

Investigation of CW-Tree for Parallel Execution of Queries to Database

Vladislav S. Shevskiy and Yulia A. Shichkina(✉)

Saint Petersburg Electrotechnical University "LETI", Professor Popov Street, 5, St. Petersburg 197022, Russia

Abstract. The efficiency of using the developed CW-tree data structure in comparison with the B+-tree were analyzed in this article. B+-tree is used in the popular MySQL relational database management system. It has been experimentally proven that B+-tree is not efficient for parallel data retrieval. The study of parallelizing queries with different numbers of threads showed that with an increase in the number of threads, the search speed becomes higher. However, when the number of threads is $\geq$4, the speed stopped changing. That means that, after four threads, there was no point in increasing the number of threads. For testing the CW-tree a separate physical drive connected via PCI-Express interface were used. The drive is INTEL MEMPEK1W016GA, it has volume of 13.41 gigabytes, its logical sector size is 512 bytes and physical sector size is 512 bytes. A database was created, filled in according to the data structure CW-tree on this intel drive. For the analysis of the CW-tree, 6 search queries were developed with different amounts of returned data. The experiment showed that executing these queries in parallel mode is faster for CW-tree than in B+-tree executing the same queries in MySQL, where B+-tree is used to index data.

Keywords: CW-tree · Database indexing algorithms · Parallel computing · Database management system

1 Introduction

1.1 Relevance of Research

In the modern world, the amount of stored data is growing very rapidly. This is happening against the background of digitalization in all areas of human life, from the financial sector to medicine. The number of mobile applications and electronic services is growing, which, based on the analysis of large amounts of data, make fore-casts and recommendations. In such analysis, fast data retrieval is especially important. Data retrieval makes a significant contribution to the efficiency of the entire system.

Physical storage systems adapt to the fast processing of big data, but this happens extremely slowly, so hardware remains a natural limiter of the acceleration of working with data. On the other hand, software in the form of a DBMS is programmatic and algorithmic support and can be improved much faster, especially when using parallel computations.

© Springer Nature Switzerland AG 2021
O. Gervasi et al. (Eds.): ICCSA 2021, LNCS 12956, pp. 324–335, 2021.
https://doi.org/10.1007/978-3-030-87010-2_23

However, not all modern DBMSs can be effectively parallelized. A study of the literature has shown that such a feature of the DBMS takes place due to the data structures used, which cannot be parallelized.

In early works [2, 3], the authors of this article proposed a new data structure CW-tree (Constantly Wide tree), developed by the authors in such a way that allows efficient use of parallelization.

This study compares the CW-tree and the most widespread in the DBMS B+-tree.

1.2 What is the CW-Tree

In [2] and [3], a description of the CW-tree algorithm is presented. CW-tree is a data structure that allows performing asynchronous tree traversal using not only the root, but also additional starting vertices. The main purpose of using the CW-tree algorithm is to speed up the search for data in the CW tree by using additional threads of the operating system that perform searches asynchronously starting from additional starting vertices.

1.3 What is the B+-Tree

B+-tree is a data structure containing nodes that store indexes as well as edges that define links between nodes. The traverse in B+-tree starts from the root vertex and continues through the nodes until the vertex with the target index is found.

Originally, the B+-tree structure was intended to store data for efficient retrieval in a block-oriented storage environment in particular for file systems; the application is due to the fact that, unlike binary search trees, B+-trees have a very high branching ratio (the number of pointers from the parent node to the children, usually of the order of 100 or more), which reduces the number of I/O operations that require searching for an element in the tree.

2 Analysis of CW-Tree

2.1 Disadvantages of B+-Tree in Parallel Mode

An important problem with the widely used B+-tree data indexing algorithm is the difficulty of using operating system threads while traversing a tree to find data.

The disadvantage of the search process in B+ is shown in Fig. 1 below, where B+-tree is displayed as an approximate one, each node of which has one index.

If it is necessary to find the vertices J, K, L and H in the tree that shown in Fig. 1, then all the paths composed of transitions between the vertices in the process of walking through the tree will intersect at the vertices A and B. And if we compare the tree passes in searches for the vertices K and L, then you can see that both paths inter-sect at the vertices A, B, E, K. It follows that if you construct an asynchronous search for the vertices J, K, L and H, that is, construct the traversal process in this way, so that the search for each of the given vertices occurs independently, then at the intersection of the vertices the threads will do unnecessary work, especially when comparing the threads searching for the vertices K and L.

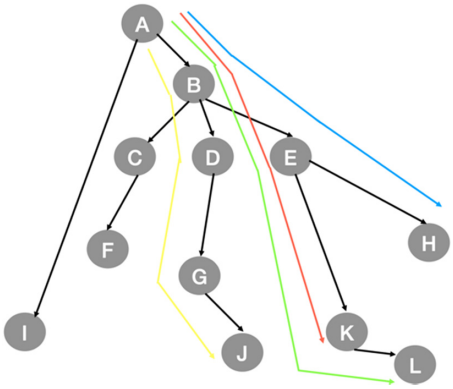

Fig. 1. An example of tree traversal when searching for different vertices

If it is necessary to find one vertex F in the tree shown in Fig. 1 and if you try to build an asynchronous search with four executing threads, then each of the four threads will do extra work due to the fact that each of the threads will perform the same path which contains passes A → B, B → C and C → F. This path follows from the property of B+-tree. Also, the root of the problem of forming a single path to find the required vertex in the tree is that in the tree there is only one starting point for searching for any vertex that is the root vertex.

In [1], an investigation of the execution time of various queries in the MySQL database, which uses indexing according to the rule of the B+-tree structure, was performed. Among the queries used for testing, various data read queries using the SELECT statement were designed. As well as subqueries of the original queries using the SELECT statement, which are called parallelized queries and divided according to the principles of data separation and task separation. As shown by the results of work [1], parallelized queries were executed faster than the original queries.

The database, which was used in testing in [1], was also used to identify the dependence of the query execution time on the amount of data read and the number of threads used within one task. For this, the following steps were performed:

1. A set of distributed queries has been formed in such a way that each one returns 1000000, 500000, 250000, 125000 first records from a table and uses 1, 2, … 10 threads;
2. Separate query executed;
3. Stored the result into a LinkedList object in the Java programming language implementation;
4. The total time of execution and saving of the result has been calculated. The result is presented in Table 1.

It is necessary to compare the execution time of queries using 1 thread with queries using more than one thread to see if there is an advantage to using parallelized queries in MySql.

Table 1. Dependence of execution time on the number of threads and the amount of data in the database table

№	Number of threads	Number of records in the table	Total thread execution time
1	1	1000000	1478
2	1	500000	507
3	1	250000	252
4	1	125000	140
5	2	1000000	1155
6	2	500000	345
7	2	250000	176
8	2	125000	90
9	3	1000000	1075
10	3	500000	313
11	3	250000	156
12	3	125000	81
13	4	1000000	1040
14	4	500000	299
15	4	250000	163
16	4	125000	84
17	5	1000000	1021
18	5	500000	295
19	5	250000	146
20	5	125000	74
21	6	1000000	995
22	6	500000	278
23	6	250000	145
24	6	125000	85
25	7	1000000	1017
26	7	500000	287
27	7	250000	145
28	7	125000	67
29	8	1000000	1024
30	8	500000	307
31	8	250000	145
32	8	125000	109

(*continued*)

Table 1. (*continued*)

№	Number of threads	Number of records in the table	Total thread execution time
33	9	1000000	1003
34	9	500000	264
35	9	250000	150
36	9	125000	79
37	10	1000000	1020
38	10	500000	295
39	10	250000	187
40	10	125000	73

The comparison was made by calculating the difference in execution time using the formula:

$$\Delta t = t_oneThread - t_nThreads \quad (1)$$

where Δt is acceleration of query processing when using n threads; *t_oneThread* is the execution time of a query that returns the specified number of data records and uses one thread; *t_nThreads* is the execution time of a query that returns the specified amount of data and uses the specified number of threads. The result is presented in Table 2.

Table 2. Taking into account the difference in execution time of a parallelized query with different data volumes compared to single-threaded queries

№	Number of threads	Number of records in the table	Time when using n threads	Time when using 1 thread	Δt
1	1	1000000	1478	1478	0
2	1	500000	507	507	0
3	1	250000	252	252	0
4	1	125000	140	140	0
5	2	1000000	1155	1478	323
6	2	500000	345	507	162
7	2	250000	176	252	76
8	2	125000	90	140	50
9	3	1000000	1075	1478	403
10	3	500000	313	507	194
11	3	250000	156	252	96

(*continued*)

Table 2. (*continued*)

№	Number of threads	Number of records in the table	Time when using n threads	Time when using 1 thread	Δt
12	3	125000	81	140	59
13	4	1000000	1040	1478	438
14	4	500000	299	507	208
15	4	250000	163	252	89
16	4	125000	84	140	56
17	5	1000000	1021	1478	457
18	5	500000	295	507	212
19	5	250000	146	252	106
20	5	125000	74	140	66
21	6	1000000	995	1478	483
22	6	500000	278	507	229
23	6	250000	145	252	107
24	6	125000	85	140	55
25	7	1000000	1017	1478	461
26	7	500000	287	507	220
27	7	250000	145	252	107
28	7	125000	67	140	73
29	8	1000000	1024	1478	454
30	8	500000	307	507	200
31	8	250000	145	252	107
32	8	125000	109	140	31
33	9	1000000	1003	1478	475
34	9	500000	264	507	243
35	9	250000	150	252	102
36	9	125000	79	140	61
37	10	1000000	1020	1478	458
38	10	500000	295	507	212
39	10	250000	187	252	65
40	10	125000	73	140	67

The result of comparing the difference in execution time Δt between queries by the criterion of the amount of data returned is shown in the diagrams below (see Figs. 2, 3, 4, and 5).

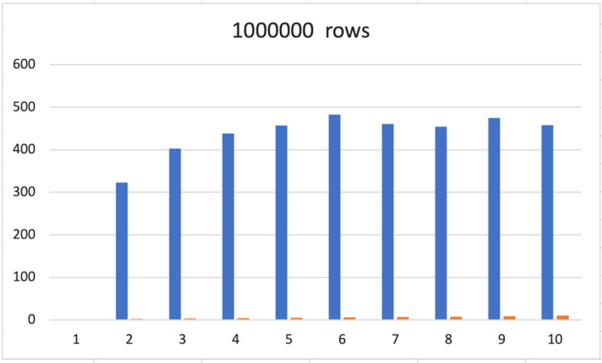

Fig. 2. Comparison of the execution time of a read query for 1000000 records based on the number of threads used during the execution of the query

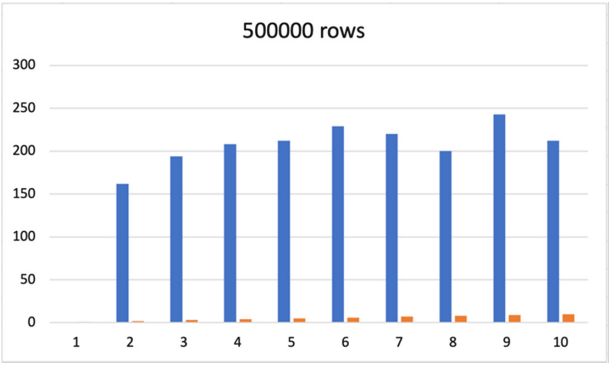

Fig. 3. Comparison of the execution time of a query to read 500000 records based on the number of threads used during the execution of the query

In the pictures (see Figs. 2, 3, 4, and 5) the time scale of queries execution in milliseconds vertically, and the number of threads used during the execution of queries horizontally shown.

The result of the experiment shown in the figures (see Figs. 2, 3, 4, and 5) shows that a parallelized query with more than 1 thread is executed faster than with 1 thread, however, as the number of threads used increases, there is no tendency to decrease the query execution time.

2.2 CW-Tree Data Structure

The CW-tree structure is designed to traverse the tree asynchronously using additional starting vertices from which the traversal will start. For such a passage to be possible, it is necessary that the CW-tree structure meets the following criteria:

– the structure should consist of two levels: the branch level, which stores the vertex indices, as well as the leaf level, which stores the data;

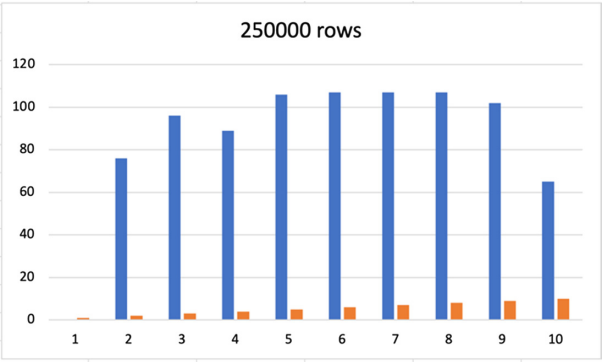

Fig. 4. Comparison of the execution time of a read query for 250000 records based on the number of threads used during the execution of the query

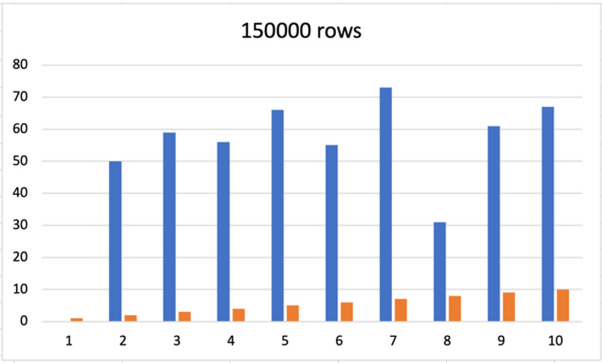

Fig. 5. Comparison of the execution time of a query to read 150000 records by the criterion of the number of threads used during the execution of the query

– adjacent vertices, which are at the same level, must be connected with each other by an edge in order to enable transition between them in the process of walking through the tree;
– in addition to the root vertex, the structure must contain additional starting vertices, from which the search for the vertex will be started.

Also, in order to carry out a parallel traversal through the CW-tree, it is necessary to determine the algorithms:

– initialization of starting vertices;
– bypassing the CW-tree.

In [3], detailed description of the mentioned algorithms, as well as a description of the features of the CW-tree data structure is presented.

Additional start vertices are initialized when data is added to the structure. In [3], a sequence of steps is described, as a result of the execution of which additional start vertices are initialized in the structure. During the execution of the data, a pass through the CW-tree structure occurs and each vertex is checked using the formula described in [3]. An example of a structure with initialized start vertices with indexes 6, 18 and 29, as well as a root node with index 16, is shown below (see Fig. 6). In addition to additional starting vertices, it is important to note that the structure contains connections between adjacent vertices that are at the same level. The sequence of steps, as a result of which the CW-tree structure has connections between neighboring vertices, is carried out at the stage of adding data to the structure and is also described in [3].

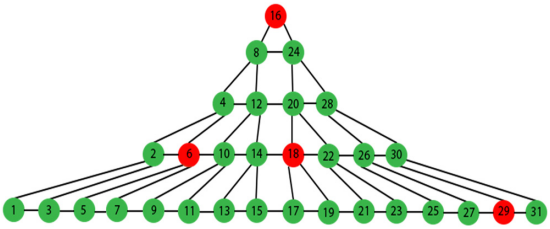

Fig. 6. Example CW-tree with initialized additional start vertices

After initialization of the starting vertices, it becomes possible to traverse the CW-tree. To traverse the tree, at each vertex, it is necessary to store not only its index, but also additional information about the neighboring vertices, including: the index of the vertex adjacent to the left, the index of the vertex neighboring on the right, the address in the permanent memory of the neighboring vertex on the left, the address in the permanent memory of the vertex adjacent to the right, etc. In [3], a complete list of information stored in each vertex is described. Thus, in the process of traversing the structure, at each step of the transition from one vertex to another, a choice is made - which vertex to go to due to the information that is known in the current vertex about neighboring vertices. The function of determining the next vertex to which the passage will be carried out within one iteration is called move. Its definition is given in [3].

An example of finding the vertex with index 1 in the CW-tree (see Fig. 6) is presented below (see Fig. 7).

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

Fig. 7. Example of finding the vertex with index 1 in a CW-tree using additional start vertices

In the example of finding a vertex with index 1 (see Fig. 7), the first column shows the vertices from which the search was started, and the remaining columns, in order from

left to right, show the vertices to which the transition was made during the search for the vertex with index 1. Thus, the search for vertex 1 from the additional starting vertex with index 6 was performed in fewer disk operations compared to the search for vertex 1 from the root vertex with index 16. As a result, the search for vertex 1 from vertex 6 will be faster than finding vertex 1 from vertex 16.

2.3 CW-Tree Application

To determine the effectiveness of the algorithm, we conducted comparative testing of query execution in the CW-tree and in the database created using the MySQL DBMS. During testing, we used two databases with the same database schema as in [1].

The number of data records in each database was equal to 500,000. The INTEL MEMPEK1W016GA device was used as a separate physical drive (volume is 13.41 gigabytes, logical sector size is 512 bytes, physical sector size is 512 bytes). The device was connected via built-in PCI-Express interface. In a database that used CW-tree indexing, 3 additional start nodes were used, so one query was executed by 4 threads of execution, one of which started searching from the root node, and the other 3 from additional start nodes. Query groups were developed, where a special parameter X was introduced which is the amount of data returned as a result. With this parameter, you can adjust the total amount of data returned. The groups of queries used in the testing process are presented in Table 3 below.

Table 3. Description of queries for testing

№	Content of the query
Query #1	select * from Table1 where A1 < X
Query #2	select * from Table1 inner join Table2 on A1 = A2 and A1 < X and A2 < X
Query #3	select * from Table1 inner join Table2 inner join Table3 on A1 = A2 and A2 = A3 and A1 < X and A2 < X and A3 < X
Query #4	select * from Table1 inner Table2 inner join Table3 inner join Table4 on A1 = A2 and A2 = A3 and A3 = D4 and A1 < X and A2 < X and A3 < X and D4 < X
Query #5	select * from Table1 where A1 < X and A1 in (select A2 from Table2 where A2 < X)
Query #6	select A1 from Table1 where A1 < X union all select A2 from Table2 where A2 < X

In Table 3, the parameter X, which indicates the amount of data returned, can be determined by the recursive formula:

$$X = \{xi = xi - 1 + \Delta, i = \{1, \ldots, 10\}, x0 = 0, \Delta = 50000, x10 = 500000\}.$$

The results of testing the execution time of the queries shown in Table 3 are presented in Figs. 2, 3, 4, 5, and 6 below.

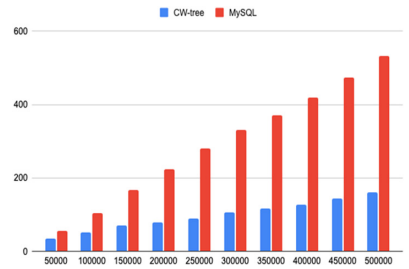

Fig. 8. Query #1 research results

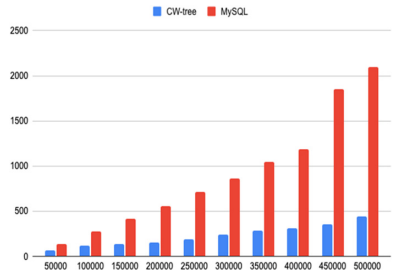

Fig. 9. Query #2 research results

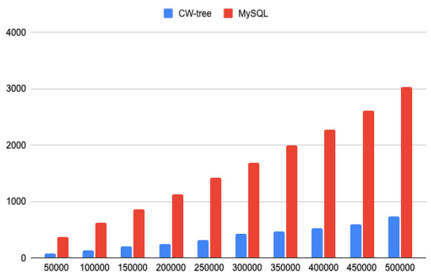

Fig. 10. Query #4 research results

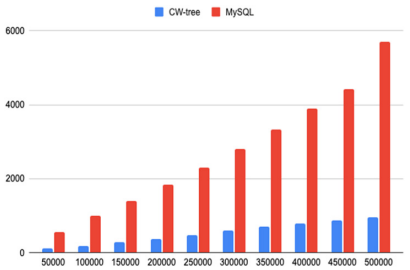

Fig. 11. Query #4 research results

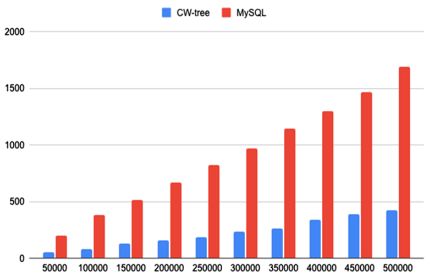

Fig. 12. Query #5 research results

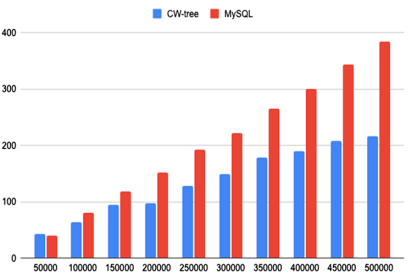

Fig. 13. Query #6 research results

In the figures (see Figs. 8, 9, 10, 11, 12, and 13), the vertical scale of the query execution time in milliseconds is displayed, the horizontal is the X parameter, described above and indicating the amount of data returned as a result of queries.

As the results in Figs. 8, 9, 10, 11, 12, and 13 show, that the execution time of queries in the CW-tree is significantly less than the execution time of the corresponding queries in MySQL, which confirms the effectiveness of using the CW-tree when performing parallel data retrieval.

3 Conclusion

The results of investigation show that one query is executed simultaneously against the MySQL databases and the CW-tree, and all six are executed faster in the CW-tree database. The fact that all queries run faster in the CW-tree proves the effectiveness of the CW-tree method when processing various SQL queries, especially those that can be divided into independent parts. Thus, CW-tree with parallel search capability is much faster than analogs.

Acknowledgements. The reported study was funded by RFBR according to the research project № 19-07-00784.

References

1. Shichkina, Y., Kupriyanov, M., Shevsky, V.: The application of graph theory and adjacency lists to create parallel queries to relational databases. In: Galinina, O., Andreev, S., Balandin, S., Koucheryavy, Y. (eds.) NEW2AN/ruSMART -2018. LNCS, vol. 11118, pp. 138–149. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01168-0_13
2. Shevskiy, V.: Constantly wide tree for parallel processing. In: Conference 2019, ISTMC, Yalta, Crimea, pp. 285–292 (2019). http://ceur-ws.org/Vol-2522/paper28.pdf. Accessed 13 Jan 2021
3. Shevskiy, V.: Indexing data based on the CW-tree algorithm using parallel data reading. In: Conference 2020, MECO, pp. 1–6 (2020). https://doi.org/10.1109/MECO49872.2020.9134263
4. Graefe, G., Kuno, H.: Modern B-tree techniques. In: 2011, IEEE 27th International Conference on Data Engineering, pp. 1370–1373 (2011)
5. Bayer, R., McCreight, E.: Organization and maintenance of large ordered indexes. In: Broy, M., Denert, E. (eds.) Software Pioneers, pp. 245–262. Springer, Heidelberg (2002). https://doi.org/10.1007/978-3-642-59412-0_15
6. Comer, D.: Ubiquitous B-tree. ACM Comput. Surveys **11**(2), 121–137 (1979)