

*Radostaw BORONSKI\**

## **INDEXES SELECTION FOR BLOCKS OF RELATED SQL QUERIES**

### **Abstract**

*This paper discusses the problem of minimizing the response time for a given database workload by a proper choice of indexes. The main objective of our contribution is to illustrate the database queries as a group and search for good indexes for the group instead of an individual query. We present queries block relation conditions for applying the concept of grouped queries index selection. In three experimental tests we provide measurements on the quality of the recommended approach.*

### **1. INTRODUCTION**

Getting database search result quickly is one of the crucial optimization problems in a relational database processing. The major strength of relational systems is their ease of use. Users interact with these systems in a natural way using nonprocedural languages that specify what data are required, but do not specify how to perform the operations to obtain those data [8]. Online Internet shops, analytics data processing or catalogue search are examples of structures where data search must be processed as quick as possible with minimal hardware resources involved. Common practice is to minimize the database search process at minimal cost. A database administrator (or a user) may redesign the physical hardware structure or reset the database engine parameters, or try to find suitable table indexes for a current query. Most vendors nowadays offer automated tools to adjust the physical design of a database as part of their products to reduce the DBMS's total cost of ownership [3]. As adding more CPUs or memory may not always be possible (i.e. limited budget) and maneuvering within hundreds of database parameter may lead to a temporary solution (wrong settings for other database queries), index optimization should be considered as being foremost.

Indexes are optional data structures built on tables. Indexes can improve data retrieval performance by providing a direct access method instead of the default

---

\* Koszalin University of Technology, ul. Śniadeckich 2, 75-453 Koszalin, Poland, e-mail: [radoslaw.boronski@ie.tu.koszalin.pl](mailto:radoslaw.boronski@ie.tu.koszalin.pl)

full table scan retrieval method [7]. In the simple case, each query can be answered either without using any index, in a given answer time or with using one built index, reducing answer time by a gain specified for every index usable for a query [14]. Hundreds of consecutive database queries together with large amount of data involved lead to a very complex combinatorial optimization problem. Time needed to obtain result of both index-less tables joined together may be up to 45 minutes. Such delays are not acceptable for production environment processes. Indexes in such cases may reduce the response time of 50% (depending on which columns are used for the indexing). The classic index selection method focuses on a tree data structure, which could limit the search area as much as possible. Literature acknowledges us with such B-tree types as: (Known in the literature are those of the type of B-tree such as)

- Sorted counted B-trees, with the ability to look items up either by key or by number, could be useful in database-like algorithms for query planning [5],
- Balanced B\*-tree that balances more neighboring internal nodes to keep the internal nodes more densely packed [12],
- Counted B-trees with each pointer within the tree and the number of nodes in the subtree below that pointer [19].

The B-tree and its variants have been widely used in recent years as a data structure for storing large files of information, especially on secondary storage devices [11]. The guaranteed small (average) search, insertion, and deletion time for these structures makes them quite appealing for database applications.

The topic of current interest in database design is the construction of databases that can be manipulated concurrently and correctly by several processes. In this paper, we discuss a simple variant of the B-tree (balanced B\*-tree, proposed by Wedekind [20] especially well-suited for use in a concurrent database system [15].

While the selection of indexes structure have a very important role in the design of database applications, one should plan the indexes structure and number of indexes at the early stage of database development operation. In such situations more important is to ask a question “how to choose a set of indexes for the selected query sets?”. It turns out that the proper selection of indexes can bring significant benefits for the database query execution time. Typical approaches found in the literature mainly focus on the search indexes only for single column or single query [16], [10], [9], [17], [4]. In this paper, an approach associated with the search query indexes for groups called blocks is presented.

In this case we will consider B-tree indexes. A B-tree index allows fast access to the records of a table whose attributes satisfy some equality or range conditions, and also enables sorted scans of the underlying table [18]. Also, we focus on databases with the same SQL queries repeated periodically. By doing

so, we eliminate database queries' low selectivity factor where no good indexes could be found due to changing queries sets.

The rest of the paper is organized as follows: in section 2, we describe a problem statement. In section 3, we briefly present classic index selection approach together with simple examples that will illustrate the subject. In section 4, we demonstrate new method of grouped queries index selection and compare examples results with classic approach. Test and comparisons with commercial tools results are presented in section 5. Section 6 presents our conclusions and further works.

## 2. PROBLEM STATEMENT

Motivation for this work is to suggest an approach of multi-queried SQL block where sub-optimal or optimal solution is to be found that gives decision makers some leeway in their decisions. The main goal is to choose a subset of given indexes to be created in a database, so that the response time for a given database workload together with indexes used to process queries are minimal.

The index selection problem has been discussed in the literature. Several standard approaches have been formulated for the optimal single-query and multi-query index selection. Some past studies have developed rudimentary on-line tools for index selection in relational databases, but the idea has received little attention until recently. In the past year, on-line tuning came into the spotlight and more refined solution was proposed. Although these techniques provide interesting insights into the problem of selecting indexes on-line, they are not robust enough to be deployed in a real system [18]. The problem is known in a literature as Index Selection Problem (ISP) According to [8] it is NP-hard. Note that in practice the space limit in the ISP is soft, because databases usually grow, thus the space limit is specified in such way that a significant amount of storage space remains free [13].

In a real life scenario, for thousands database queries (Fig. 1) compromising hundreds of tables and thousands of columns, the search space is huge and grows exponentially with the size of the input workload.

Considered case of Index Selection Problem can be defined in following way. Given is a set of tables:

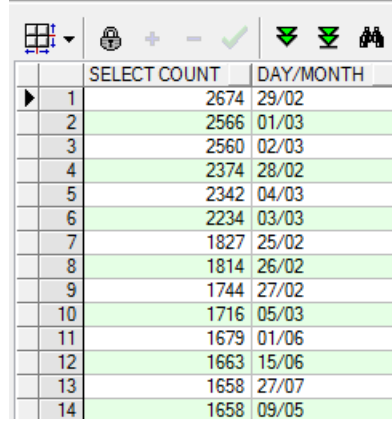
$$T = \{T_1, \dots, T_i, \dots, T_n\}, \quad (1)$$

described by a set of columns included in the tables:

$$K = \{k_{1,1}, \dots, k_{1,l(1)}, \dots, k_{i,j}, \dots, k_{n,1}, \dots, k_{n,l(n)}\}, \quad (2)$$

where:  $k_{i,j}$  is a  $j$ -th column of table  $T_i$ .

Each column  $k_{i,j}$  corresponds to set of values  $V(k_{i,j})$  (tuples set) included in this column.



	SELECT COUNT	DAY/MONTH
1	2674	29/02
2	2566	01/03
3	2560	02/03
4	2374	28/02
5	2342	04/03
6	2234	03/03
7	1827	25/02
8	1814	26/02
9	1744	27/02
10	1716	05/03
11	1679	01/06
12	1663	15/06
13	1658	27/07
14	1658	09/05

**Fig. 1. Example of number of database queries in a given day for a production data warehouse**

For the set of tables  $T$  various queries  $Q_i$  can be formulated (in SQL these are SELECT queries). These queries are put against the specified set of columns  $K_i^* \subseteq K$ . The result of query  $Q_i$  is set as:

$$A_i \subseteq \prod_{k_{i,j} \in K_i^*} V(k_{i,j}), \quad (3)$$

where:  $\prod_{i=1}^n Y_i = Y_1 \times Y_2 \times \dots \times Y_n$  is a cartesian product of sets  $Y_1, \dots, Y_n$ . For a given database  $DB$  it is taken into account that  $A_i$  is a result of following function:

$$A_i = Q_i(K_i^*, Op(DB)), \quad (4)$$

where:  $K_i^*$  is a subset of used columns,  $Op(DB)$  is set of operators available in database  $DB$  of which relation describing query  $Q_i$  is built.

The time associated with the determination of the set  $A_i$  is depended on the  $DB$  database used (search algorithms, indexes structures) and adopted set of indexes  $J \subseteq \mathcal{P}(K^*)$  (where  $\mathcal{P}(K^*)$  - is a power set of  $K_i^*$ ). It is therefore assumed that the query execution time  $Q_i$  in given database  $DB$ , is determined by the function:  $t(Q_i, J, DB)$ . In short the value of execution time for query  $Q_i$ , data base  $DB$  and set of indexes  $J$  will be define as:  $t_i(J)$ .

In the context of the so-defined parameters, a typical problem associated with the ISP responds to the question:

*What set of indexes  $J \subseteq \mathcal{P}(K_i^*)$  minimizes the query  $Q_i$  execution time:  $t_i(J) \rightarrow \min$  ?*

When a multi-component set of queries  $Q = \{Q_1, \dots, Q_m\}$  is considered, question takes the form:

*What set of indexes  $J \subseteq \mathcal{P}(K^*)$  minimizes the queries block  $Q$  execution time:  $\sum_{Q_i \in Q} t_i(J) \rightarrow \min$  ?*

### 3. CLASSIC INDEX SELECTION APPROACH

Classic index selection approach focuses on individual query and tries to find good index or indexes set for tables in a single query in a given block. Such approach does not take into consideration queries in a block as a whole. By doing so, a database user may expose database to create excess number of indexes which could be redundant or not used for more than one query in an examined block. This could also result in utilizing too much disk space and time needed for the indexes creation. Finding good index group for a large database queries' block was never an easy task to do and usually users and database administrators rely on their experience and good practice. In the commercial use one may find tools that support the index selection process, such as SQL Access Advisor (Fig. 2) [6], Toad, SQL Server Database Tuning Advisor [1].

Let us consider three examples where given is a group of three database queries  $Q = \{Q_1, Q_2, Q_3\}$ :

$Q_1$ : *SELECT \* FROM  $T_1, T_2$  WHERE  $k_{1,1} < k_{2,2}$  AND  $k_{1,3} = [const]$ ,*  
 $Q_2$ : *SELECT \* FROM  $T_2, T_3$  WHERE  $k_{2,2} = k_{3,2}$ ,*  
 $Q_3$ : *SELECT \* FROM  $T_2$  WHERE  $k_{2,1} > [const]$ .*

Interpretation of this type of queries (according to (4)) is as following:

$Q_1$ : searching for a set of triples:  $A_i = \{(a, b, c): a \in V(k_{1,1}), b \in V(k_{2,2}), c \in V(k_{1,3}); a < b, c = [const]\}$ ,  
 set  $K_1^* = \{k_{1,1}, k_{2,2}, k_{1,3}\}$ .

$Q_2$ : searching for a set of pairs:  $A_i = \{(a, b): a \in V(k_{2,2}), b \in V(k_{3,2}); a = b\}$ ,  
 set  $K_2^* = \{k_{2,2}, k_{3,2}\}$ .

$Q_3$ : searching for a set:  $A_i = \{a: a \in V(k_{2,1}); a = [const]\}$ ,  
 set  $K_3^* = \{k_{2,1}\}$ .

Tables  $T_1, T_2, T_3$  contain  $1 \cdot 10^6$  records each. No indexes are built on either table:  $J = \emptyset$ . With the first test run, database returned following response times:  $t_1(J) = 2040s, t_2(J) = 3611s, t_3(J) = 345s$  respectively, resulting in full table scans for each  $Q$ . Queries  $Q$  ran on database Oracle 11.2.0.1 installed on server with Redhat 6 operating system with 64GB memory and ASM used for disk storage.

The classic approach requires treating every database query individually. Hence indexes are built:  $k_{1,1}$  and  $k_{1,3}$  on table  $T_1$ ;  $k_{2,1}, k_{2,2}$  on table  $T_2$ ;  $k_{3,2}$  on table  $T_3$ . This kind of indexes are represented by the set:  $J = \{\{k_{1,1}, k_{1,3}\}, \{k_{2,2}\}, \{k_{3,2}\}, \{k_{2,1}\}\}$  containing four sets. Each element (set) of  $J$  contains the columns which are used to build the indexes. For example, the set  $\{k_{1,1}, k_{1,3}\}$  means that we have to build one index for columns  $k_{1,1}, k_{1,3}$ .

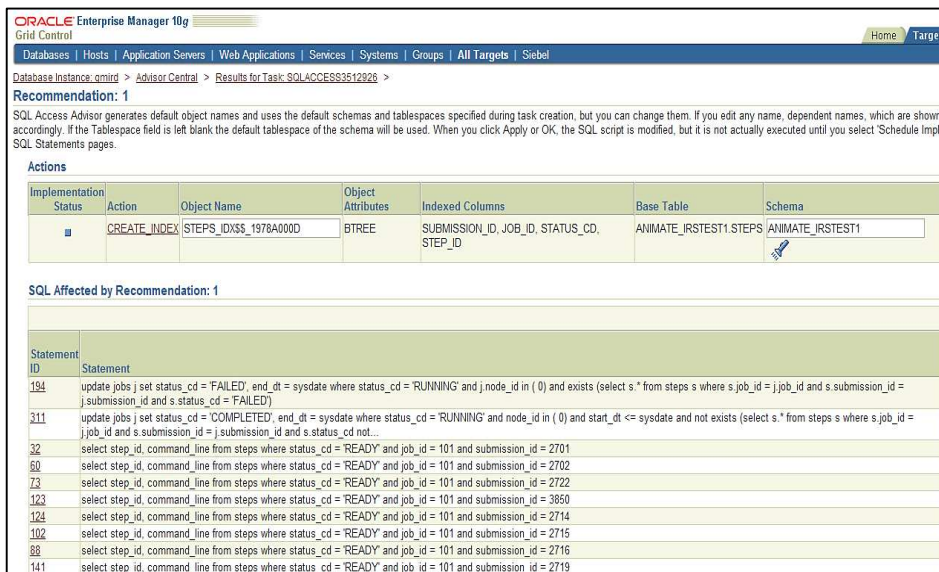


Fig. 2. Oracle's 10g2 SQL Access Advisor

The set of indexes  $J$  is built for three different tables, resulting in use of 2GB of additional disk space. With the second test run, database returned following response times:  $t_1(J) = 2612s, t_2(J) = 2580s, t_3(J) = 5s$  respectively. As the response time is better by approximately 10%, there is still unreasonable disk space used and time needed for creating 4 large indexes. Creating 4 indexes forced query optimizer to use them, and instead of decreasing  $Q_1$  execution time, it got increased. This is because optimizer decided to read  $k_{1,1}$  column index

content first and because it couldn't find values for  $k_{1,3}$  column, it performed full table scan for table  $T_1$ . Examples shows that selected indexes may increase the query execution performance where in other cases may have the opposite effect.

#### 4. GROUPED QUERIES APPROACH

In this paper we focus on related queries group and because of this relation on the number of indexed columns. We take into account the search for a good index for the entire queries block. We propose a new approach by using multi-query SQL block selection. Such block consists tabular relations between queries, meaning that the number of tables columns used in previous query is present in other queries. The proposed approach could be an alternative to the classic index selection method, where one common index set can be found. Grouped queries approach has to be studied for its effectiveness and authenticity via a series of numerical tests. Furthermore, to compare the performance of the method we use commercial tools to compare results.

For previous examples, we suggest to create a pool of all columns taking part in all queries in a group and build sub-optimal indexes set for queried tables. Such task involves creating the weighted list that will include all the index candidate query-related columns and their number of occurrence in the examined queries block:

$$KW = ((k_{1,1}, 1), (k_{1,3}, 1), (k_{2,1}, 1), (k_{2,2}, 2), (k_{3,2}, 1)). \quad (5)$$

Of course, only  $k_{2,2}$  column (marked by the box in (5)) is a query-related candidate column that could be used for the index creation. Nevertheless, other columns from remaining tables could also be revised. In that context, we suggest to create composite index for the same table  $T_2$  on columns  $k_{2,1}$  and  $k_{2,2}$ :  $J = \{\{k_{2,1}, k_{2,2}\}\}$ . By doing so, user not only speeds up block execution but also saves significant volume of disk space. With the third test run, database returned following response times:  $t_1(J) = 1235s$ ,  $t_2(J) = 2430s$ ,  $t_3(J) = 5s$ , respectively, decreasing total execution time of 35% and saving disk space of 60%. This is due to the fact that only index is used or full table scan for non-indexed table resulting in smaller response times for  $Q_1$  and  $Q_2$ . Database optimizer does not need to perform an additional read operation (separate for index and if values not found and separate for a table). This proves that indexes should be selected with care.

Determining the answers to a set of queries can be improved by creating some indexes.

Classic index selection focuses on each query individually and final indexes set is a sum of indexes sub-sets for each query.

We show that groups of queries, one can get better indexes set if such group is treated as a whole.

Grouped queries index search can only benefit and have an advantage over single query search, only if queries in the group satisfy the condition of mutual dependence. Queries  $Q_1, Q_2, Q_3$ , from previous examples are dependent so below statement applies. Such dependency must be clearly defined.

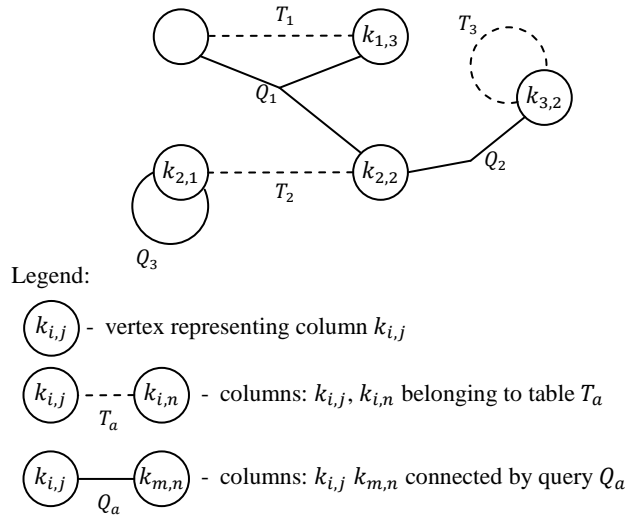
In the present case, the dependence set of queries  $Q$  is determined by connectivity of hypergraph  $G(Q)$ .

Example of a hypergraph for considered queries  $Q$  is presented on Fig. 3.

In this type of graph vertices represent the columns used in queries  $Q$ , edges connect those vertices which combined make table  $T_a$  (dashed line hyper edge) or related queries  $Q_i$  (solid line hyper edge). For example, hyper edge connecting vertices  $k_{1,1}, k_{2,2}, k_{1,3}$  represents relation with query  $Q_1$ .

**It is assumed that the query set  $Q$  is related if corresponding hypergraph  $G(Q)$  is consistent.**

In this context, the group queries indexes set creation can benefit compared to classic index selection only for related sets.



**Fig. 3. Hypergraph for considered set of queries  $Q$**



As a counterexample, given is a group of three database queries  $Q^* = \{Q_1^*, Q_2^*, Q_3^*\}$ :

$Q_1^*$ : *SELECT \* FROM T<sub>1</sub>, T<sub>2</sub> WHERE k<sub>1,1</sub> > k<sub>1,2</sub>,*  
 $Q_2^*$ : *SELECT \* FROM T<sub>2</sub>, T<sub>3</sub> WHERE k<sub>2,1</sub> = k<sub>3,2</sub>,*  
 $Q_3^*$ : *SELECT \* FROM T<sub>4</sub> WHERE k<sub>4,1</sub> > [const].*

Example of a hypergraph for considered queries  $Q^*$  is presented on Fig. 4. This kind of hypergraph presented is inconsistent. For this reason queries  $Q^*$  are treated as the unrelated queries.

Unrelated queries for index selection process means they cannot be treated as a group. In such cases best index set is a set determined for each query individually:

$$J^* = \{\{k_{1,1}, k_{1,2}\}, \{k_{2,1}\}, \{k_{3,2}\}, \{k_{4,1}\}\}. \quad (6)$$

Weighted list for  $Q^*$  that includes all the index candidate columns:

$$KW^* = ((k_{1,1}, 1), (k_{1,2}, 1), (k_{2,1}, 1), (k_{3,2}, 1), (k_{4,1}, 1)). \quad (7)$$

One can notice there are no query-related candidate columns (single column occurrence) that could be used for the grouped queries index set creation. Each table  $T_i$  will have to be indexed separately for each individual query  $Q^*$ .

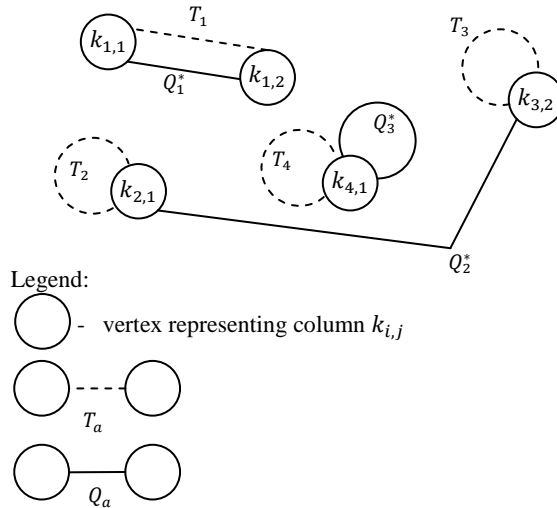


Fig. 4. Hypergraph for considered set of queries  $Q^*$

## 5. EXPERIMENTAL TESTS

In the previous section we show two examples where grouped queries approach may be beneficial for SQL blocks with related queries, which is blocks of queries that can be graphically represented by a consistent hypergraph (see Fig. 3.).

In such context, a question needs to be asked: how does the efficiency of obtained indexes (calculated as a response time for a given query set) depend on the degree of density of these types of hypergraphs?

In order to answer this question we carry out 3 experiments that involve index selection for 3 different queries blocks with changing hypergraph density degree.

Each of the analyzed query blocks:  $Q^1$ ,  $Q^2$ ,  $Q^3$  consists of three queries which characterize relations between columns of three database tables:  $T = \{T_1, T_2, T_3\}$ , containing  $10 \times 10^6$  rows each.

For experimental purposes we use Oracle database, version 10.2.0.3 installed on server with Redhat 6 operating system with 64GB memory and ASM used for disk storage.

Each of the queries blocks  $Q^1$ ,  $Q^2$ ,  $Q^3$  are presented (using the SQL language notation) in Tab. 1, Tab. 2, Tab. 3, respectively.

Database queries are constructed so that the corresponded hypergraph (presented in Fig. 5,6,7) has a varied density. It is assumed that the density  $\rho_i$  of a hypergraph describes common relations between queries of block  $Q^i$  and in example of a block with 3 queries, density is defined as follows:

$$\rho_i = \frac{|(K_{i,1}^* \cap K_{i,2}^* \cap K_{i,3}^*) \cup (K_{i,1}^* \cap K_{i,2}^*) \cup (K_{i,1}^* \cap K_{i,3}^*) \cup (K_{i,2}^* \cap K_{i,3}^*)|}{|K_{i,1}^* \cup K_{i,2}^* \cup K_{i,3}^*|}$$

, where:  $\rho_i \in [0,1]$  ,  $\rho_i = 0$  – describes no relations between queries in a block, and

$\rho_i = 1$  describes presence of each column in each query in a block.

$K_{i,j}^*$  is a subset of columns used in query  $Q_j^i$

Density of a hypergraph is calculated as a proportion of number of common columns to number of all columns used in the block queries. Densities of analyzed blocks from this experiment are as follows:

- block  $Q^1$  (Tab. 1):  $\rho_1 = 0$
- block  $Q^2$  (Tab. 2):  $\rho_2 = \frac{8}{14} = 0,57$
- block  $Q^3$  (Tab. 3):  $\rho_3 = \frac{11}{12} = 0,91$

The presented values should be interpreted as follows: density of a hypergraph of queries block  $Q^1$  is zero ( $\rho_1 = 0$ ), meaning there are no relations between queries (no relations). In turn, density of a hypergraph of queries block  $Q^3$  is 0,91, meaning relation between queries are very strong (high relation – density is close to 1).

Each of the three experimental queries blocks is examined by three different tests so that a good index group for each query block is found:

1. index selection with use of advisory tools,
2. classic index selection approach,
3. grouped queries index selection approach

In the first test we use 2 different index selection advisory tools. One is the Oracle SQL Access Advisor, provided together with the server database installation package. Another is TOAD package, developed by Quest company. Oracle's software has ability to search for indexes not only for individual queries but also for a queries block (SQL Tuning Set).

TOAD tool treats every SQL query within a group as an individual and indexes are selected individually, too.

For 2 other tests (classic and grouped queries approach) we use our own index selection adaptive algorithm.

The results for all 3 tests we carry out are shown below:

**Test 1:** For queries blocks with recommendations of index selection advisory tools, each block execution times are as follows:

- for block  $Q^1$  - 12s
- for block  $Q^2$  - 267s
- for block  $Q^3$  - 368s.

**Test 2:** For queries blocks with recommendations of classic index selection approach, each block execution times are as follows:

- for block  $Q^1$  - 3s
- for block  $Q^2$  - 253s
- for block  $Q^3$  - 320s.

**Test 3:** For queries blocks with recommendations of grouped queries index selection approach, each block execution times are as follows:

- for block  $Q^1$  - 3s
- for block  $Q^2$  - 245s
- for block  $Q^3$  - 289s.

Based on the above results, differences between advisory tools, classic and grouped queries approach for blocks execution times are calculated as follows:

- 0s for queries block with no relations (block  $Q^1$ ).
- 8s (3%) for queries block with low relations (block  $Q^2$ ).
- 31s (9.5%) for queries block with high relations (block  $Q^3$ ).

The obtained results show that together with the increase in queries' relations ( $\rho_i$  density increase), the efficiency of the grouped queries approach against classic index selection approach also increases. We don't notice efficiency increase for queries block with no relation (block  $Q^1$ ). Furthermore, for this block indexes developed from classical approach are identical to those with grouped queries index selection method (see Tab. 1).

It is worth noting that the commercial advisory tools seem to be useful only for non-related block queries  $Q^1$  ( $\rho_1 = 0$ ). For other queries blocks ( $Q^2, Q^3$ ) advisors are unable to recommend any indexes whatsoever (see Tab. 2, Tab. 3). As it seems, with block queries density increase the effectiveness of such tools decrease.

**Tab. 1. Database queries  $Q^1$  with no relations ( $\rho_1 = 0$ ) and indexes recommendations**

<p><i>Database queries set with no relations:</i></p> <p>Q<sub>1</sub>: SELECT T1_2.KOL4, T1_1.KOL5 FROM TEST1 T1_1, (SELECT KOL3, KOL4 FROM TEST1) T1_2 WHERE T1_1.KOL1 BETWEEN T1_1.KOL2 AND T1_2.KOL4 AND T1_2.KOL3 = 1234 GROUP BY T1_2.KOL4, T1_1.KOL5;</p> <p>Q<sub>2</sub>: SELECT TEST2.KOL1, TEST2.KOL4 FROM TEST2 WHERE TEST2.KOL4 &gt; 100 AND TEST2.KOL1 &lt; 100 AND TEST2.KOL3 &gt; ANY (SELECT TEST2.KOL3 FROM TEST2 WHERE TEST2.KOL2 &lt; 100) GROUP BY TEST2.KOL1, TEST2.KOL4 ORDER BY 2;</p> <p>Q<sub>3</sub>: SELECT KOL2, KOL4 FROM TEST3 WHERE KOL4 &lt; 1000 AND KOL1 IN (0,5,10) UNION ALL SELECT KOL2, KOL5 FROM TEST3 WHERE KOL2 &gt; 1000 AND KOL5 IN (1,10,100);</p>	<p><i>Oracle SQL Advisor + TOAD suggestions:</i></p> <p>CREATE INDEX k1_col3_col4_idx ON T<sub>1</sub>(k<sub>1,3</sub>, k<sub>1,4</sub>); CREATE INDEX k2_col1_col3_idx ON T<sub>2</sub>(k<sub>2,1</sub>, k<sub>2,3</sub>); CREATE INDEX k2_col2_col3_idx ON T<sub>2</sub>(k<sub>2,2</sub>, k<sub>2,3</sub>); CREATE INDEX k3_col1_col2_col4_idx ON T<sub>3</sub>(k<sub>3,1</sub>, k<sub>3,2</sub>, k<sub>3,4</sub>); CREATE INDEX k3_col2_col5_idx ON T<sub>3</sub>(k<sub>3,2</sub>, k<sub>3,5</sub>);</p> <p><i>Classic index selection approach:</i></p> <p>CREATE INDEX k1_col3_idx ON T<sub>1</sub>(k<sub>1,3</sub>); CREATE INDEX k2_col1_col2_col4_idx ON T<sub>2</sub>(k<sub>2,1</sub>, k<sub>2,2</sub>, k<sub>2,4</sub>); CREATE INDEX k2_col2_idx ON T<sub>2</sub>(k<sub>2,2</sub>); CREATE INDEX k3_col1_col4_idx ON T<sub>3</sub>(k<sub>3,1</sub>, k<sub>3,4</sub>); CREATE INDEX k3_col5_idx ON T<sub>3</sub>(k<sub>3,5</sub>);</p> <p><i>Grouped queries approach:</i></p> <p>CREATE INDEX k1_col3_idx ON T<sub>1</sub>(k<sub>1,3</sub>); CREATE INDEX k2_col1_col2_col4_idx ON T<sub>2</sub>(k<sub>2,1</sub>, k<sub>2,2</sub>, k<sub>2,4</sub>); CREATE INDEX k2_col2_idx ON T<sub>2</sub>(k<sub>2,2</sub>); CREATE INDEX k3_col1_col4_idx ON T<sub>3</sub>(k<sub>3,1</sub>, k<sub>3,4</sub>); CREATE INDEX k3_col5_idx ON T<sub>3</sub>(k<sub>3,5</sub>);</p>
---	---

**Tab. 2. Database queries  $Q^2$  for low relations ( $\rho_2 = 57$ ) and indexes recommendations**

<p><i>Database queries set with low relations:</i></p> <p>Q<sub>1</sub>: SELECT T3.KOL1,T3.KOL2 FROM TEST1 T1, (SELECT T2.KOL3, T2.KOL5 FROM TEST2 T2, TEST1 T1 WHERE T2.KOL3=T1.KOL5) T2, TEST3 T3 WHERE T1.KOL5 = T3.KOL4 AND T3.KOL1 = T2.KOL3 AND T3.KOL5 = ANY (SELECT T2.KOL5 FROM TEST2 T2, TEST1 T1 WHERE T2.KOL4=T1.KOL3) ORDER BY 1,2;</p> <p>Q<sub>2</sub>: SELECT DISTINCT T1.KOL , T1.KOL2 , COUNT(*) FROM TEST1 T1, TEST3 T3, (SELECT T2.KOL4, T2.KOL1 FROM TEST2 T2, TEST3 T3 WHERE T2.KOL3=T3.KOL5) T2 WHERE T1.KOL1 = T2.KOL1 AND T2.KOL4 = T3.KOL4 GROUP BY T1.KOL1, T1.KOL2 ORDER BY 1 DESC;</p> <p>Q<sub>3</sub>: SELECT DISTINCT T1.KOL2, T2.KOL5, COUNT(2) FROM TEST2 T2, TEST1 T1, TEST3 T3 WHERE T1.KOL4 = T3.KOL4 AND T1.KOL1 = T2.KOL3 AND T1.KOL5 &gt; ANY (SELECT T2.KOL5 FROM TEST2 T2 WHERE T2.KOL1=1000) AND (T3.KOL3 &gt; T2.KOL3) GROUP BY T1.KOL2, T2.KOL5 ORDER BY 1,2 DESC;</p>	<p><i>Oracle SQL Advisor + TOAD suggestion:</i></p> <p>NO INDEXES</p> <hr/> <p><i>Classic index selection approach:</i></p> <p>CREATE INDEX k1_col1_col2_idx ON T<sub>1</sub>(k<sub>1,1</sub>, k<sub>1,2</sub>); CREATE INDEX k1_col5_idx ON T<sub>1</sub>(k<sub>1,5</sub>);</p> <p>CREATE INDEX k2_col1_col3_idx ON T<sub>2</sub>(k<sub>2,1</sub>, k<sub>2,3</sub>); CREATE INDEX k2_col3_col4_idx ON T<sub>2</sub>(k<sub>2,3</sub>, k<sub>2,4</sub>); CREATE INDEX k2_col4_idx ON T<sub>2</sub>(k<sub>2,4</sub>);</p> <p>CREATE INDEX k3_col1_idx ON T<sub>3</sub>(k<sub>3,1</sub>); CREATE INDEX k3_col3_idx ON T<sub>3</sub>(k<sub>3,3</sub>); CREATE INDEX k3_col4_idx ON T<sub>3</sub>(k<sub>3,4</sub>);</p> <hr/> <p><i>Grouped queries approach:</i></p> <p>CREATE INDEX k1_col1_idx ON T<sub>1</sub>(k<sub>1,1</sub>);</p> <p>CREATE INDEX k2_col1_col3_col4_idx ON T<sub>2</sub>(k<sub>2,1</sub>, k<sub>2,3</sub>, k<sub>2,4</sub>);</p> <p>CREATE INDEX k3_col2_col4_idx ON T<sub>3</sub>(k<sub>3,2</sub>, k<sub>3,4</sub>);</p>
---	---

**Tab. 3. Database queries  $Q^3$  with high relations ( $\rho_3 = 0,91$ ) and indexes recommendations**

<p><i>Database queries set with high relations:</i></p> <p>Q<sub>1</sub>: SELECT COUNT(*) FROM TEST1 INNER JOIN TEST2 ON TEST1.KOL1 = TEST2.KOL2 AND TEST1.KOL2 = TEST2.KOL3 AND TEST1.KOL3 = TEST2.KOL4 INNER JOIN TEST3 ON TEST2.KOL2 = TEST3.KOL1 AND TEST2.KOL4 = TEST3.KOL3 AND TEST2.KOL5 = TEST1.KOL3;</p> <p>Q<sub>2</sub>: SELECT COUNT(*) FROM TEST1 INNER JOIN TEST2 ON TEST1.KOL1 = TEST2.KOL1 AND TEST1.KOL3 = TEST2.KOL3 AND TEST1.KOL2 = TEST2.KOL4 AND TEST2.KOL2 = TEST1.KOL2 INNER JOIN TEST3 ON TEST2.KOL1 = TEST3.KOL1 AND TEST2.KOL2 = TEST3.KOL2 AND TEST2.KOL3 = TEST3.KOL3 AND TEST2.KOL5 = TEST3.KOL5;</p> <p>Q<sub>3</sub>: SELECT COUNT(*) FROM TEST1 INNER JOIN TEST2 ON TEST1.KOL1 = TEST2.KOL5 AND TEST2.KOL3 = TEST1.KOL3 INNER JOIN TEST3 ON TEST2.KOL5 = TEST3.KOL1 AND TEST2.KOL1 = TEST3.KOL5 AND TEST3.KOL3 = TEST2.KOL3 AND TEST1.KOL2 = TEST3.KOL5;</p>	<p><i>Oracle SQL Advisor suggestion + TOAD suggestion:</i></p> <p>NO INDEXES</p> <hr/> <p><i>Classic index selection approach:</i></p> <p>CREATE INDEX k1_col1_col3_idx ON T<sub>1</sub>(k<sub>1,1</sub>, k<sub>1,3</sub>); CREATE INDEX k1_col2_idx ON T<sub>1</sub>(k<sub>1,2</sub>);</p> <p>CREATE INDEX k2_col1_col2_idx ON T<sub>2</sub>(k<sub>2,1</sub>, k<sub>2,2</sub>); CREATE INDEX k2_col3_col5_idx ON T<sub>2</sub>(k<sub>2,3</sub>, k<sub>2,5</sub>); CREATE INDEX k2_col4_idx ON T<sub>2</sub>(k<sub>2,4</sub>);</p> <p>CREATE INDEX k3_col1_idx ON T<sub>3</sub>(k<sub>3,1</sub>); CREATE INDEX k3_col3_idx ON T<sub>3</sub>(k<sub>3,3</sub>); CREATE INDEX k3_col5_idx ON T<sub>3</sub>(k<sub>3,5</sub>);</p> <hr/> <p><i>Grouped queries approach:</i></p> <p>CREATE INDEX k1_col1_col2_col3_idx ON T<sub>1</sub>(k<sub>1,1</sub>, k<sub>1,2</sub>, k<sub>1,3</sub>);</p> <p>CREATE INDEX k3_col1_col3_col5_idx ON T<sub>3</sub>(k<sub>3,1</sub>, k<sub>3,3</sub>, k<sub>3,5</sub>);</p>
---	---

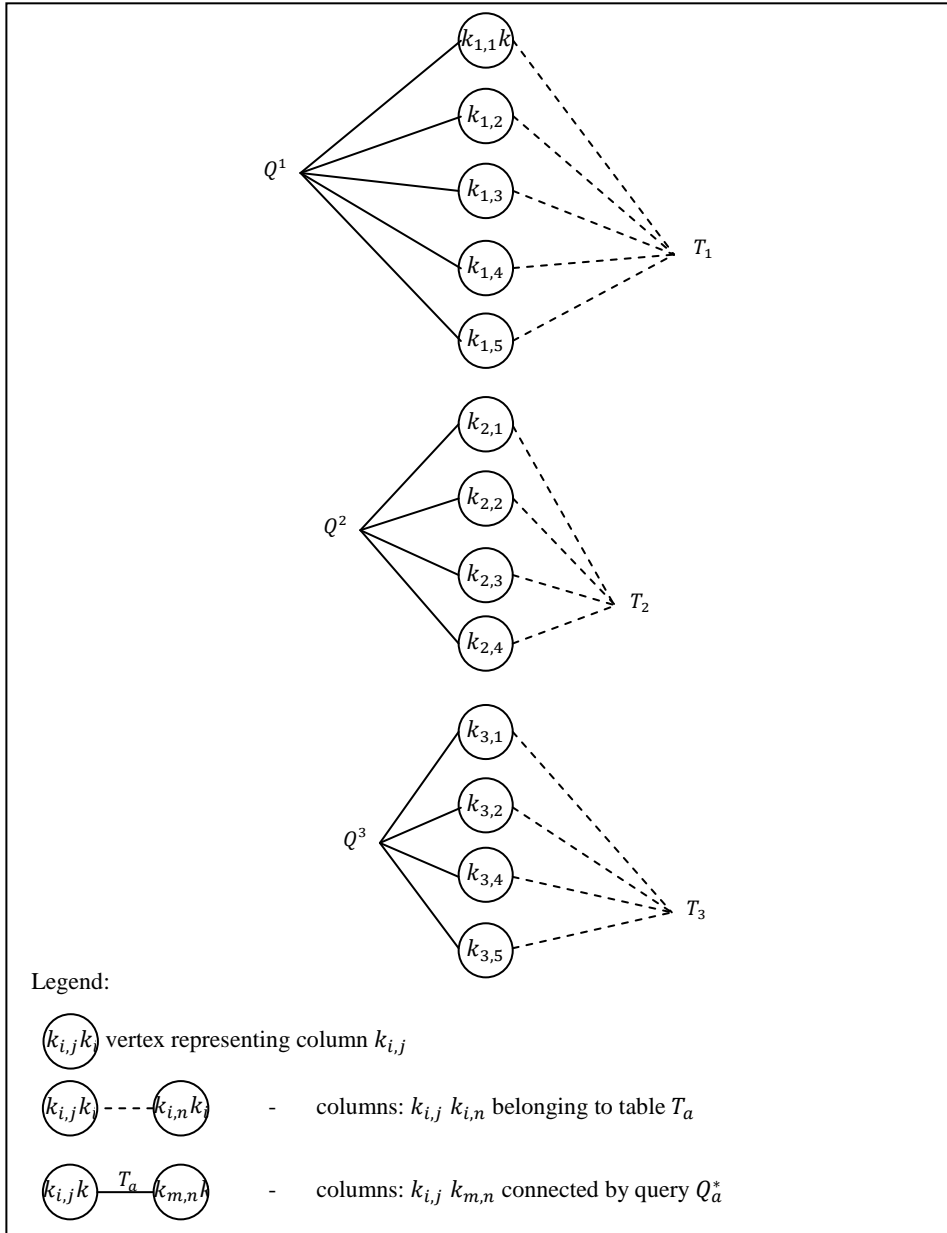


Fig. 5. Hypergraph for example set of queries  $Q^1$  with no relations:  $\rho_1 = 0$

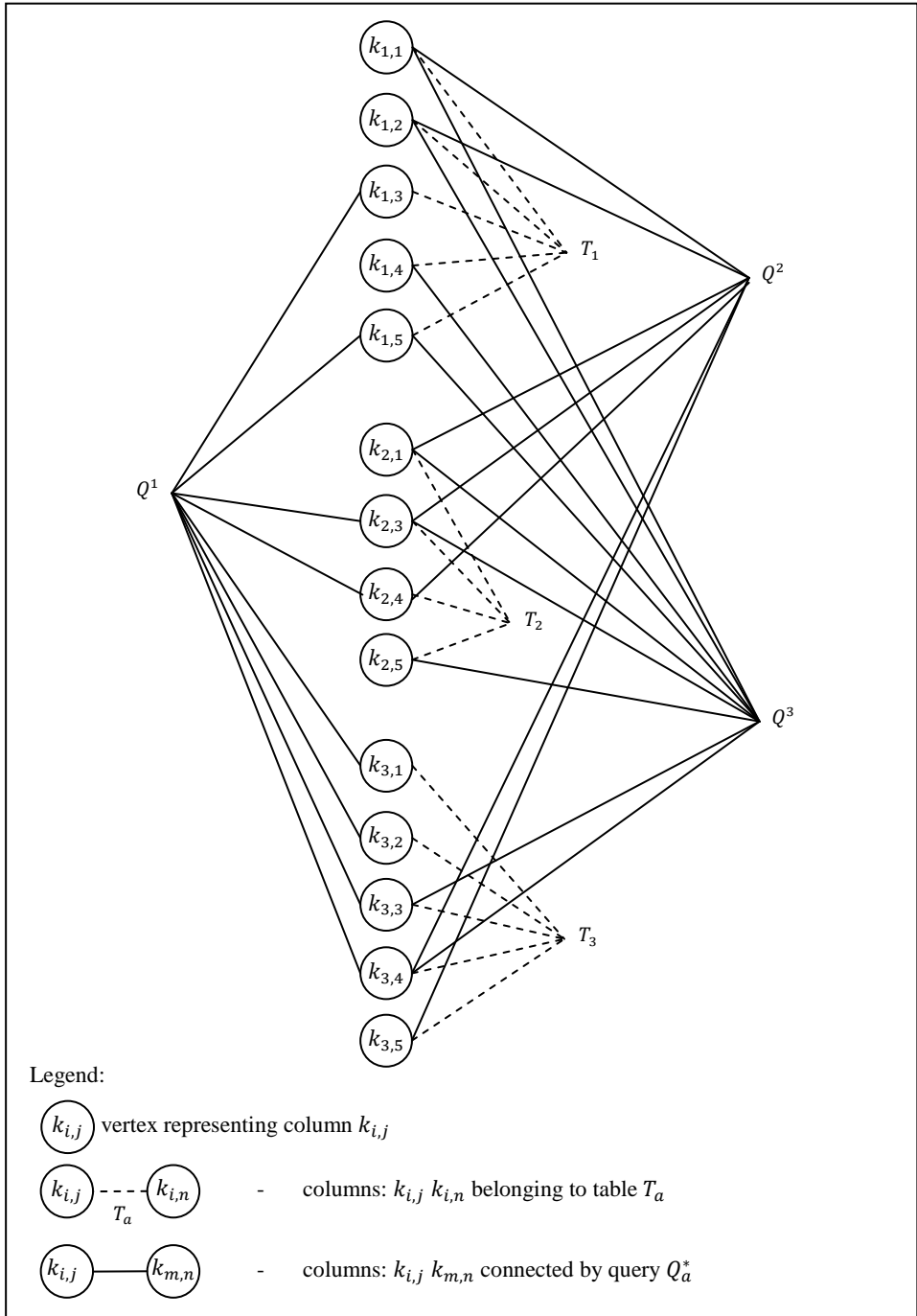


Fig. 6. Hypergraph for set of queries  $Q^2$  with low relations:  $\rho_2 = 0, 57$

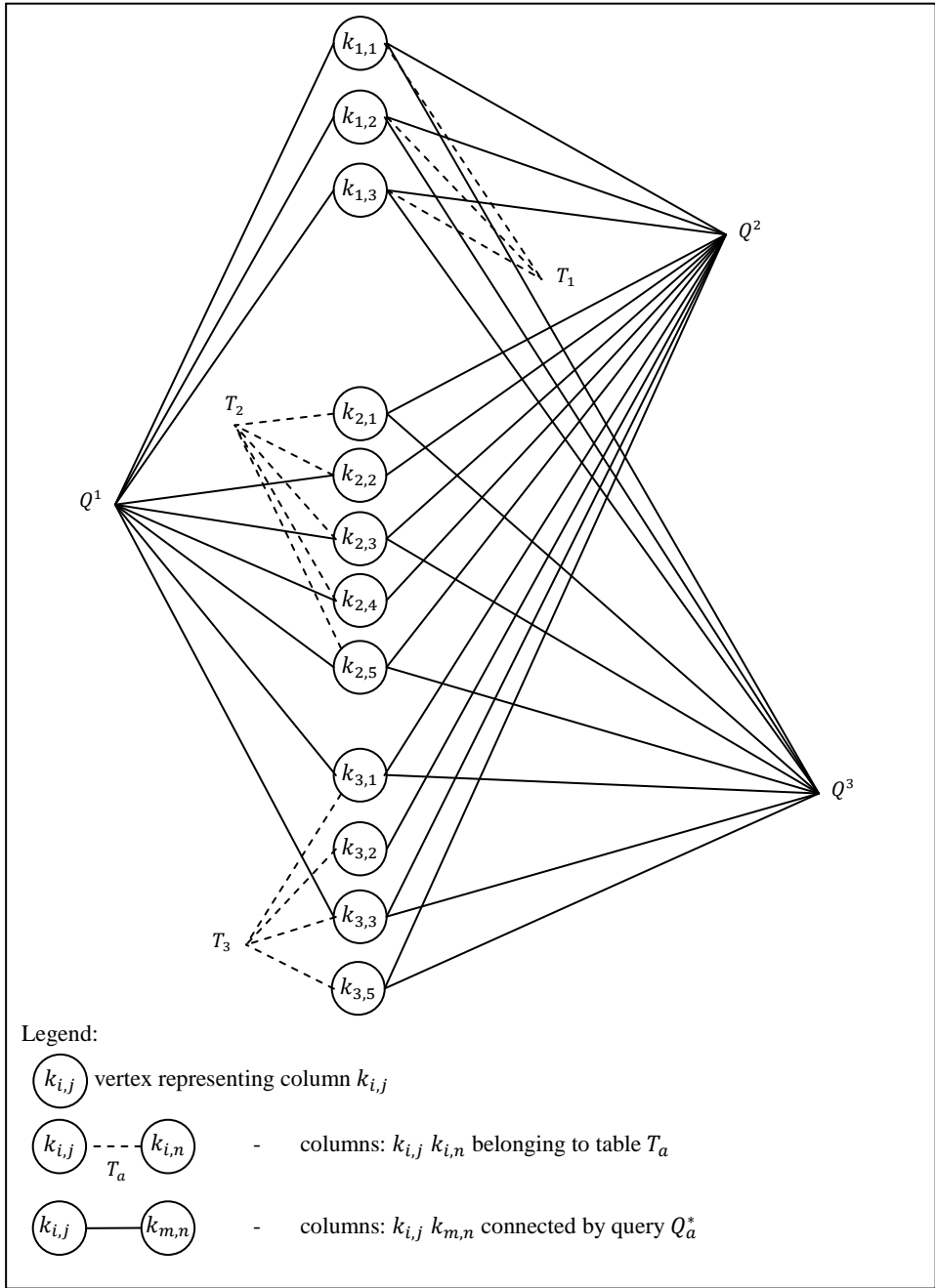


Fig. 7. Hypergraph for set of queries  $Q^3$  with concentrated relations:  $\rho_3 = 0, 91$



## 6. CONCLUSIONS

Finding a good index or indexes set for a table is very important for every relational database processing not only from the performance point but also cost aspect. Indexes can be crucial for a relational database to process queries with reasonable efficiency, but the selection of the best indexes is very difficult.

Presented examples show that there is a need for finding an automatic index selection mechanism with grouped queries-oriented rather than a classic (single query) approach for blocks with related queries. Practice shows that index focus on grouped queries gives better results and enables user to save time needed for index creation. It also saves system hardware resources. In the examples we show grouped queries indexes set are more effective than individual queries indexes because queries  $Q^2, Q^3$  satisfy the relation condition (see Tab. 2,3). For blocks with no related queries we show grouped queries indexes set are not more effective than individual queries indexes because queries  $Q^1$  do not satisfy the relation condition (see Tab. 1).

One should note that the experiments we carry out are to determine index sets that minimize queries blocks execution time only. What is important in the general case are different parameters such as: index creation cost, number of indexes and disk storage allocation. Future research will take into account the resources needed to create an index and storage resources.

For the automatic index selection, the system continuously monitors queries block and gathers information on columns used in queries. The administrator (or user) can summon the automatic system at any time to be presented with the current index recommendation, or tune it to the queries block needs. The system also presents the user index set and allows user to choose best option. User decides whether to reject or accept proposed set. Due to index interactions, the user's decisions might affect other indexes in the configuration, so the recommendation would need to be regenerated, taking the user's constraints into account.

In the presented examples we show three situations of database queries block execution, one without indexes, one with classic separate queries indexing and one with grouped queries indexing. Examples showed that one should create grouped indexes only for related queries. In that context presented relationship may be treated as sufficient condition for the evaluation of grouped queries indexing.

Our current works are focused on grouped queries index selection method with the use of genetic algorithm [2] that analyzes database queries, suggests indexes' structure and tracks indexes influence on the queries' execution time. We work on the system that will be used in an attempt to find better indexes for a critical part of long-running database queries in testing and production

database environment. Recording queries with good indexes together with their total execution time is a starting point for broader searches in the future. Simple test presented in this article proves effectiveness of this method. The developed system is scalable: there is a potentiality of combining smaller queries' blocks into larger series and finding better solution based on execution history.

## REFERENCES

- [1] AGRAWAL S., CHAUDHURI S., KOLLAR L., MARATHE A., NARASAYYA V., and SYAMALA M.. Database Tuning Advisor for Microsoft SQL Server 2005. In Proceedings of the 30th International Conference on Very Large Databases, 2004.
- [2] BACK T., "Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms", Oxford University Press Oxford, UK, 1996.
- [3] BRUNO N., CHAUDHURI S., "Automatic physical database tuning: a relaxation-based approach", SIGMOD '05 Proceedings of the 2005 ACM SIGMOD international conference on Management of data, ACM New York, NY, USA, 2005, pp.227-238.
- [4] CHAUDHURI S., NARASAYYA V., "An efficient Cost-Driven Index Selection Tool for MS SQL Server", Very Large Data Bases Endowment Inc, 1997.
- [5] COMERS D., "The Ubiquitous B-Tree", Computing Surveys 11 (2), doi:10.1145/356770.356776, pp. 123-137.
- [6] DAGEVILLE B., DAS D., DIAS K., YAGOUB K., ZAIT M., and ZIAUDDIN M.. "Automatic SQL Tuning in Oracle 10g". In Proceedings of the 30th International Conference on Very Large Databases, 2004.
- [7] DAWES C., BRYLA B., JOHNSON J., WEISHAN M., "OCA Oracle 10g Administration I", Sybex, 2005, pp.173.
- [8] FINKELSTEIN S., SCHKOLNICK M., TIBERIO P., "Physical database design for relational databases", ACM Trans. Database Syst. 13(1), (1988), pp.91-128.
- [9] FRANK M., OMIECINSKI M., "Adaptive and Automated Index Selection in RDBMS", Proceedings of EDBT, 1992.
- [10] GUPTA H., HARINARAYAN V., RAJARAMAN A., and ULLMAN J. D., "Index Selection for OLAP", In Proceedings of the Internatoinal Conference on Data Engineering, Birmingham, U.K., April 1997, p. 208-219.
- [11] KNUTH D., "The Art of Computer Programming", vol. 3, Sorting and Searching. Addison-Wesley, Reading, Mass., 1973.
- [12] KNUTH D., "Sorting and Searching, The Art of Computer Programming", Volume 3 (Second ed.), Addison-Wesley.
- [13] KOŁACZKOWSKI P., RYBIŃSKI H., "Automatic Index Selection in RDBMS by Exploring Query Execution Plan Space", Studies in Computational Intelligence, vol. 223, Springer, 2009, pp.3-24
- [14] KRATICA J., LJUBIC I., TOSIC D., "A Genetic Algorithm for the Index Selection Problem", EvoWorkshops'03 Proceedings of the 2003 international conference on Applications of evolutionary computing, 2003.
- [15] LEHMAN P.L., "Efficient locking for concurrent operations on B-trees", ACM Transactions on Database Systems (TODS), Volume 6 Issue 4, Dec. 1981, pp.650-670.
- [16] MAGGIE Y., IP L., SAXTON L. V., and VIJAY RAGHAVAN V., "On the Selection of an Optimal Set of Indexes", IEEE Transactions on Software Engineering, 9(2), March 1983, p.135-143.
- [17] SCHKOLNICK M., "The Optimal Selection of Indices for Files", Information Systems, V.1, 1975.

- [18] SCHNAITTER K., "On-line Index Selection for Physical Database Tuning", ProQuest, UMI Dissertation Publishing, 2011.
- [19] TATHAM S., "Counted B-Trees", <http://www.chiark.greenend.org.uk/~sgtatham/algorithms/cbtree.html>.
- [20] WEDEKIND H., "On the selection of access paths in a data base system. In Data Base Management", KLIMBIE J.W. and KOFFEMAN K.L., Eds. North-Holland, Amsterdam, 1974, pp. 385-397.