Constantly Wide Tree for Parallel Processing*

Vladislav Shevskiy[0000-0001-5121-201X]

Saint Petersburg Electrotechnical University "LETI", Saint Petersburg 197376, Russia
lncs@springer.com

Abstract. Improving the performance of modern computing devices requires the adaptation of data processing algorithms for the efficient use of hardware resources. This article provides comparisons of existing database management systems and shows their disadvantages when used in conjunction with parallel processing algorithms. An approach to data storage is presented that can be used to optimize the execution of database queries using parallel processing. A method for constructing the proposed data structuring storage algorithm is described. At the same time, the advantages of the algorithm and the ways of interaction with it to achieve greater benefits in comparison with the existing analogs are shown.

Keywords: Database; Tree algorithm; Parallel computing; Data reading.

1 Introduction

With increasing productivity of modern computing systems [6], it is necessary to adapt the software in such a way as to make optimal use of the available computing resources. Modern high-performance systems, including database management systems (DBMS), use the interfaces for controlling the threads and cores of the working CPU.

In the article [19] there was considered the process of paralleling queries and executing them on MySQL DBMS. The authors have established the fact that MySQL on the InnoDB engine does not work well with processing parallel queries and the use of parallelized queries is inconsequential compared to simple ones.

In this article, for more efficient reading of data from the database, we propose the use of a modified tree algorithm.

2 Problems using tree algorithms

The most popular algorithm currently used by the DBMS is B + -tree. It belongs to the B-tree group, which are balanced trees. The main advantage of using B + tree in a DBMS is the method of data storage - all data is stored in the leaves of the tree, which

---

can be linked to each other. This allows storing data in adjacent blocks, which reduces the chances of missing a cache in the process of paging data from the main memory. Because of the fact that the number of reading operations from a disk decreases, the overall data reading speed will be increased.

However, despite the increase in data reading performance when using B + -tree, the algorithms for their application are not optimal when processing a large number of database queries and when using multi-thread processing of query data[5]. This is because each thread will pass through the upper levels of the tree, and therefore will make many requests to the tree and do extra work. This problem is illustrated in Figure 1:

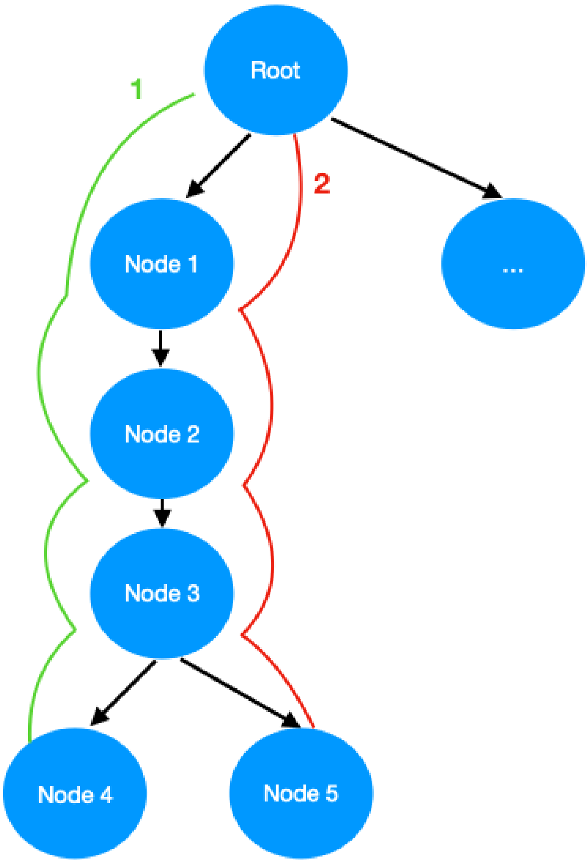

Fig. 1. Processing queries in a database whose data storage is based on B + tree.

Figure 1 shows the processing of two queries for data on a fragment of the B + tree. The first query reads data from node 4, and the second from node 5. As can be seen in the figure, for the successful execution of each of the query processes it is necessary that both must go through node 1, node 2 and node 3, and only after that, each of them reaches the goal.

Another type of tree that is used for parallel processing is the k-d tree. Its main advantage is the effective search for keys in a given range. Such a tree is useful to use in databases designed to store geometric data, for example, coordinates on a map. The authors of the article [1] created a STIG tree (Spatio-Temporal Indexing using GPUs) based on the k-d tree. This algorithm was designed to optimally process queries on the input spatial data. To make this possible, the authors included in the STIG physical storage of the keys of the tree according to the basic principles of the k-d tree, and the actual records are stored separately in the form of data blocks that will be processed by the cores of the graphical process. The results presented in the article [1] show the efficiency of the algorithm. However, the possibilities of using such a tree are severely limited. It may be useful in solving the problem of creating a database that stores spatial data; however, a more unified solution is required in commercial development. Thus, despite the obvious advantages of the STIG algorithm, it is not suitable for use in a data management system used in solving business problems.

3 Proposed cw-tree algorithm

The basic idea of creating a tree is that the tree should be distributed according to the number of physical cores available on the machine on which the DBMS is to be installed.

The tree must comply with the rules:

- it must have a main vertex or root of a tree;
- the tree consists of subtrees, which we will call child trees, each of which has its own root, which we will call child roots;
- the root has connections with the child roots, each of which represents a root for its part of the tree;
- the number of daughter roots must be equal to the number of physical cores of the involved processor for more efficient use of resources;
- each subtree is traversed according to the rules of that subtree. Thus, all subtrees may not necessarily be of the same type, but on the contrary, each subtree can be implemented in accordance with various algorithms, including various traversal algorithms of this subtree.

Creating a tree is done in accordance with the rules:

Before determining which of the subtrees should add a new element, it is necessary to find the necessary child root. For a uniform distribution, the direction of passage from the child root with index 0 to the child root with index N alternates with the opposite direction. At each iteration, the following algorithm is executed:

```
start {
if(one_of_child_trees_ is_empty) {
if (current_ child_tree_ is_empty) {
current_child_tree.Add(new_element);
sort(all_nodes);
}
else {
go_to_next_in_order_child_root(); } }
else if (more_than_one_of_child_trees_is_empty) {
If (current_ child_tree_ is_empty) {
current_child_tree.Add(new_element); } }
else {
If (current_direction_ equals_B) {
function_1();
}
else if (current_direction_ equals_A) {
function_4(); } }
}
function_1 {
If (new_element < maximum_of_current_node && new_element
>minimum_of_current_node) {
current_child_tree.Add(new_element); }
function_2();
}
function_2 {
If (new_element < minimum_current_subtree AND new_element
> maximum_of_next_subtree) {
next_child_tree.Add(new_element);
from узел_n to узел_0 {
function_3 (node_i); }
}
else {
go_to_next_in_order_child_root(); } }
function_3 {
If (size_of_current_subtree >
Size_of_next_in_order_subtree) {
next_child_tree.Add(minimum_of_current_subtree);
current_child_tree.Remove(minimum_of_current_subtree); }
}
function_4 {
If (new_element < maximum_of__current_subtree &&
new_element >Minimum_of_current_subtree) {
current_child_tree.Add(new_element); }
function_5 ();
}
function_5 {
if (new_element > maximum_of_current_subtree AND
new_element <minimum_of_next_subtree) {
next_child_tree.Add(new_element);
```

```
from node_0 to node_n {
function_6 (node_i); }
}
else {
go_to_next_in_order_child_root(); }
}
function_6{
if (size_of_current_subtree >
size_of_next_in_order_subtree) {
next_child_tree.Add(maximum_of_current_node); }
}                                                        (1)
```

The operation «Add» involves adding an element to the appropriate subtree according to the rules of that subtree. Performing the above described algorithm spends additional time at the stage of storing data in the tree, however, it allows to sort the data in the corresponding subtrees. This reduces the number of reads from the tree, and therefore increases the overall read performance of the tree.

The nodes should store the keys of each record without data. The target data set that the user expects to receive is to be stored in the leaves of the tree. This will minimize the number of reading operations from memory, thus speeding up queries. And also, it separates the logic of finding the target element in the tree and accessing the memory. Thus it is possible to use various computer resources for each operation. For example, in article [1], the authors applied an algorithm to process only the bottom level of the kd-tree tree, that is, database records using GPU tools.

Based on the presented description of the algorithm, the scheme of the resulting tree is shown (see Fig. 2).

Figure 2 shows an example of a created tree for working with data under the control of a processor that has 4 physical cores. The user configures the number of child roots during the creation of the database.

The root of the tree from which the DBMS starts processing queries has connections with the child roots. At the first stage of processing, it is necessary to make a choice in which of the child trees to search for the requested data. Its physical processors must process each of the child trees. If one of the cores cannot be used on the computer, it is possible to use threads. Next, there is a need to make a pass through the corresponding subtree. Since there is an abstraction on subtrees, any tree structure can be applied to its subtree. However, the proposed use of B + - the tree is because it is well adapted to the storage of data of various types, in contrast to, for example, from k-d trees.

In the article [2], the authors state that b-tree is suitable for parallel processing. In the article [2], the authors obtained experimentally that when using b-tree, the algorithm for traversing a tree wide (BFS) is more suitable for parallel computing, unlike the types of algorithms for traversing a tree in-depth (DFS). Based on the conclusions that the authors have made, it is expected that using the B-tree together using the tree-wide traversal algorithm is best suited for parallelized queries.

The database created on the implementation of the proposed tree algorithm will faster process queries for reading data on high-performance hardware. A special increase in performance will be noticeable for databases in which read operations are most often performed.

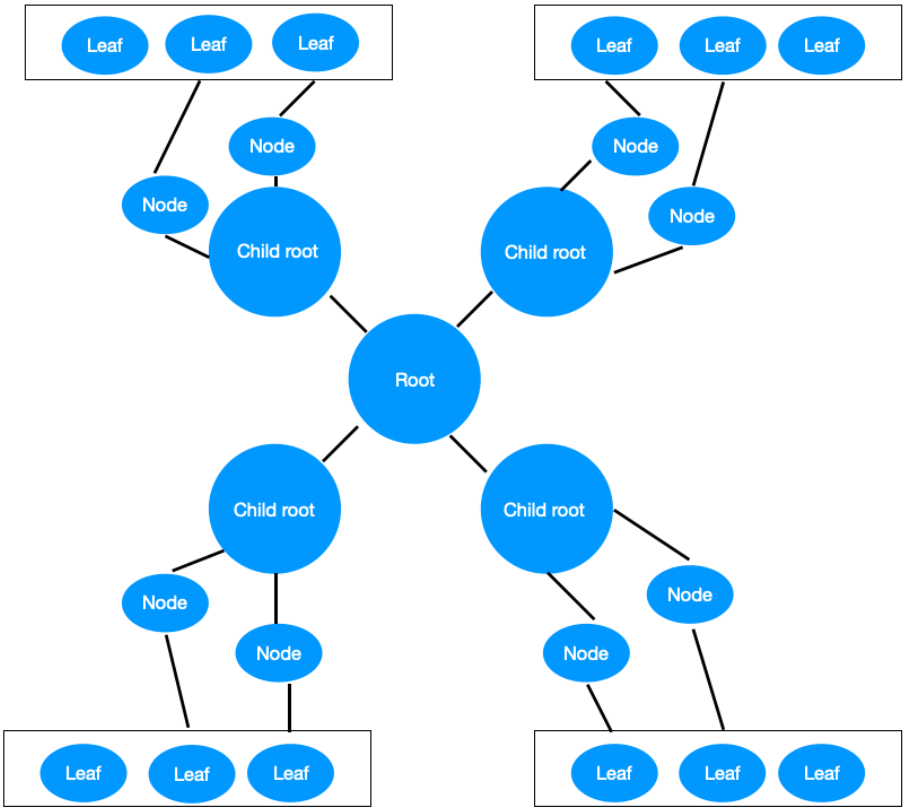

Fig. 2. The scheme of the resulting tree

In order to optimize database queries, it is possible to use different types of trees. However, even the most frequently used ones have their drawbacks, which consist of poor adaptation to parallel processing.

4 Conclusion

A tree algorithm has been developed that will efficiently process database queries using the processing power of the processor, which performs the search for keys and also the processing power of the graphics device that searches for data records using the received key.

Based on the proposed algorithm, it is planned to create a new database management system and create an experimental database in it in order to conduct an experiment.

As a result, temporary assessments and an assessment of the overall efficiency of the use of this tree will be obtained.

Acknowledgments

The research was funded by RFBR and CITMA according to the research project №18-57-34001

References

1. Anthony Fox, Chris Eichelberger, James Hughes, Skylar Lyon Commonwealth Computer Research; Spatio-temporal Indexing in Non-relational Distributed Databases; IEEE 2013, 978-1-4799-1293-3/13;
2. R.K. Ghosh; Parallel Search Algorithms for Graphs and Tree; Information sciences 67, 137- 165 1993, 0020-0255/93;
3. Rupak Biswas, Zhang Jiang, Kostya Kechezhi, Sergey Knysh, Salvatore Mandra, Bryan O'Gorman, Alejandro Perdomo-Ortiz, Andre Petukhov, John Realpe-Gomez, Eleanor Rieffel, Davide Venturelli, Fedir Vasko, Zhihui Wang; A NASA perspective on quantum computing: Opportunities and challenges; Parallel Computing, Volume 64, May 2017, pp 81-98, DOI 10.1016/j.parco.2016.11.002;
4. Weiming Lu, Yaoguang Wang, Jingyuan Juang, Jian Liu, Yapeng Shen, Baogang Wei; Hybrid storage architecture and efficient MapReduce processing for unstructured data; Parallel Computing, Volume 69, November 2017, pp.63-77, DOI 10.1016/j.parco.2017.08.008;
5. Peiquan Jin, Puyuan Yang, Lihua Yue; Optimizing B+-tree for hybrid storage systems; Distributed and Parallel Databases, Volume 33, September 2015, Issue 3, pp 449–475, DOI 10.1007/s10619-014-7157-7;
6. Qiong Luo, Jens Teubner; Special issue on data management on modern hardware; Distributed and Parallel Databases, 2015, Volume 33, pp.415–416, DOI 10.1007/s10619-014-7168-4;
7. Abdurrahman Yasar, Bugra Gedik, Hakan Ferhatosmanoglu; Distributed block formation and layout for disk-based management of large-scale graphs; Distributed and Parallel Databases, 2017, Volume 35, Number.1, pp.23-53, March 2017, DOI 10.1007/s10619-017-7191-3;
8. Daichi Amagata, Takashiro Hara, Shojiro Nishio; Sliding window top-k dominating query processing over distributed data streams; Distributed and Parallel Databases, December 2016, Volume 34, Issue 4, pp 535–566; DOI 10.1007/s10619-015-7187-9;
9. Research in Mobile Database Query Optimization and Processing, Agustinus Borgy Waluyo, Bala Srinivasan, and David Taniar, Mobile Information Systems, Volume 1 (2005), Issue 4, pp. 225-252
10. Spiliopoulou M., Hatzopoulos M., Ttanslation of SQL queries into a graph structure: query transformations and pre-optimization issues in a pipeline multiprocessor environment, Informafion Sysfems 1992, Vol. 17, No. 2, pp. 161-170. https://doi.org/10.1016/0306-4379(92)90010-K
11. Yao S.B. Optimization of query evaluation algorithms. ACM Trans. Database Syst. 4(2), 133-155 (1979). DOI: 10.1145/320071.320072

12. Mikkilineni K. P., Su S. Y. W. An evaluation of relational join algorithms in a pipelined query processing environment. IEEE Trans. Software Engng 14(6), 838-848 (1988). DOI: 10.1109/32.6162
13. Jarke M., Koch J. Query optimization in database systems. ACM Comput. Suru. 16(2), 111-152 (1984). DOI: 10.1145/356924.356928
14. Smith J. M., Chang P. Y. T. Optimizing the performance of a relational algebra database interface. CACM 18(10). 5688579 (1975). DOI: 10.1145/361020.361025
15. Sai Wu, Feng Li, Sharad Mehrotra, Beng Chin Ooi, Query Optimization for Massively Parallel Data Processing, Proceedings of the 2011 SoCC Conference, Oct. 2011. DOI: 10.1145/2038916.2038928
16. Lila Shnaiderman, Oded Shmueli, A Parallel Tree Pattern Query Processing Algorithm for Graph Databases using a GPGPU, Workshop Proceedings of the EDBT/ICDT 2015 Joint Conference (March 27, 2015, Brussels, Belgium) on CEUR-WS.org (ISSN 1613-0073)
17. Dex: High-performance and scalable graph database management system. http://www.sparsity-technologies.com/dex.
18. Shichkina, Y.A., Kupriyanov, M.S., Applying the list method to the transformation of parallel algorithms into account temporal characteristics of operations, Proceedings of the 19th International Conference on Soft Computing and Measurements, SCM 2016, 7519759, c. 292-295. DOI: 10.1109/SCM.2016.7519759
19. Article URL, http://2018.nscf.ru/TesisAll/05_Reshenie_zadach_optimizatsii/206_ShichkinaYA.pdf, last accessed 2019/04/14.
20. Julian Shun, Guy E. Blelloch; A Simple Parallel Cartesian Tree Algorithm and its Application to Parallel Suffix Tree Construction; Carnegie Mellon University; ACM Trans. Parallel Comput. 1, 1, Article 8 (September 2014); http://dx.doi.org/10.1145/2661653.
21. Zhila Nouri and Yi-Cheng Tu, GPU-Based Parallel Indexing for Concurrent Spatial Query Processing; University of South Florida; Proceedings of 30th International Conference on Scientific and Statistical Database Management; Bozen-Bolzano, Italy, July 9–11, 2018 (SSDBM ’18), 12 pages; https://doi.org/10.1145/3221269.3221296.
22. Giovanni Mariani, Andreea Anghel, Rik Jongerius, Gero Dittmann; Parallel Computing 66 (2017) 1–21; http://dx.doi.org/10.1016/j.parco.2017.04.006.
23. K. Amunts, A. Lindner, K. Zilles; The human brain project: neuroscience perspectives and German contributions, e-Neuroforum 5 (2) (2014) 43–50; doi:10.1007/s13295-014-0058-4.
24. A. Anghel, G. Rodríguez, B. Prisacari, C. Minkenberg, G. Dittmann; Quantifying communication in graph analytics, in: High Performance Computing - 30th International Conference, ISC High Performance 2015, Frankfurt, Germany, July 12–16, 2015, Proceedings, 2015a, pp. 472–487; DOI:10.1007/978- 3- 319- 20119- 1_33.
25. M. Charrad, N. Ghazzali, V. Boiteau, A. Niknafs, Nbclust; an r package for determining the relevant number of clusters in a data set, J Stat Softw 61 (1) (2014) 1–36; DOI:10.18637/jss.v061.i06.
26. G. Chetsa, L. Lefevre, J.-M. Pierson, P. Stolf, G. da Costa; A user friendly phase detection methodology for HPC systems’ analysis; Green Computing and Communications (GreenCom); 2013 IEEE and Internet of Things (iThings/CPSCom), IEEE International Conference on and IEEE Cyber, Physical and Social Computing, 2013, pp. 118–125; DOI:10.1109/GreenCom-iThings-CPSCom.2013.43.
27. L. Fiorin, E. Vermij, J. Van Lunteren, R. Jongerius, C. Hagleitner; An energy-efficient custom architecture for the SKA1-Low central signal processor; Proceedings of the 12th ACM International Conference on Computing Frontiers, in: CF ’15, ACM, New York, NY, USA, 2015, pp. 5:1–5:8; DOI:10.1145/ 2742854.2742855.