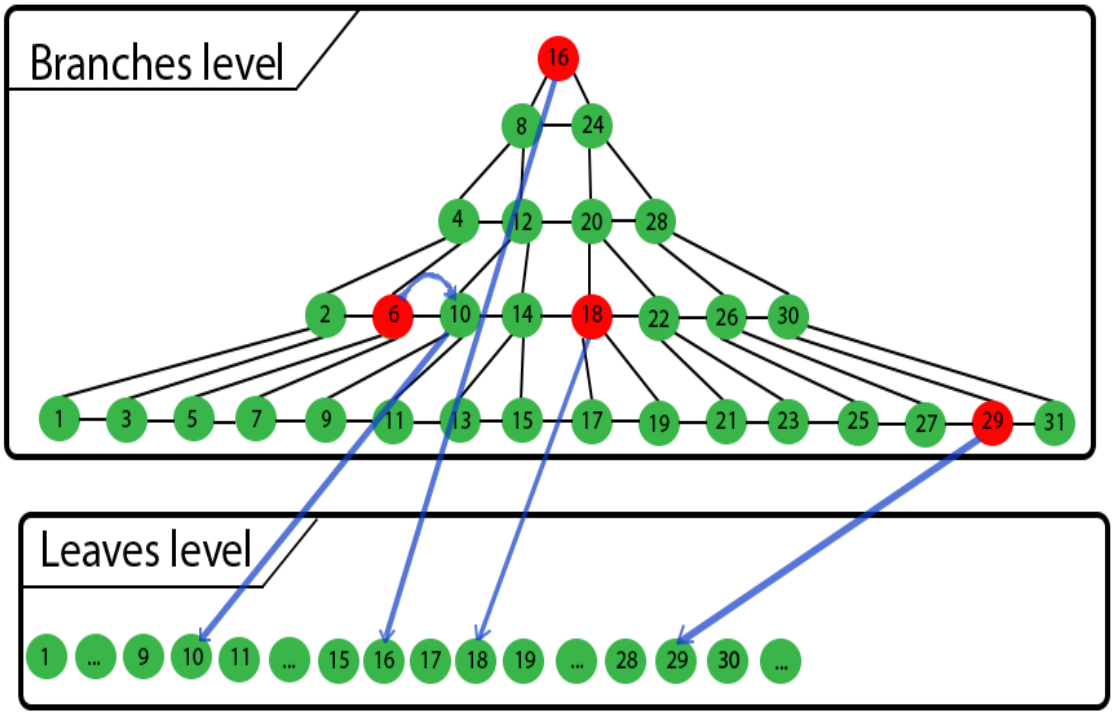

Рисунок 3 – Поиск вершины в CW-tree с индексом равным или более 10 на уровне ветвей
Figure 3 – Retrieval a vertex in a CW-tree with an index equal to or more than 10 at the branch level

Для демонстрации процесса поиска рассмотрим пример обработки запроса на получение всех данных, у которых ключевой индекс больше или равен 10. Процесс поиска таких данных в CW-tree будет выглядеть следующим образом (Рисунок 3).

Как было показано на Рисунке 3, поддерево содержит 4 стартовые вершины с индексами 6, 16, 18, и 29. Пусть потоки, которые выполняют обход CW-tree на уровне ветвей из стартовых вершин обозначены как I, II, III, и IV для соответствующих стартовых вершин. Потоки II, III и IV заканчивают поиск среди ветвей за один шаг, поскольку индексы 16, 18, и 29 больше 10. Поток 1 инициализируется в вершине 6, затем переходит в соседнюю вершину 10 и также завершает поиск среди ветвей. Таким образом, как следует из примера, поиск в потоке I завершится тогда, когда будет найдена первая вершина, удовлетворяющая условие запроса, а именно – вершина с индексом 10. С учетом того, что запрос содержит условие, удовлетворить которое могут более одной вершины, все четыре потока переходят на уровень листьев. Отметим, что, если бы запрос содержал условие, которые может удовлетворить только одна вершина, проход по CW-tree закончился бы сразу после нахождения вершины на уровне ветвей. Переход из уровня ветвей в уровень листьев возможен благодаря тому, что каждая вершина-ветвь хранит информацию о соответствующей вершине-листе. Отсюда следует, что поток I переходит в вершину 10 на уровне листьев, поток II переходит в вершину 16, поток III переходит в вершину 18 и поток IV переходит в вершину 29. Данный процесс представлен на Рисунке 3. Затем выполняется проход по уровню листьев (Рисунок 4).

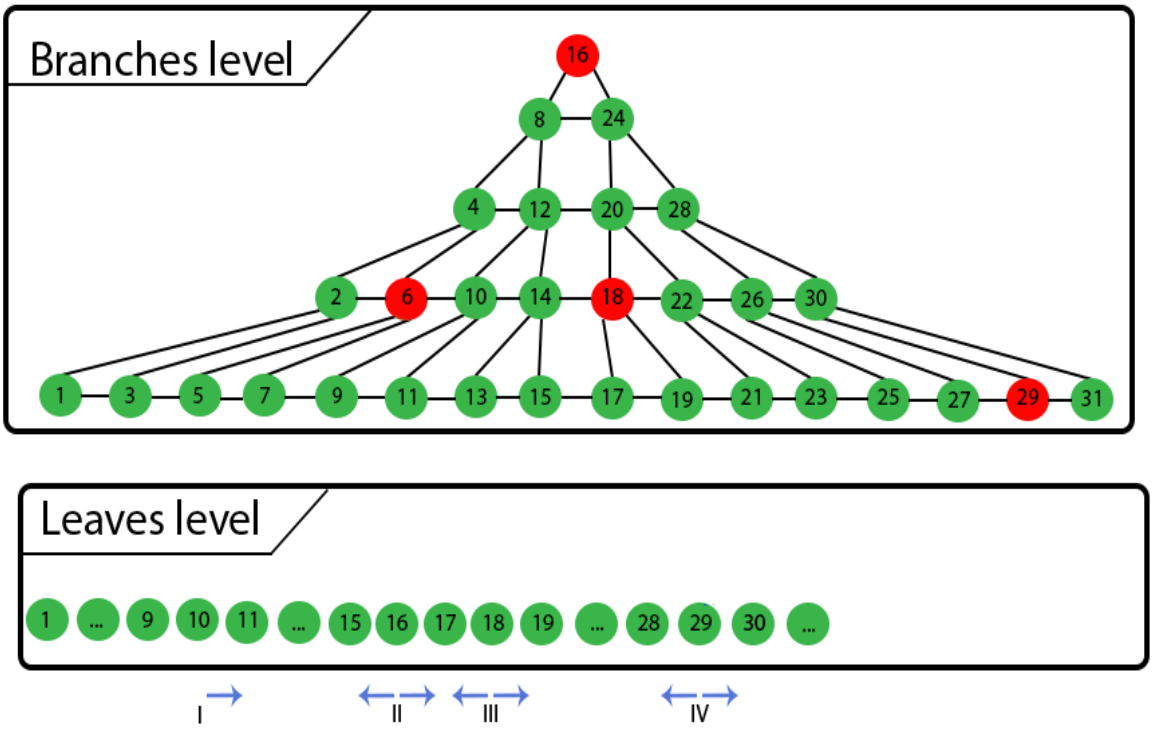

Рисунок 4 – Проход по уровню листьев CW-tree
Figure 4 – Traversal through the leaf level CW-tree

Поток I выполняет проход только в сторону увеличения индексов. Это связано с тем, что еще в начале прохода на уровне листьев было очевидно, что все вершины, которые хранятся левее вершины с индексом 10 не будут удовлетворять первоначальному условию. Потоки II, III и IV выполняют проход как в сторону увеличения, так и в сторону уменьшения индексов. Ниже представлено общее формализованное описание метода обхода CW-tree на уровне листьев.

Входные данные:

Количество потоков, которые будут использованы для обхода CW-tree:

$$z \in Z \quad (1)$$

где Z – поле целых чисел.

Набор целочисленных значений, которые обозначают состояния потоков. Каждое состояние отвечает за продолжение выполнения обхода:

$$S = \{s_i\}, i = \{0, \dots, z\} \quad (2)$$

Набор вершин, из которых начинается проход в CW-tree на уровне листьев

$$V = \{v_i\}, i = \{0, \dots, z\} \quad (3)$$

Набор вершин, составляющий уровень листьев CW-tree, по которому будет осуществляться проход:

$$D = \{d_i\}, i = \{0, \dots, |D|\} \quad (4)$$

Вершина-лист CW-tree:

$$d \in D \quad (5)$$

Двумерный набор вершин, выбранных в процессе обхода в сторону уменьшения индексов вершин из D для каждого потока, участвующего в обходе D:

$$L = \{l_i\}, i = \{0, \dots, z\} \quad (6)$$

Двумерный набор вершин, выбранных в процессе обхода в сторону увеличения индексов вершин из D для каждого потока, участвующего в обходе D:

$$R = \{r_i\}, i = \{0, \dots, z\} \quad (7)$$

Условие выбора данных, задаваемое пользователем в SQL запросе:

$$condition(d) \quad (8)$$

Используя определения (1)-(8), можно представить формулу асинхронного обхода в сторону уменьшения индексов вершин из *D*:

$$F(v) = \begin{cases} H(v) & W(v) \,!= 1 \wedge W(v) \,!= 3 \wedge i > 0 \\ exit & !\,(W(v) \,!= 1 \wedge W(v) \,!= 3 \wedge i > 0) \end{cases} \quad (9)$$

Функция (9) выполняет обход в сторону уменьшения индексов вершин с помощью формулы *H(v)*, которая будет определена чуть позже. Обход происходит до тех пор, пока состояние вершины, которая имеет самый большой индекс из тех, которые меньше *v*, из множества *V*, не равно 1 и не равно 3, а также счетчик вершин больше нуля.

Вспомогательные формулы, необходимые для определения формулы (9):

$$T(v) = \{v - V_i\}, i = \{0, \dots, |V|\}, v \in V, (v - V_i > 0) \quad (10)$$

Функция (10) возвращает набор всех разностей индексов двух вершин, где уменьшаемым является индекс вершины v, а вычитаемым – индексы всех остальных вершин из множества *V*, описанного в *(3)*, при условии, что эта разность является положительной.

$$W(v) = \{V_i \mid V_i < v \,\&\, (v - V_i) = \min\big(T(v)\big), i = \{0, \dots, |V|\}, \quad (11)$$

Функция (11) возвращает вершину из множества *V*, описанного в (3), индекс которой меньше индекса вершины v и при этом такую, что разность индекса вершины v и индекса вершины из множества *V* (3) будет минимальной.

$$G(v) = s, \exists!\ s \in S, v \in V \quad (12)$$

Функция (12) возвращает значение состояния *S*, описанные в (2), по ключу *v*, который является аргументом функции (12). Данная функция является реализацией структуры данных «ключ-значение».

$$B(v) = l, \exists!\, l \in L, v \in V \quad (13)$$

Функция (13) возвращает вершину из множества *L*, представленного в (6), по ключу *v*, который является аргументом функции (13). Данная функция является реализацией структуры данных «ключ-значение».

$$P(v, d) = \begin{cases} L = L \cup d & condition(d) \\ G\big(W(v)\big) = 3, v = W(v) & d \le W(v) \end{cases} \quad (14)$$

Если условие пользователя удовлетворяются для вершины *d*, то есть функция *condition(d)* возвращает значение true, то функция (14) добавляет аргумент *d* в множество *L*, представленное в (6). Если индекс вершины *d* меньше либо равен индексу вершины, которая имеет самый большой индекс из тех, которые меньше v, из множества *V*, то выполняется 2 действия: 1) функция (14) помечает статус значением 3 для вершины, которая имеет самый большой индекс из тех, которые меньше v, из множества *V*; 2) вместо вершины, которая имела самый большой индекс из тех, которые меньше v, из множества V, присваивает вершине v статус стартовой вершины.

$$H(v) = (G(v) = G(v)||1), P(v, D_i), i = i - 1, H(v) \quad (15)$$

Функция (15) выполняет рекурсивный обход уровня листьев CW-поддерева в сторону, как указывалось ранее, уменьшения индексов вершин. На каждой итерации обхода функция (15) выполняет несколько действий: 1) побитовую операцию «или», где первым операндом является состояние вершины v, а вторым – константа 1; 2) вызывает функцию *P*, определенную в (14); 3) декрементирует счетчик текущей вершины.

Формула асинхронного обхода в сторону увеличения индексов вершин из D:

$$Q(v) = \begin{cases} J(v) & Z(v) \,!= 2 \wedge Z(v) \,!= 3 \wedge i < |D| \\ exit & !\,(Z(v) \,!= 2 \wedge Z(v) \,!= 3 \wedge i < |D|) \end{cases} \quad (16)$$

Функция (16) выполняет обход в сторону увеличения индексов вершин с помощью функции *J(v)*, которая будет определена ниже. Обход происходит до тех пор,

пока состояние вершины, которая имеет самый малый индекс из тех, которые больше индекса вершины v, из множества V, не равно 2 и не равно 3, а также счетчик вершин не превышает количество вершин-листьев.

Вспомогательные формулы, необходимые для определения формулы (16):

$$Z(v) = V_i, i = \{0, \dots, |V|\}, v \in V, V_i > v, (V_i - v) = \min\,(T(v)) \quad (17)$$

Функция (17) возвращает вершину из множества V(3), индекс которой больше индекса вершины v и такую, что разность индекса этой вершины из множества V(3) и индекса вершины v будет минимальной.

$$N(v) = r, \exists! r \in R, v \in V \quad (18)$$

Функция (18) возвращает вершину из множества R(7) по ключу v, который является аргументом функции (18). Данная функция является реализацией структуры данных «ключ-значение».

$$X(v,d) = \begin{cases} R = R \cup d & condition(d) \\ G\big(Z(v)\big) = 3, v = Z(v) & d \geq Z(v) \end{cases} \quad (19)$$

Если условие пользователя удовлетворяются для вершины d, то есть функция *condition(d)* возвращает значение true, то функция (19) добавляет аргумент d в множество R (7). Если индекс вершины d больше либо равен индексу вершины, которая имеет самый малый индекс из тех, которые больше индекса вершины v, из множества V, то выполняется 2 действия: 1) функция (19) помечает статус значением 3 для вершины, которая имеет самый малый индекс из тех, которые больше индекса вершины v, из множества V; 2) вместо вершины, которая имела самый малый индекс из тех, которые больше v, из множества V, присваивает вершине v статус стартовой вершины.

$$J(v) = (G(v) = G(v)||2), X(v, D_i), i == i + 1, J(v) \quad (20)$$

Функция (20) выполняет рекурсивный обход уровня листьев CW поддерева в сторону увеличения индексов. На каждой итерации обхода функция (20) выполняет несколько действий: 1) побитовую операцию «или», где первым операндом является состояние вершины v, а вторым – константа 2; 2) вызывает функцию X (19); 3) инкрементирует счетчик текущей вершины.

Формула обхода CW-tree на уровне ветвей:

$$move(x) == \begin{cases} Go\ to\ CL, if\ \text{MNC} \leq x,\ \ x \leq MXC,\ \ x < e; \\ Go\ to\ CR, if\ \text{MNC} \leq x,\ \ x \leq \text{MXC},\ \ x > e; \\ Go\ to\ LNP, if\ x \leq LN; \\ Go\ to\ LNP, if\ x < MNPK; \\ Go\ to\ RNP, if\ x \geq \text{RN}; \\ Go\ to\ RNP, if\ x > MXPK; \\ Go\ to\ PP, if\ x < MNC, x > LN; \\ Go\ to\ PP, if\ x > MXC,\ \ x < RN \end{cases} \quad (21)$$

Формула (21) содержит условия выбора, в какую вершину необходимо перейти на каждом шаге обхода CW-tree на уровне ветвей. Полное описание формулы (21) а также всех используемых в формуле обозначений представлено в работе [13].

Выходные данные:

Результирующий набор данных, полученных из вершин D, содержимое которых зависит от запроса пользователя:

$$Result = \{e\}, e \in D \quad (22)$$

Рассмотрим метод обхода по CW-tree на уровне листьев.

Шаг 1. Разделить SQL запрос на подзапросы таким образом, чтобы было возможно выполнить каждый такой подзапрос независимо от других в асинхронном режиме.

Шаг 2. Осуществить обход CW-tree на уровне ветвей в соответствии с формулой (21) для каждого подзапроса, полученного на предыдущем шаге.

Шаг 3. Ожидание обхода CW-tree на уровне ветвей. Поскольку поиск происходит в два последовательных этапа – поиск на уровне ветвей и затем – поиск на уровне листьев, прежде чем начать поиск на уровне листьев, необходимо дождаться завершения поиска на уровне ветвей. В результате данного обхода будет получено $C(1)$ стартовых вершин, из который начнется обход по *D*. Дальнейшие шаги, за исключением шага 9, выполняются для каждого подзапроса в асинхронном режиме.

Шаг 4. Считать из постоянной памяти данные вершин на уровне листьев, которые были найдены в результате выполнения метода (21).

Шаг 5. Асинхронный обход в сторону уменьшения индексов вершин из *D* в соответствии с формулой (9).

Шаг 6. Асинхронный обход в сторону увеличения индексов вершин из *D* в соответствии с формулой (16).

Шаг 7. Дождаться завершения асинхронных обходов в стороны уменьшения и увеличения индексов вершин из *D*.

Шаг 8. Объединить результаты выполнения функций *F(v)* и *Q(v)* с помощью функции (23):

$$O(v) = B(v) \cup N(v) \quad (23)$$

Шаг 9. Дождаться окончания выполнения всех подзапросов.

Шаг 10. Объединить результаты выполнения всех подзапросов.

Шаг 11. Конец метода.

Далее проведем тестирование выполнения запросов в базе данных с применением CW-tree и в базе данных MySQL.

Результаты

Для того, чтобы оценить эффективность применения алгоритма индексирования данных на основе CW-tree, было проведено сравнительное тестирование выполнения запросов к базе данных на основе CW-tree и к базе данных MySQL. Обе базы данных содержали информацию в одинаковом объеме об одних и тех же объектах. Первая база данных была создана на основе структуры CW-tree, вторая база данных была создана в СУБД MySQL на основе таблиц типа InnoDB. Схема обеих баз данных представлена на Рисунке 5.

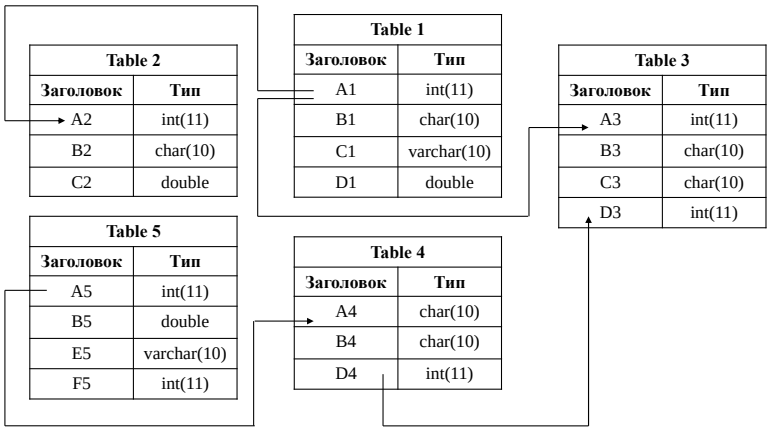

Рисунок 5 – Схема базы данных, на которой выполнялось тестирование
Figure 5 – Database schema on which testing was performed

Для каждой таблицы обеих баз данных было сгенерировано 500000 синтетических данных. Для осуществления параллельного чтения данных с физического устройства хранения данных был использован накопитель NVME через разъем M.2 (возможно также использование PCI-E при наличии соответствующего адаптера). Для физического хранения баз данных было выбрано устройство со следующими характеристиками:

Модель – INTEL MEMPEK1W016GA;

Объем – 13.41GB;

Логический размер сектора – 512B;

Физический размер сектора – 512B;

Интерфейс подключения – PCI-Express.

Для каждой базы данных было использовано отдельное устройство.

В процессе тестирования были выполнены следующие запросы:

1) select avg(A1) from Table1 union all select avg(A2) from Table2 union all select avg(D3) from Table3 union all select avg(D4) from Table4 union all select avg(F5) from Table5.

2) select * from Table3.

3) select A5, E5 from Table5 where B5 in (select (D4 + 0.3) from Table4 where B4='ddddddd' and D4 >= 150000 and D4 <= 165000).

4) select sql_no_cache B3 from Table3 union select A5 from Table5.

Время выполнения этих четырех запросов в MySQL представлены в Таблице 1. Замер времени производился с помощью последовательных MySQL команд:

Set profiling = 1;

‘query’;

show profiles;

По итогу выполнения запроса значение времени выполнения будет отображено в колонке Duration.

Результаты замера времени выполнения запросов 1, 2, 3 и 4 представлены в Таблице 1.

В соответствии с шагом 1 метода обхода CW-tree на уровне листьев, представленном в п.4, до начала обхода дерева из 4 первоначальных запросов были выделены подзапросы. Похожим образом MySQL оптимизатор формирует план выполнения запроса. Ниже представлены подзапросы в SQL форме для описания плана выполнения запросов в CW-tree, сформированные на основе каждого запроса:

1) select avg(A1) from Table1 union all select avg(A2) from Table2 union all select avg(D3) from Table3 union all select avg(D4) from Table4 union all select avg(F5) from Table5.

Таблица 1 – Время выполнения запросов 1, 2, 3 и 4 в MySQL базе данных
Table 1 – Execution times of queries 1, 2, 3 and 4 in MySQL database

Номера запросов	Время выполнения запросов в миллисекундах
1	546
2	285
3	173
4	1162

Запрос 1 разбивается на 5 подзапросов таким образом, чтобы объединение результатов выполнения этих 5 подзапросов содержало тот же набор данных, которые

будут получены при выполнении запроса 1. Подзапросы для запроса 1 представлены ниже:

1.1) select avg(A1) from Table1;
1.2) select avg(A2) from Table2;
1.3) select avg(D3) from Table3;
1.4) select avg(D4) from Table4;
1.5) select avg(F5) from Table5.
2) select * from Table3.
3) select A5, E5 from Table5 where B5 in (select (D4 + 0.3) from Table4 where B4='ddddddd' and D4 >= 150000 and D4 <= 165000):

Запрос 3 разбивается на 2 подзапроса таким образом, чтобы объединение результатов выполнения этих 2 подзапросов содержало тот же набор данных, которые будут получены при выполнении запроса 3. Подзапросы для запроса 3 представлены ниже:

3.1) select (D4 + 0.3) from Table4 where B4='ddddddd' and D4 >= 150000 and D4 <= 165000;
3.2) select A5, E5 from Table5 where B5 = result of (3.1).
4) select sql_no_cache B3 from Table3 union select A5 from Table5:

Запрос 4 разбивается на 2 подзапроса таким образом, чтобы объединение результатов выполнения этих 2 подзапросов содержало тот же набор данных, которые будут получены при выполнении запроса 4. Подзапросы для запроса 4 представлены ниже:

4.1) select B3 from Table3;
4.2) select A5 from Table5.

Замер времени в базе данных CW-tree производился с помощью C++ функции chrono::high_resolution_clock::now(). Время выполнения описанных выше подзапросов к базе данных на основе CW-tree представлены в Таблице 2.

Первый запрос был разделен на 5 независимых подзапросов. Так как в тестовом CW-tree было определено 4 стартовые вершины, каждый из 5 подзапросов был обработан 4 независимыми потоками. То есть, первый подзапрос сперва выполняется 4 потоками, которые сперва начинают поиск на уровне ветвей начиная из 4 стартовых вершин, в соответствии с формулой (21), затем, после того, как была найдена первая вершина, удовлетворяющая условие первого подзапроса, потоки переходят на уровень листьев и выполняют проход по нему в соответствии с формулами (9) и (16). Независимо от первого подзапроса, выполняются остальные 4 по тому же алгоритму.

Таблица 2 – Время выполнения запросов 1, 2, 3 и 4 в базе данных CW-tree
Table 2 – Execution times of queries 1, 2, 3 and 4 in the CW-tree database

Номера запросов	Время выполнения запросов в миллисекундах
1	524
2	143
3	74
4	243

Каждый подзапрос выполняется в своем потоке. Каждый поток начинает выполнение с одной из стартовых вершин, адреса которых сохраняются на этапе добавления новых элементов в CW поддерево. После обработки всех подзапросов

своими потоками было произведено объединение всех результатов в один общий результат. Время выполнения запроса, представленное в Таблице 2, включает общее время выполнения запроса с учетом объединения результатов.

Поскольку во втором запросе не было логических частей, которые могли бы выполниться независимо друг от друга, второй запрос не был разбит на подзапросы и был выполнен 4 независимыми потоками с помощью стартовых вершин, хранящихся в постоянной памяти.

Третий запрос был разбит на два подзапроса. Сначала был выполнен первый подзапрос с помощью 4 потоков. Затем был выполнен второй подзапрос с помощью 4 потоков на основе результата первого подзапроса. Это означает, что при выполнении второго подзапроса каждый кортеж таблицы Table5 базы данных был подвергнут проверке на соответствие значения поля B5 одному из значений, полученных из подзапроса 3.1.

Четвертый запрос был разбит на два независимых подзапроса, каждый из которых был обработан 4 потоками в асинхронном режиме. После обработки результаты обоих подзапросов были объединены в один общий результат.

Время выполнения запросов к обеим базам данных представлено на Рисунке 6.

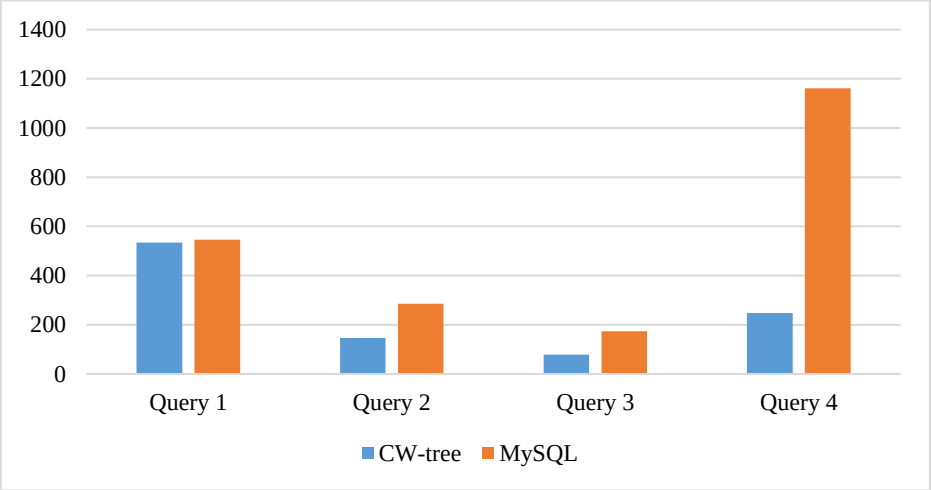

Рисунок 6 – Время выполнения (мс) 4 исходных запросов в MySQL и CW-tree в миллисекундах
Figure 6 – Execution time (milliseconds) of 4 initial queries in MySQL and CW-tree in milliseconds

Из Рисунка 6 видно, что запрос 1 выполнялся примерно одинаковое время как в MySQL, так и CW-tree; запросы 2 и 3 выполнялись быстрее в CW-tree; запрос 4 выполнялся значительно быстрее в CW-tree. Это означает, что выполнение поисковых запросов с ключевым словом SELECT с конструкциями avg, union all, in, а также для запросов, осуществляющих full table scan, в базе данных с применением индексирования CW-tree выполняется быстрее, чем в базе данных с индексированием, применяемым в MySQL.

Заключение

Проблема медленной обработки данных в базах данных больших объемов требует применение алгоритмов поиска данных, которые могут эффективно использовать аппаратные ресурсы используемой вычислительной системы, один из таких алгоритмов был предложен в данной статье.

Полученные в ходе тестирования результаты показывают, что один запрос выполняется одинаково в MySQL и в CW-tree и три выполняются быстрее в CW-tree. Тот факт, что большинство запросов быстрее выполнилось в CW-tree, доказывает эффективность алгоритма CW-tree при обработке различных SQL запросов, особенно таких, которые могут быть разделены на независимые части.

Алгоритм CW-tree позволяет оптимизировать запрос благодаря распараллеливанию поиска данных в CW дереве. Данные в дереве сортируются благодаря ключам, которые индексируют каждый элемент дерева. Именно благодаря такой сортировке, поиск данных в дереве выполняется быстрее, чем поиск в неупорядоченном множестве.

Алгоритм CW-tree подходит для индексирования таблиц, в которых ключевой индекс является уникальным и для которого определены логические операции сравнения. Например, таким может быть ключевой индекс целочисленного типа.

ЛИТЕРАТУРА

1. Research and Markets. The world’s largest market research store. Доступен по: https://www.researchandmarkets.com (дата обращения: 05.02.2021)
2. Reinsel D., Gantz J., Rydning J. The Digitization of the World From Edge to Core. Framingham: *International Data Corporation*. 2018. Доступно по: https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf (дата обращения: 05.02.2021)
3. Vladislav Shevskiy. Constantly Wide Tree for Parallel Processing. *2019 Information Systems and Technologies in Modeling and Control on CEUR-WS.org (ISSN 1613-0073). ISTMC 2019 Conference*. 2019;50-56.
4. Berchtold S.; Keim D.A., Kriegel H-P. The X-Tree: An Index Structure for High-Dimensional Data. *Readings in multimedia computing and networking.* 2001;451.
5. Wang X., Meng W., Zhang M. A novel information retrieval method based on R-tree index for smart hospital information system. *International Journal of Advanced Computer Research.* 2019;9(42):133-45. DOI: 10.19101/IJACR.2019.940030.
6. Roumelis G., Vassilakopoulos M., Corral A., Manolopoulos Y. Efficient query processing on large spatial databases: A performance study. *Journal of Systems and Software*. 2017;132:165-85. DOI: 132. 10.1016/j.jss.2017.07.005.
7. Lehman T. J., Carey M. J. A study of index structures for main memory database management systems. *University of Wisconsin-Madison Department of Computer Sciences*. 1985.
8. Leis V., Kemper A., Neumann T. The adaptive radix tree: ARTful indexing for main-memory databases. *2013 IEEE 29th International Conference on Data Engineering (ICDE).* 2013;38-49. Доступно по: https://db.in.tum.de/~leis/papers/ART.pdf (дата обращения: 05.02.2021). DOI: 10.1109/ICDE.2013.6544812
9. Kim Ch., Chhugani J., Satish N., Sedlar E., Nguyen A., Kaldewey T., Lee V.W., Brandt S.A., Dubey P. FAST: fast architecture sensitive tree search on modern CPUs and GPUs. *In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 2010;339-350. Доступно по: http://kaldewey.com/pubs/FAST__SIGMOD10.pdf (дата обращения: 05.02.2021). DOI: 10.1145/1807167.1807206.
10. Guang-Ho Ch., Chin-Wan Ch. The GC-tree: a high-dimensional index structure for similarity search in image databases. *IEEE Transactions on Multimedia.* 2002;4(2):235-247. DOI: 10.1109/TMM.2002.1017736.

11. Deniziak S., Michno T. New content-based image retrieval database structure using query by approximate shapes. *2017 Federated Conference on Computer Science and Information Systems (FedCSIS)*. 2017;613-621. DOI: 10.15439/2017F457.
12. Nowakova J.,Prilepok M., Snasel V. Medical Image Retrieval Using Vector Quantization and Fuzzy S-tree. *Journal of Medical Systems*. 2017;41(2):1-6. DOI: 10.1007/s10916-016-0659-2.
13. Fonseca M.J., Jorge, J.A. NB-Tree: An Indexing Structure for Content-Based Retrieval in Large Databases. *In Proceedings of the 8th Int. Conf. on Database Systems for Advanced Applications.* 2003;267-274
14. Shetty P., Spillane R., Malpani R., Andrews B., Seyster J., Zadok E. Building workload-independent storage with VT-trees. *11th {USENIX} Conference on File and Storage Technologies ({FAST} 13)*. 2013;17-30.
15. Ngu H.C.V., Huh J. B+-tree construction on massive data with Hadoop. *Cluster Computing*. 2019;22(1):1011–1021. Доступно по: https://doi.org/10.1007/s10586-017-1183-y (дата обращения: 05.02.2021)
16. Byung H.S., Lee B., Kyung T.K., Hee Y.Y. Enhanced query processing using weighted predicate tree in edge computing environment. *2017 IEEE Conference on Standards for Communications and Networking (CSCN).* 2017;48-53. Доступно по: https://ieeexplore.ieee.org/abstract/document/8088597 (дата обращения: 05.02.2021) DOI: 10.1109/CSCN.2017.8088597.
17. Kubiatowicz J. Introduction to Parallel Architectures and Pthreads. *Short Course on Parallel Programming.* 2013.

REFERENCES

1. Research and Markets. The world's largest market research store. Available at: https://www.researchandmarkets.com (accessed 05.02.2021)
2. Reinsel D., Gantz J., Rydning J. The Digitization of the World From Edge to Core. Framingham: *International Data Corporation*. 2018. Available at: https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf (accessed 05.02.2021)
3. Vladislav Shevskiy. Constantly Wide Tree for Parallel Processing. *2019 Information Systems and Technologies in Modeling and Control on CEUR-WS.org (ISSN 1613-0073). ISTMC 2019 Conference*. 2019;50-56.
4. Berchtold S.; Keim D.A., Kriegel H-P. The X-Tree: An Index Structure for High-Dimensional Data. *Readings in multimedia computing and networking.* 2001;451.
5. Wang X., Meng W., Zhang M. A novel information retrieval method based on R-tree index for smart hospital information system. *International Journal of Advanced Computer Research.* 2019;9(42):133-45. DOI: 10.19101/IJACR.2019.940030.
6. Roumelis G., Vassilakopoulos M., Corral A., Manolopoulos Y. Efficient query processing on large spatial databases: A performance study. *Journal of Systems and Software*. 2017;132:165-85. DOI: 132. 10.1016/j.jss.2017.07.005.
7. Lehman T. J., Carey M. J. A study of index structures for main memory database management systems. *University of Wisconsin-Madison Department of Computer Sciences*. 1985.
8. Leis V., Kemper A., Neumann T. The adaptive radix tree: ARTful indexing for main-memory databases. *2013 IEEE 29th International Conference on Data Engineering (ICDE).* 2013;38-49. Available at: https://db.in.tum.de/~leis/papers/ART.pdf (accessed 05.02.2021). DOI: 10.1109/ICDE.2013.6544812

9. Kim Ch., Chhugani J.a, Satish N., Sedlar E., Nguyen A., Kaldewey T., Lee V.W., Brandt S.A., Dubey P. FAST: fast architecture sensitive tree search on modern CPUs and GPUs. *In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 2010;339-350. Available at: http://kaldewey.com/pubs/FAST__SIGMOD10.pdf (accessed 05.02.2021). DOI: 10.1145/1807167.1807206.
10. Guang-Ho Ch., Chin-Wan Ch. The GC-tree: a high-dimensional index structure for similarity search in image databases. *IEEE Transactions on Multimedia.* 2002;4(2):235-247. DOI: 10.1109/TMM.2002.1017736.
11. Deniziak S., Michno T. New content-based image retrieval database structure using query by approximate shapes. *2017 Federated Conference on Computer Science and Information Systems (FedCSIS)*. 2017;613-621. DOI: 10.15439/2017F457.
12. Nowakova J., Prilepok M., Snasel V. Medical Image Retrieval Using Vector Quantization and Fuzzy S-tree. *Journal of Medical Systems*. 2017;41(2):1-6. DOI: 10.1007/s10916-016-0659-2.
13. Fonseca M.J., Jorge, J.A. NB-Tree: An Indexing Structure for Content-Based Retrieval in Large Databases. *In Proceedings of the 8th Int. Conf. on Database Systems for Advanced Applications.* 2003;267-274
14. Shetty P., Spillane R., Malpani R., Andrews B., Seyster J., Zadok E. Building workload-independent storage with VT-trees. *11th {USENIX} Conference on File and Storage Technologies ({FAST} 13)*. 2013;17-30.
15. Ngu H.C.V., Huh J. B+-tree construction on massive data with Hadoop. *Cluster Computing.* 2019;22(1):1011–1021. Available at: https://doi.org/10.1007/s10586-017-1183-y (accessed 05.02.2021)
16. Byung H.S., Lee B., Kyung T.K., Hee Y.Y. Enhanced query processing using weighted predicate tree in edge computing environment. *2017 IEEE Conference on Standards for Communications and Networking (CSCN).* 2017;48-53. Available at: https://ieeexplore.ieee.org/abstract/document/8088597 (accessed 05.02.2021) DOI: 10.1109/CSCN.2017.8088597.
17. Kubiatowicz J. Introduction to Parallel Architectures and PThreads. *Short Course on Parallel Programming.* 2013.

ИНФОРМАЦИЯ ОБ АВТОРАХ / INFORMATION ABOUT AUTHORS

Шевский Владислав Сергеевич, аспирант кафедры Вычислительной Техники (ВТ), Санкт-Петербургский государственный электротехнический университет «ЛЭТИ» им. В.И. Ульянова (Ленина), Санкт-Петербург, Российская Федерация
email: ImmortalGhost@yandex.ru
ORCID: 0000-0001-5121-201X

Vladislav S. Shevskiy, Postgraduate Student of the Department of Computer Science and Engineering, Saint-Petersburg, Russian Federation

Шичкина Юлия Александровна, доктор технических наук, профессор кафедры Вычислительной Техники (ВТ), Санкт-Петербургский государственный электротехнический университет «ЛЭТИ» им. В.И. Ульянова (Ленина), Санкт-Петербург, Российская Федерация
email: strange.y@mail.ru
ORCID: 0000-0001-7140-1686

Yulia A. Shichkina, Doctor of Technical Science, Department of Computer Science and Engineering, Saint-Petersburg, Russian Federation