UDC 004.65

V. Nikitin, E. Krylov, Y. Kornaga, V. Anikin

COMBINED INDEXING METHOD IN NOSQL DATABASES

Abstract: Any system must process requests quickly. This is one of the main conditions for a successful system. Higher data processing rates come along with new technologies. An example is 5G technology, which allows data to be exchanged at speed of up to 100 Mbps for downloads and up to 50 Mbps for uploads. The operation of the database depends on many factors, including the characteristics of the server, the number of requests to the server and the requests themselves. Improperly worded queries can degrade the speed of the system in general. The situation can be corrected by indexing, which allows you to increase the speed of searching for information in the database itself.

Keywords: database, SQL, NoSQL, index, binary tree, hashing

Formulation of the problem

Any information system cannot be without a database. This automatically becomes one of the potential problem areas. Performance declines when queries are misused, active connections are exceeded, or too many data is used. If the system has a small number of users, it is possible to clean the database and / or optimize abstractions. If the system targets a large amount of data, it will not help. There are special objects called indexes to solve the problem of data retrieval. The process of creating an index is called indexing [1].

Currently, NoSQL databases are very common, so the ability to speed up the search for information remains relevant. Databases such as CoachDB and MongoDB have the ability to index documents. Without indexing, the listed databases must scan each document and select the ones that match the query. This process is very inefficient and requires a lot of data processing [2].

An overview of existing solutions

It is common in NoSQL databases that existing indexing methods are binary trees and hashing.

B-trees are balanced trees in which the time of standard operations is proportional to the height. The main difference is that they have been created specifically to work with disk memory. The structure aims to reduce the number of input and output operations. When constructing a B-tree, the factor t is used, which is called the minimum degree. There is a limitation on this indicator, which is that each node must have more than $t - 1$ and less than $2 \cdot t - 1$ keys [3].

© V. Nikitin, E. Krylov, Y. Kornaga, V. Anikin

The keys are stored in ascending order in each node, but more important property is the number of keys in the node which affects the range of descendant keys. Suppose that the set of node keys is *k[i]*, where *t* is $1 < I < 2 \cdot t - 1$, and the set of descendants is *c[i]*. Then for any key in the subtree with the root *c[i]* the following inequality holds:

$$[i-1] \leq c[i] \leq [i] \quad (1)$$

Another condition of the B-tree is the location of all the leaves on the same level, which is the height of the tree. It follows an important statement that the height of a tree with $n \geq 1$ and a minimum degree $t \geq 2$ does not exceed $log_t(n+1)$.

Searching in the B-tree is similar to searching in the binary tree, but the choice is possible not only from two options, but few more. The algorithm is as follows:

- We pass on keys of a root node while value is less / more than necessary;
- We pass to the descendant which is to the left / right;
- We go on the keys of the current node;
- If we find the necessary element, we stop. Otherwise, repeat the algorithm.

The complexity of the algorithm is $O(t \cdot log_t n)$, where t is the minimum degree. But it is worth noting that the number of disk operations is $O(log_t n)$.

Adding elements to a B-tree is more difficult because of the possible violation of properties. If the sheet node is not filled, the insert is simple. Otherwise, the sheet node is divided into two, each of them contains *t - 1* key. The middle element moves to the upper node. If the upper node is also full, then the same operations are performed for it. The complexity of the algorithm is also $O(t \cdot log_t n)$, and the number of disk operations $O(h)$, where h is the height of the tree.

Deleting an element in a B-tree is more difficult than adding elements. This is because each removal requires rebuilding the tree as a whole. The algorithm of the process of removing the key from the sheet node is as follows:

- If the node has more keys than $t-1$ after removal, then everything is fine. Just delete the item and that’s all;
- Otherwise, take a separate element in the upper node, and insert it into the node where the element is removed.
- The element from the second descendant passes to the upper node and becomes a separator.

If the element is removed from the upper node, the algorithm is as follows:

- Delete the item;

 ISSN 1560-8956

- Take the first element from the right node of the descendant, for which the deleted element is a separator.

The complexity of the algorithm is also $O(t \cdot log_t n)$, and the number of disk operations $O(h)$, where h is the height of the tree.

The main disadvantage of B-trees is the lack of effective means of data sampling (i.e. the method of traversing the tree), arranged differently from the selected key.

Database hashing is used as a way to directly find data on disk without using a typical index structure. It is used to index and retrieve data from a database by a short hash value, because it is faster than searching by value. The data is stored in the form of blocks, the addresses of which are generated using a hash function. The address of the data block is used as an argument.

The hashing method is used in the following cases:

- The hashing method is used to index and retrieve items in the database because it is faster to search for that particular item using a shorter hashed key instead of using its original value;
- Hashing is an ideal method for calculating the direct location of data written to disk without using an index structure.

There are two types of hashing methods:

- In static hashing, the resulting address of the data segment remains unchanged. This means that using the same argument for the hash function, we will get the same address. Therefore, the number of segments in memory always remains constant [4].

  When we need to insert a new record, we can create an address using its hash key. When the address is generated, the record is automatically saved in that location.

  When reading information, the hash function is useful for obtaining the address of the segment where the data should be stored.

  In its turn, static hashing is divided into open and closed.

  — Open hashing does not overwrite old information, but uses the following block for data. This method is also called linear sounding.

  — In the closed hashing method, when segments are filled, a new segment is allocated for the same hash, and the result is bound to the previous one.
- Dynamic hashing offers a mechanism by which data segments are added and removed dynamically and on demand. In this hashing, the hash function helps to create a large number of values [4].

The disadvantage of hashing is collisions. A collision is a condition in which the resulting hashes of two or more data in a set incorrectly represent the same place in the hash table [5].

A T-tree is a balanced index tree data structure optimized for when both index and actual data are fully stored in memory, just as a B-tree is an index structure optimized for storage on block-oriented secondary storage devices such as hard drives. T-trees seek to benefit from the performance of wood structures in memory, such as AVL trees, while avoiding the large loads they share.

T-trees do not store copies of indexed data fields in the nodes of the index tree. Instead, they take advantage of the fact that the actual data is always in main memory along with the index, so they simply contain pointers to the actual data fields. "T" in the T-tree refers to the shape of the node data structures in the original work, which first has described this type of index [6].

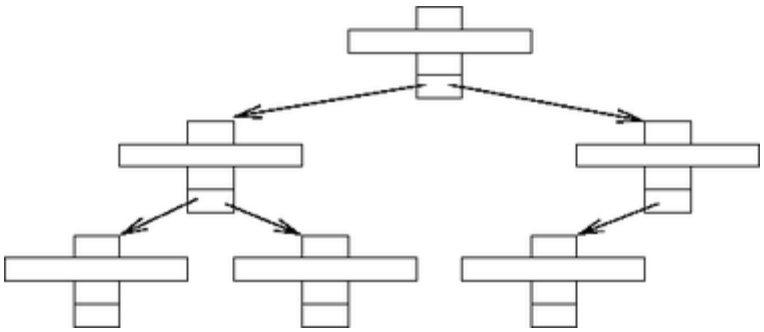

Figure 1. T-tree nodes distribution

A T-node typically consists of pointers to the top node, a left and right child node, an ordered array of data pointers, and some additional control data. For each internal node, there are nodes of the leaf or half of the leaf that contain the predecessor of the smallest data value (called the largest lower bound) and the one that contains the successor of its largest data value. Nodes can contain any number of data elements from one to the maximum size of the data array. Internal nodes keep busy between a predetermined minimum and maximum number of elements.

The main disadvantage is the amount of memory required. The cost of memory to store even a single index that includes actual values can exceed dozens or even hundreds of terabytes.

The O_2 tree is basically a red-black binary search tree in which leaf nodes are formed into index blocks or fragments that store recordings of "key-value, record pointer" pairs. Internal nodes contain copies of only the key-value values of the middle pairs "key-value, record-pointer", which separate the blocks of sheet nodes when they are filled.

 ISSN 1560-8956

These internal nodes are formed into a simple binary search tree, which is balanced using Red-Ebony rotation algorithms. The binary mahogany algorithm is less complex than the AVL tree algorithm, which has more stringent balancing conditions [7].

The O_2 tree is associated with the order of the tree denoted by m. The order is the maximum number of "key-value, write pointer" pairs that a sheet node can contain. Data is stored in sheet nodes; internal nodes are simply binary space holders that facilitate or direct the bypass of the tree to the leaf node. All successful or unsuccessful searches always end at the leaf node. This is similar to the B+-Tree search process, except that internal nodes now contain only individual key values as opposed to m key values.

Searching for an O_2 tree is similar to searching for a red-and-black binary tree. However, the internal red-black tree structure serves as space holders to find the actual key in the leaf node. Unlike T-Tree and B + -Tree, the search is performed with only one key comparison in the internal node. T-Tree and B+ -Tree perform an average of $\mathrm{m}/2$ comparisons before continuing the search.

The task of the Top-Down algorithm is to ensure that when a new internal node is inserted, there is no need to move up the tree again. In this way, it ensures that when a new internal node is inserted, its parent element will not be red.

The expected use of the drive, due to the fact that it increases and decreases according to the splitting and merging of the blocks, is $O(ln\,2)$. The memory usage indicated is the total number of stored keys divided by the total capacity of all nodes.

The O_2 tree has the disadvantage that it does not have asynchronous permanent storage.

Conclusion

Thus, indexing is a popular tool to speed up data retrieval. It is available in both relational and non-relational databases. In both cases, similar algorithms are used. The most popular indexing methods are B-Tree, T-Tree, O_2-Tree and hashing. Each of these methods has significant disadvantages. The main disadvantage is the excessive consumption of memory allocated for the index. To solve this problem, it is possible to combine tree structure and hashing to obtain the most optimal and reliable structure.

This data structure is extended hashing. The basic idea is that the hash table is represented as a directory, and each cell will point to a container with a certain capacity. The hash table itself will have a global depth, and each of the capacities has a local depth. The global depth shows how many last bits will be used to determine the capacity in which you want to store the value. From the difference between the local

and global depth, you can understand how many cells refer to the capacity [8]. This can be calculated using the formula:

$$K = 2^{G-L} \tag{2}$$

where G – global depth, L – local depth, K – cell amount which has references.
The algorithm itself:

1) Convert the value to binary. Based on the last G bits, we decide in which capacity to send the value;
2) If the container has a free space, then add this value. We look at the local depth in the opposite case.
3) If the local depth is less than the global depth, then there is an overlap with division into two parts. The value of the local depth is increased by one and the values are entered;
4) If the local depth is equal to the global one, then we increase the global depth by 1, doubling the number of cells, the number of pointers on the tank, and increase the number of last bits by which we distribute the value. Then the local depth of the overflowed tank becomes smaller and we repeat the previous algorithm, that is, we mix the capacity, dividing it into two containers, and so on.

Having implemented B-tree data structures and extended hashing using the Java programming language, the following results were obtained in table.1 and table.2.

Table 1. **Processing time of search and insert operations in the B-tree**

Data amount, pieces/operation	Insertion, s	Searching, s
300 000	3	1
500 000	3	2
1 000 000	5	6
1 200 000	15	7

Randomly generated rows were used as test data and entered into the instances of the structures. Also, it should be noted that the study was conducted using only RAM. In the future it is planned to apply the algorithm for hard media and make comparisons.

Using a combined indexing method can significantly speed up the insertion and search operations. For 1 million test values, we see that the insert operation is 66.67% more efficient when using extended hashing, and 500% more efficient when searching. With 1 million and 200 thousand test data, we see a significant difference. The insert operation is 275% more efficient when using extended hashing, and 600% more efficient when searching.

 ISSN 1560-8956

Table 2. **Processing time of search and insert operations in the structure of extendible hashing**

Data amount, pieces /operation	Insertion, s	Searching, s
300 000	0	0
500 000	1	0
1 000 000	3	1
1 200 000	4	1

REFERENCES

1. Корнага Я. І. Методи моніторингу подій та обробки запитів в гетерогенних розподілених базах даних на основі векторно-матричних операцій :дис. канд техн. наук: 05.13.06 / Держ. ун-т телекомунікацій. - Київ, 2015.

2. Redmond E. Seven Databases in Seven Weeks, 2nd Edition – 358 p., Pragmatic Bookshelf, April, 2018. ISBN-13: 9781680502534.

3. Grd P., Baca M.. Analysis of B-tree data structure and its usage in computer forensics. URL: https://www.researchgate.net/publication/210381551_Analysis_of_B-tree_data_structure_and_its_usage_in_computer_forensics.

4. Sumit Thakur. Hashing Algorithm And Its Techniques In DBMS. URL: https://whatisdbms.com/hashing-algorithm-and-its-techniques-in-dbms/.

5. MongoDB: Hashed Indexes. URL: https://docs.mongodb.com/manual/core/index-hashed/.

6. Hongjun Lu, Yuet Yeung Ng, Zengping Tia. T-Tree or B-Tree: Main Memory Database Index Structure Revisited // 2000 IEEE Proceedings 11th Australasian Database Conference. ADC 2000 (Cat. No.PR00528). – IEEE, 2000.

7. Ohene-Kwofie D., J.Otoo E., Nimako G.. O2-Tree: A Fast Memory Resident Index for In-Memory Databases // 2012 IACSIT International Conference on Information and Knowledge Management (ICIKM 2012). - IACSIT, 2012.

8. Robert Sedgewick. Algorithms in Java, Third Edition, Parts 1-4: Fundamentals, Data Structures, Sorting, Searching: 3rd Edition – 768 p., Addison-Wesley, 23 July, 2003. ISBN-13 : 9780201361209.