SQLite RDBMS Extension for Data Indexing Using B-tree Modifications

Anton Rigin[1], Sergey Shershakov[2]
Faculty of Computer Science
National Research University – Higher School of Economics
Moscow, Russia
[1]amrigin@edu.hse.ru, [2]sshershakov@hse.ru

The B-tree structure has several modifications which are, by default, not supported in the popular open-source RDBMS SQLite. In the scope of this work an extension for SQLite is developed, based on some modifications of the B-tree structure. The modifications of the base structure were developed as a C++ library. The library is connected to the SQLite using the C-C++ cross-language API. The SQLite extension also implements the novel algorithm for selecting the best possible index structure (B-tree or one of its modifications) for some table of a database.

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

I. 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 most 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 best 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.

II. B-tree and Its Modifications

A. 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: $t - 1 \le k \le 2t - 1$, where k is the number of keys in the node [2];
- for root node in the non-empty tree the following is true: $1 \le k \le 2t - 1$, 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 $O(t\log_t n)$, 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 $O(t\log_t n)$, memory usage is $O(t\log_t n)$ 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 $O(t\log_t n)$, memory usage is $O(t\log_t n)$ 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.

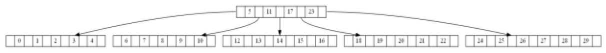


Fig. 1. The B-tree example, tree order $t = 6$.

B. 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 $t \le k \le 2t$ 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.

III. Implementation and Tools

A. 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.

B. 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.

C. 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.

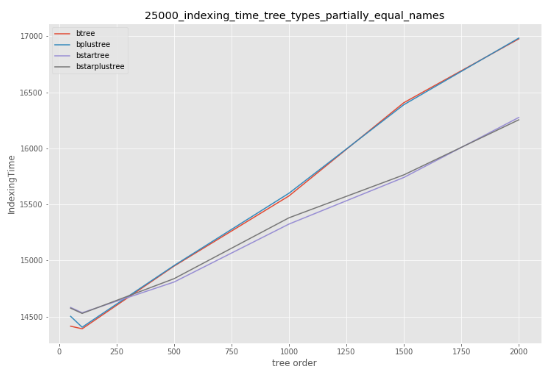


Fig. 2. The chart with the indexing time dependence on the tree order for a file where the "names" (keys) of the rows are uniformly distributed, with the size of 25000 rows.

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.

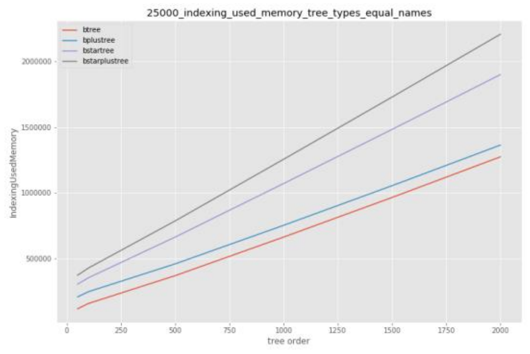


Fig. 3. The chart with the indexing memory usage dependence on the tree order for a file where all the "names" (keys) of the rows are equal, with the size of 25000 rows.

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.

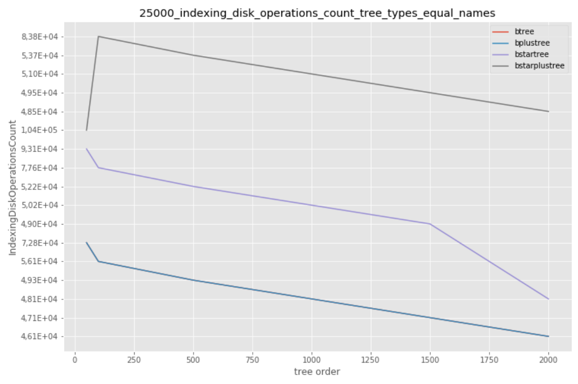


Fig. 4. The chart with the indexing disk operations count dependence on the tree order for a file where all the "names" (keys) of the rows are equal, with the size of 25000 rows.

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

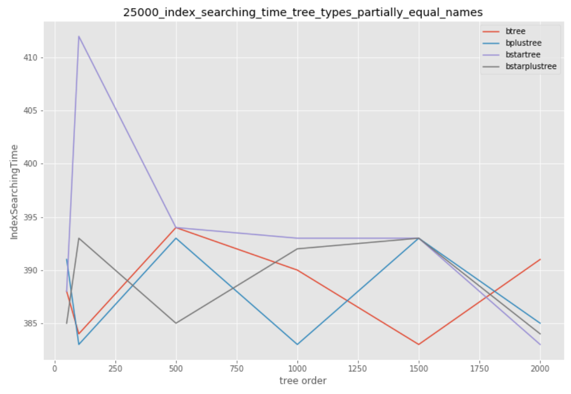


Fig. 5. The chart with the index searching time dependence on the tree order for a file where the "names" (keys) of the rows are uniformly distributed, with the size of 25000 rows.

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.

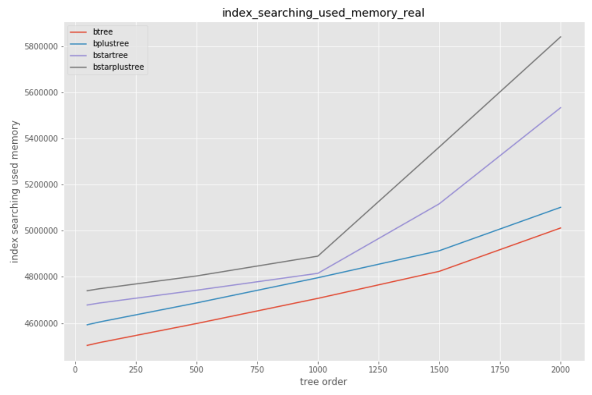


Fig. 6. The chart with the index searching memory usage dependence on the tree order for a file with real (not randomly generated) data.

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.

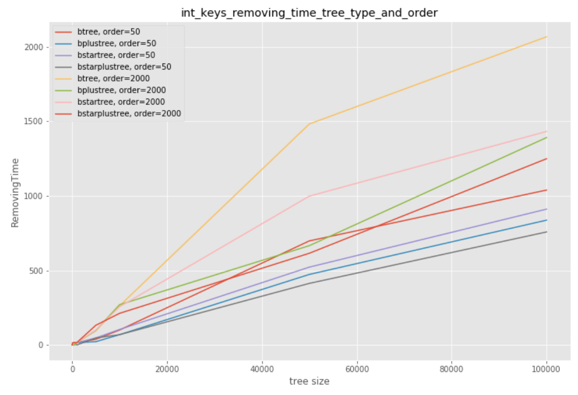


Fig. 7. The chart with the keys removing time dependence on the tree size.

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

IV. Working with Indices While Manipulating DB Data

A. *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 I.

TABLE I. 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, a B-tree or one of its modifications is created using the algorithm of selecting the best index's structure (see the section V) 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.

B. *Index Structure's Graphical Representation and Main Information Outputting*

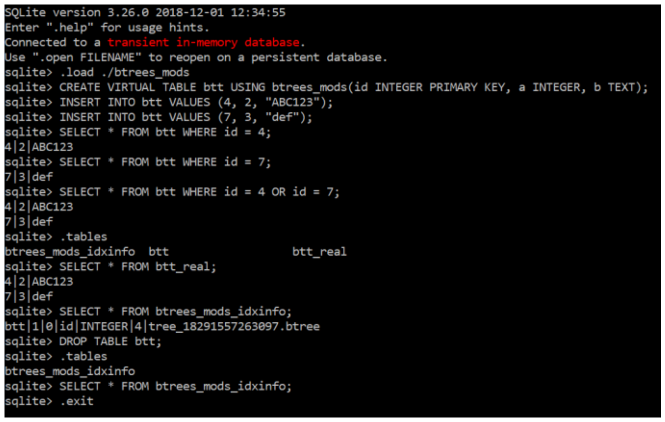
Also, the several methods are available to output the index structure's graphical representation and main information. They are presented in the Table II.

TABLE II. 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.

C. SQLite Extension's Usage Example

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



```
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.

V. ALGORITHM OF SELECTING THE BEST INDEX STRUCTURE

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

The algorithm considers B-tree and its modifications (B^+-tree, B^*-tree and B^{*+}-tree) for using as an index structure. The best index structure is defined by the step 6 of the algorithm.

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 count of the operations on the tree is equal to 0 or more than 10000 or not a multiple of 1000, then exit the algorithm, otherwise go to step 2.

2. If the current count of the modifying operations on the tree (key insertions, key deletions) is less than 10 % of the current total count of the operations on the tree, then exit the algorithm, otherwise go to step 3.

3. If the current count of the key insertion operations is more than 90 % of the current count of the modifying operations on the tree, then select the B^*-tree as the index structure and go to step 6, otherwise go to step 4.

4. If the current count of the key insertion operations is not less than 60 % of the current count of the modifying operations on the tree, then select the B^{*+}-tree as the index structure and go to step 6, otherwise go to step 5.

5. Select the B^+-tree as the index structure and go to step 6.

6. If the new index structure was selected in the steps 3 – 5, then rebuild the existing index structure into the new selected index structure (which is considered as the best index structure) saving all the data stored in the existing index structure.

The tree order of the B-trees and their modifications used in the SQLite extension developed in this work equals 100.

VI. EXPERIMENT CONDUCTED USING THE DEVELOPED SQLITE EXTENSION

The experiment on the counting the empirical computational complexity is conducted using the developed in this work SQLite extension. The operations' times were counted using the SQLiteStudio GUI manager [7]. The results are presented in the Table III.

TABLE III. EXPERIMENT RESULTS

Operation on the table	Total execution time (ms)	Mean execution time per row (ms)
Table creation	21	-
First 500 rows insertion	9128	18.3
Next 500 rows insertion	9802	19.6
1001st row insertion (including the B-tree into the B^*-tree rebuilding)	50	50
Next 499 rows insertion	9168	18.4
Last 500 rows insertion	8933	17.9

Operation on the table	Total execution time (ms)	Mean execution time per row (ms)
First 500 rows deletion	9888	19.8
Next 500 rows deletion (including the B*-tree into the B^{*+}-tree rebuilding)	9974	19.9
Next 500 rows deletion	9784	19.6
Last 500 rows deletion (including the B^{*+}-tree into the B$^+$-tree rebuilding)	9201	18.4
1000 rows insertion	18113	18.1
Next 4800 rows insertion (including the B$^+$-tree into the B^{*+}-tree rebuilding)	86465	18

According to the data in the Table III, the key insertion on the B*-tree is faster than on the B-tree as expected. The key deletion on the B$^+$-tree and B^{*+}-tree is faster than on the B-tree and B*-tree respectively. Also, the key insertion on the B^{*+}-tree is slightly faster than on the B$^+$-tree.

VII. 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 best index selecting is developed and implemented and the experiment is conducted using the developed in this work SQLite extension.

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 SQLite B-tree modifications extension can be used by all the developers who use this DBMS.

Acknowledgment

This work is supported by RFBR according to the Research project No. 18-37-00438 “mol_a”.

References

[1] J. Manyika *et al.*, “Big data: The next frontier for innovation, competition, and productivity,” McKinsey Global Institute, May 2011. Accessed: Jan. 20, 2019. [Online]. Available: 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

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

[3] K. Pollari-Malmi. (2010). B$^+$-trees [PDF paper]. Available: https://www.cs.helsinki.fi/u/mluukkai/tirak2010/B-tree.pdf

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

[5] A. Rigin, “On the Performance of Multiway Trees in the Problem of Structured Data Indexing,” (in Russian), coursework, Dept. Soft. Eng., HSE, Moscow, Russia, 2018.

[6] “SQLite Home Page.” SQLite.org. Available: https://www.sqlite.org/ (accessed Jan. 20, 2019).

[7] “SQLiteStudio”. SQLiteStudio. Available: https://sqlitestudio.pl/ (accessed: Jan. 26, 2019).