STUDIA INFORMATICA
Nr 1-2(17) Systems and information technology 2013

The influence of indexing methods on effective functioning of the database

Andrzej Barczak[1], Dariusz Zacharczuk[1], Anna Korzeniecka[1]
[1] Uniwersytet Przyrodniczo-Humanistyczny, Instytut Informatyki
ul. 3 Maja 54, 08-110 Siedlce, Poland

Abstract: The article describes the different types of indexes. Their characteristic determines whether and when they can be used to improve database performance. Then studies are performed using different indices for different situations. The conclusions of the study can be serve as a guide to correct use of indexes.

Keywords: database optimization, T-SQL, MS Server, indices

1 Introduction

Regardless of the type of business, well-organized data maintenance is a key of any project. To do so, indices come here with help. Well used, can significantly optimize the performance of the database. The index is used primarily to improve query performance. Database index consists of records with two fields:

1. key: includes values of the attributes on which index is established
2. pointer: block contains records, that values of attribute are equal to the index key. Pointer determines the physical location of the row in the database table. With index, we can effectively retrieve data from a table.

In some way, indices function as a shortcut to the data table. Indices are mostly used for a SELECT query. The aim is to find relevant information in a database. The way the indices work is as follows: at the time of execution of query the index is searched. Then, based on the index, suitable records are found. How exactly indices affect the performance of the database instance? Let's do example:

```
  CREATE TABLE customers (id_klienta number, name varchar2 (30), name
varchar2 (30));
```

All "Smith" are show by the following query:

```
  SELECT id_klienta, name, name FROM customers WHERE name = 'Smith';
```

When table has a lot of records query takes a long time. Why is this happening? Query is searching table step by step, line by line. In this case, you can create an index on the column `name`:

```
CREATE INDEX ON indeks_klienta customers (name);
```

When the table has a marginal number of records, with the name of 'Smith', the index immediately locates the exact position of the block in the table. In the case where the table has a large number of matching results, omit index and search each block in the table would by efficiently and immediately.

Above example shows how important it is, that the implementation of the database indices were carefully thought out.

2 Index characteristics

Creating indices take up disk space. They are not stored together with the tables, but are defined on a table on one or more columns. Selecting columns, for which the index is created, is very significant, because the indices in some cases may increase the time of inserting or modifying data operations. Indices are creating primarily for:

- Columns of the primary key constraints (PRIMARY KEY)
- Column of the foreign key constraints (FOREIGN KEY) and columns used to join tables
- The columns that contain data used as a search argument,
- Often ordered columns containing data.

Index must be removed when:

- It is no longer needed;
- Has become invalid and need to be re-build by removal;
- There is a need to move the index to another tablespace;
- In an indexed table will be performed large inserts or updates.

After deleting index by DROP instruction, the space occupied by it, will be returned to the system. However, there are exceptions to this rule. Namely indices created automatically for columns, while giving them the refinements UNIQUE or PRIMARY KEY, can not be removed that easily. They can be removed by following the appropriate ALTER TABLE.

2.1 Type of indices

indices can be divided according to: **index attribute characteristics**, **the number of levels**, **the structure**, **the number of indexed attributes**, unique key value, **the number of index indications to the data file**, the sequence of key values, manner of storage and **applications**. In this article the five bolded above will be discussed.

Index attributes

Stands out for:

- primary index. For this type of index, records indicate from index leading directly to the data blocks. Primary index is founded on ordering attribute of indices file. This attribute specifies the order of the records in the file. Values of unique attribute are keep in order. Not all records in the data file in the index have a primary index records. This record contains the values for a specific data block address, where the data record with index attribute value equal to that values.
- clustered index. Clustered index is also founded on ordering attribute, but the values are not unique. Indexed record of clustered index for a particular value, contains the address to data block, where the first record of the attribute value equal to the value of the index. This file organization is a problem when inserting records, because after the addition of the data, records order must remain unchanged. The solution to this problem would be to book the entire block on the records with the same value. Another solution is the use of redundant units. Added records are stored in the free space of main block and when it fill-in, pointer from main block indicates to appropriate excess block.
- Secondary index. Secondary index is founded on index attribute, which is not an attribute of the file ordering. This index is also structured. Each record has its counterpart in the index record. Secondary index record consists of two fields: an index field value and a pointer to a record or data block that contains the record.

Number of index level

- one level indices: one index is created for the data file. Finding data using this kind of index requires search the index file. Through the use of founded index records, read of data are made. Index file is searched by binary algorithm because it is a structured file. This algorithm is not very efficient. For this reason, a multi-level indices introduced, which are searched in efficient manner.
- Multilevel indices: there is another index creates for the first one. One of the main concepts of this type of index is the structure of ISAM (Indexed Sequential Access Method). Conceptually, this structure consists of two layers. At the first level cylinder are indexed. At this level the index records contain pairs of values each key and the address of the tracks index. At the second level disk paths are indexed. It contain pairs of values: the key and the search path address. It is strongly connected to the hardware. ISAM is a static index. This means that they do not have sophisticated mechanisms for modifying the structure where a change to the contents of the indexed file is made. Thus, removing record will cause the empty space in the index block and new records will be added to the overflow blocks. As a result, the structure of the ISAM index becomes ineffective.

Structure

- B-tree: usually used in OLTP systems. Is defined for the attributes of high selectivity. In case of the B-tree with the indexed key a list of addresses of records, where the attribute values are equal to the index key, maintain. Index of this type offers efficient operations of conjunction, equal worth queries and intervals, testing unique attribute, sorting, grouping, calculating the minimum and maximum values and the elimination of duplication. B-tree is a balanced tree structure. This means that the distance from the root to any leaf is the same. The inside tops of trees are used to support the search for records, and leaf nodes contain records with pointers to records in the data files. In order to ensure effective implementation of appropriate queries leaf nodes are bidirectional list. Search the record requires the transition away from the root to leaf. Internal node B-tree is: a pointer to the node, the value of the index key, the next indicator, the next value, etc. The number of indicators is always one more than the number of key values. Leaf is: the value of the index key, a pointer to the record (block) with the key value.

Finding the right leaf requires reading 3 blocks of index: root, inner, and leaf node. Using the pointer of leaf we need to read one block of data that contains the search record. In order to go through the tree and reach a specific record, required a few pages accesses. The state of the tree must be continuously monitored and the branches transformed as needed. Balance the tree is very important its property. If you modify a single record it generated low cost of this modification. The high cost is obtained when modifying a group of records.
Indices whether they are grouped or single are created mostly in the form of a balanced tree. This ensures logarithmic time operations such as insert, search, or delete items.

- bitmap index: it is the most widely used in OLAP systems. Is defined only for attributes with low selectivity. With the index bitmap key is stored. The bitmap is a table, where each cell contains a single bit corresponding to one record in the table. The bit is set to 1 when attribute of record has a specific value, otherwise is set to 0. Number of bitmap corresponds to the number of different attribute values for this attribute. Bitmaps records are represented in the order in which they appear in the table. Bitmap index is used in queries with the terms with operator "=". It is used quite often in queries looking for blank values. The size of this index is strongly dependent on the size of the index attribute fields. When modifying a single record, this type of index generates a high cost modification. If you modify a group of records the cost of modification is lower.

Number of attributes in the key

- simple indices is characterized by the fact, that the index key contains only one attribute indexed.
- complex index: the index key contains not one but more attributes of the relationship. Combinations of attributes: X, XY and XYZ of the index founded on attributes XYZ, is a leading part of the key, as opposed to

a combination of Y, YZ and Z. Complex index is submitted on the attributes that occur frequently together in a WHERE clause and attributes often read together by multiple queries. Attributes used quite frequently in the WHERE clause should be part of the leadership key. In the case where the frequency is the same, the first attribute should be the one, by which the data values are sorted.

Use

- functional indices: established on attributes which are often used in queries as a parameter of functions eg. UPPER(name) or are part of expressions eg. base_price*1.23. Index of this type can be implemented either as an index B-tree index type as well as a bitmap.
- Connected bitmap indices: are defined for operations combine two or more relationships. For each values of indexed attribute for one relationship, another relationship addresses are stored, which have the same value of linking attribute.

Number of indications to data file

- dense indices: contains records for each value indexed data file.
- sparse indices: has records only for selected values of an indexed data file.

3 The concept of a database

The database does not need to be very complicated but it should contain thousands or even millions of records. ERD diagram is shown below.

The customer should be able to freely browse, edit and find information quickly. Indices, in certain situations, can make adding difficult. The same applies to the updating of data already stored in the database. System, that meets the above requirements, should allow users to work comfortably.

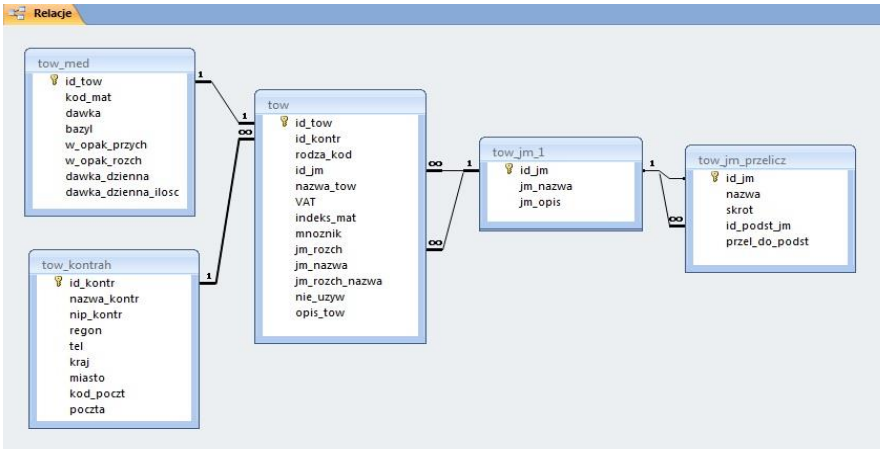

Figure 3.1 Database logical model

3.1 Queries structure

In the created database, queries will have to display user-selected information, add new and update existing data. Designed queries will be used to verify the impact of indexing on effectiveness of the database. There will be 8 and they will vary in terms of the obtained results and the complexity:

1. Searching for materials that contain the name of a specific sequence of letters.
2. Search for materials by ID that contain the name of a string of letters.
3. Searching for materials that match the pattern.
4. Searching for materials for which the ID within the specified range and the contractor city is e.g. Warsaw.
5. Searching for materials for which the ID units within the specified range and the name fits the pattern
6. Find the number of materials for each unit of measure, which units ID within the specified range.
7. Search materials, whose name contains the string of letters and the name of the package contains "szt".
8. Search materials, whose name matches a certain pattern and dose contains the name of the string.

Besides SELECT, to test a database will be used INSERT and UPDATE. They will be used to check the database behaves when you enter new and update old data in the absence and together with indices:

9. Enter sample data into the table `tow`
10. Modification of indexed fields id_jm.
11. Modification not indexed fields.

3.2 The implementation of the database

Implementation of the system will be in T-SQL on SQL Server. After creating a database, tables will be filled with data. Instructions for testing the aforementioned points are as follows:

1. `SELECT nazwa_tow FROM materialy.dbo.tow WHERE nazwa_tow like '%an%' GROUP BY nazwa_tow order by nazwa_tow`
2. `SELECT nazwa_tow, id_jm FROM materialy.dbo.tow WHERE nazwa_tow like '%an%' AND id_jm = 1 GROUP BY nazwa_tow, id_jm order by nazwa_tow`
3. `SELECT t.nazwa_tow, tm.jm_nazwa FROM materialy.dbo.tow t, materialy.dbo.tow_jm tm WHERE nazwa_tow like '%an%' AND t.id_jm = tm.id_jm GROUP BY nazwa_tow, tm.jm_nazwa order by nazwa_tow`
4. `SELECT count(*), t.nazwa_tow, t.indeks_mat, tjm.jm_nazwa, tk.nazwa_kontr, tm.bazyl FROM tow t, tow_kontrah tk, tow_jm tjm, tow_med tm WHERE tm.id_tow=t.id_tow AND t.id_jm=tjm.id_jm AND t.id_kontr=tk.id_kontr AND tjm.id_jm<20 AND tk.miasto =`

```
   'Warszawa' GROUP BY t.nazwa_tow, t.indeks_mat, tjm.jm_nazwa,
   tk.nazwa_kontr, tm.bazyl;
5. SELECT count(*), t.nazwa_tow, t.indeks_mat, tjm.jm_nazwa,
   tm.bazyl FROM tow t, tow_jm tjm, tow_med tm WHERE
   tm.id_tow=t.id_tow AND t.id_jm=tjm.id_jm AND tjm.id_jm<20 AND
   t.nazwa_tow LIKE '%Cew%' GROUP BY t.nazwa_tow, t.indeks_mat,
   tjm.jm_nazwa, tm.bazyl;
6. SELECT count(*), tjm.jm_nazwa FROM tow t, tow_jm tjm, tow_med
   tm WHERE tm.id_tow=t.id_tow AND t.id_jm=tjm.id_jm AND
   tjm.id_jm<20 AND tjm.jm_nazwa=t.jm_rozch_nazwa GROUP BY
   tjm.jm_nazwa;
7. SELECT tm.* FROM tow_med tm, tow t WHERE t.id_tow=tm.id_tow AND
   t.nazwa_tow LIKE '%Złączka%' AND tm.w_opak_przych LIKE '%szt%'
8. SELECT tm.* FROM tow_med tm, tow t WHERE t.id_tow=tm.id_tow AND
   t.nazwa_tow LIKE '%SODIUM%' AND tm.dawka LIKE '%CM%'

9. INSERT INTO tow (ID_KONTR, RODZA_KOD, ID_JM, NAZWA_TOW,VAT,
   INDEKS_MAT, MNOZNIK,JM_ROZCH, JM_NAZWA, JM_ROZCH_NAZWA,
   NIE_UZYW, OPIS_TOW) VALUES (5, 'SPM', 1, 'FILTR INFUZYJNY
   NOWORODKOWY F 62', 8, 'SJ-06-0005', 1, 1, 'opak.', 'opak.', 0,
   NULL)
10. UPDATE tow SET nazwa_tow='Test mod.', id_jm=10, indeks_mat =
   'SJ-01-010101' WHERE id_tow = 6684675
11. UPDATE tow SET nazwa_tow = 'Test mod.2', indeks_mat = 'SJ-01-
   020101', opis_tow = 'Test opisu' WHERE id_tow = 6684673
```

3.3 Creating indices

A database is not complicated nor complex. It was created in SQL Server, which supports two types of indices: grouped and ungrouped. Both types store information using a standard B-tree. Therefore, to test the effect of indexing methods for the effective functioning of the database, three types of indices were selected: simple, complex, clustered.

Most searches results relates to table \`tow\`, so at the beginning we created **simple index** for the table column \`nazwa_tow\` .

```
CREATE NONCLUSTERED INDEX ind_1_prosty ON tow(nazwa_tow ASC);
```

In addition, we created a simple index on a column \`jm_nazwa\` in the table \`tow_jm \`.

```
CREATE NONCLUSTERED INDEX ind_3_zlozony_prosty
ON tow_jm(jm_nazwa ASC);
```

Composite index is create on the attributes often occur together in the WHERE clause queries and attributes often read together by multiple queries. Composite index created on columns \`nazwa_tow\` and \`id_tow\` on table \`tow\`.

```
CREATE NONCLUSTERED INDEX ind_1_zlozony
ON tow(id_tow ASC, nazwa_tow ASC);
```

For the purpose of search queries using the unit of measure, a **composite index** on columns `nazwa_tow` and `id_jm` was created:

```
CREATE NONCLUSTERED INDEX ind_2_3_zlozony
ON tow(id_jm ASC, nazwa_tow ASC);
```

A **clustered index (grouped)** was founded on the columns `nazwa_tow` and `id_jm`:

```
CREATE NONCLUSTERED INDEX ind_1_2_3_klastrowany
ON tow(nazwa_tow ASC, id_jm ASC);
```

Now it's time for gathering results.

4 Research process of impact indexing methods on the efficiency of the database

Each of the created queries tested several times in four different cases: 1) a database with no indices, 2) with simple, 3) complex 4) and clustered index. Testing performed on two types of factors, that affect the effectiveness of the database. Namely, these are the query execution time and the volume occupied by indices. The study was conducted in MS Windows, which is not real-time system. Therefore, query execution time on different computers can vary and be dependent on factors such as CPU utilization, CPU clock speed, amount of memory, system load, number of processes running on the system, and many others.

4.1 Query execution time

The tables below summarizes the execution times of individual queries to the database without and with indices. Queries times for SELECT are given in seconds, accurate to the thousandth of it.

First query is relatively simple: search for records from one table with only one column. Therefore, the results are very similar. Although it can be noted, that the worst results were obtained with a complex index. This is due to the fact, that the query is based only on one column but this type of index is founded on two, which slows down the performance of the query.

Table 4.1. Times summary of query No. 1

Query	No index	Simple index	Complex index	Clustered index
#1	23,234	22,371	23,971	21,115
	23,561	22,201	24,013	20,945
	23,011	22,112	24,71	20,899
Average	23,269	22,228	24,231	20,986

A better result was achieved when the index is not used or the use of a simple index. The best results were obtained for the clustered index. This is because the index contains the address of the block in which the first data record with the value of indexed attribute equal to that value.

Table 4.2. Times summary of query No. 2

Query	No index	Simple index	Complex index	Clustered index
#2	25,69	24,501	22,512	22,311
	26,11	24,087	22,411	22,113
	25,918	24,211	22,417	22,415
Average	25,906	24,266	22,447	22,28

Query #2 search records from one table with two columns. Despite the fact, that the results are close, you can see exactly the worst result was obtained without the use of an index. This is due to the fact that the records are not sorted in any way, and then the query table must be searched from the beginning to the end. Slightly better results were obtained using a simple index. Much better results were for the complex and the clustered index.

Table 4.3. Times summary of query No. 3

Query	No index	Simple index	Complex index	Clustered index
#3	24,612	23,15	22,071	21,912
	24,332	23,088	21,819	22,012
	24,387	22,978	22,009	22,142
Average	24,444	23,072	21,966	22,022

This query uses two tables. Definitely the worst result was obtained without use of an index. In this query similar results and also the best, were obtained for the clustered and complex index. This time, composed index works well, because it uses two columns used in the WHERE clause.

Table 4.4. Times summary of query No. 4

Query	No index	Simple index	Complex index	Clustered index
#4	65,968	57,543	56,225	51,056
	65,745	57,439	55,798	51,211
	66,012	56,811	55,922	50,698
Average	65,908	57,264	55,982	50,988

Request #4 is more complicated. Uses four tables and counting function, so the execution times are longer and more varied. Once again, the worst results were obtained for tables without indices, and the best for the clustered one.

Table 4.5. Times summary of query No. 5

Query	No index	Simple index	Complex index	Clustered index
#5	33,945	27,231	24,234	16,645
	33,594	26,493	24,087	16,102
	33,289	26,748	23,452	16,273
Average	33,609	26,824	23,924	16,340

Request #5 is similar to the previous query. However, in the condition indexed column has been used, which makes the execution times are shorter than previously and is clearly different. Again, the worst without indices, and the best for the clustered index. Note that in this case, execution time for the grouped index is half less, compared to the result obtained without the use of indices.

Table 4.6. Times summary of query No. 6

Query	No index	Simple index	Complex index	Clustered index
#6	50,652	37,411	36,211	21,127
	50,321	37,210	35,486	20,723
	50,268	37,129	36,045	21,298
Average	50,414	37,250	35,914	21,049

Above query uses three tables and function count(), so the test results are quite varied. The worst were obtained again without the use of indices, and the best for the clustered index. The difference between the longest and the shortest execution time request is approximately 30 seconds. Given the low complexity of the database is a very significant difference.

Table 4.7. Times summary of query No. 7

Query	No index	Simple index	Complex index	Clustered index
#7	36,411	29,468	27,149	22,234
	35,697	28,658	26,954	22,128
	36,129	28,832	26,734	22,087
Average	36,079	28,986	26,946	22,150

Request #7 finds records containing the names matching the pattern. The results are quite different, but once again the longest query execution time by far, was achieved without use of indices. Top times for the simple index and the complex one. The fastest query is finished for the clustered index.

Table 4.8. Times summary of query No. 8

Query	No index	Simple index	Complex index	Clustered index
#8	30,225	26,896	26,210	20,736
	29,798	26,621	25,736	21,163
	30,289	26,795	25,938	21,240
Average	30,104	26,771	25,961	21,046

Request #8 is very similar to the previous one. It has only slightly different conditions in the WHERE clause. This similarity has led to results, that do not differ significantly from the query #7.
In addition to the index impact on the duration of the SELECT queries, you can also explore the impact on adding new record (INSERT) and update existing ones (UPDATE). The study takes place in the same way as the previous queries. The only difference is that the results are given in milliseconds. This is due to the fact, that for the low complexity of the database is easier to see the difference when the measuring unit is more accurate. Table 4.9. presents the time results for INSERT command. It has been tested only once, because the command is not changing, so repeated testing would the same results. UPDATE examines in two situations: when the indexed and non indexed fields are modified.

Table 4.9. Times summary of query No. 9

Query	No index	Simple index	Complex index	Clustered index
#9	41	43	45	52
#10	596	663	1176	1305
#11	756	613	590	556

For queries inserting a new record into a indexed table, the worst result was obtained for clustered indices, the best for a table without indices. This is due to the fact that, a new record is added to the end of the table, which does not last long.
For the first updating query, the worst result obtained in the case of a clustered index, the best for a table without any. The query includes modifying the indexed fields and its increases with the complexity of the indices.
For the second updating data query weakest result was obtained for tables without and grouped indices, the best for the simple index. This is due to the fact that the query does not modify the indexed fields.

4.2 Capacitive factor

Study capacitive factor is nothing but a verification of the assumed size of the database indices. Indices, in addition to its advantages - speed up work requests, has also disadvantages - very big size of space needed for storage. With tables with millions of rows indices are beginning to address the mass of disk space. It is therefore important to analyze this factor, as it often happens that a lot of indices involved to a small extent help to improve performance. The results appear in Table 4.10. obtained by checking the properties of the size of the database before adding the index and after its creation.

Table 4.10. Capacitive factor results

Index	Table	Capacity before	Capacity after
ind_1_2_3_klastrowany	tow	1171 MB	1823 MB
ind_1_prosty	tow	1171 MB	1418 MB
ind_1_zlozony	tow	1171 MB	1443 MB
ind_2_3_prosty	tow	1171 MB	1417 MB
ind_2_3_zlozony	tow	1171 MB	1443 MB

Database capacity before using indices was 1171 MB. After adding the indices it increased accordingly. Minimum space is needed by a simple index. A little more for composite index. By far the largest space for storage needs clustered one.

5 Summary

When examining the impact of indexing methods two factors were considered: time and capacity. Test results show if and how each method affects the indexing database. However the differences are not large. Today's systems are able to deal better and more quickly with the performance of this type of queries. Despite that, small differences can easily interpret the time results.

By far the worst results were obtained without the use of indices. Slightly better results have been obtained using a simple index. This is understandable because in the case of a simple index, records are sorted by a specific column. With sorting at the start, we reject a large amount of unnecessary records. It is therefore no longer necessary to view the entire table to find specific data. In the case of the complex index, query execution times are generally better, than the time obtained using simple indices. The composite index is characterized in that, the index key has more than one relationship attribute. Records are sorted by more than one column. Therefore, in the WHERE clause when we have more conditions relating to the various columns, composite index makes it easy to search for information. Sometimes, however, the times are worse. This happens when you create a composite index with mismatched columns.

By far the best results were achieved for clustered indices (grouped). Indexed record of grouped index for a particular value, contains the address of data block, where the first record of the searched value is located. In this way, data mining takes place much faster than using grouped indices. This shows the positive impact of indexing on performance of a SELECT command.

For INSERT command results are unlike in the case of a SELECT. This is due to the complexity of grouped index file organizations. Such an organization may causes problems with inserting records, because the records order modifications must remain unchanged. The best result is achieved when you insert data in the table on which any indices was established. This study shows the negative impact indices have for the duration of commands such as INSERT.

UPDATE commands shows both positive and negative impact on the performance of indices. The first contains modification of indexed field `id_jm`, for that the time of its performance increases with the complexity of the index. The second command does not modify the indexed fields. By contrast, finds the records by `id_tow` (WHERE clause) thereby making this type would be more optimal using composite index. The data write command is executed at the same time for each of the variants, but to search the position of "id_tow", the indices will operate faster.

The second factor is verification of the assumed size of the database indices. Table 4.10. shows the obtained results. The indices need quite a space for storage. With the rise of rows in a table, increases the amount of space needed to store indices on disk. Databases containing large amounts of indices therefore require larger drives, better equipment which of course is associated with additional costs.

Conclude is that, the indices makes a noticeable acceleration of the query execution for small tables, but it also brings great benefits for complex and large amounts of data. Therefore, if a table is mainly used to read the data, there a larger number of indices can support the operation on the database. Unfortunately, in addition to such important advantages are also significant drawbacks, such as a fairly significant size space required for their storage and increasing system load. indices also slow down the operations of data entry and editing. If the table is modified quite often a better solution is to reduce the number of indices.

At the end – indices are and must be used in databases for performance reasons, but their use must be well considered.

References

1. Barczak A., Florek J., Sydoruk T.: Bazy danych. Wydawnictwo Akademii Podlaskiej, Siedlce 2007.
2. Taniar D., Rahayu J. W.: A Taxonomy of Indexing Schemes for Parallel Database Systems. Distributed and Parallel Databases, Volume 12, Number 1, Kluwer Academic Publishers, pp. 73-106, 2002

3. Liebeherr J., Omiecinski E., Akyildiz I. F.: The Effect of Index Partitioning Schemes on the Performance of Distributed Query Processing. IEEE Transactions on Knowledge and Data Engineering archive, Volume 5, Issue 3, 1993, pp. 510-522
4. Helmer S., Moerkotte G.: A performance study of four index structures for set-valued attributes of low cardinality. VLDB Journal, 12(3): pp. 244-261, October 2003
5. Bertino E. et al.: Indexing Techniques for Advanced Database Systems. Kluwer Academic Publishers, Boston Dordrecht London,1997
6. Elmasri R., Wprowadzenie do systemów baz danych. Helion, Gliwice 2005.
7. Johnson E., Jones J., tł. Moch W.: Modelowanie danych w SQL Server 2005 I 2008. Przewodnik. Helion, Gliwice 2009.
8. Kuhn D., Alapati S. R., Padfield B., Expert Indexing in Oracle Database 11g: Maximum Performance for your Database. Apress, New York 2011.
9. Majczak A.: SQL od podstaw, Translator S.C., Warszawa 2001.
10. Matthew N., Stones R.: Od podstaw Bazy danych i MySQL. Helion, Gliwice 2003.
11. Pankowski T.: Podstawy bazy danych. PWN, Warszawa 1992.
12. Szeliga M., Tablice informatyczne. SQL. Helion, Gliwice 2005.
13. Szeliga M., Transact-SQL. Czarna księga. Helion, Gliwice 2005.
14. Vieira R., SQL Server 2005. Programowanie. Od podstaw. Helion, Gliwice 2007.
15. Whitehorn M., Marklyn B.: Relacyjne bazy danych. Helion, Gliwice 2002.