UDC 004.657

A. Stepanov, Y. Kornaga, E. Krylov, V. Anikin

FEATURES OF INDEXING IN DATABASES AND THE CHOICE OF THE OPTIMAL IMPLEMENTATION

Abstract: Database management systems use indexing to improve performance and speed up search queries. There are several possible indexing implementations. The problem is the choice of the optimal implementation depending on certain conditions. To address this issue a review of the main indexing implementations used in modern database management systems is provided. The data structures underlying indexes are considered. Examples and features of using each of the main indexing implementations are given.

Keywords: database management systems (DBMS), databases, indexing, indexes, data structures.

Introduction

One of the main goals in software engineering is to speed up the operation of applications, reduce the execution time of applications and requests, and also minimize costs and used resources. The challenge touched upon in this publication is to speed up database searches. To solve this problem, database management systems use indexing. This involves creating and using indexes on values from columns in tables that exist in the database. The created index is a separate database object and is a table of the values that are supposed to be searched and pointers to the corresponding values in the table of the database itself [1] (Fig. 1):

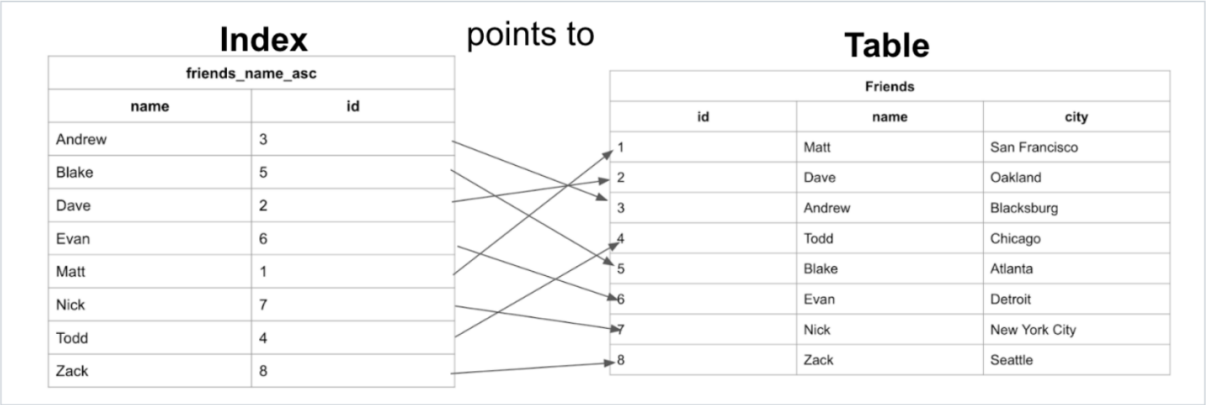

Figure 1. Index concept in databases

The index can be formed from the values of one or several columns of the table. In the second case such an index is called a composite index or multi-column index. Also, indexes can be unique when they refer to no more than one attribute of a table and non-unique when they correspond to several attributes at the same time.

 ISSN 1560-8956

The simplest example of using indexes in real life is the table of contents for a book. So, instead of going through all the pages of the book the reader can open a specific chapter of the book by getting the number of its starting page from the table of contents. Indexes in databases serve the same purpose. They allow you to get data from a specific record without going through all the records in the table.

Problem Statement

The speedup when using indexes is achieved primarily because the index has a structure optimized for search - a B-tree for example. Also, indexes are used when it is necessary to perform fast selection, sorting, performing a join of database tables.

If it is necessary to find a friend from a table with a specific name without using indexes the search will be performed by searching the table completely. The time complexity of such an algorithm will be O(n) or linear. You can create an index to the friend's name field in which names are sorted alphabetically. Search for a friend by name in a table with such an index will be carried out by an algorithm with a time complexity of O(log n) or logarithmic. This will be a binary search with halving. For example, if you need to find a friend named Zack the first iteration would be to check for a match in the middle of the friends list in the index. If there is no match, we discard the first half of the list, since the friends are sorted alphabetically in the index and we know that the letter “z” will be in the second half. Further, according to a similar algorithm a search is carried out in the second part and subsequent parts. Thus, using an index can significantly reduce the search time in the table especially as the amount of data in it grows.

The downside of using an index is increased memory usage. The index needs to create and store a separate table in the database. The number of database fields to which indexes are created also plays a role. Indeed, for each created index to a database field, it will be necessary to store a separate table, if we are talking about a simple index. This raises the problem of choosing the optimal number of fields for which you need to create an index. Typically, these are the fields that are searched for.

Another problem when using an index is the need to rebuild the index every time the database changes. Indexes cannot be used until they are updated. The developers of some database management systems declare support for the ability to query the database during index updates in the future, but this feature is not yet massively available. Thus, the question arises about the advisability of using and maintaining the operation of indexes in conditions of high intensity of records in the database.

If the decision to use indexes is made, one of the main problems becomes the problem of choosing the optimal implementation of the index, depending on the task and conditions. Possible solutions to this problem will be discussed below.

An overview of existing solutions

B-tree Index

There are several possible database indexing implementations. One of the most commonly used and also often the default indexes in database management systems is the B-tree or balanced tree [2, 3] (Fig.2).

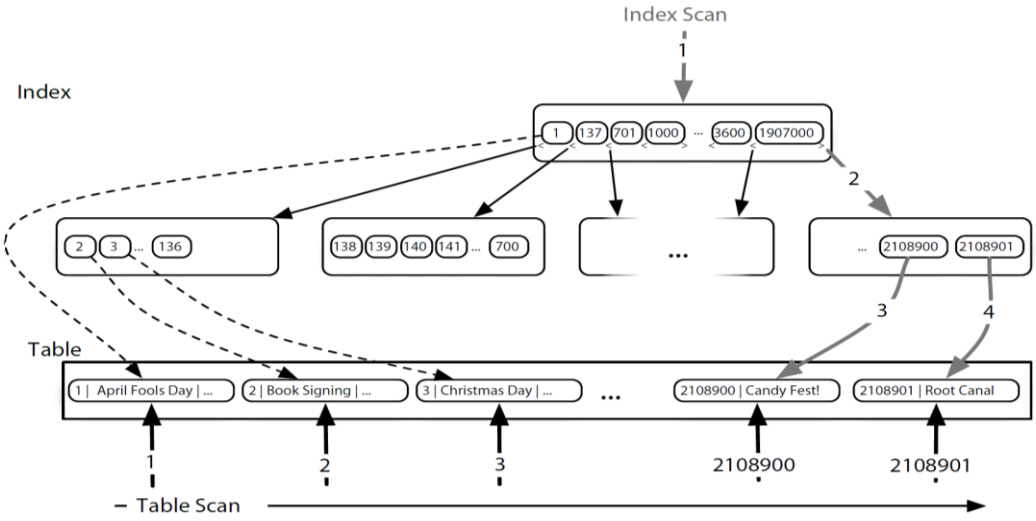

Figure 2. B-tree index in databases

B-tree is a data structure that is a balanced tree that stores data in its nodes in sorted order. Balanced means that the length of any two paths from root node to leaf node differs by no more than one. The B-tree stores data so that each node contains keys in ascending order. Each of these keys has links to child nodes. The number of such links for each node in the tree is determined by the degree of the tree. The higher the degree of the tree the more links to child nodes and the lower the nesting level of the tree nodes. Databases usually use a tree with a degree of nesting with root node, branch nodes and leaf nodes.

The B-tree index is not picky about the data of the fields, but it means that the fields can be compared with each other for a larger, smaller value and equality. B-tree indexing allows you to search by full key value, by a range of values, by the prefix part of the value, as well as by the index itself. B-tree indexing does not allow sampling without key prefix [4].

Hash Index

Another type of index used for indexing in databases is Hash index (Fig.3).

Hash indexes are based on the hash table (hash map) data structure. In other words, it is an associative array storing key-value pairs. Hash indexes are used as entry points for memory-optimized tables. Reading rows from a table requires an index that locates the data in memory. The Hash index consists of a collection of containers organized into an array. The hash function maps the index keys to the corresponding containers in the Hash index. If two index keys are matched to the same hash container a collision occurs - a hash conflict. A large number of hash conflicts can negatively impact read operations [5, 6].

 ISSN 1560-8956

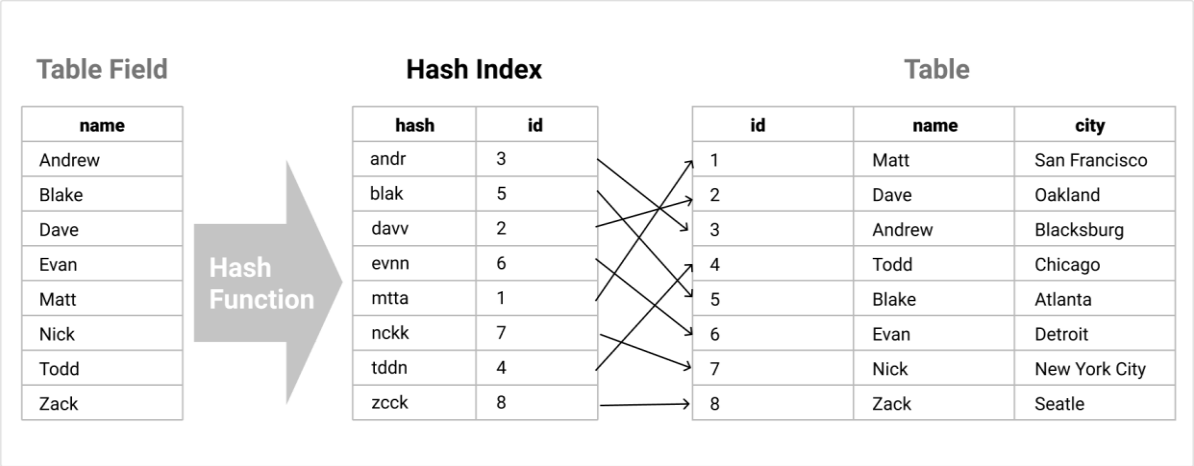

Figure 3. Hash index in databases

R-tree Index

Indexing some data types will not satisfy either the B-tree index or the Hash index. For example, these types of data include spatial, geographic, cartographic data. The R-tree index is used to index such data types [7] (Fig. 4).

R-tree index allows you to index values that may overlap with each other. The construction method of the R-tree index is similar to the B-tree index. The difference lies in the information written to the intermediate pages in the tree. R-tree index is actively used for geographic data types [8].

Other Indexes

The above indexes are the most basic, common and frequently used. However, there are a number of other indexes that are also widely used.

Inverted Index. An index commonly used for full-text searches. It can also be used to search through arrays and other non-atomic fields consisting of elements (by JSON object for example). It is a key-value pair where the key is search elements (words, numbers) and the key are pointers to the table fields where this element occurs [9].

Functional Index. The index in which all keys are derived from the function. For example, if you have a column of images, and the function is to determine the dominant color, then you can create an index as a result of this function. Such an index will allow you to quickly obtain all images with the same dominant color, without performing the function again [10].

Bitmap Index. An index based on the bitmap data structure (bitset, bitmask, bit array). Allows you to compactly store information about fields about compliance with any Boolean condition. Thus, the bit index value is a sequence of 0 or 1. Bit index can be used by the database to combine multiple indexes.

Partial Index. An index that may index not all fields. Used to optimize and reduce memory usage. For example, it can exclude NULL fields from indexing, thereby reducing the size of the index itself.

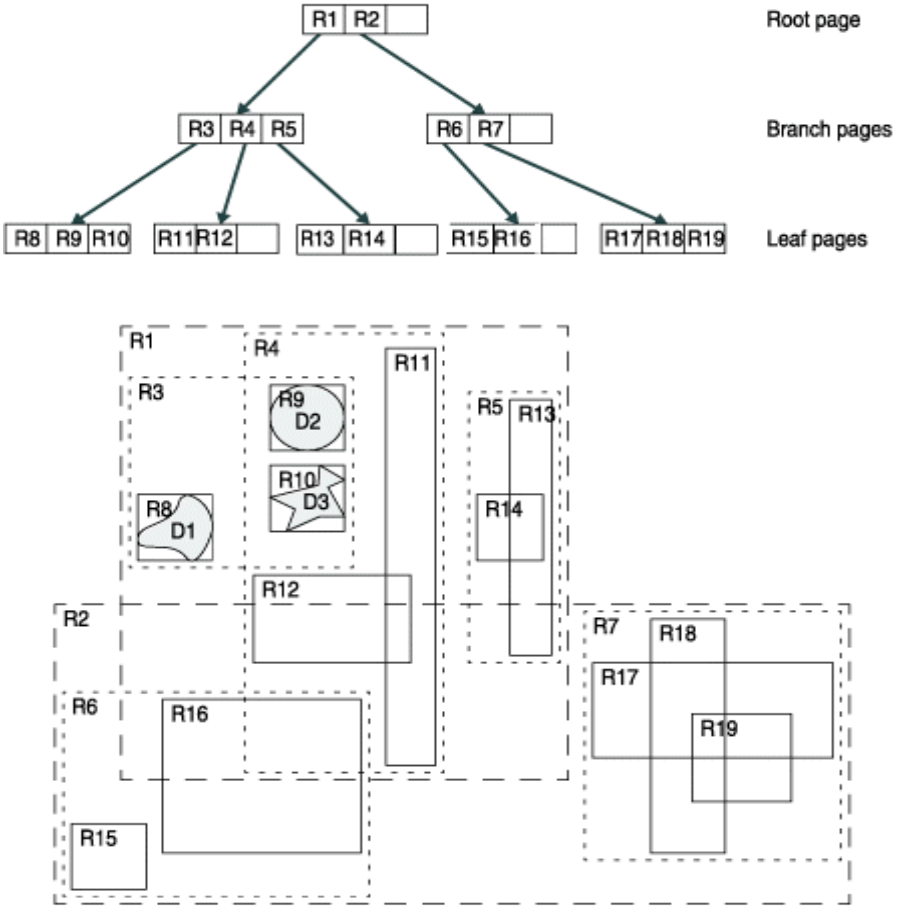

Figure 4. R-tree index in databases

Covering Index. An index with enough data to stop accessing the table. Allows increasing the speed of query execution. The index can be configured to cover more fields. However, this should not be overused as the index itself becomes larger.

Clustered Index. It is used to optimize the execution of queries by ordering the data of the table itself. By default, the data in the table is not ordered in any way. Using a clustered index implies ordering the table data in the same order as in the index. This helps to reduce the execution time of queries using this index. There can be only one clustered index in a table [11].

There are also other indexes that are not as widespread or used as often. It can also be a variety of the above-described indexes, or indexes unique to a specific database. Among these:

- **GiST index, Spatial index** – indexes most often based on the R-tree data structure;
- **GIN index, Text index** – similar to Inverted index;
- **Function-Based index** – similar to Functional index;
- **Reverse index** – index designed to eliminate index hot spots on insert application;
- **Unique index** – index that helps maintain data integrity by ensuring that no two rows of data in a table have identical key values;
- **Bidirectional index** – index that allows scans in both the forward and reverse directions;

 ISSN 1560-8956

- **Expression-based index** – index that can includes expressions;

- **BRIN** (Block Range Index) – index designed for handling very large tables in which certain columns have some natural correlation with their physical location within the table [12, 13, 14].

Choosing the optimal index

In certain tasks, under certain conditions it is advisable to use a certain index. Also, indexes can be combined. For the optimal choice of an index one should take into account the advantages, disadvantages and peculiarities of using each of them. To do this let's make a comparative characteristic of the basic indexes:

Table 1

Comparative characteristic of the basic indexes

Index	Features	Advantages	Disadvantages
B-tree	The most common type of index. Default index in many databases	Stores data sorted. Undemanding to the fields themselves. Efficient at processing inequality queries	Not as fast as Hash indexes for equality queries. Requires a relatively large amount of space
Hash	Index that implements the hash table data structure	Very efficient at equality lookups. The lookups take constant time which is independent of the number of rows in table	Not efficient at inequality queries. Require more space than range indexes. Possible collisions will slow down requests
R-tree	Used for geographic and spatial data	The ability to search for arbitrary points and regions	Redundancy in data storage. Because of this - slow data refresh
Inverted	Used for full text search	Best option for full-text search and other complex fields	Inserting and updating data is relatively slow
Functional	Index from function by field	Uses a B-tree structure, with the ensuing benefits	Increased the request time each time you insert and update a field
Bitmap	An index based on bitmap as data structure	Allows to store data about Boolean fields of the table using a relatively small amount	Inability to change the encoding method when updating the table
Partial	Used to reduce memory usage	Reduces memory usage by not indexing NULL fields	Time spent on index creation, complexity of index creation
Covering	Used to optimize query execution	Stores query data and allows you to refuse to access the table	Increases the amount of used memory while trying to cover a large number of fields
Clustered	Used to order the table itself	Allows you to reduce the execution time of a query by aligning the index and table	Usually, it is performed once and requires repeated calls later

Conclusion

The indexing process in database management systems is the creation of indexes - individual entities, tables in the database. Indexes are created to existing database tables and are used to speed up searching, sorting, and comparing information stored in the database. Effective use of indexes can significantly speed up database query processing. However, indexes take up additional space in the database and also slow down the database when inserting and updating information. Consider this before using indexes.

Once the decision on the advisability of using indexes is made, the main problem becomes the choice of the optimal implementation of the index. There can be several indexes and they can be combined. One or another index is effective depending on the tasks and conditions. To successfully include indexes in the database, you need to consider the specifics of using each of them.

REFERENCES

1. Indexing [Electronic resource] // The Data School by Chartio. – 2020. – URL: https://dataschool.com/sql-optimization/how-indexing-works.

2. Redmond E. Seven Databases in Seven Weeks, Second Edition / E. Redmond, J. Wilson. – Dallas, Texas; Raleigh, North Carolina U.S.: The Pragmatic Bookshelf, 2018.

3. Lehman P. Efficient locking for concurrent operations on B-trees / P. Lehman, S. Yao. // ACM Transactions on Database Systems. – 1981. – Pages 650–670.

4. Graefe G. Modern B-Tree Techniques / G. Graefe, H. Kuno. // IEEE 27th International Conference on Data Engineering. – 2011. – Pages 1370–1373.

5. Hash Indexes [Electronic resource] // Microsoft SQL Server 2014 documentation. – URL: https://docs.microsoft.com/en-us/previous-versions/sql/2014/database-engine/hash-indexes.

6. Index Table pattern [Electronic resource] // Azure Product documentation. – 2017. – URL: https://docs.microsoft.com/en-us/azure/architecture/patterns/index-table.

7. IBM Informix R-Tree Index User's Guide [Electronic resource] // IBM Informix Server V11.50 documentation. – URL: https://www.ibm.com/support/knowledgecenter/en/SSGU8G_11.50.0/com.ibm.rtree.doc/sii-overview-27706.htm.

8. Böhm C. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases / C. Böhm, S. Berchtold, D. Keim. // ACM Computing Surveys. – 2001. – Pages 322–373.

9. Black P. Inverted index [Electronic resource] / Paul E. Black // Dictionary of Algorithms and Data Structures. – 2017. – URL: https://www.nist.gov/dads/HTML/invertedIndex.html.

10. Functional Indexes [Electronic resource] // IBM Informix Server V12.1 documentation. – URL: https://www.ibm.com/support/knowledgecenter/en /SSGU8G_12.1.0/com.ibm.adref.doc/ids_adr_0325.htm.

11. Clustered and Nonclustered Indexes Described [Electronic resource] // Microsoft SQL Server 2019 documentation. – 2019. – URL: https://docs.microsoft.com/en-us/sql/relational-databases/indexes/clustered-and-nonclustered-indexes-described?view=sql-server-ver15.

12. Types of indexes [Electronic resource] // IBM Db2 Warehouse documentation. – URL: https://www.ibm.com/support/knowledgecenter/SSCJDQ/com.ibm.swg.im.dashdb.kc.doc/welcome.html.

13. Using a Different Index Type [Electronic resource] // Oracle Database Performance Tuning Guide. – URL: https://docs.oracle.com/en/database/oracle/oracle-database/19/tgdba/designing-and-developing-for-performance.html#GUID-38FC5A9F-89E6-4812-8EE4-F9949B69BCFC.

14. BRIN Indexes [Electronic resource] // PostgreSQL 9.5.22 Documentation. – URL: https://www.postgresql.org/docs/9.5/brin-intro.html.