УДК 004.657 DOI: 10.26102/2310-6018/2021.32.1.013

Разработка алгоритма индексирования данных на основе структуры данных CW-tree с применением параллельных вычислений

В.С. Шевский

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

Резюме. Цель данного научного исследования состояла в разработке алгоритма индексирования данных с использованием технологий параллельных вычислений, применение которого приводило бы к уменьшению времени на обработку запросов к базе данных. В ходе исследования был разработан алгоритм индексирования данных на основе специальной структуры хранения данных на базе дерева, получившей название CW-tree (Constantly-Wide tree). Проход по CW-tree в соответствии с предлагаемым алгоритмом индексирования осуществляется в асинхронном режиме, что достигается благодаря использованию дополнительных стартовых вершин. Как показало тестирование, по сравнению с существующими аналогами алгоритмов индексирования предложенный алгоритм на основе CW-tree обладает преимуществами, главным из которых является полное распараллеливание поиска данных как на верхнем уровне, так и на уровне листьев дерева. Для установления эффекта от использования алгоритма CW-tree было проведено сравнительное исследование, а именно: чтение данных с применением B-tree, которое широко используется практически во всех современных системах управления базами данных, и чтение данных с применением предложенного CW-tree. В ходе исследования запросы к базе данных строились таким образом, чтобы проводился поиск по определенному ключу, устанавливалось, какой подход быстрее обработает некоторое число запросов поиска. Результаты проведенного тестирования показали, что на поиск данных в CW-tree затрачивается значительно меньше времени, чем на поиск в B-tree.

***Ключевые слова*:** алгоритмы индексирования данных, структура данных дерева, параллельные вычисления, многопоточный режим, базы данных, запросы поиска, стартовая вершина, целевая вершина.

Для цитирования: Шевский В.С. Индексирование данных на основе алгоритма CW-tree с применением параллельного чтения данных. *Моделирование, оптимизация и информационные технологии.* 2021;9(1). Доступно по: Доступно по: https://moitvivt.ru/ru/journal/pdf?id=905 DOI: 10.26102/2310-6018/2021.32.1.013

Development of a data indexing algorithm based on the CW-tree data structure using parallel computing

V.S. Shevskiy

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

***Abstract*:** The purpose of this scientific research was to develop an algorithm for data indexing using parallel computing technologies, the use of which would lead to a decrease in the time for processing queries to the database. As part of the study, an algorithm for data indexing was developed based on a specific tree-based data storage structure called CW-tree (Constantly Wide tree). The CW-tree traversal under the proposed indexing algorithm is carried out in an asynchronous mode, which is achieved using additional start vertices. Testing has shown that, in comparison with existing analogs of indexing algorithms, the proposed algorithm based on the CW-tree has advantages. The main advantage is the

complete parallelization of data retrieval both at the top level and at the level of the tree leaves. To establish the effect of using the CW-tree algorithm, a comparative study was carried out, namely: reading data using the B-tree, which is widely used in almost all modern database management systems, and reading data using the proposed CW-tree. In the course of the study, queries to the database were built in such a way that a search was carried out for a specific key, it was established which approach would process a certain number of search queries faster. Testing results suggest that searching for data in the CW-tree takes much less time than search in the B-tree.

***Keywords*:** data indexing algorithms, tree data structures, parallel computing, multithreading, databases, search queries, start node, target node.

***For citation*:** Shevskiy V.S. Indexing data with CW-tree algorithm using parallel data reading. *Modeling, optimization and information technology*. 2021;9(1). Available from: https://moitvivt.ru/ru/journal/pdf?id=905 DOI: 10.26102/2310-6018/2021.32.1.013(In Russ.)

Введение

В настоящее время существуют различные виды алгоритмов нахождения необходимых данных из заданного множества данных по задаваемому условию. В основе таких алгоритмов часто используются алгоритмы индексирования данных с применением многопоточной обработки на исполняемой машине. Данные алгоритмы используются как в системах управления базами, так и в других сферах обработки и поиска данных.

В работе [1] предлагается алгоритм, предназначенный для хранения данных, который использует в основе kd-дерево (k-dimensional). Алгоритм получил название STIG (Spatio-Temporal Indexing using GPUs). Этот подход предполагает разделение дерева на 2 уровня: верхний и нижний. Верхний уровень содержит вершины с ключами для поиска нужной записи и играет роль ветвей kd-дерева. Нижний уровень содержит набор вершин с записями базы данных и играет роль листьев дерева. Интерес в данном алгоритме представляет поиск записей базы данных по некоторому заданному условию. Поиск целевой записи начинается с верхнего уровня в соответствии с алгоритмом поиска в kd-дереве. На этом этапе поиск происходит в однопоточном режиме. После нахождения вершины на верхнем уровне происходит проход по нижнему уровню с задействованием GPU (Graphics Processing Unit) в многопоточном режиме. Преимуществом данного подхода является уменьшение временных затрат на обработку листьев в дереве, т. е. более быстрый поиск записей базы, а также распределение нагрузки между CPU (Central Processing Unit) и GPU в процессе поиска целевой записи базы данных.

В работе [2] используется декартово дерево для обработки строковых данных. Изначальный массив символов, который необходимо обработать, разделяется на подмассивы, из которых формируются декартовы деревья. Данное разделение позволяет алгоритму асинхронные операции на каждом сформированном дереве путем независимой обработки каждого дерева отдельным, независимым потоком. В результате выполнения обработки формируется единое декартово дерево. Интерес в данном алгоритме представляет принцип «разделяй и властвуй» – разделение исходных данных на независимые деревья и асинхронная обработка каждого из них. Несомненным преимуществом данного подхода является уменьшение временных затрат на обработку входных данных с использованием нескольких процессоров в качестве блоков обработки. Недостатком данного подхода является увеличение временных затрат на обработку при задействовании одного процессора в качестве блока обработки. Данный недостаток отмечен авторами.

С другой стороны, некоторые исследования не ограничиваются алгоритмами на деревьях и рассматривают другие пути оптимизации запросов к базам данных, как показано в работе [3]. В исследовании [4] представлены подходы к оптимизации производительности интерфейса базы данных с использованием реляционной алгебры. Также ведутся работы и в области оптимизации запросов для массовой параллельной обработки данных [5]. В [6] рассматриваются оптимизации с использованием GPGPU для графических баз данных, в [7] исследуется MPI фреймворк для параллельного поиска в больших биологических базах данных. Особое место в оптимизации запросов к базам данных занимают масштабируемые параллельные и распределенные в разработке экспертных систем на основе баз данных с прогнозируемой балансировкой нагрузки, что подробно представлено в работе [8]. Оптимизация запросов к базе данных с целью регулирования энергоэффективности на уровне процессорного элемента CPU-GPU при интенсивных вычислениях SIMD / SPMD показано в [9]. В [10] рассматривается оптимизация методов индексации на базе автоматов с поддержкой twig-запросов для широковещательной передачи XML-данных.

На основе анализа литературы можно сделать вывод, что алгоритмы на деревьях (как, например, показано в [11]) являются основным способом индексации данных в базах данных, но обладают несовершенствами, соответственно, настоящее исследование представляется актуальным.

Таким образом, целью данного научного исследования является разработка алгоритма индексирования данных с использованием технологий параллельных вычислений, применение которого приводило бы к уменьшению времени на обработку запросов к базе данных.

Материалы и методы

Необходимо разработать алгоритм поиска данных, который позволит осуществить поиск данных по ключу быстрее, чем поиск данных по ключу в алгоритме B-tree [11].

В настоящей работе предлагается алгоритм индексирования данных, который получил название CW-tree (Constantly-Wide tree). Алгоритм CW-tree является алгоритмом структурирования данных, основанным на алгоритме B-tree.

CW-tree предназначен для хранения и считывания данных с использованием многопоточной обработки в виде группы поддеревьев, каждое из которых является CW-поддеревом
и количество которых может быть различным – одним либо более. В ходе описания тестирования в данной работе был использован алгоритм CW-tree с одним поддеревом.

CW-поддерево состоит из вершин двух типов: верхнего уровня, предназначенного для поиска нужной записи, а также уровня листьев, предназначенного для хранения данных и хранящегося в формате двусвязного списка. Процесс поиска целевых записей базы данных по определенному условию запроса, как в алгоритме STIG [1], состоит из 3-х этапов:

1) поиск соответствующего CW-поддерева;

2) прохождение верхнего уровня в поисках первой вершины, удовлетворяющей заданному условию запроса;

3) прохождения через листья в поисках всех вершин, удовлетворяющих этому условию.

Каждое поддерево содержит несколько вершин на верхнем уровне, из которых может быть начат поиск, – корневую вершину, а также дополнительные стартовые вершины. Количество дополнительных стартовых вершин обусловлено количеством

доступных ядер процессора на исполняемой машине. Поиск целевой записи происходит в многопоточном режиме как на этапе прохода по верхнему уровню, так и на этапе прохода по листьям.

В данной работе используются определения *CW-tree* и *CW-поддерево*. CW-tree обозначает объединение алгоритмов инициализации стартовых вершин и алгоритма параллельного обхода CW-поддерева. Вместе с этим CW-tree обозначает структуру данных, состоящую из некоторого количества поддеревьев, которые получили названия CW-поддеревья. CW-поддерево обозначает структуру данных в формате дерева, которое основано на уже существующей структуре данных, в данной работе это B-tree, CW-поддерево содержит дополнительные связи между вершинами, а также дополнительные стартовые вершины.

Алгоритм индексирования данных, описанный в данной работе, основан на специальном разбиении CW-tree на CW-поддеревья, каждое из которых содержит несколько стартовых вершин, при этом для хранения этих вершин и корня используется специальный блок данных. Общий алгоритм CW-tree включает в себя алгоритмы инициализации стартовых вершин и параллельного обхода CW-поддерева.

Данная работа содержит описание разработанных методов: «Инициализация стартовых вершин» для формирования дополнительных стартовых вершин, с помощью которых будет осуществляться поиск данных по ключу в параллельном режиме; «Параллельный обход CW-поддерева» для поиска данных по ключу в параллельном режиме с помощью дополнительных стартовых вершин.

Далее перейдем к описанию разработанного метода для инициализация стартовых вершин. Алгоритм хранения данных CW-tree предполагает хранение одного или более поддеревьев. CW-поддеревья реализуются по модифицированному алгоритму уже существующего алгоритма индексирования B-tree. При использовании обычного дерева без модификации в качестве CW-поддеревьев нельзя получить выгоду при параллельной обработке данного дерева. Данный раздел статьи описывает модифицированный алгоритм индексирования данных на основе B-tree, а именно добавление в дерево дополнительных стартовых вершин.

Основная идея заключается в том, чтобы начинать поиск в поддереве не только из корневой вершины, но и из дополнительных стартовых вершин, формируемых алгоритмом CW-tree. Данные вершины формируются таким образом, чтобы равномерно распределить пути поиска из всех стартовых вершин до запрашиваемой вершины. Количество дополнительных вершин должно соответствовать количеству ядер процессора либо максимально доступному количеству потоков, которое может выделить процессор на исполняемой машине.

В качестве примера сформировано дерево в соответствии с алгоритмом B-tree, обозначенное *B*. Количество вершин в *B* обозначено «l». Набор дополнительных стартовых вершин, начиная с которых, как и из корневой вершины, должен начинаться поиск, обозначен «*W*». Дерево, которое будет получено в результате преобразований исходного дерева *B* в CW-поддерево, обозначено как «*R*». Представление дерева *B* показано на Рисунке 1.

Следует отметить, что в приведенном в качестве примера дереве *B*, каждая вершина хранит один ключ. Однако это не означает, что вершина B-tree обязана хранить только один ключ. Максимальное количество хранимых ключей в вершине задается константой на этапе реализации дерева в программном коде. Для того чтобы сформировать *R*, первоначально необходимо определить размер множества *W*, а также порядок расстановки его элементов. Формирование множества *W* осуществляется при обходе дерева от корня (сверху вниз). Вместе с этим переход по узлам одного уровня

осуществляется слева направо в порядке прохода через список элементов, как показано на Рисунке 2.

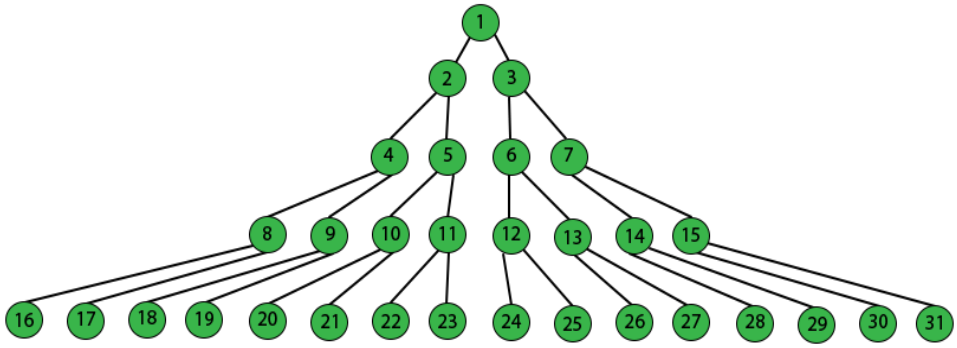

Рисунок 1 – Представление дерева B
Figure 1 – B-tree representation

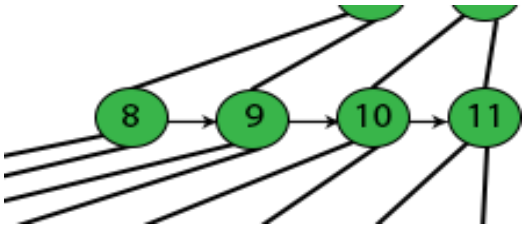

Рисунок 2 – Представление алгоритма прохода дерева с целью поиска оптимальной стартовой вершины
Figure 2 – Representation of an algorithm for traversing a tree in order to find the optimal starting vertex

Целью данного прохода является поиск оптимальных стартовых вершин. Под оптимальными стартовыми вершинами понимается такой набор вершин, который даст наибольший выигрыш при их использовании, т. е. поиск из таких вершин будет происходить быстрее, чем поиск из корневой вершины.

Спуск осуществляется в крайний левый элемент следующего уровня при условии, что проход по текущему уровню окончен. В процессе обхода в каждом узле выполняется проверка, будет ли он добавлен в множество *W*. В случае удовлетворения условий проверки узел добавляется в множество *W*, иначе происходит переход в следующий узел. Следует отметить, что множество *W* хранится в той же области памяти, что и корневая вершина. Это означает, что поиск с дополнительных стартовых вершин начинается за то же время, что и поиск с корневой вершины. Ниже представлен алгоритм инициализации дополнительных стартовых вершин.

Входные данные – составляющие множества исходного дерева:

– *V* – множество вершин;

– *E* – множество ребер;

– дерево, B-tree – B = (*V*, *E*).

Выходные данные – составляющие множества модифицированного CW-поддерева:

– – множество дополнительных стартовых вершин;

– *Z* = *V* \ *W* – множество, состоящее из всех вершин *V*, за исключением дополнительных стартовых вершин (разность множеств *V* и *W*);

– множество дополнительных ребер, описанное далее.

Определим принцип формирования наборов вершин $A(x)$ и $H(y)$, которые представляют пути от корня до вершин x и y соответственно, где x, y – соседние вершины, находящиеся на одном уровне; A – множество вершин, составляющих путь от корня до вершины x; H – множество вершин, составляющих путь от корня до вершины y.

Также необходимо использовать определения, представленные далее:

$$\begin{cases} d_1 = (a_1 = r),\ e_1 = (h_1 = r) \\ d_2 = (a_n = x),\ e_2 = (h_m = y) \end{cases} \quad (1)$$

где r – корневая вершина; a_1 – первая вершина в множестве A; a_n – последняя вершина в множестве A; h_1 – первая вершина в множестве, полученном при помощи выполнения формулы H; h_m – последняя вершина в множестве H; n – количество вершин в множестве A; m – количество вершин в множестве H.

Определения e и d по формуле (1) обозначают условия того, что некоторая вершина из множества A или множества H является корневой либо соседними вершинами x или y.

Используя определения множеств $A(x)$ и $H(y)$, а также (1), определим формулу для дополнительных ребер:

$$U = \{\{x, y\} \mid x, y \in V \cap (x \neq y) \cap (\ |A(x) \subseteq E \cap d_1 \cap d_2| = |H(y) \subseteq E \cap e_1 \cap e_2|\ )\}, \quad (2)$$

где V – множество вершин; E – множество ребер.

Определим множество вершин V’ для CW-поддерева:

$$V' = Z \cup W. \quad (3)$$

Определим множество ребер E’ для CW-поддерева:

$$E' = E \cup U. \quad (4)$$

И тогда CW-поддерево можно будет представить так:

$$R = V' \cup V' \neq \emptyset. \quad (5)$$

Этапы инициализации дополнительных стартовых вершин:

Шаг 1. Сформировать дерево B.

Шаг 2. Используя дерево B, сформировать множество дополнительных ребер U в соответствии с формулой (2).

Шаг 3. Сформировать множество вершин W. Введем вспомогательные переменные:

$$\begin{cases} p; \\ K = \{\{k_{i,1}, k_{i,2}\} \mid k_{i,1}, k_{i,2} \in V\},\ i = 1, 2, \ldots, \\ C = c_1 \wedge c_2 \wedge c_3, \end{cases} \quad (6)$$

где K – множество ребер, составляющих путь от дополнительной стартовой вершины с индексом m до текущей вершины; i – номер текущей вершины графа B, в котором совершается обход в порядке, представленном на Рисунке 2; p – количество ребер в множестве K; C – множество таких наборов ребер K, которые составляют пути от дополнительных стартовых вершин до текущей вершины, C определяется по следующим условиям: $c_1 = K \subseteq (E \cup U)$ – описывает множество всех ребер (основные и дополнительные); $c_2 = (k_{1,1} = w_m)$ – самая первая вершина дополнительного пути K должна равняться стартовой вершине; $c_3 = (a_{n,2} = V_i)$ – самая последняя вершина должна

равняться последней вершине из множества вершин *V*; индексы *m, n = 1, .., (i–1)*– номера сформированных вершин *W*; k_i – дополнительные вершины; *w* – стартовые вершины.

Также определим дополнительное условие *D*, описывающее количество еще не рассмотренных дополнительных вершин, которые актуальны с учетом еще не инициализированных дополнительных стартовых вершин.

$$D = \sum_{h=j+1}^{k+2} (h) \tag{7}$$

где *k* – константа, определяющая размер множества вершин *W*; *j* – номер текущей по очереди вершины из множества *W*; *h* – номер следующей по очереди вершины из множества *W*.

Используя условия (6) и (7), определим условие занесения вершины в множество *W*:

$$(\min(|C|) < D + (k - i)) \wedge ((\mathrm{l} - i) > (\min(|C|) * (k - i)) + (k - i)) \wedge (\min(|C|) \geq (j + 1)) \Rightarrow W_j \leftarrow V_i \tag{8}$$

где *l* – мощность множества вершин *V* дерева *B*. Иллюстрация формулы (8) была представлена выше на Рисунке 2.

В соответствии с алгоритмом прохода по CW-поддереву (Рисунок 2) при необходимости сохранения *W* либо при необходимости перезаписи *W* осуществляется проход по CW-поддереву, и на каждой вершине применяется подход, который опишем далее. В данном подходе проверяется вершина на соответствие различным условиям. Если вершина удовлетворяет всем условиям, то она помечается и в дальнейшем сохраняется в постоянной памяти как *W* вершина, т. е. адрес вершины сохраняется в специальный блок памяти запоминающего устройства. Помимо ранее описанных обозначений переменных, данный подход подразумевает использование некоторых ключевых фрагментов. Фрагмент $|C|$ означает длину пути в дереве *B* от одной из *W* вершин, которые уже были сохранены в постоянной памяти, либо от корня, если не было сохранено ни одной *W* вершины. Фрагмент $\min(|C|)$ означает минимальную длину пути в дереве *B* от одной из *W* вершин, которые уже были сохранены в постоянной памяти, либо от корня, если не было сохранено ни одной *W* вершины. Фрагмент *D* означает минимально необходимое количество вершин дерева *B*, которые еще не пройдены на текущем шаге в соответствии с алгоритмом прохода дерева, иллюстрация которого представлена на Рисунке 2. Алгоритм прохода дерева:

Шаг 4. Сформировать множество вершин *Z*.

Шаг 5. Сформировать множество вершин *CW*-поддерева в соответствии с формулой (3).

Шаг 6. Сформировать множество ребер *CW*-поддерева в соответствии с формулой (4).

Шаг 7. Сформировать *CW*-поддерево в соответствии с формулой (5).

Шаг 8. Конец алгоритма.

Формула (8) определяет дополнительные стартовые вершины, равномерно распределяя *W* по дереву, чтобы каждая вершина в дереве могла быть найдена с помощью одной из *W* вершин за меньшее количество шагов, чем считывание без помощи *W*, начиная с корня поддерева. Количество *W* вершин определяется в соответствии с аппаратными ресурсами вычислительной машины, на которой осуществляется работа алгоритма CW-tree. Чем больше это количество, тем больший охват вершин *W* будет по CW-поддереву и, соответственно, увеличивается шанс того, что вершина, которую будет необходимо найти пользователю, будет найдена быстрее

с помощью одной из вершин *W*, нежели с помощью только корневой вершины поддерева.

Далее рассмотрим метод параллельного обхода CW-поддерева.

В основе алгоритма параллельного обхода лежит разбиение каждого CW-поддерева в исходном дереве таким образом, чтобы использовать ядра процессора для параллельного поиска целевого элемента в дереве. Для того чтобы параллельный проход по дереву стал возможен, необходимо:

1. Определить несколько стартовых узлов, с которых может начинаться проход по дереву.
2. Определить алгоритм поиска целевой вершины из каждого стартового узла.
3. Обеспечить параллельный проход по дереву по каждому определенному ранее маршруту независимыми потоками.

Проблему определения стартовых узлов решает алгоритм формирования CW-поддерева. Данный алгоритм добавляет связи от корня в такие вершины, чтобы минимизировать проход до любой вершины. Возьмем в качестве примера CW-поддерево, основанное на B-tree. Пример представлен на Рисунке 3.

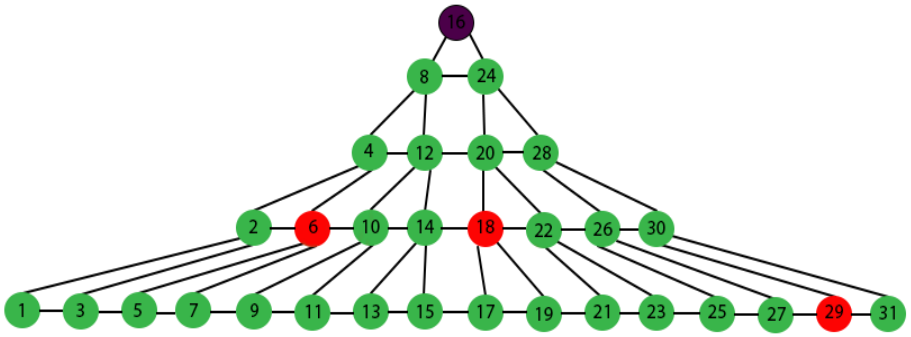

Рисунок 3 – Пример CW-поддерева
Figure 3 –CW-subtree example

Обозначим дерево, представленное на Рисунке 3, через *R*. Каждая вершина дерева *R* имеет свой номер, который соответствует ее целочисленному ключу. Индексом 16 отмечена корневая вершина. Индексами 6, 18 и 29 отмечены дополнительные вершины, с которых начинается параллельный поиск при использовании дополнительных потоков. Каждая вершина в CW-поддереве должна хранить следующую информацию:

– индекс текущей вершины;

– индекс левого соседа относительно текущей вершины, который обозначим как *T*;

– указатель на левого соседа относительно текущей вершины, который обозначим как *S*;

– индекс правого соседа относительно текущей вершины, который обозначим как *I*;

– указатель на правого соседа относительно текущей вершины, который обозначим как *O*;

– указатель на родительскую вершину, который обозначим как *P*;

– указатели на дочерние вершины, которые обозначим как *F* и *Q*, где *F* – левая дочерняя вершина, а *Q* – правая дочерняя вершина;

– индекс минимальной вершины среди дочерних вершин, который обозначим как *N*;

– индекс максимальной вершины среди дочерних вершин, который обозначим как *M*;

– элементы, хранящие пользовательские данные, ключом текущего элемента обозначим *e* (под пользовательскими данными понимаются те данные, которые содержат информацию из реального мира, которая не участвует в процессе индексирования дерева; в разделе «Тестирование» в качестве пользовательских данных будут использоваться данные о пользователе: идентификатор пользователя, имя пользователя, фамилия пользователя);

– минимальный ключ родительской вершины, который обозначим как *L*;

– максимальный ключ родительской вершины, который обозначим как *J*.

Ниже представлен алгоритм прохода по CW-поддереву.

Входные данные – составляющие множества исходного дерева:

– множество вершин *V*:

$$V = \{v_i\},\ i = 1, 2, \ldots, l, \tag{9}$$

где *l* – мощность множества *V*; *i* – номер вершины;

– множество ребер *E*:

$$E = \{\{x, y\} \vee x, y \in V \cap x \neq y\}, \tag{10}$$

где *x*, *y* – вершины из множества *V*;

– множество дополнительных стартовых вершин:

$$W = \{w_i\},\ i=1, 2, \ldots, l;\ W \subset V; \tag{11}$$

– ключ целевого элемента – *k*:

Выходные данные – элемент дерева с ключом *k* обозначен *e*.

Обход CW-поддерева осуществляется в случае, когда необходимо найти данные в дереве по некоторому условию, например, найти данные, хранящиеся в вершине, у которой индекс равен 10. Таким образом, дан запрос с условием: индекс равен 10. Обход в CW-поддереве будет продолжаться до тех пор, пока вершина с индексом 10 не будет найдена либо пока не будет завершен обход со всех стартовых вершин. Условие запроса будет храниться в переменной *x* для того, чтобы данная переменная использовалась в процессе обхода в целях проверки. Этапы прохода по CW-поддереву:

Шаг 1. Начать поиск с ранее инициализированных дополнительных стартовых вершин *W*.

Шаг 2. Проверить, содержит ли текущая вершина элемент с ключом *k*, простым методом перебора.

Шаг 3. Если текущая вершина не содержит целевой элемент, применить функцию *move(x)* в соответствии с формулой

$$move(x) = \begin{cases} Go\ to\ F, & if\ N \le x, & if\ N \le x, & if\ x \le M, & if\ x < e \\ Go\ to\ Q, & if\ N \le x, & if\ N \le x, & if\ x \le M, & if\ x > e \\ Go\ to\ S. & & if\ x \le T & & \\ Go\ to\ S & & if\ x < L & & \\ Go\ to\ O & & if\ x \ge I & & \\ Go\ to\ O & & if\ x > J & & \\ Go\ to\ P & & if\ x < N, & if\ x > T & \\ Go\ to\ P & & if\ x > M, & if\ x < I & \end{cases}, \tag{12}$$

где *x* – условие запроса. Перейти к шагу 2.

Шаг 4. Если текущая вершина содержит целевой элемент, вернуть элемент вызывающей функции.

Шаг 5. Конец алгоритма.

Продемонстрируем работу формул (9) – (12) на практике. Пусть дан запрос на поиск элемента с ключом 1 в дереве *R*. Даны 4 потока для параллельного прохода по дереву. Тогда получим результат поиска вершины 1 (Таблица 1).

Таблица 1 – Результат параллельного прохода по дереву R для 4 потоков в поиске вершины с ключом 1
Table 1 – Result of a parallel traversal of the tree R for 4 threads in search of a vertex with key 1

Вершина 16	**Вершина 6**	**Вершина 18**	**Вершина 29**
16	6	18	29
8	2	14	27
4	1	10	25
2	-	6	23
1	-	2	21
-	-	1	19
-	-	-	17
-	-	-	15
-	-	-	13
-	-	-	11
-	-	-	9
-	-	-	7
-	-	-	5
-	-	-	3
-	-	-	1

Для того чтобы проследить за процессом поиска вершин, как показано в Таблице 1, нужно перейти к Рисунке 3. Следует отметить, что разделы статьи «Инициализация стартовых вершин» и «Параллельный обход CW-поддерева» относятся к разным этапам. Инициализация стартовых вершин происходит во время добавления новых данных в CW-поддерево, этап параллельного обхода CW-поддерева – уже после добавленных в CW-поддерево данных и после сформированных дополнительных стартовых вершин.

В Таблице 1 каждая строка представляет последовательность проходов по дереву для каждого потока. Названия соответствующих потоков указаны в первой колонке (16, 8, 3, 2, 1). Достижение вершины 1 в каждой колонке означает окончание поиска. Поскольку корневая вершина дерева *R* имеет ключ 16, строка с заголовком «Вершина 16» показывает поиск целевой вершины от корня. Таким образом, можно увидеть, что, если бы не использовались дополнительные потоки, мы бы нашли вершину 1 за 5 дисковых операций. Параллельный поиск от вершин 6, 18 и 29 завершится за 3, 6 и 15 дисковых операций соответственно. Поскольку поиск от вершины 6 закончится раньше, чем от всех остальных вершин, мы учитываем только его результат, не дожидаясь завершения поиска прочих потоков. Таким образом, при использовании 4 потоков мы можем найти вершину 1 за 3 дисковых операции. Если сравнивать с поиском в однопоточном режиме, мы получаем сокращение в 2 дисковые операции.

Количество сокращенных дисковых операций при использовании параллельного поиска с использованием дополнительных стартовых вершин по сравнению с однопоточным поиском от данной вершины отображено на Рисунке 4.

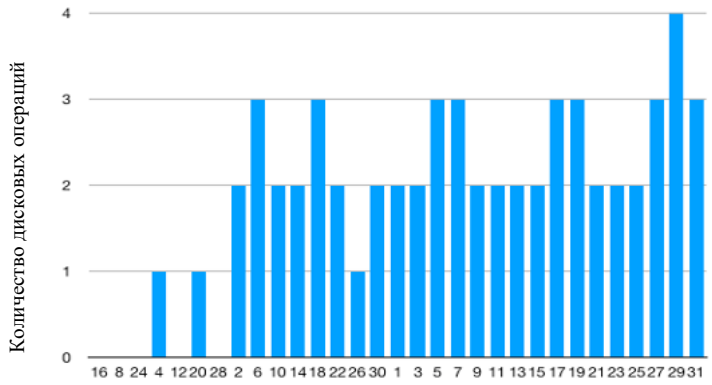

Рисунок 4 – Гистограмма сокращенных дисковых операций параллельного поиска по сравнению с классическим однопоточным поиском от поисковой вершины

Figure 4 – Histogram of reduced disk parallel search operations versus classic single-threaded search from search vertex

Например, если необходимо найти вершину 4 и использовать дополнительные стартовые вершины, то в процессе поиска будет использовано на одну дисковую операцию меньше, чем если производить поиск вершины 4 только из корневой вершины. Если необходимо найти вершину 2, то в процессе поиска будет использовано на две дисковые операции меньше, чем если производить поиск вершины 2 только из корневой вершины. На Рисунке 4 видна тенденция к снижению дисковых записей, необходимых для поиска вершины, что, в свою очередь, демонстрирует повышение эффективности алгоритма пропорционально росту уровня дерева, на котором находится целевая вершина.

Пример дерева, приведенный в данной главе, является полным. Однако эффект использования алгоритма параллельного обхода CW-поддерева будет таким же, если дерево будет неполным. Например, возьмем дерево, представленное на Рисунке 3, и удалим у него следующие вершины: 1, 3, 5, 7, 9, 11, 13, 15. У него остается прежняя корневая вершина с индексом 16, а также прежние дополнительные стартовые вершины с индексами 6, 18 и 29. Благодаря тому, что у каждой вершины есть связи с дочерними вершинами, родительской вершиной, левой и правой соседними вершинами, поиск любой вершины на уровне 4, который начинается с вершины 17, будет завершен максимум за 2 шага. Поиск любой вершины на уровне 3, который начинается с вершины 2, также будет завершен максимум за 2 шага. Аналогично поиск будет происходить и на 2 уровне.

Переходим к экспериментальному исследованию, рассмотрим методику тестирования. Наиболее показательна разница между использованием классического алгоритма индексирования, такого как B-tree, и использованием алгоритма CW-tree при хранении и обработке данных будет при высокой нагрузке. Для выполнения тестирования было сгенерировано дерево, которое хранит следующие данные о пользователе: идентификатор пользователя, имя пользователя, фамилия пользователя. Идентификатор пользователя соответствует индексу таблицы. Дерево состоит из 600 000 записей. Каждая запись последовательно сохранялась в дерево с инкрементируемым индексом от 1 до 600 000 по правилам добавления нового элемента в B-дерево. Для всех записей были использованы имя John и фамилия Kurt. B-дерево позволяет хранить элементы в узлах. Каждый узел может хранить более одной записи. В результате выполнения алгоритма инициализации стартовых вершин, было инициализировано 4 стартовые вершины – 1 корневая вершина и 3 дополнительные

стартовые вершины. Данные стартовые вершины содержат записи базы данных с индексами, которые приведены в Таблице 2.

Таблица 2 – Индексы записей базы данных, хранящиеся в стартовых вершинах тестового CW-поддерева
Table 2 – Database record indexes stored in the starting vertices of the test CW subtree

Корневая вершина	W_1	W_2	W_3
159999	419	1 619	2 400
319999	439	1 639	2 401
-	459	1 659	2 402
-	479	1 679	2 403
-	499	1 699	2 404
-	519	1 719	2 405
-	539	1 739	2 406
-	559	1 759	2 407
-	579	1 779	2 408
-	599	1 799	2 409
-	619	1 819	2 410
-	639	1 839	2 411
-	659	1 859	2 412
-	679	1 879	2 413
-	699	1 899	2 414
-	719	1 919	2 415
-	739	1 939	2 416
-	759	1 959	2 417
-	779	1 979	2 418

В Таблице 2 хранятся индексы записей базы данных, которые совпадают с идентификаторами пользователей, у которых значения остальных колонок Имени и Фамилии равны “John” и “Kurt” соответственно. Следует отметить, что множество дополнительных стартовых вершин W одно, а W_1, W_2 и W_3 – это дополнительные стартовые вершины с соответствующими номерами. Заголовок каждой колонки Таблице 2 описывает стартовую вершину, а значения, которые представлены в столбцах, описывают идентификаторы пользователей, которые хранятся в соответствующих им стартовых вершинах. Таким образом, в корневой вершине хранятся записи “John” и “Kurt” с идентификаторами пользователя 159999 и 319999. В вершине W_1 хранятся записи “John” и “Kurt” с идентификаторами пользователя 419, 439, 459, 479 и т. д.

Результаты

Для проведения тестирования было выбрано следующее накопительное устройство INTEL MEMPEK1W016GA со следующими параметрами:

– memory size – 13,41 GB;
– logical sector size –512 B;
– physical sector size – 512 B;
– connection interface – PCI-Express.

Одна вершина занимает один сектор объемом 512 байт. Каждая вершина хранит информацию об адресах соседних вершин, некоторые индексы соседних вершин и т. д. Эта информация будет использована на этапе выполнения параллельного обхода CW-поддерева для того, чтобы определить, в какую сторону необходимо выполнять проход

CW-поддерева, чтобы найти необходимую вершину. Помимо этого, вершина хранит информацию о записях базы данных. Количество хранимых записей в вершинах варьируется от 1 до установленной константы. Значение константы выбирается из расчета на максимально доступное пространство в одном секторе. В данном тесте значение константы равно 40.

Поскольку тестовое CW-поддерево содержит 4 стартовые вершины (корневая вершина и вершины W_1, W_2 и W_3), то в процессе поиска необходимой записи по ее индексу в данном CW-поддереве использовалось 4 потока. В качестве языка реализации алгоритма поиска в CW-поддереве, а также алгоритма выбора дополнительных стартовых вершин был использован C++14. В качестве компилятора был использован MSVC 14, дополнительные аргументы запуска программы не использовались. В качестве инструмента распараллеливания алгоритма поиска вершины в CW-поддереве был использован класс thread библиотеки STL.

На Рисунке 5 представлен результат выполнения поиска записи базы данных при 10 000 запросах по ее ключу в двух различных вариантах: с использованием только корневой вершины и с использованием дополнительных стартовых вершин Z, поиск из которых выполняется асинхронно отдельными потоками. На Рисунке 5 по горизонтали описаны целевые индексы записей, а по вертикали – время выполнения 10 000 запросов на поиск записи в миллисекундах, т. е. показан результат выполнения 10 000 запросов каждой записи из множества, отмеченных по горизонтали.

В каждом столбце колонка слева отображает время выполнения 10 000 последовательных запросов поиска записи по индексу в однопоточном режиме, начиная поиск из корневой вершины. Каждая колонка справа отображает время выполнения 10 000 последовательных запросов поиска записи по индексу в многопоточном режиме, начиная поиск из корневой, а также дополнительных стартовых вершин.

Гистограмма, отображающая количество сокращенных дисковых операций при использовании параллельного поиска с использованием дополнительных стартовых вершин по сравнению с однопоточным поиском от данной вершины, была представлена выше на Рисунке 4.

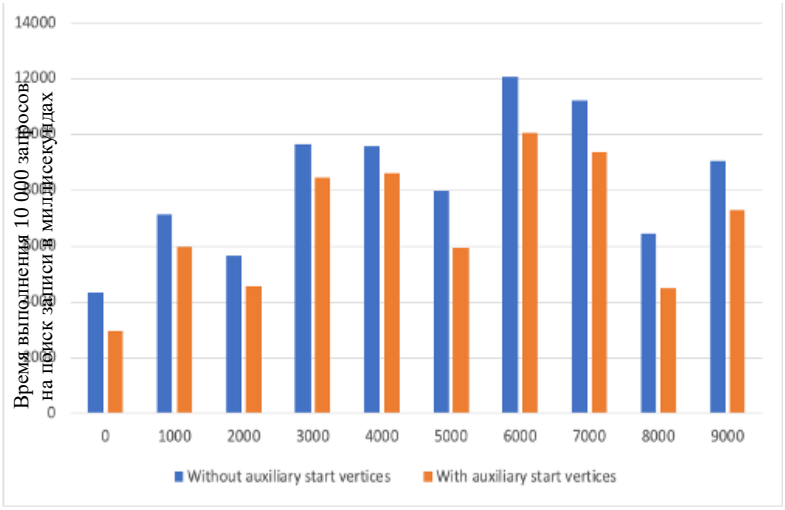

Рисунок 5 – Результат нахождения вершин (Without auxiliary start vertices – Без вспомогательных стартовых вершин; With auxiliary start vertices – Со вспомогательными стартовыми вершинами)

Figure 5 – The result of finding the vertices (x-axis: without auxiliary start vertices and with auxiliary start vertices)

Составление данных для Рисунке 5 выполнялось в соответствии со следующими этапами:

1. Выполнение 10 000 последовательных запросов поиска записи с индексом 0 без применения дополнительных стартовых вершин.
2. Подсчет времени выполнения 10 000 запросов поиска записи с индексом 0 без применения дополнительных стартовых вершин.
3. Выполнение 10 000 запросов поиска записи с индексом 0 с применением дополнительных стартовых вершин.
4. Подсчет времени выполнения 10 000 запросов поиска записи с индексом 0 с применением дополнительных стартовых вершин.
5. Добавление результатов времени выполнения запросов поиска записи с индексом 0 (Рисунок 5).
6. Выполнение 10 000 последовательных запросов поиска записи с индексом 1 000 без применения дополнительных стартовых вершин.
7. Подсчет времени выполнения 10 000 запросов поиска записи с индексом 1 000 без применения дополнительных стартовых вершин.
8. Выполнение 10 000 запросов поиска записи с индексом 1 000 с применением дополнительных стартовых вершин.
9. Подсчет времени выполнения 10 000 запросов поиска записи с индексом 1000 с применением дополнительных стартовых вершин.
10. Добавление результатов времени выполнения запросов поиска записи с индексом 1 000 (Рисунок 5).
11. Выполнение шагов 1–5 для записей с индексами 2 000, 3 000, 4 000, 5 000, 6 000, 7 000, 8 000 и 9 000.

При больших запросах на данные по ключу использование CW-tree алгоритма позволяет сократить необходимое количество дисковых операций и время на обработку таких запросов (см. Рисунок 5).

Заключение

Была проведена разработка алгоритма индексирования данных с использованием технологий параллельных вычислений, применение которого привело к уменьшению времени на обработку запросов к базе данных. Разработанный алгоритм индексирования данных базируется на использовании специальной структуры хранения данных на базе дерева, получившей название CW-tree.

Проход по CW-tree в соответствии с предлагаемым алгоритмом индексирования осуществляется в асинхронном режиме, что достигается благодаря использованию дополнительных стартовых вершин. Как показало тестирование, по сравнению с существующими аналогами алгоритмов индексирования, предложенный алгоритм на основе CW-tree обладает преимуществами, главным из которых является полное распараллеливание поиска данных как на верхнем уровне, так и на уровне листьев дерева.

Было проведено сравнительное исследование разработанного алгоритма и аналогов, а именно: чтение данных с применением B-tree, которое широко используется во многих современных СУБД, и чтение данных с применением CW-tree алгоритма. Для этого запросы к базе данных строились на поиск по определенному ключу. Устанавливалось, какой подход быстрее обработает некоторое число запросов поиска.

По результатам сравнительного исследования между поиском данных из B-tree и поиском данных из CW-tree, времени на выполнение 10 000 запросов записей базы данных по каждому индексу записи, использованному в процессе тестирования и

отображенному на Рисунке 5, требуется меньше при поиске на CW-tree, чем при поиске на B-tree. Из этого следует, что при использовании многопоточной технологии проход по CW-tree оказывается эффективнее, чем проход по B-tree. Планы дальнейшего исследования – сравнить операции чтения данных из CW-tree и из MySQL.

ЛИТЕРАТУРА

1. Nouri Z., Tu Y. GPU-Based Parallel Indexing for Concurrent Spatial Query Processing; Distributed and Parallel Databases. *SSDBM '18: Proceedings of the 30th International Conference on Scientific and Statistical Database Management*. 2018;1-12. Доступно по: https://doi.org/10.1145/3221269.3221296 (дата обращения: 20.02.2020)
2. Shun J., Blelloch G.E. A Simple Parallel Cartesian Tree Algorithm and its Application to Parallel Suffix Tree Construction. *Proceedings of the Thirteenth Workshop on Algorithm En-gineering and Experiments (ALENEX)*. 2011;(1):48-58.
3. Jarke M., Koch J. Query optimization in database systems. *ACM Computing Surveys (CsUR)*. 1984;16(2):111-152.
4. Smith J. M., Chang P. Y. T. Optimizing the performance of a relational algebra database interface. *Communications of the ACM* (*CACM)*. 1975;18(10):568-579.
5. Wu S., Li F., Mehrotra S., Chhin Ooi B. Query Optimization for Massively Parallel Data Processing. *SoCC Conference*. 2011;(12):1-13.
6. Shnaiderman L., Shmueli O. A Parallel Tree Pattern Query Processing Algorithm for Graph Databases using a GPGPU. *Proceedings of the Workshops of the EDBT/ICDT 2015 Joint Conference (EDBT/ICDT).* 2015;1330(24):141. Доступно по: http://ceur-ws.org/Vol-1330/paper-24.pdf (дата обращения: 20.01.2020)
7. Battre D., Angulo D. S. MPI framework for parallel searching in large biological databases. *Journal of Parallel and Distributed Computing*. 2006;66(12):1503-1511.
8. Dewan H. M., Stolfo S. J. Scalable Parallel and Distributed Expert Database Systems with Predictive Load Balancing. *Journal of Parallel and Distributed Computing*. 1994;22(3):506-522.
9. Ren D. Q. Algorithm level power efficiency optimization for CPU-GPU processing element in data intensive SIMD/SPMD computing. *Journal of Parallel and Distributed Computing*. 2011;72(2):245-253.
10. Liu P., Sun W., Zhang J., Zheng B. An automaton-based index scheme supporting twig queries for on-demand XML data broadcast. *Journal of Parallel and Distributed Computing*. 2015;(86):82-97.
11. Wani M. A., Bhat W. A. Dataset for forensic analysis of B-tree file system. *Data in Brief*. 2018;(18):2013-2018.

REFERENCES

1. Nouri Z., Tu Y. GPU-Based Parallel Indexing for Concurrent Spatial Query Processing; Distributed and Parallel Databases. *SSDBM'18: Proceedings of the 30th International Conference on Scientific and Statistical Database Managemen*t. 2018;1-12. Available at: https://doi.org/10.1145/3221269.3221296 (accessed 20.01.202)
2. Shun J., Blelloch G. E. A Simple Parallel Cartesian Tree Algorithm and its Application to Parallel Suffix Tree Construction. *Proceedings of the Thirteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*. 2011;(1):48-58.
3. Jarke M., Koch J. Query optimization in database systems. *ACM Computing Surveys (CsUR).* 1984;16(2):111-152.

4. Smith J.M., Chang P.Y.T. Optimizing the performance of a relational algebra database interface. *Communications of the ACM* (*CACM)*. 1975;18(10):568-579.
5. Wu S., Li F., Mehrotra S., Chhin Ooi B. Query Optimization for Massively Parallel Data Processing. *SoCC Conference*. 2011;(12):1-13.
6. Shnaiderman L., Shmueli O. A. Parallel Tree Pattern Query Processing Algorithm for Graph Databases using a GPGPU. *Proceedings of the Workshops of the EDBT/ICDT 2015 Joint Conference (EDBT/ICDT).* 2020;1330(24):141. Available at: http://ceur-ws.org/Vol-1330/paper-24.pdf (accessed 20.01.2020)
7. Battre D., Angulo D.S. MPI framework for parallel searching in large biological databases. *Journal of Parallel and Distributed Computing*. 2006;66(12):1503-1511.
8. Dewan H.M., Stolfo S.J. Scalable Parallel and Distributed Expert Database Systems with Predictive Load Balancing. *Journal of Parallel and Distributed Computing*. 1994;22(3):506-522.
9. Ren D.Q. Algorithm level power efficiency optimization for CPU-GPU processing element in data intensive SIMD/SPMD computing. *Journal of Parallel and Distributed Computing*. 2011;72(2):245-253.
10. Liu P., Sun W., Zhang J., Zheng B. An automaton-based index scheme supporting twig queries for on-demand XML data broadcast. *Journal of Parallel and Distributed Computing*. 2015;86:82-97.
11. Wani M.A., Bhat W.A. Dataset for forensic analysis of B-tree file system. *Data in Brief*. 2018;18:2013-2018.

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

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