DOI: 10.15514/ISPRAS-2019-31(3)-16

SQLite RDBMS Extension for Data Indexing Using B-tree Modifications

A.M. Rigin, ORCID: 0000-0003-4081-9144 <amrigin@edu.hse.ru>
S.A. Shershakov, ORCID: 0000-0001-8173-5970 <sshershakov@hse.ru>
National Research University – Higher School of Economics,
20, Myasnitskaya st., Moscow, 101000, Russia

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-trees, including B^+-trees, B^*-trees and B^{*+}-trees 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. In this research the B^{*+}-tree was developed as the data structure which combines the main B^+-tree and B^*-tree features together. Also, in the research the empirical computational complexities of different operations on the B-tree and its modifications were measured as well as the memory usage. The purpose of the current 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 in the SQLite RDBMS. The modifications of the base data structure were developed as a C++ library. The library is connected to the SQLite using the C-C++ cross-language API which is developed in the current work. The SQLite extension implements the novel algorithm for selecting the index structure (one of B-tree's modifications) for some table of a database. The provided SQLite extension is adopted by the SQLite EventLog component of the LDOPA process mining library. In addition, the experiment on the counting the empirical computational complexities of operations on the trees of different types is conducted using the developed in this work SQLite extension.

Keywords: B-tree; data indexing; SQLite; DBMS; RDBMS; multiway tree

For citation: Rigin A.M., Shershakov S.A. SQLite RDBMS Extension for Data Indexing Using B-tree Modifications. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 3, 2019. pp. 203-216. DOI: 10.15514/ISPRAS-2019-31(3)-16

Acknowledgements. This work is supported by RFBR according to the Research project No. 18-37-00438 «mol_a» and the Basic Research Program of the National Research University – Higher School of Economics.

Компонент-расширение РСУБД SQLite для индексирования данных модификациями B-деревьев

А.М. Ригин, ORCID: 0000-0003-4081-9144 <amrigin@edu.hse.ru>
С.А. Шершаков, ORCID: 0000-0001-8173-5970 <sshershakov@hse.ru>
Национальный исследовательский университет «Высшая школа экономики»,
101000, Россия, г. Москва, ул. Мясницкая, д. 20

Аннотация. Сильно ветвящиеся деревья являются одним из наиболее популярных решений для индексирования больших объёмов данных. Наиболее распространённой разновидностью сильно ветвящихся деревьев являются B-деревья. Существуют различные модификации B-деревьев, в том числе, рассматриваемые в настоящей работе B^+-деревья, B^*-деревья и B^{*+}-деревья, однако данные модификации не поддерживаются по умолчанию в популярной реляционной СУБД с открытым исходным кодом SQLite. Данная работа выполняется на основе проведённого ранее исследования эффективности сильно ветвящихся деревьев в задаче индексирования структурированных данных, с использованием разработанной в рамках него C++-библиотеки структур данных – сильно ветвящихся деревьев. В этом исследовании было разработано B^{*+}-дерево как структура данных, совмещающая в себе основные свойства B^+-дерева и B^*-дерева. Также в исследовании были измерены эмпирические вычислительные сложности различных операций над B-деревом и его модификациями и объём потребляемой данными операциями оперативной памяти. Целью настоящей 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 в РСУБД SQLite. Модификации базовой структуры данных были разработаны в виде C++-библиотеки. Данная библиотека подключается к SQLite, используя разработанный для неё в рамках настоящей работы API на языке C. Расширение для SQLite также реализует новый алгоритм выбора индексирующей структуры данных (одной из модификаций B-дерева) для заданной таблицы в базе данных. Предложенное расширение используется компонентом SQLite EventLog библиотеки LDOPA алгоритмов и структур данных для process mining. Кроме того, проведён эксперимент по сравнению эмпирической вычислительной сложности операций на деревьях разных типов в разработанном расширении для SQLite.

Ключевые слова: B-дерево; индексирование данных; SQLite; СУБД; РСУБД; сильно ветвящееся дерево

Для цитирования: Ригин А.М., Шершаков С.А. Компонент-расширение РСУБД SQLite для индексирования данных модификациями B-деревьев. Труды ИСП РАН, том 31, вып. 3, 2019 г., стр. 203-216 (на английском языке). DOI: 10.15514/ISPRAS-2019-31(3)-16

Благодарности. Работа выполнена при поддержке РФФИ (проект № 18-37-00438) и Программы фундаментальных исследований Национального исследовательского университета – Высшей школы экономики.

1. Introduction

Last decades, the amount of data volume is growing substantially, which exposes the well-known problem of big data [1]. Many companies and laboratories need to collect, store and process big data. There exist many algorithmic and software solutions to cope with these problems. One of these solutions is using indices which are usually represented by data structures such as hash tables and trees.

Using indices creates a new problem – when data are stored on slow carriers, it is more efficient to load data batches from a storage instead of splitting to individual elements. Multiway trees solve this problem. One type of them is a B-tree which was initially described by Bayer and McCreight in 1972 [2]. The B-tree also has several modifications. In this paper, the following B-tree modifications are considered: B^+-tree [3], B^*-tree [4] and B^{*+}-tree (the latter is developed by the author of this paper data structure, which combines the main B^+-tree and B^*-tree features) [5].

This paper extends the research made in the framework of the term project [5].

One of the popular open-source relational database management systems (RDBMS) is SQLite [6]. It is used in mobile phones, computers and many other devices. However, this RDBMS does not support using B^+-tree or B^*-tree as data index structures by default.

The main goals of the work are the following:

- to add B-tree modifications such as B^+-tree, B^*-tree and B^{*+}-tree to SQLite;
- to develop and implement an algorithm that would allow selecting the appropriate indexing data structure (B-tree, B^+-tree, B^*-tree or B^{*+}-tree) when a user manipulates a table.

The work includes linking of B-tree modifications from a C++ library (developed by the author of this work previously) to SQLite using a C-C++ cross-language API and developing an algorithm for selecting an indexing data structure.

The rest of the paper is organized as follows. Firstly, B-tree, B^+-tree, B^*-tree and B^{*+}-tree are shortly described. After this, the SQLite, its indexing algorithms and extensions are presented. Then, the B-

tree modifications C++ library and connecting it to the SQLite RDBMS is described. After this, our previous researches conducted using this library are presented. These researches have proved the main theoretical B-tree modifications complexity hypotheses and they show the abilities of this library. Then, the indexing approach, the methods for outputting the index representation and information and the development of algorithm of selecting the index structure for table are discussed, after which the experiment conducted using the developed in this work SQLite extension is described. After this, the main points of the paper are summarized in conclusion and used references are presented.

2. B-tree and its modifications

2.1 B-tree

The B-tree is a multiway tree. It means that each node may contain more than one data key. Furthermore, each node except of the leaf nodes contains more than one pointer to the children nodes. If some node contains k keys than it contains exactly $k + 1$ pointers to the children nodes [2].

The B-tree depends on its important parameter which is called B-tree order. The B-tree order is such a number t that:

- for each non-root node, the following is true: $t - 1 \leq k \leq 2t - 1$, where k is the number of keys in the node [2];
- for root node in the non-empty tree the following is true: $1 \leq k \leq 2t - 1$, where k is the number of keys in the node [2];
- for root node in the empty tree the following is true: $k = 0$, where k is the number of keys in the node [2].

B-tree operations complexities are the following (t is the tree order, n is the tree total keys count):

- for the searching operation: time complexity is $O(tlog_t n)$, memory usage is $O(t)$ and disk operations count is $O(log_t n)$ [2];
- for the nodes split operation (the part of the insertion operation): time complexity is $O(t)$, memory usage is $O(t)$ and disk operations count is $O(1)$ [2];
- for the insertion operation (includes the nodes split operation): time complexity is $O(tlog_t n)$, memory usage is $O(tlog_t n)$ for simple recursion and $O(t)$ for tail recursion and disk operations count is $O(log_t n)$ [2];
- for the deletion operation: time complexity is $O(tlog_t n)$, memory usage is $O(tlog_t n)$ for simple recursion and $O(t)$ for tail recursion and disk operations count is $O(log_t n)$ [2].

B-tree is usually used as the data index [2].

The example of B-tree is shown on the fig. 1.

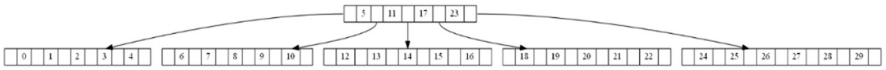

Fig. 1. The B-tree example, tree order t = 6

2.2 B-tree modifications

B^+-tree is the B-tree modification in which only leaf nodes contain real keys (real data), other nodes contain router keys for searching real keys. Leaf nodes in B^+-tree contain $t \leq k \leq 2t$ keys, where t is the tree order, the rules for other nodes are the same as in B-tree [3]. Keys deletion in B^+-tree is expected to be faster than in B-tree since it is always performed on the leaf nodes.

B^*-tree is the B-tree modification in which each node (except of the root node) is filled at least by 2/3 not 1/2 [4]. Keys insertion in B^*-tree is expected to be faster than in B-tree.

B^{*+}-tree is the B-tree modification developed by the author of this paper which combines the main B^+-tree and B^*-tree features together. In this data structure only leaf nodes contain real keys (real data) as in B^+-tree and each node (except of the root node) is filled at least by 2/3 as in B^*-tree.

3. Implementation and tools

3.1 SQLite and its extensions

The SQLite is the popular open-source C-language library which implements the SQLite relational database management system (RDBMS) [6]. The SQLite default index algorithms are hash-table and B-tree. The SQLite does not implement B^+-tree and B^*-tree based indices.

Nevertheless, SQLite supports loading its extensions at run-time, which can add new functionality to the SQLite. For example, it can be a new index structure implementation. One of such extensions is the R-tree. The R-tree is a B-tree modification which allows to index geodata. It is loaded by the SQLite as the extension and delivered together with the SQLite RDBMS default build.

3.2 B-tree modifications C++ library

The B-tree modifications C++ library was developed by the author of this paper previously. It contains B-tree, B^+-tree, B^*-tree and B^{*+}-tree implementations written in C++ [5].

In the current work this library is connected to the SQLite as the run-time loadable extension. For this goal the C-C++ cross-language API is implemented. It is possible to do using the *extern "C" { ... }* C++ statement. The other tasks are to implement base SQLite extension's methods and to use Makefiles to make this extension run-time loadable correctly. The extension provides module for creating virtual tables (tables which encapsulate callbacks instead of simple reading from database and writing to database) based on this module.

3.3 Research conducted using the library

The B-tree modifications C++ library was previously used for conducting a research on the performance of multiway trees in the problem of structured data indexing by the author of this paper [5].

The CSV files with random content were generated for the indexing, with sizes of 25000, 50000, 75000, 100000 rows. The value of the first cell of each row was considered as a key («name») of the row and was saved in the tree together with the bytes offset of the row in the indexed CSV file. The charts of different dependencies were built using the Python 2.

The chart with the indexing time dependence on the tree order for a file where the «names» (keys) of the rows are uniformly distributed, with the size of 25000 rows is shown on the fig. 2.

According to this chart, B^*-tree and B^{*+}-tree have a better time performance on the keys insertion than B-tree and B^+-tree, as expected. These results are confirmed by the experiments with other parameters (for example, on the larger files with different keys).

However, the better time performance of B^*-tree and B^{*+}-tree on the keys insertion has a cost of a larger memory usage as shown on the fig. 3.

The monotonous dependence of the keys searching on the tree order is not detected as shown on the fig. 5.

The B^*-tree and B^{*+}-tree require more memory during the keys searching than the B-tree and B^+-tree as shown on the fig. 6.

In addition, the B^+-tree and B^{*+}-tree have a better time performance on the keys removing than B-tree and B^*-tree as expected and shown on the Fig. 7. This chart also proves that the B^{*+}-tree has the best time performance on the keys removing among all the considered in this paper multiway trees and that the dependence of keys removing time on the tree size is logarithmic.

Therefore, the main theoretical hypotheses were confirmed [5].

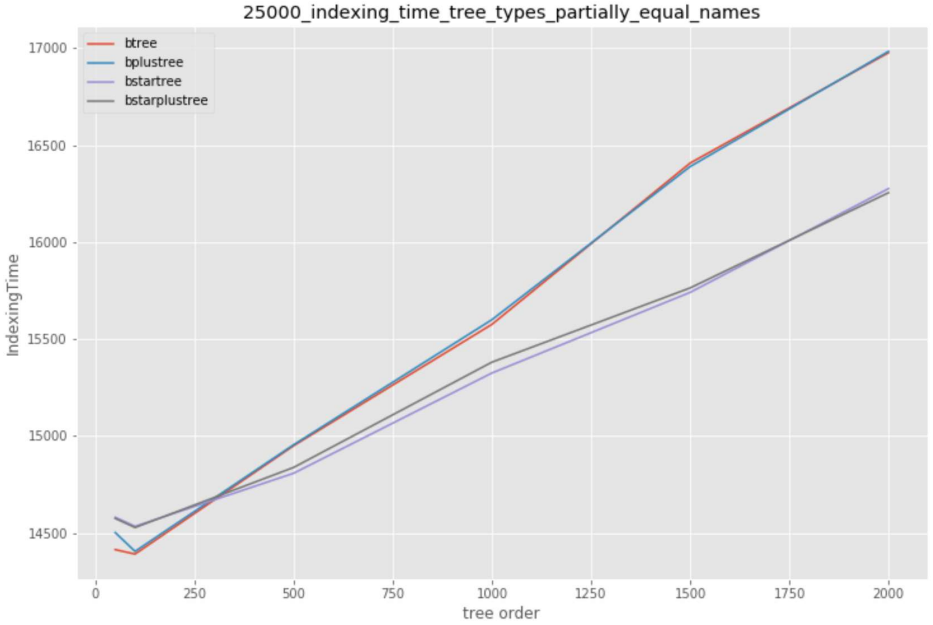

Fig. 2. The chart with the indexing time dependence on the tree order for a file where the «names» (keys) of the rows are uniformly distributed, with the size of 25000 rows

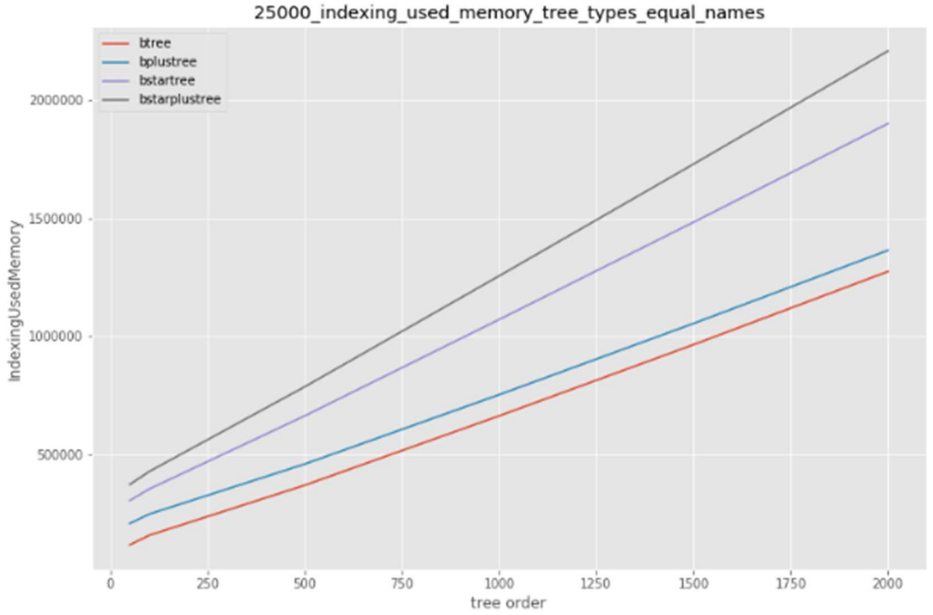

Fig. 3. The chart with the indexing memory usage dependence on the tree order for a file where all the «names» (keys) of the rows are equal, with the size of 25000 rows

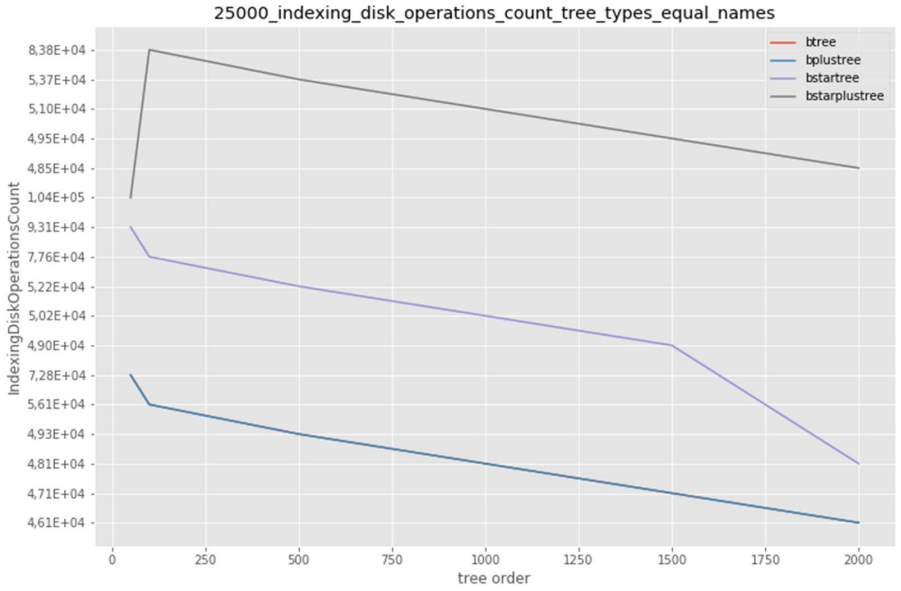

Fig. 4. The chart with the indexing disk operations count dependence on the tree order for a file where all the «names» (keys) of the rows are equal, with the size of 25000 rows

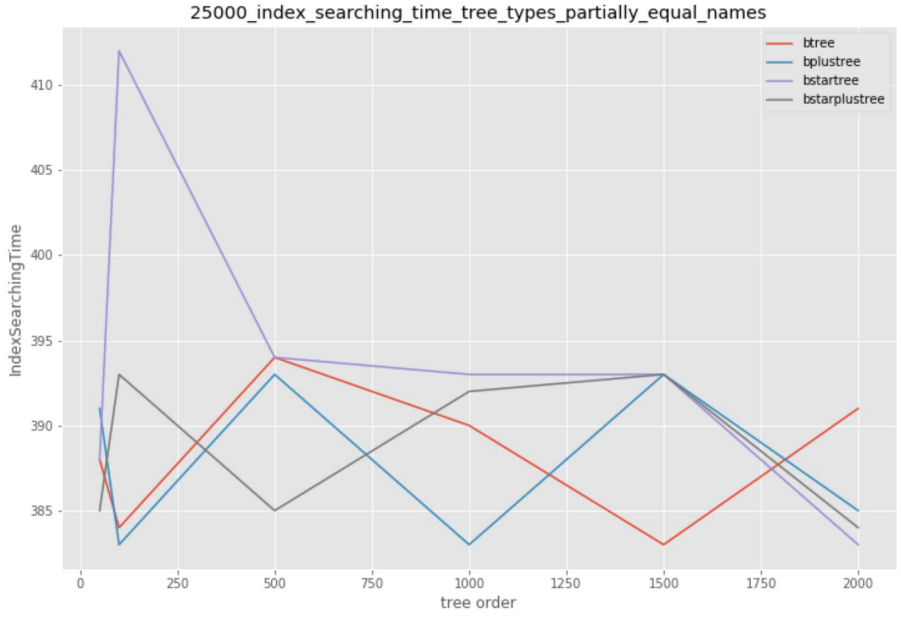

Fig. 5. The chart with the index searching time dependence on the tree order for a file where the «names» (keys) of the rows are uniformly distributed, with the size of 25000 rows

Also, indexing using B-tree or B*+-tree requires more disk operations than indexing using B-tree or B+-tree as shown on the fig. 4.*

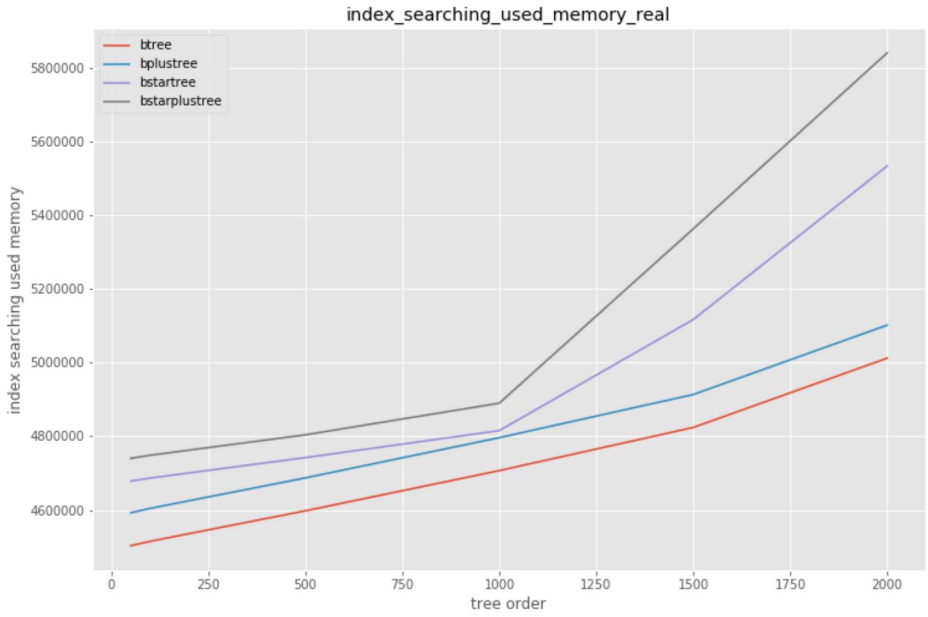

Fig. 6. The chart with the index searching memory usage dependence on the tree order for a file with real (not randomly generated) data

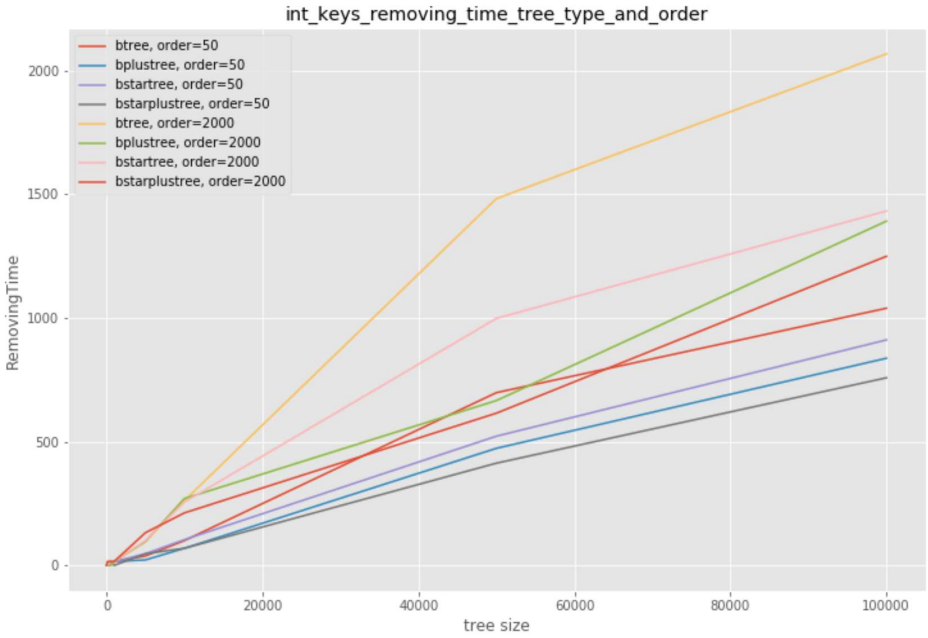

Fig. 7. The chart with the keys removing time dependence on the tree size

4. Working with indices while manipulating DB data

4.1 Table creation, data search and updating

In the current work B-tree modifications based indices are built over the existing SQLite table implementation which is represented in the storage as pages of a B-tree by default.

The table creation and main data operations (inserting, searching, deleting and updating) use the methods presented in the Table. 1.

Table. 1. Main extension methods

Method	Purpose
btreesModsCreate(sqlite3*, void*, int, const char* const*, sqlite3_vtab**, char**)	Creates a new table.
btreesModsUpdate(sqlite3_vtab*, int, sqlite3_value**, sqlite_int64*)	Inserts, deletes or updates a value of a row in the table.
btreesModsFilter(sqlite3_vtab_cursor*, int, const char*, int, sqlite3_value**)	Searches for a row in the table.

The extension with the B-tree modifications based indices provides module for creating virtual tables. User should create a virtual table using the module called *btrees_mods* in order to use one of the B-tree modifications as index for the table. When a user creates such virtual table, the *btreesModsCreate()* method of the extension is called and the matching real table is created in the database. Also, one of B-tree's modifications is created using the algorithm of selecting the index's structure (see the section 5) and the information about the created table and index's structure (including the name of the file with the B-tree or its modification and the attributes of the primary key of the table) is stored in a special table.

When a user inserts a row into a table, the *btreesModsUpdate()* method of the extension is called and a corresponding record for the index structure is created. The record consists of the primary key value of this row and the row id. This record is saved as a data key into the index structure (B-tree or one of its modifications).

When a user searches for a row in a table, the *btreesModsFilter()* method of the extension is called and the value of the primary key of the row being searched is compared with the keys of the index structure. During the key searching only the primary key value part of the tree's keys is compared with the value of the primary key of the row being searched. If the necessary tree's key is found, the row id is extracted from the key and a row found in the table by the row id is considered as a result of the searching.

When a user deletes a row from a table, the *btreesModsUpdate()* method of the extension is called, the primary key of the deleted row is found in the index structure using the same approach as in the search case. The found key is deleted from the index structure.

When a user updates the value of the primary key of a row in a table, the *btreesModsUpdate()* method of the extension is called. The old value of the primary key is deleted from the index structure and the new value is inserted to the index structure.

4.2 Index structure's graphical representation and main information outputting

Also, the several methods are available to output the index structure's graphical representation and main information. They are presented in the Table. 2.

Table. 2. Index structure's information and graphical representation outputting extension methods

Method	Purpose
btreesModsVisualize(sqlite3_context*, int, sqlite3_value**)	Outputs the graphical representation of the table's index structure (tree) into the GraphViz DOT file. It is called after the SQL query such as *SELECT btreesModsVisualize("btt", "btt.dot");*, where *btt* is the table name, *btt.dot* is the outputting GraphViz DOT file name.
btreesModsGetTreeOrder(sqlite3_context*, int, sqlite3_value**)	Outputs the order of the tree used as the table's index structure. It is called after the SQL query such as *SELECT btreesModsGetTreeOrder("btt");*, where *btt* is the table name.
btreesModsGetTreeType(sqlite3_context*, int, sqlite3_value**)	Outputs the type of the tree (1 – B-tree, 2 – B$^+$-tree, 3 – B*-tree, 4 – B^{*+}-tree) used as the table's index structure. It is called after the SQL query such as *SELECT btreesModsGetTreeType("btt");*, where *btt* is the table name.

```
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|1|0|id|INTEGER|4|tree_18291557263097.btree
sqlite> DROP TABLE btt;
sqlite> .tables
btrees_mods_idxinfo
sqlite> SELECT * FROM btrees_mods_idxinfo;
sqlite> .exit
```

Fig. 8. SQLite extension's usage example

4.3 SQLite extension's usage

The developed in this work SQLite extension's usage example is presented on the screenshot (fig. 8).

The provided SQLite extension is adopted by the SQLite EventLog component of the Library for Dynamic Operational Process Analysis (LDOPA) [7].

5. Algorithm of selecting the index structure

In this work an algorithm for selecting the index structure for a table is developed and implemented in the following way.

The algorithm considers B-tree's modifications (B$^+$-tree, B*-tree and B^{*+}-tree) for using as an index structure.

The algorithm is executed at the start of each operation on the table (search, insertion, deletion or update of the table's row) which uses the *btrees_mods* module. The algorithm consists of the following steps.

1) If the current total number of the operations on a tree is equal to 0, or more than 10000, or not a multiple of 1000, then the algorithm stops, otherwise it goes to step 2.
2) If the current number of the modifying operations (key insertions, key deletions) on the tree is less than 10 % of the current total number of the operations on the tree, then the algorithm stops, otherwise it goes to step 3.
3) If the current number of the key insertion operations is more than $p = 73.97$ % of the total number of the modifying operations on the tree, then the algorithm selects the B*-tree as the index structure and goes to step 5, otherwise it goes to step 4.
4) The algorithm selects the B^{*+}-tree as the index structure and goes to step 5.
5) If the new index structure has been selected at the steps 3 – 4, then the algorithm rebuilds the existing index structure replacing it by the new selected index structure and copies all the data stored in the previous index structure to the new index structure.

The tree order of the B-trees and their modifications used in the SQLite extension developed in this work equals 750. For selecting this tree order the average times (for all the four tree types – B-tree, B$^+$-tree, B*-tree and B^{*+}-tree) of performing 1000 modifying operations (insertions and deletions) on the tree were measured, for each of the tree orders from 100 to 1000 inclusive with the step of 50 (100, 150, 200, …, 1000). The least average time was achieved for the tree order of 750 and it was equal to 9.55 ms (for 1000 modifying operations on the tree).

The $p = 73.97$ % constant was selected in the following way. The splines for the plots of the average time of performing 1000 modifying operations (insertions and deletions) on the tree depending on the percentage of the insertions among all the modifying operations were drawn for all the four tree types (B-tree, B$^+$-tree, B*-tree and B^{*+}-tree) using the Python 2 language. The abscissa of the intersection point of the splines for B*-tree and B^{*+}-tree was equal to $p = 73.97$ %. This intersection point is shown on the fig. 9.

The B$^+$-tree is used as the default index structure in the developed SQLite extension since its operations have the least memory usage according to the previously conducted experiments (see the section 3.3).

6. Experiment conducted using the developed SQLite extension

The experiment on the counting the empirical computational complexities of operations on the trees of different types is conducted using the developed in this work SQLite extension. The operations' times were counted using the SQLiteStudio GUI manager [8]. The results are presented in the Table 3.

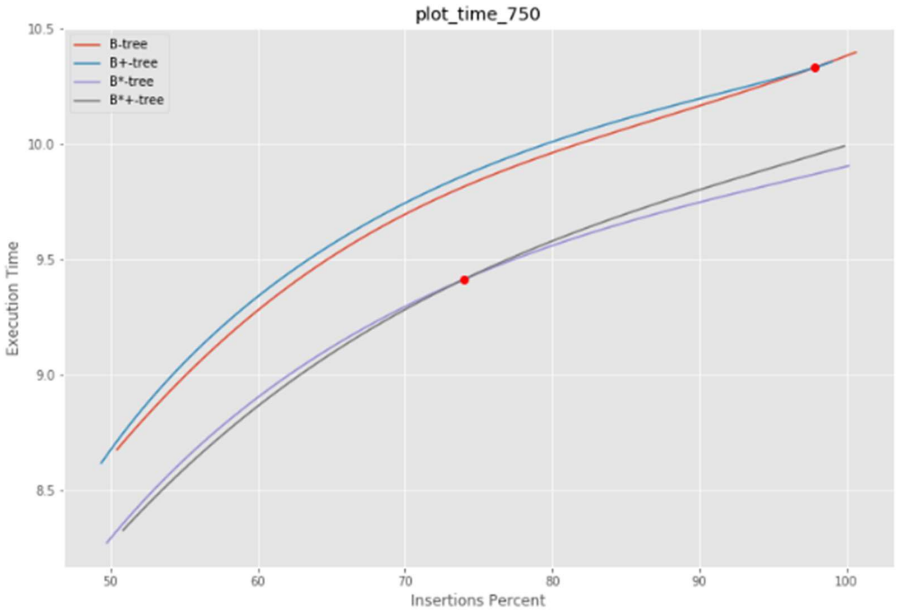

Fig. 9. The splines for the plots of the average time of performing 1000 modifying operations on the tree depending on the percentage of the insertions among all the modifying operations.

Table. 3. Experiment results

Operation on the table	Total execution time (ms)	Mean execution time per row (ms)
Table creation	20	-
First 500 rows insertion	10301	20.6
Next 500 rows insertion	10322	20.6
1001st row insertion (including the B^+-tree into the B^*-tree rebuilding)	40	40
Next 499 rows insertion	9386	18.8
Last 500 rows insertion	9032	18.1
First 500 rows deletion	11558	23.1
Next 500 rows deletion	10708	21.4
1001st row insertion (including the B^*-tree into the B^{*+}-tree rebuilding)	62	62
Next 499 rows deletion	9418	18.9
Last 500 rows deletion	8863	17.7
1000 rows insertion	18890	18.9
Next 5000 rows insertion (including the B^{*+}-tree into the B^*-tree rebuilding)	92395	18.5

According to the data in the Table. 3, the key insertion into the B^*-tree was faster than into the B^+-tree during the experiment. The key deletion from the B^{*+}-tree was faster than from the B^*-tree during the experiment. Also, the key insertion into the B^*-tree was slightly faster than into the B^{*+}-tree during the experiment.

The search in a table took about 1 ms on all the B-tree modifications considered in this work.

7. Conclusion

The big data problem currently affects the world. There are many mathematical and software solutions for collecting, storing and processing big data including the data indexing. Many of the index data structures are tree-based ones such as B-tree and its modifications. B-tree is used as an index structure in many DBMSs including the popular open-source RDBMS SQLite. However, the SQLite does not support its modifications which may be more appropriate for some tasks than the original B-tree. In the current work this problem is elaborated.

Firstly, the B-tree modifications C++ library is connected to the SQLite as the extension using C-C++ cross-language API. After this, the algorithm of the index structure selection is developed and implemented and the experiment is conducted using the developed in this work SQLite extension.

The developed B^{*+}-tree has smaller running time for keys insertion and deletion than B-tree, however it has greater memory usage, which is confirmed by the experiments conducted using the B-tree modifications C++ library.

This work tests new data indexing approaches using the SQLite as an example. The results of the work can be used by researchers and professors in this field and their students. The developed SQLite B-tree modifications extension can be used by all the developers who use this DBMS.

References / Список литературы

[1]. Manyika J., Chui M., Brown B., Bughin J., Dobbs R., Roxburgh C., Hung Byers A. Big data: The next frontier for innovation, competition, and productivity. McKinsey Global Institute, May 2011. Available at: 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, accessed Jan. 20, 2019.

[2]. Bayer R., McCreight E. Organization and maintenance of large ordered indexes. Acta Informatica, vol. 1, no. 3, 1972, pp. 173 – 189.

[3]. Pollari-Malmi K. B^+-trees. Available at: https://www.cs.helsinki.fi/u/mluukkai/tirak2010/B-tree.pdf, accessed Dec. 24, 2018.

[4]. B^*-tree. NIST Dictionary of Algorithms and Data Structures. Available at: https://xlinux.nist.gov/dads/HTML/bstartree.html, accessed Dec. 24, 2018.

[5]. Rigin A.M. On the Performance of Multiway Trees in the Problem of Structured Data Indexing. Coursework, Dept. Soft. Eng., HSE, Moscow, Russia, 2018 (in Russian) / Ригин В.М. Исследование эффективности сильно ветвящихся деревьев в задаче индексирования структурированных данных. Курсовая работа, Департамент программной инженерии, ФКН, ВШЭ, Москва, 2018.

[6]. SQLite Home Page. Available at: https://www.sqlite.org/, accessed Jan. 20, 2019.

[7]. Library for Dynamic Operational Process Analysis (LDOPA). xiart.ru Projects. Available at: https://prj.xiart.ru/projects/ldopa, accessed Jul. 1, 2019.

[8]. SQLiteStudio. Available at: https://sqlitestudio.pl/, accessed Jan. 26, 2019.

Информация об авторах / Information about authors

Антон Михайлович РИГИН получил степень бакалавра в области программной инженерии в 2019 г. в Национальном исследовательском университете «Высшая школа экономики» (Москва, Россия). Его исследовательские интересы включают программную инженерию,

алгоритмы и структуры данных и их применение в задачах хранения и индексирования данных в СУБД, включая использование B-деревьев и их модификаций для решения этих задач.

Anton Mikhailovitch RIGIN received his bachelor's degree in software engineering from National Research University – Higher School of Economics (Moscow, Russia) in 2019. His research interests include software engineering, algorithms and data structures and their usage in the problems of data indexing and storage in DBMSs, which involves the usage of the B-trees and their modifications in these problems solving.

Сергей Андреевич ШЕРШАКОВ получил степень магистра в области программной инженерии в Национальном исследовательском университете «Высшая школа экономики» (Москва) в 2012 году. В настоящий момент он является научным сотрудником научно-учебной лаборатории процессно-ориентированных информационных систем факультета компьютерных наук Высшей школы экономики. В число научных интересов входят извлечение и анализ процессов (process mining), верификация программного обеспечения, архитектуры информационных систем и преподавание программной инженерии.

Sergey Anreevitch SHERSHAKOV received the MS degree in software engineering from National Research University — Higher School of Economics (Moscow, Russia) in 2012. He is currently a researcher at PAIS Lab of the Faculty of Computer Science at Higher School of Economics. His research interests include process mining, software verification, information systems architectures and teaching software engineering.