ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет компьютерных наук
Департамент программной инженерии

УТВЕРЖДАЮ
Академический руководитель
образовательной программы
«Программная инженерия»,
профессор департамента программной
инженерии, канд. техн. наук

_____ В.В. Шилов
«___» _____ 2019 г.

Выпускная квалификационная работа

на тему «Компонент-расширение РСУБД SQLite для индексирования данных модификациями B-деревьев»

по направлению подготовки 09.03.04 «Программная инженерия»

Научный руководитель

Старший преподаватель
департамента программной
инженерии

С.А. Шершаков

_____
Подпись, Дата

Выполнил
студент группы БПИ153
4 курса бакалавриата
образовательной программы
«Программная инженерия»

А.М. Ригин

_____
Подпись, Дата

Москва 2019

Реферат

Сильно ветвящиеся деревья являются одним из наиболее популярных решений для индексирования больших объёмов данных. Наиболее распространённой разновидностью сильно ветвящихся деревьев является B-дерево. Существуют различные модификации B-дерева, в том числе, рассматриваемые в настоящей работе B^+-дерево, B^*-дерево и B^{*+}-дерево, однако данные модификации не поддерживаются по умолчанию в популярной реляционной СУБД с открытым исходным кодом SQLite.

Данная работа выполняется на основе выполненной ранее работы по исследованию эффективности сильно ветвящихся деревьев в задаче индексирования структурированных данных, с использованием разработанной в рамках неё C++-библиотеки сильно ветвящихся деревьев.

Целью работы является разработка расширения для реляционной СУБД SQLite, позволяющего использовать модификации B-дерева (B^+-дерево, B^*-дерево и B^{*+}-дерево) в качестве индексирующих структур данных.

Задачами работы являются:

1. обзор основных источников для работы;
2. обзор существующих решений;
3. реализация API на C для имеющейся C++-библиотеки сильно ветвящихся деревьев;
4. разработка расширения для SQLite, позволяющего использовать модификации B-дерева (B^+-дерево, B^*-дерево и B^{*+}-дерево) в качестве индексирующих структур данных, на основе имеющейся C++-библиотеки сильно ветвящихся деревьев, а также позволяющего выводить графическое изображение B-дерева или его модификации, используемой в данной таблице, в формате DOT для GraphViz, и основные данные о дереве;
5. разработка и реализация алгоритма выбора структуры данных для индексации таблицы из числа модификаций B-дерева (B^+-дерева, B^*-дерева и B^{*+}-дерева);
6. тестирование разработанного программного продукта;
7. разработка технической документации в соответствии с ЕСПД.

Ключевые слова – B-дерево, сильно ветвящееся дерево, индексация данных, SQLite, СУБД, РСУБД.

Работа содержит 33 страницы, 3 главы, 7 рисунков, 11 источников, 4 приложения.

Abstract

Multiway trees are one of the most popular solutions for the big data indexing. The most commonly used kind of the multiway trees is the B-tree. There exist different modifications of the B-tree, including B^+-tree, B^*-tree and B^{*+}-tree considered in this work. However, these modifications are not supported by the popular open-source relational DBMS SQLite.

This work is based on the previous research on the performance of multiway trees in the problem of structured data indexing, with the previously developed multiway trees C++ library usage.

The purpose of the work is the development of the SQLite RDBMS extension which allows to use B-tree modifications (B^+-tree, B^*-tree and B^{*+}-tree) as index structures.

The issues of the work are:

1. the literature review;
2. the existing solutions review;
3. implementation of the C-C++ cross-language API for the existing multiway trees C++ library;
4. development of the SQLite RDBMS extension which allows to use B-tree modifications (B^+-tree, B^*-tree and B^{*+}-tree) as index structures based on the existing multiway trees C++ library and to output the graphical representation of the B-tree or its modification used in the given table in the GraphViz DOT file format and the main data related to the tree;
5. development and implementation of the algorithm of selecting the index structure for a table indexing from the B-tree modifications (B^+-tree, B^*-tree and B^{*+}-tree);
6. testing of the developed software product;
7. development of the technical documentation.

Key words – B-tree, multiway tree, data indexing, SQLite, DBMS, RDBMS.

The paper contains 33 pages, 3 chapters, 7 illustrations, 11 bibliography items, 4 appendices.

Обозначения и сокращения

1. СУБД – система управления базами данных.
2. ЕСПД – единая система программной документации.

Содержание

Введение

В настоящее время в мире стремительно растёт объём обрабатываемых данных. Возникает проблема работы с большими данными (big data) [4]. Для её решения разрабатываются различные алгоритмы работы с крупными объёмами данных, в том числе, алгоритмы индексирования данных, использующие, как правило, структуры данных, основанные на хэш-таблицах и деревьях.

Одним из видов таких структур данных являются сильно ветвящиеся деревья, которые могут хранить в каждом своём узле более одного ключа (элемента данных) и более одной ссылки на дочерний узел, и большая часть узлов которых, как правило, хранится на постоянном запоминающем устройстве, а не в оперативной памяти. Наиболее популярной разновидностью сильно ветвящихся деревьев является B-дерево. Оно используется в качестве индекса по умолчанию во многих СУБД, в том числе, в популярной реляционной СУБД с открытым исходным кодом SQLite [7].

Существуют различные модификации B-дерева – в данной работе рассматриваются B^+-дерево, B^*-дерево и B^{*+}-дерево. Однако, данные модификации по умолчанию не поддерживаются в SQLite в качестве индексирующей структуры данных. Тем не менее, SQLite поддерживает разработку расширений, дополняющих базовую функциональность данной СУБД. Расширения подключаются к SQLite как динамические библиотеки [6].

Разработка расширения, позволяющего использовать в качестве индексирующей структуры данных одну из модификаций B-дерева (B^+-дерево, B^*-дерево или B^{*+}-дерево), даёт возможность сравнивать B-дерево и его модификации по различным индикатором эффективности (например, времени выполнения операций) на разных таблицах, а также использовать модификации B-дерева в качестве индексирующих структур данных в SQLite.

Данная работа выполняется на основе выполненной ранее работы по исследованию эффективности сильно ветвящихся деревьев в задаче индексирования структурированных данных, с использованием разработанной в рамках неё C++-библиотеки сильно ветвящихся деревьев [11].

Целью работы является разработка расширения для реляционной СУБД SQLite, позволяющего использовать модификации B-дерева (B^+-дерево, B^*-дерево и B^{*+}-дерево) в качестве индексирующих структур данных.

Задачами работы являются:

1. обзор основных источников для работы;
2. обзор существующих решений;

3. реализация API на C для имеющейся C++-библиотеки сильно ветвящихся деревьев;
4. разработка расширения для SQLite, позволяющего использовать модификации B-дерева (B^+-дерево, B^*-дерево и B^{*+}-дерево) в качестве индексирующих структур данных, на основе имеющейся C++-библиотеки сильно ветвящихся деревьев, а также позволяющего выводить графическое изображение B-дерева или его модификации, используемой в данной таблице, в формате DOT для GraphViz, и основные данные о дереве;
5. разработка и реализация алгоритма выбора структуры данных для индексации таблицы из числа модификаций B-дерева (B^+-дерева, B^*-дерева и B^{*+}-дерева);
6. тестирование разработанного программного продукта;
7. разработка технической документации в соответствии с ЕСПД.

Оставшаяся часть работы организована следующим образом. В главе 1 представлен обзор основных источников для работы и существующих решений. В главе 2 описаны B-дерево и его модификации, а также описан алгоритм выбора индексирующей структуры данных. В главе 3 описана реализация программного продукта. В заключении подведены основные результаты работы, а также определены пути дальнейших разработок в данном направлении. В приложениях представлена техническая документация к программному продукту, разработанная в соответствии с ЕСПД.

Глава 1. Обзор источников и существующих решений

В данной главе проводится обзор основных источников, используемых в настоящей работе, а также обзор существующих решений (аналогов).

1.1. Обзор основных источников, используемых в настоящей работе

B-дерево было впервые предложено как индексирующая структура данных Р. Байером и Э. МакКрейтом в 1972 году [1]. B-дерево описывается в данном источнике как сбалансированное дерево поиска, большая часть узлов которого хранится не в оперативной памяти, а на постоянном запоминающем устройстве (например, жёстком диске), и узлы которого могут содержать более одного ключа (элемента данных) и более одного указателя на дочерний элемент (в связи с чем такие деревья называются сильно ветвящимися). Байер и МакКрейт в качестве назначения B-дерева указывают именно индексирование данных.

B^+-дерево как модификация B-дерева было введено также в 1970-х годах разными авторами. Д. Комер в 1979 году описывает B^+-дерево как разновидность B-дерева, в которой реальные данные хранятся исключительно в листовых узлах. В этой же статье упоминается B^*-дерево, как разновидность B-дерева, в которой каждый узел (кроме корневого) заполняется как минимум на 2/3, а не на 1/2, как в B-дереве. Кроме того, в статье упоминается, что удаление из B^+-дерева является более простой процедурой, чем в B-дереве, так как всегда выполняется на листовых узлах [2]. Из этого можно сделать предположение, что удаление в B^+-дереве выполняется, в среднем, быстрее, чем в B-дереве, что, впрочем, необходимо проверить экспериментально. Также в статье описано отличие вставки в B^*-дерево от вставки в B-дерево – при вставке в B^*-дерево вместо разбиения узла на два, по возможности, происходит перераспределение ключей между данным узлом и его соседними узлами, а когда это невозможно – разбиение двух соседних узлов на три [2]. Так как, в связи с этим, дорогостоящая операция разбиения узлов выполняется реже, то можно предположить, что вставка в B^*-дереве выполняется, в среднем, быстрее, чем в B-дереве, что, впрочем, также необходимо проверить экспериментально. Среди более современных источников по B^+-дереву, использованных в настоящей работе, можно отметить статью преподавателя Университета Аалто (г. Хельсинки, Финляндия) К. Поллари-Малми 2010 года [5].

B^{*+}-дерево было разработано ранее автором настоящей работы как модификация B-дерева, сочетающая в себе основные характеристики B^+-дерева и B^*-дерева – в B^{*+}-дереве реальные данные, как и в B^+-дереве, хранятся только в листовых узлах, при этом каждый

узел такого дерева (кроме корневого) заполняется минимум на 2/3, как и в B^*-дереве. Ожидается, что оно должно выигрывать у B-дерева как по скорости вставки ключей в дерево, так и по скорости удаления ключей из дерева [11].

Основным источником по СУБД SQLite в настоящей работе является официальный сайт SQLite [7].

1.2. Обзор существующих решений и аналогов

SQLite не поддерживает работу с модификациями B-дерева (B^+-дерево, B^*-дерево, B^{*+}-дерево) по умолчанию, хотя использует само B-дерево как основную (используемую по умолчанию) индексирующую структуру данных [3]. Готовых расширений для SQLite, которые позволяли бы использовать такие модификации в качестве индексирующих структур данных, в открытом доступе не обнаружено.

Одно из расширений для SQLite, поставляемых вместе со стандартной сборкой данной СУБД, даёт возможность работать с R-деревом – модификацией B-дерева, позволяющей работать с геоданными [9]. Тем не менее, R-дерево не рассматривается в данной работе и по этой причине данное расширение аналогом программного продукта, разрабатываемого в настоящей работе, не является.

Таким образом, аналогов настоящей работы не обнаружено.

Глава 2. Теоретическая основа работы

В данной главе описаны B-дерево и его модификации, а также описан алгоритм выбора индексирующей структуры данных.

2.1. B-дерево

B-дерево – сбалансированное сильно ветвящееся дерево поиска. Важным параметром построения B-дерева является порядок B-дерева – такое число t, что минимальное и максимальное количества ключей в узле дерева линейно зависят от t. По этой причине высота B-дерева равняется $O(log_t n)$, где t – порядок B-дерева, а n – общее количество ключей в B-дереве [10].

Формально, B-дерево задаётся следующими правилами:

1. Для любого узла B-дерева, кроме корневого, верно, что $t - 1 \leq k \leq 2t - 1$, где k – количество ключей в узле, а t – порядок дерева.
2. Для корневого узла непустого B-дерева верно, что $1 \leq k \leq 2t - 1$, где k – количество ключей в узле, а t – порядок дерева.
3. Для корневого узла пустого B-дерева верно, что $k = 0$.
4. Для любого узла B-дерева, кроме листовых, верно, что $p = k + 1$, где p – количество указателей на дочерние узлы, k – количество ключей в узле.
5. Для любого листового узла B-дерева верно, что $p = 0$, где p – количество указателей на дочерние узлы.

Поскольку B-дерево является деревом поиска, то ключи в нём хранятся упорядоченно. Формально, можно сказать, что для каждого внутреннего (нелистового) узла верно следующее: $s(p_1) \leq k_1 \leq s(p_2) \leq k_2 \leq s(p_3) \leq \cdots \leq s(p_n) \leq k_n \leq s(p_{n+1})$, где k_i – i-й ключ узла, p_i – i-й указатель на дочерний узел, а $s(p_i)$ – любой ключ из узлов поддерева с корнем в узле, на который указывает p_i.

Пример B-дерева для порядка $t = 6$ приведёт на рис. 1.

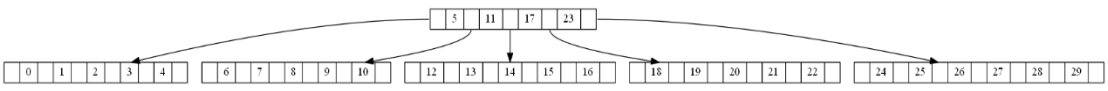

Рисунок 1. B-дерево порядка $t = 6$, с высотой 2

2.1.1. Поиск по B-дереву

Поиск по B-дереву выполняется путём обхода B-дерева, начиная с корня, рекурсивно. В каждом из узлов, через которые проходит алгоритм поиска, перебираются ключи соответствующего узла, до тех пор, пока не будет найден ключ, больший или

равный искомому, либо пока не будут перебраны все ключи в данном узле. Если в узле найден ключ, равный искомому, то он рассматривается как результат поиска. В противном случае, если текущий узел является внутренним (не является листовым), поиск по B-дереву рекурсивно продолжается в соответствующем дочернем узле. Если искомый ключ не найден в листовом узле, то считается, что он отсутствует в дереве [10].

2.1.1.1. Сложность поиска по B-дереву

В процессе поиска по B-дереву производится проход от корневого узла дерева, в направлении к листовым узлам дерева. Таким образом, количество пройденных алгоритмом узлов, будет не больше высоты дерева, то есть $O(log_t n)$, где t – порядок B-дерева, а n – общее количество ключей в B-дереве. Внутри каждого узла, проходимого алгоритмом, выполняется линейный перебор ключей данного узла. Количество ключей в узле не превышает $2t - 1$, где t – порядок B-дерева, поэтому вычислительная сложность поиска по B-дереву будет равна $O(tlog_t n)$, где t – порядок B-дерева, а n – общее количество ключей в B-дереве [10].

Если поиск по B-дереву организовывать в форме обычной рекурсии, то в каждый момент времени придётся хранить в оперативной памяти текущий узел и все узлы, пройденные до него, вплоть до корневого узла, вместе со всеми ключами в этих узлах. Поскольку количество пройденных поисковым алгоритмом узлов не будет превышать высоты дерева, а количество ключей в каждом из узлов линейно зависит от порядка дерева, сложность поиска по памяти будет равна $O(tlog_t n)$, где t – порядок B-дерева, а n – общее количество ключей в B-дереве [11].

Если поиск по B-дереву организовывать в форме хвостовой рекурсии или цикла, то в каждый момент времени будет достаточно хранить лишь текущий узел дерева, и тогда сложность поиска по памяти будет равна $O(t)$, где t – порядок B-дерева [11].

При поиске по B-дереву из дисковых операций применяется только операция чтения, при этом применяется она по одному разу на каждый проходимый алгоритмом узел дерева, а значит, число дисковых операций не будет превышать высоту дерева, то есть $O(log_t n)$, где t – порядок B-дерева, а n – общее количество ключей в B-дереве [11].

2.1.2. Вставка в B-дерево

Вставка новых ключей в B-дерево производится всегда в листовых узлах дерева. Нужный листовой узел для вставки подбирается рекурсивным алгоритмом поиска, таким же, как описанный в п. 2.1.1 алгоритм поиска по B-дереву, за тем лишь исключением, что

ищется не конкретный ключ, а место для вставки нового ключа в листовом узле и сам листовой узел. Когда алгоритм вставки проходит через заполненный узел B-дерева (то есть через узел, в котором количество ключей равняется $2t - 1$, где t – порядок B-дерева), то данный узел разбивается на два равных узла, при этом ключ, который располагается посередине разбиваемого узла, перемещается в родительский узел [10].

2.1.2.1. Сложность вставки в B-дерево

Вычислительная сложность операции разбиения узла в B-дереве равняется $O(t)$, где t – порядок B-дерева, поскольку не зависит от высоты дерева, но может потребовать сдвиг ключей в родительском узле вправо для перемещения среднего ключа разбиваемого узла, и, таким образом, линейно зависит от порядка дерева. Поскольку операция разбиения затрагивает только два узла – разбиваемый узел и родительский для него узел, то сложность по памяти для операции разбиения равняется также $O(t)$, где t – порядок B-дерева, а количество дисковых операций не превысит $O(1)$ [11].

Так как операция вставки в B-дерево состоит из прохода по дереву от корня к листьям и разбиений узлов, то её сложность зависит от высоты дерева и его порядка. Таким образом, вычислительная сложность вставки в B-дерево равняется $O(tlog_t n)$, где t – порядок B-дерева, а n – общее количество ключей в B-дереве [11].

Аналогично операции поиска по B-дереву, операция вставки в B-дерево при использовании обычной рекурсии будет иметь сложность по памяти $O(tlog_t n)$, где t – порядок B-дерева, а n – общее количество ключей в B-дереве. При использовании же хвостовой рекурсии или цикла операция вставки в B-дерево будет иметь сложность по памяти $O(t)$, где t – порядок B-дерева [11].

Количество дисковых операций не превысит высоту B-дерева, то есть $O(log_t n)$, где t – порядок B-дерева, а n – общее количество ключей в B-дереве [11].

2.1.3. Удаление из B-дерева

При удалении из B-дерева, в первую очередь, находится удаляемый ключ, при помощи алгоритма поиска по B-дереву, аналогичного описанному в п. 2.1.1, со следующими изменениями.

Если удаляемый ключ находится во внутреннем (не листовом) узле, то выполняются следующие проверки и действия. В начале проверяется дочерний узел, предшествующий удаляемому ключу. Если он содержит не менее t ключей, то в поддереве с корнем в данном дочернем узле ищется и рекурсивно удаляется наибольший

по значению ключ, после чего он подставляется на место удаляемого ключа. Если же данный дочерний узел содержит $t-1$ ключей, где t – порядок дерева, то проверяется дочерний узел, следующий за удаляемым ключом. Если он содержит не менее t ключей, то в поддереве с корнем в данном дочернем узле ищется и рекурсивно удаляется наименьший по значению ключ, после чего он подставляется на место удаляемого ключа. Если же и данный дочерний узел содержит $t-1$ ключей, то выполняется процедура слияния двух данных дочерних узлов, при этом удаляемый ключ, лежащий между этими дочерними узлами, становится средним ключом в объединённом узле, после чего из него рекурсивно удаляется [11].

Если ключ отсутствует в текущем внутреннем (не листовом) узле дерева, то определяется поддерево, в котором он может находиться, и проверяется дочерний узел, являющийся корнем этого поддерева. Если данный дочерний узел содержит не менее t ключей, то алгоритм рекурсивного удаления сразу выполняется на нём. Если же данный дочерний узел содержит только $t-1$ ключей, то проверяются его соседние узлы, то есть узлы, имеющие тот же родительский узел, и такие, что между соседним узлом и данным узлом в родительском узле расположен только один ключ, который назовём медианой. Если один из двух соседних узлов содержит не менее t ключей, то выполняется процедура «переливания» одного ключа из соответствующего соседнего узла в данный, при этом медиана перемещается в данный узел, а крайний ключ из соответствующего соседнего узла перемещается на место медианы, при этом выполняется необходимые сдвиги ключей. После этого алгоритм удаления ключа рекурсивно выполняется на данном дочернем узле. Если же оба соседних узла содержат только по $t-1$ ключей, то выполняется процедура слияния данного дочернего узла с одним из его соседних узлов, после чего алгоритм удаления ключа рекурсивно выполняется на объединённом узле [11].

Если ключ находится в текущем листовом узле, то он просто удаляется из него, со сдвигом следующих за ним ключей на его место [11].

Если ключ отсутствует в текущем листовом узле, то он отсутствует в дереве [11].

2.1.3.1. Сложность удаления из B-дерева

Вычислительная сложность операции удаления линейно зависит от высоты дерева и его порядка и равняется $O(tlog_t n)$, где t – порядок B-дерева, а n – общее количество ключей в B-дереве [10].

Поскольку в некоторых случаях бывает необходимо извлекать и рекурсивно удалять наименьший или наибольший ключ в поддереве, то удаление можно реализовать

лишь в форме обычной рекурсии, и его сложность по памяти всегда будет равна $O(tlog_t n)$, где t – порядок B-дерева, а n – общее количество ключей в B-дереве [11].

Количество дисковых операций зависит от высоты B-дерева, то есть равняется $O(log_t n)$, где t – порядок B-дерева, а n – общее количество ключей в B-дереве [11].

2.2. Модификации B-дерева

2.2.1. B⁺-дерево

В B⁺-дереве реальные данные (реальные ключи) хранятся лишь в листовых узлах, в остальных (внутренних) узлах хранятся ключи-маршрутизаторы, предназначенные для поиска реальных ключей [5].

При разбиении листового узла крайний ключ в одном из узлов, являющихся продуктами разбиения, копируется в родительский узел, копия становится ключом-маршрутизатором. По этой причине для листовых узлов в B⁺-дереве верно, что $t \leq k \leq 2t$, где k – количество ключей в узле, а t – порядок дерева.

Асимптотика у операций поиска, вставки и удаления из B⁺-дерева совпадает с асимптотикой соответствующих операций в B-дереве. Однако, фактическая вычислительная сложность операции удаления из B⁺-дерева при проведённых ранее экспериментах оказалась ниже, чем у операции удаления из B-дерева (так как удаление в B⁺-дереве всегда производится из листовой вершины), что показано на графике на рис. 2.

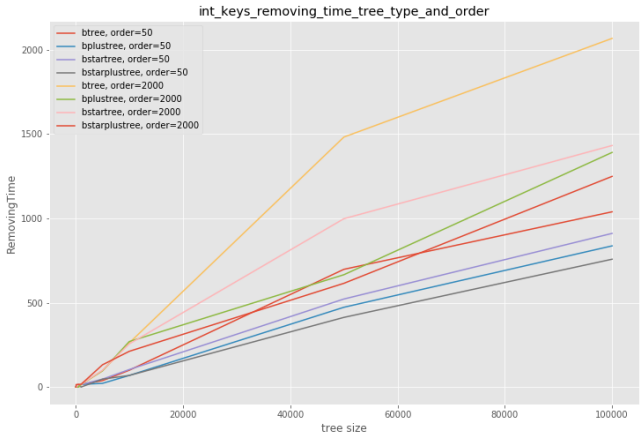

Рисунок 2. График зависимости времени выполнения удаления всех ключей в дереве от количества ключей (размера дерева) для значений порядка дерева, равных 50 и 2000 [11]

2.2.2. B-дерево*

В B*-дереве все узлы (кроме корневого) заполняются минимум на 2/3, а не на 1/2, как в B-дереве. Вместо обычного разбиения узла при вставке выполняются операции «переливания» ключей (такие же, как в B-дереве выполняются при удалении ключа из

дерева), а в тех случаях, когда такую операцию совершить невозможно – операция разбиения двух узлов на три. Благодаря этому, дорогостоящая по времени операция разбиения выполняется реже и, таким образом, при ранее проведённых экспериментах вставка ключа в дерево выполнялась быстрее, что показано на графике на рис. 3.

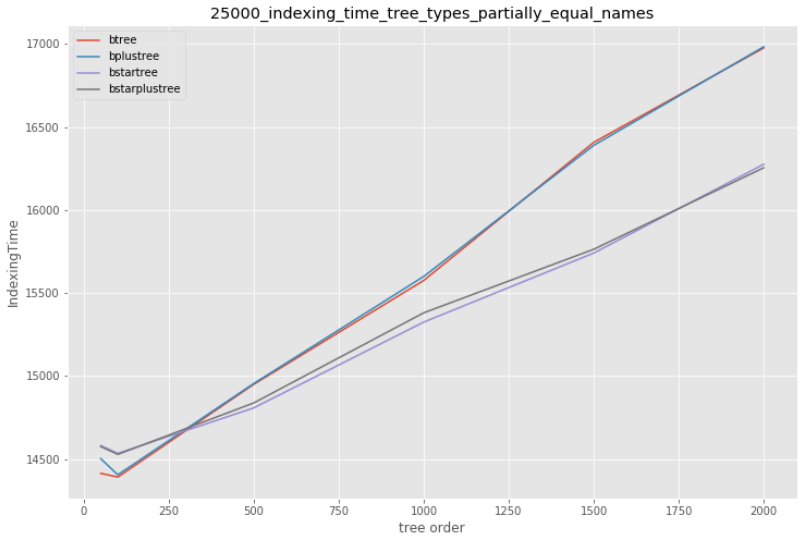

Рисунок 3. График зависимости времени индексации CSV-файла размером 25000 строк от порядка дерева [11]

2.2.3. B^{*+}-дерево

B^{*+}-дерево разработано ранее автором данной работы и совмещает в себе основные свойства B^{+}-дерева и B^{*}-дерева – в нём реальные ключи, как и в B^{+}-дереве, хранятся лишь в листовых узлах, а в остальных узлах хранятся ключи-маршрутизаторы. При этом все вершины (кроме корневой) в нём заполняются минимум на 2/3, как и в B^{*}-дереве.

B^{*+}-дерево на проведённых ранее экспериментах показало лучший результат в плане вычислительной сложности, чем B-дерево, как на операциях вставки, так и на операциях удаления, что показано на графиках на рис. 2 и 3. Тем не менее, оно, как и B^{*}-дерево, в рамках этих экспериментов потребляло больше оперативной памяти, как показано на графике на рис. 4. Согласно этому графику, в рамках проведённых ранее экспериментов, B^{*}-дерево и B^{*+}-дерево на операциях вставки потребляли приблизительно в два раза больше оперативной памяти, чем B-дерево и B^{+}-дерево соответственно.

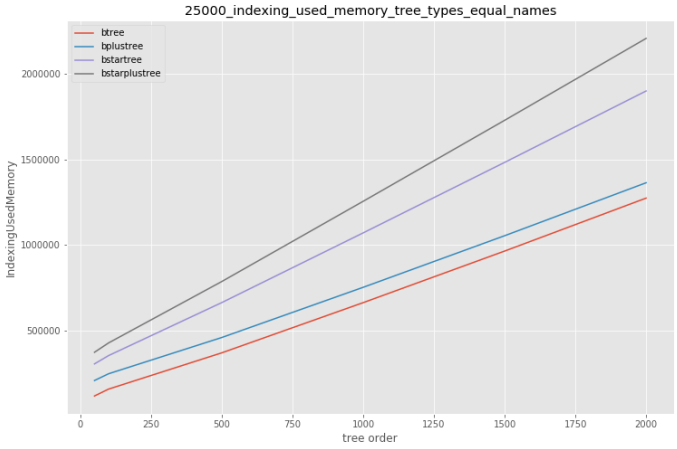

Рисунок 4. График зависимости объёма занимаемой памяти (максимальной за время операции) во время индексации от порядка дерева для CSV-файла с размером 25000 строк [11]

2.3. Алгоритм выбора индексирующей структуры данных и порядок дерева

В рамках разработки расширения, позволяющего использовать модификации B-дерева (B^+-дерево, B^*-дерево и B^{*+}-дерево), также разработан и реализован алгоритм выбора индексирующей структуры данных из модификаций B-дерева (B^+-дерева, B^*-дерева и B^{*+}-дерева).

Так как в предыдущей работе по исследованию эффективности сильно ветвящихся деревьев в задаче индексирования структурированных данных было обнаружено, что разные модификации B-дерева имеют лучшую производительность по времени на разных модифицирующих операциях (в зависимости от конкретного типа операции – вставки в дерево или удаления из дерева), при этом других связей типа дерева со скоростью выполнения операций обнаружено не было [11], то было решено в качестве критерия для работы алгоритма использовать соотношение различных типов модифицирующих операций (вставка в дерево, удаление из дерева) между собой.

Также в рамках настоящей работы необходимо было выбрать порядок дерева для разработанного в рамках данной работы расширения, по критерию наименьшего среднего времени выполнения операций. Для этого было посчитано среднее по четырём типам деревьев (B-дерево, B^+-дерево, B^*-дерево и B^{*+}-дерево) время выполнения 1000 модифицирующих операций (операций вставки в дерево и удаления из дерева) для порядков дерева от 100 до 1000 включительно с шагом 50, то есть для порядков дерева 100, 150, 200, …, 950, 1000. Все замеры выполнялись только для операций непосредственно с деревом, так как время выполнения служебных операций в РСУБД

SQLite не зависит от типа дерева и его порядка. Наименьшее среднее по четырём указанным выше типам деревьев время выполнения было достигнуто при порядке дерева, равном 750 – время выполнения 1000 модифицирующих операций с деревом составило приблизительно 9,55 мс. По этой причине для B-дерева и его модификаций, используемых в расширении, разработанном в рамках данной работы, выбран порядок 750.

Кроме того, проведён замер максимального значения используемой в рамках выполнения операций с деревом в куче (heap) оперативной памяти в течение выполнения 1000 модифицирующих операций с деревом. Для B-дерева и B^+-дерева, это значение, в среднем, составило 0 байт, то есть для операций не приходилось выделять дополнительной памяти в куче. Это можно объяснить тем, что при используемом порядке дерева не приходилось создавать новых узлов дерева, достаточно было использовать корневой узел дерева, который изначально хранится в оперативной памяти. Для B^*-дерева и B^{*+}-дерева, это значение, в среднем, составило 120 и 136 байт соответственно. По той причине, что, как показано выше, B^+-дерево использует наименьшее количество оперативной памяти (в куче) среди всех трёх рассматриваемых в данной работе модификаций B-дерева, оно используется в разработанном в рамках данной работы расширении в качестве типа дерева, используемого по умолчанию.

Также произведён выбор соотношения количеств модифицирующих операций с деревом разных типов (вставка в дерево, удаление из дерева), разделяющего выбор B^*-дерева и B^{*+}-дерева в качестве индексирующей структуры данных для таблицы. Для этого, с использованием языка программирования Python 2 и библиотек NumPy, Pandas, Matplotlib, SciPy и Shapely, был построен график, показанный на рис. 5.

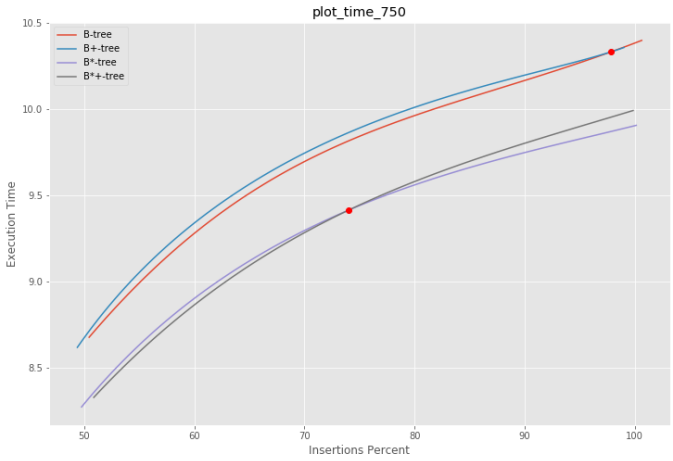

Рисунок 5. Сплайны, приближённые к графикам зависимости времени выполнения 1000 модифицирующих операций на дереве от процента числа вставок среди этих операций, для B-дерева и его модификаций порядка 750

На графике на рис. 5 показаны сплайны, приближённые к графикам зависимости времени выполнения 1000 модифицирующих операций на дереве от процента числа вставок среди этих операций, для B-дерева и его модификаций порядка 750. Данные сплайны были вычислены при помощи библиотеки SciPy и отображены при помощи библиотеки Matplotlib. Точки пересечения вычислены при помощи библиотеки Shapely. Точкой пересечения сплайна для B-дерева и сплайна для B^+-дерева является точка (97,84; 10,33). Точкой пересечения сплайна для B^*-дерева и сплайна для B^{*+}-дерева является точка (73,97; 9,42). Справа от этой точки (при проценте операций вставки большем, чем 73,97 % от общего числа модифицирующих операций с деревом) лучшую производительность показывает B^*-дерево, а слева (при проценте операций вставки меньшем, чем 73,97 % от общего числа модифицирующих операций с деревом) – B^{*+}-дерево. Таким образом, при проценте операций вставки большем, чем 73,97 % от общего числа модифицирующих операций с деревом, в качестве индексирующей структуры данных будет выбираться B^*-дерево, а в противном случае – B^{*+}-дерево.

В случае, если большинство операций (более 90 %) с деревом являются операциями поиска, то перестраивание дерева не выполняется, так как не обнаружено зависимости скорости выполнения операций поиска от типа дерева, которая позволила бы определить дерево, на котором операция поиска, в среднем, выполняется быстрее [11]. Чтобы операция перестраивания дерева не занимала слишком много времени при использовании разработанного в рамках настоящей работы расширения для SQLite, она выполняется лишь на каждой 1000-й операции с деревом и только для первых 10000 операций.

Алгоритм запускается при каждой операции с таблицей, созданной с использованием этого расширения – поиске строки в таблице, вставке строки в таблицу, обновлении строки в таблице, удалении строки из таблицы, и формально может быть описан следующим образом:

1. Если текущее общее количество операций с деревом равно 0, или больше 10000, или не кратно 1000, то выйти из алгоритма, иначе перейти к шагу 2.
2. Если текущее количество операций с изменением данных в дереве (вставок ключей в дерево, удалений ключей из дерева) составляет менее 10 % от текущего общего количества операций с деревом, то выйти из алгоритма, иначе перейти к шагу 3.
3. Если текущее количество операций вставки в дерево составляет более 73,97 % от текущего количества операций с изменением данных в дереве, то выбрать в качестве индексирующей структуры данных B^*-дерево и перейти к шагу 5, иначе перейти к шагу 4.

4. Выбрать в качестве индексирующей структуры данных B^{*+}-дерево и перейти к шагу 5.
5. Если в шагах 3 – 4 была выбрана новая индексирующая структура данных, то перестроить имеющуюся индексирующую структуру данных на выбранную в шагах 3 – 4, сохранив все имеющиеся в ней данные.

Глава 3. Реализация программного продукта

В данной главе описана реализация программного продукта, разрабатываемого в рамках данной работы – компонента-расширения РСУБД SQLite для индексирования данных модификациями B-деревьев.

3.1. Функциональные требования

Расширение для SQLite должно удовлетворять следующим функциональным требованиям:

1. Расширение должно позволять создавать таблицу, использующую B^+-дерево из данного расширения в качестве индексирующей структуры данных, с указанием столбца, являющегося первичным ключом таблицы.
2. Расширение должно позволять удалять таблицу, использующую B-дерево или его модификацию из данного расширения в качестве индексирующей структуры данных.
3. Расширение должно позволять производить поиск строки/строк в таблице, использующей B-дерево или его модификацию из данного расширения в качестве индексирующей структуры данных, по признаку равенства значения/значений первичного ключа искомой/искомых строки/строк таблицы заданному значению/заданным значениям.
4. Расширение должно позволять производить вставку строки/строк в таблицу, использующую B-дерево или его модификацию из данного расширения в качестве индексирующей структуры данных.
5. Расширение должно позволять производить удаление строки/строк в таблице, использующей B-дерево или его модификацию из данного расширения в качестве индексирующей структуры данных, по признаку равенства значения/значений первичного ключа искомой/искомых строки/строк таблицы заданному значению/заданным значениям.
6. Расширение должно позволять производить обновление значений ячеек (включая ячейку с первичным ключом) строки/строк в таблице, использующей B-дерево или его модификацию из данного расширения в качестве индексирующей структуры данных, по признаку равенства значения/значений первичного ключа искомой/искомых строки/строк таблицы заданному значению/заданным значениям.

7. Расширение должно позволять переименовывать таблицу, использующую B-дерево или его модификацию из данного расширения в качестве индексирующей структуры данных.
8. Расширение должно при каждой операции с таблицей, использующей B-дерево или его модификацию из данного расширения в качестве индексирующей структуры данных, запускать алгоритм выбора индексирующей структуры данных из модификаций B-дерева (B^+-дерева, B^*-дерева и B^{*+}-дерева) и перестраивать имеющуюся индексирующую структуру данных на новую (если была выбрана новая), сохраняя все имеющиеся в ней данные.
9. Расширение должно поддерживать сохранение таблицы, использующей B-дерево или его модификацию из данного расширения в качестве индексирующей структуры данных, вместе с базой данных на постоянном запоминающем устройстве.
10. Расширение должно поддерживать открытие сохранённой вместе с базой данных на постоянном запоминающем устройстве таблицы, использующей B-дерево или его модификацию из данного расширения в качестве индексирующей структуры данных.
11. Расширение должно позволять для таблицы, использующей B-дерево или его модификацию из данного расширения в качестве индексирующей структуры данных, выводить графическое изображение индексирующей структуры данных (дерева) таблицы в DOT-файл для GraphViz.
12. Расширение должно позволять для таблицы, использующей B-дерево или его модификацию из данного расширения в качестве индексирующей структуры данных, выводить тип используемого дерева (1 – B-дерево, 2 – B^+-дерево, 3 – B^*-дерево, 4 – B^{*+}-дерево).
13. Расширение должно позволять для таблицы, использующей B-дерево или его модификацию из данного расширения в качестве индексирующей структуры данных, выводить порядок используемого дерева.

Данные функциональные требования к расширению для SQLite также представлены в графическом виде на диаграмме прецендентов использования на рис. 6.

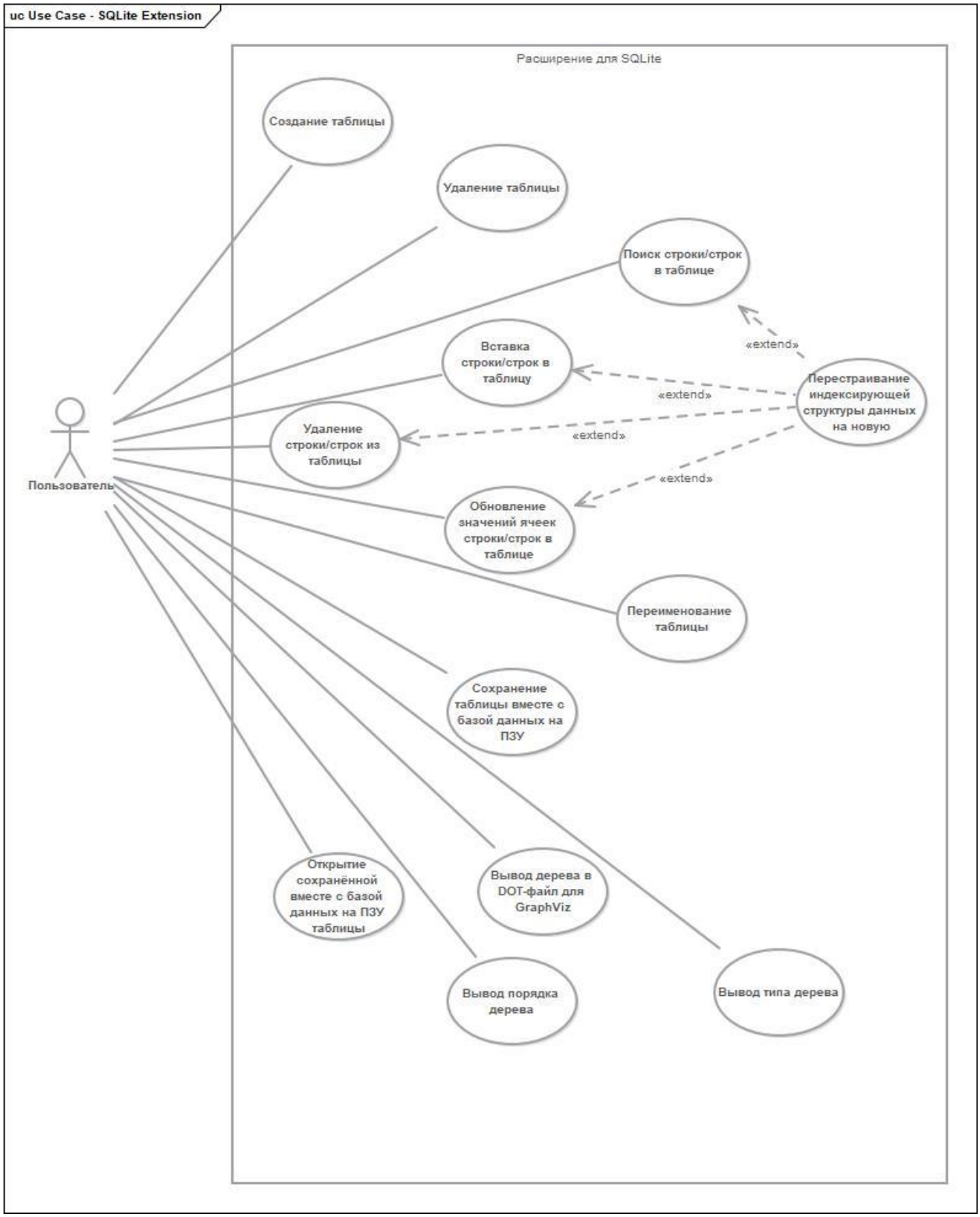

Рисунок 6. Диаграмма прецендентов использования расширения для SQLite

3.2. Средства и инструменты разработки

Для разработки расширения для SQLite в качестве языков программирования используются языки C (так как на нём написана РСУБД SQLite) и C++ (так как на нём ранее разработана библиотека сильно ветвящихся деревьев, используемая в настоящей

работе), а в качестве среды разработки – CLion 2018.3 от JetBrains. В качестве компилятора используется версия компилятора GCC для C++ (G++) с ключами *-g* и *-shared*, согласно рекомендациям на официальном сайте SQLite [6]. Также использованы ранее разработанная автором данной работы C++-библиотека сильно ветвящихся деревьев и SQLite C API, в том числе заголовочный файл sqlite3ext.h, предназначенный для разработки расширений для SQLite.

3.3. Реализация расширения для SQLite

Расширение для SQLite использует API на C для имеющейся C++-библиотеки сильно ветвящихся деревьев. API на C реализовано с использованием конструкции *extern “C” { … }*.

Расширение для SQLite регистрирует в СУБД модуль виртуальной таблицы с названием *btrees_mods*, который «перехватывает» все обращения к виртуальным таблицам, созданным с использованием этого модуля. Виртуальная таблица – любая таблица, созданная с использованием модуля, поставляемого любым расширением для SQLite и перехватывающего обращения к созданным с его использованием таблицам.

Кроме того, расширение для SQLite регистрирует в СУБД функции *btreesModsVisualize* (вывод графического представления дерева в DOT-файл для GraphViz), *btreesModsGetTreeOrder* (вывод порядка дерева) и *btreesModsGetTreeType* (вывод типа дерева: 1 – B-дерево, 2 – B^+-дерево, 3 – B^*-дерево, 4 – B^{*+}-дерево).

В табл. 1 представлены описания методов расширения для SQLite, к которым непосредственно обращается СУБД.

Таблица 1

Методы расширения для SQLite

Название метода	**Описание метода**
btreesModsCreate	Создаёт виртуальную таблицу с использованием модуля *btrees_mods* (в ходе выполнения запроса вида «CREATE VIRTUAL TABLE tableName USING btrees_mods(…);»).
btreesModsConnect	Подключается к ранее созданной с использованием модуля *btrees_mods* виртуальной таблице.

Название метода	Описание метода
btreesModsBestIndex	Подготавливает ранее созданную с использованием модуля *btrees_mods* виртуальную таблицу к поиску строки в ней.
btreesModsDisconnect	Выполняет необходимые действия при отключении от ранее созданной с использованием модуля *btrees_mods* виртуальной таблицы.
btreesModsDestroy	Удаляет ранее созданную с использованием модуля *btrees_mods* виртуальную таблицу (в ходе выполнения запроса вида «DROP TABLE tableName;»), в том числе, удаляет файл с деревом с диска.
btreesModsOpen	Инициализирует курсор, необходимый для поиска строки в ранее созданной с использованием модуля *btrees_mods* виртуальной таблице.
btreesModsClose	Выполняет действия, необходимые при уничтожении курсора, созданного при помощи метода btreesModsOpen.
btreesModsFilter	Осуществляет поиск строки в ранее созданной с использованием модуля *btrees_mods* виртуальной таблице (в ходе выполнения SQL-запроса вида «SELECT * FROM tableName WHERE id = …;»). В случае, если несколько строк удовлетворяют условию, то после выполнения этого метода, курсор, созданный при помощи метода btreesModsOpen, указывает на первую из таких строк.
btreesModsNext	Перемещает курсор, созданный при помощи метода btreesModsOpen, на следующую строку, удовлетворяющую условию, из числа найденных методом btreesModsFilter.

Название метода	Описание метода
btreesModsEof	Возвращает 1, если строки, найденные методом btreesModsFilter закончились, 0 в противном случае.
btreesModsColumn	Возвращает значение i-й ячейки строки, на которую в данный момент указывает курсор, созданный при помощи метода btreesModsOpen.
btreesModsRowid	Возвращает rowid строки, на которую в данный момент указывает курсор, созданный при помощи метода btreesModsOpen.
btreesModsUpdate	Осуществляет изменение данных в ранее созданной с использованием модуля *btrees_mods* виртуальной таблице (вставку строки, удаление строки либо обновление ячеек строки).
btreesModsRename	Осуществляет переименование ранее созданной с использованием модуля *btrees_mods* виртуальной таблицы (в ходе выполнения запроса вида «ALTER TABLE tableName RENAME TO newTableName;»).
btreesModsVisualize	Выводит графическое представление дерева, используемого в качестве индексирующей структуры данных в созданной с использованием модуля *btrees_mods* виртуальной таблице, в DOT-файл для GraphViz. Вызывается при запросе вида «SELECT btreesModsVisualize(“btt”, “btt.dot”);», где btt – название виртуальной таблицы, а btt.dot – название сохраняемого DOT-файла.
btreesModsGetTreeOrder	Выводит порядок дерева, используемого в качестве индексирующей структуры данных в созданной с использованием модуля

Название метода	Описание метода
	btrees_mods виртуальной таблице. Вызывается при запросе вида «SELECT btreesModsGetTreeOrder(“btt”);», где btt – название виртуальной таблицы.
btreesModsGetTreeType	Выводит тип дерева (1 – B-дерево, 2 – B^+-дерево, 3 – B^*-дерево, 4 – B^{*+}-дерево), используемого в качестве индексирующей структуры данных в созданной с использованием модуля *btrees_mods* виртуальной таблице. Вызывается при запросе вида «SELECT btreesModsGetTreeType(“btt”);», где btt – название виртуальной таблицы.

Часть описанных в табл. 1 методов вызывают вспомогательные методы, написанные в рамках разработки расширения для SQLite, в том числе, метод *rebuildIndexIfNecessary()*, реализующий алгоритм выбора индексирующей структуры данных модификаций B-дерева (B^+-дерева, B^*-дерева и B^{*+}-дерева), описанный в п. 2.3. Кроме того, расширение для SQLite использует специально разработанные вспомогательные структуры данных, описанные в табл. 2.

Таблица 2

Вспомогательные структуры данных расширения для SQLite

Название структуры данных	Назначение	Поля
indexParams	Параметры индексирующей структуры данных таблицы.	• int bestIndex – номер типа используемой индексирующей структуры данных (1 – B-дерево, 2 – B^+-дерево, 3 – B^*-дерево и 4 – B^{*+}-дерево). • int indexColNumber – номер столбца

Название структуры данных	Назначение	Поля
		таблицы, представляющего собой первичный ключ таблицы. • char* indexColName – имя столбца таблицы, представляющего собой первичный ключ таблицы. • char* indexDataType – название типа данных первичного ключа таблицы. • int indexDataSize – размер типа данных первичного ключа таблицы. • char* treeFileName – имя файла с индексирующей структурой данных таблицы (деревом).
indexStats	Статистика использования индексирующей структуры данных таблицы.	• int searchesCount – количество поисков по индексирующей структуре данных таблицы (дереву). • int insertsCount – количество вставок в индексирующую структуру данных таблицы (дерево).

Название структуры данных	Назначение	Поля
		• int deletesCount – количество удаление из индексирующей структуры данных таблицы (дерева). • int isOriginalStats – 1, если объект indexStats был создан вместе с созданием таблицы, 0 в ином случае.
btreesModsVirtualTable	Объект виртуальной таблицы.	• sqlite3_vtab base – объект базовой структуры. • sqlite3* db – указатель на подключение к базе данных SQLite. • char* tableName – название виртуальной таблицы. • FileBaseBTree* tree – указатель на индексирующую структуру данных виртуальной таблицы (B-дерево или его модификация). • indexParams params – параметры индексирующей

Название структуры данных	Назначение	Поля
		структуры данных виртуальной таблицы. • indexStats stats – статистика использования индексирующей структуры данных виртуальной таблицы.
btreesModsCursor	Курсор виртуальной таблицы.	• sqlite3_vtab_cursor base – объект базовой структуры. • sqlite_int64* rowsIds – массив rowid найденных строк. • int currentRowIdIdx – текущий индекс в масстве rowsIds. • int rowsIdsCount – размер массива rowsIds.

3.4. Пример тестирования и использования расширения для SQLite

Загрузим расширение *btrees_mods* при помощи команды *.load*. Создадим виртуальную таблицу *btt* с целочисленным первичным ключом *id*, целочисленным полем *a* и текстовым полем *b*. Вставим в таблицу две строки с *id = 4* и *id = 7*. Найдём данные строки в таблице при помощи запросов SELECT. Изучим содержимое таблиц *btt_real* (реальная таблица, к которой обращается модуль виртуальных таблиц *btrees_mods* при обработке запросов к таблице *btt*) и *btrees_mods_idxinfo* (таблица с данными о созданных при помощи модуля *btrees_mods* виртуальных таблицах). Удалим таблицу *btt* при помощи

запроса DROP TABLE, проверим список таблиц при помощи команды *.tables*. Данные действия и их результаты показаны на скриншоте на рис. 7.

```
SQLite version 3.26.0 2018-12-01 12:34:55
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite> .load ./btrees_mods
sqlite> CREATE VIRTUAL TABLE btt USING btrees_mods(id INTEGER PRIMARY KEY, a INTEGER, b TEXT);
sqlite> INSERT INTO btt VALUES (4, 2, "ABC123");
sqlite> INSERT INTO btt VALUES (7, 3, "def");
sqlite> SELECT * FROM btt WHERE id = 4;
4|2|ABC123
sqlite> SELECT * FROM btt WHERE id = 7;
7|3|def
sqlite> SELECT * FROM btt WHERE id = 4 OR id = 7;
4|2|ABC123
7|3|def
sqlite> .tables
btrees_mods_idxinfo  btt                  btt_real
sqlite> SELECT * FROM btt_real;
4|2|ABC123
7|3|def
sqlite> SELECT * FROM btrees_mods_idxinfo;
btt|2|0|id|INTEGER|4|tree_33051558297088.btree
sqlite> DROP TABLE btt;
sqlite> .tables
btrees_mods_idxinfo
sqlite> SELECT * FROM btrees_mods_idxinfo;
sqlite> .exit
```

Рисунок 7. Пример тестирования и использования расширения для SQLite

3.5. Эксперимент по сравнению вычислительной сложности операций на деревьях разных типов в разработанном расширении для SQLite

Проведён эксперимент по сравнению вычислительной сложности операций на деревьях разных типов в разработанном расширении для SQLite. Вычислительная сложность операции с таблицей в рамках эксперимента рассматривалась как время выполнения соответствующей операции и измерялась в миллисекундах (мс). Для измерения времени выполнения операции использовался менеджер SQLite с графическим пользовательским интерфейсом SQLiteStudio [8].

Таблица при помощи расширения *btrees_mods* всегда первоначально создаётся с использованием B^+-дерева. Время на создание таблицы в рамках эксперимента составило 20 мс. Время вставки первых 500 строк в таблицу составило 10301 мс, таким образом, среднее время вставки одной строки составило 20,6 мс. Время вставки последующих 500 строк в таблицу составило 10322 мс, таким образом, среднее время вставки одной строки также составило 20,6 мс. Во время вставки 1001-й строки в таблицу произошло перестраивание B^+-дерева в B^*-дерево, согласно алгоритму, описанному в п. 2.3. Время вставки вместе с перестраиванием дерева составило 40 мс. Время вставки последующих

499 строк в таблицу составило 9386 мс, то есть, в среднем, 18,8 мс на одну строку. После этого в таблицу было вставлено ещё 500 строк, общее время их вставки составило 9032 мс, то есть, в среднем, 18,1 мс на одну строку. Таким образом, на B^*-дереве вставка новых элементов в рамках данного эксперимента выполняется быстрее, чем на B^+-дереве.

Далее из таблицы было удалено 500 первых (в порядке вставки) строк. При каждом удалении строки выполняется две операции с деревом – поиск удаляемого элемента и собственно его удаление. Удаление данных 500 строк заняло 11558 мс, то есть, в среднем, 23,1 мс на одну строку. Удаление последующих 500 строк заняло 10708 мс, то есть, в среднем, 21,4 мс на одну строку. При удалении 1001-й строки произошло перестраивание B^*-дерева в B^{*+}-дерево, согласно алгоритму, описанному в п. 2.3. Время удаления вместе с перестраиванием дерева заняло 62 мс. Далее было удалено ещё 499 строк, что заняло 9418 мс, то есть, в среднем, 18,9 мс на одну строку. Последние 500 строк из таблицы были удалены за 8863 мс, то есть, в среднем, 17,7 мс на одну строку. Таким образом, на B^{*+}-дереве удаление в рамках данного эксперимента выполняется быстрее, чем на B^*-дереве.

После этого в таблицу было вставлено 1000 строк, что заняло 18890 мс, то есть, в среднем, 18,9 мс на одну строку. Далее было вставлено ещё 5000 строк, что включило в себя перестраивание B^{*+}-дерева в B^*-дерево, согласно алгоритму, описанному в п. 2.3. Это заняло 92395 мс (включая перестраивание дерева) то есть, в среднем, 18,5 мс на одну строку. Таким образом, выигрыш в скорости вставки у B^*-дерева перед B^{*+}-деревом в рамках данного эксперимента незначителен.

Поиск строки в таблице занял, в среднем, 1 мс, на всех типах деревьев.

Заключение

В рамках работы проведён обзор основных источников, включая статьи о B-дереве и его модификациях и официальный сайт SQLite. Аналогов программного продукта, разработанного в рамках выполнения данной работы, в открытом доступе не обнаружено.

Реализовано API на языке C для имеющейся C++-библиотеки сильно ветвящихся деревьев, с использованием конструкции *extern "C" { ... }*. Разработан и протестирован компонент-расширение РСУБД SQLite для индексирования данных модификациями B-деревьев, с реализацией алгоритма выбора индексирующей структуры данных из модификаций B-дерева (B^+-дерева, B^*-дерева и B^{*+}-дерева), также позволяющий выводить графическое изображение B-дерева или его модификации, используемой в данной таблице, в формате DOT для GraphViz, и основные данные о дереве. Разработана техническая документация для реализованного в рамках данной работы программного продукта. Таким образом, все поставленные задачи выполнены.

Результаты данной работы могут быть использованы разработчиками и исследователями, для сравнения параметров эффективности (например, времени выполнения операций) модификаций B-дерева (B^+-дерева, B^*-дерева и B^{*+}-дерева) и использования указанных модификаций B-дерева в качестве индексирующих структур данных в SQLite, в том числе, в учебных и научных целях.

Направлениями дальнейших разработок могут стать различные доработки разработанного расширения для SQLite, например, поиск строк в таблице по условиям «меньше», «меньше или равно», «больше», «больше или равно» для значения первичного ключа таблицы, поддержка транзакционности и поддержка команд с JOIN, а также разработка плагина для одного из SQLite-менеджеров с графическим пользовательским интерфейсом, для более удобной работы с B-деревьями и их модификациями. Кроме того, возможны доработки C++-библиотеки сильно ветвящихся деревьев, например, использование циклов вместо рекурсии, где это возможно, для экономии используемой оперативной памяти, и замена обычного линейного прохода по ключам узла бинарным поиском, для уменьшения вычислительной сложности операций с деревом.

Список использованных источников

1. Bayer R. Organization and Maintenance of Large Ordered Indices / Bayer R., McCreight E. // Acta Informatica. – 1972 – No. 1 (3) – P. 173 – 189.
2. Comer D. The Ubiquitous B-Tree // ACM Computing Surveys. – 1979. – June (vol. 11, no. 2). – P. 121 – 137.
3. Database File Format // [Электронный ресурс]: SQLite. Режим доступа: https://www.sqlite.org/fileformat.html, свободный. (дата обращения: 22.04.2019).
4. Manuika J. Big data: The next frontier for innovation, competition, and productivity / Manuika, J., Chui, M., Brown, B., Bughin, J., Dobbs, R., Roxburgh, C., and Hung Byers, A. // [Электронный ресурс]: McKinsey Global Institute, McKinsey & Company, May 2011. Режим доступа: https://www.mckinsey.com/~/media/McKinsey/Business%20Functions/McKinsey%20Digital/Our%20Insights/Big%20data%20The%20next%20frontier%20for%20innovation/MGI_big_data_exec_summary.ashx, свободный. (дата обращения: 20.03.2018).
5. Pollari-Malmi K. B^+-trees // [Электронный ресурс]: Computer Science | University of Helsinki. Режим доступа: https://www.cs.helsinki.fi/u/mluukkai/tirak2010/B-tree.pdf, свободный. (дата обращения: 07.12.2017).
6. Run-Time Loadable Extensions // [Электронный ресурс]: SQLite. Режим доступа: https://www.sqlite.org/loadext.html, свободный. (дата обращения: 22.04.2019).
7. SQLite Home Page // [Электронный ресурс]: SQLite. Режим доступа: https://www.sqlite.org/index.html, свободный. (дата обращения: 22.04.2019).
8. SQLiteStudio // [Электронный ресурс]: SQLiteStudio. Режим доступа: https://sqlitestudio.pl/index.rvt, свободный. (дата обращения: 06.05.2019).
9. The SQLite R*Tree Module // [Электронный ресурс]: SQLite. Режим доступа: https://www.sqlite.org/rtree.html, свободный. (дата обращения: 22.04.2019).
10. Кормен Т. Алгоритмы: построение и анализ. 3-е изд. / Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн. — М.: ИД «Вильямс». — 2013. — 1324 с.
11. Ригин А.М. Исследование эффективности сильно ветвящихся деревьев в задаче индексирования структурированных данных : Курсовая работа / Ригин Антон Михайлович; НИУ ВШЭ. – М., 2018. – 37 с.