

Grouped Queries Indexing For Relational Database

Radosław Boroński

Dept. of Electronics and Computer Science
Koszalin University of Technology
Koszalin, Poland
radoslaw.boronski@tu.koszalin.pl

Grzegorz Bocewicz

Dept. of Electronics and Computer Science
Koszalin University of Technology
Koszalin, Poland
bocewicz@ie.tu.koszalin.pl

Robert Wójcik

Institute of Computer Engineering, Control and Robotics
Wrocław University of Technology
Wrocław, Poland
robert.wojcik@pwr.wroc.pl

Abstract-This paper discusses the problem of minimizing the response time for a given database workload by a proper choice of indexes. We propose to look at the database queries as a group and search for good indexes for the group instead of an individual query. We present condition for applying the concept of grouped queries index selection. Such condition is illustrated by three practical examples.

Keywords-database;index;ISP;grouped queries;related queries

I. 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.

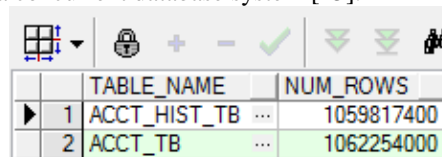
Indexes are optional data structures built on tables. Indexes can improve data retrieval performance by providing a direct access method instead of the default full table scan retrieval method [7]. In the simple case, each query can be answered either without using any index, in a given answer time or 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. Two sample tables in a data warehouse of an international automobile factory contain over 1 billion records each (Fig. 1). 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:

- 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].



	TABLE_NAME	NUM_ROWS
1	ACCT_HIST_TB ...	1059817400
2	ACCT_TB ...	1062254000

Figure 1. Example of large number of rows for two data warehouse tables

While the selection of indexes structure have a very important role in the design of database management tools so far avoided interference in the structure of indexes at the stage of the database 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].

The rest of the paper is organized as follows: in Section II we describe a problem statement. In Section III, we briefly present classic index selection approach together with simple examples that will illustrate the subject. In Section IV, we demonstrate new method of grouped queries index selection and compare examples results with the classic approach. Section V and VI present our conclusions and future works.

II. 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 solutions 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. 2) 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

Figure 2. Example of number of database queries in a given day for a production data warehouse

For 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 $K^* \subseteq K$. The result of query Q_i is set as:

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

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:

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

where: K^* is a subset of available indexes, $Op(DB)$ 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 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^*). 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 $J \subseteq \mathcal{P}(K^*)$ 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: $\sum_{Q_i \in Q} t_i(J) \rightarrow \min$?

III. 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. 3) [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 T₁, T₂ WHERE k_{1,1} < k_{2,2} AND k_{1,3}=[const],*
 Q_2 : *SELECT * FROM T₂, T₃ WHERE k_{2,2} = k_{3,2},*
 Q_3 : *SELECT * FROM T₂ 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 $K^* = \{k_{1,1}, k_{2,2}, k_{1,3}\}$.
 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 $K^* = \{k_{2,2}, k_{3,2}\}$.
 Q_3 : searching for a set: $A_i = \{a : a \in V(k_{2,1}); a = [const]\}$,
 set $K^* = \{k_{2,1}\}$.

Tables T_1, T_2, T_3 contain $1 * 10^6$ 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 screenshot shows the Oracle Enterprise Manager 10g interface. At the top, it says 'ORACLE Enterprise Manager 10g Grid Control'. Below that is a navigation bar with 'Home' and 'Targets'. The main content area shows 'Database Instance: qmird > Advisor Central > Results for Task: SQLACCESS3512926 >'. A 'Recommendation: 1' is displayed, with a description: 'SQL Access Advisor generates default object names and uses the default schemas and tablespaces specified during task creation, but you can change them. If you edit any name, dependent names, which are shown accordingly. If the Tablespace field is left blank the default tablespace of the schema will be used. When you click Apply or OK, the SQL script is modified, but it is not actually executed until you select 'Schedule Implementation' pages.'

Under 'Actions', there is a table with the following data:

Implementation Status	Action	Object Name	Object Attributes	Indexed Columns	Base Table	Schema
■	CREATE INDEX	STEPS_IDX\$\$_1978A000D	BTREE	SUBMISSION_ID, JOB_ID, STATUS_CD, STEP_ID	ANIMATE_IRSTEST1.STEPS	ANIMATE_IRSTEST1

Below the table, it says 'SQL Affected by Recommendation: 1'. There is a table with 'Statement ID' and 'Statement' columns:

Statement ID	Statement
194	update jobs j set status_cd = 'FAILED', end_dt = sysdate where status_cd = 'RUNNING' and j.node_id in (0) and exists (select s.* from steps s where s.job_id = j.job_id and s.submission_id = j.submission_id and s.status_cd = 'FAILED')
311	update jobs j set status_cd = 'COMPLETED', end_dt = sysdate where status_cd = 'RUNNING' and node_id in (0) and start_dt <= sysdate and not exists (select s.* from steps s where s.job_id = j.job_id and s.submission_id = j.submission_id and s.status_cd not...)
32	select step_id, command_line from steps where status_cd = 'READY' and job_id = 101 and submission_id = 2701
60	select step_id, command_line from steps where status_cd = 'READY' and job_id = 101 and submission_id = 2702
73	select step_id, command_line from steps where status_cd = 'READY' and job_id = 101 and submission_id = 2722
123	select step_id, command_line from steps where status_cd = 'READY' and job_id = 101 and submission_id = 3850
124	select step_id, command_line from steps where status_cd = 'READY' and job_id = 101 and submission_id = 2714
102	select step_id, command_line from steps where status_cd = 'READY' and job_id = 101 and submission_id = 2715
88	select step_id, command_line from steps where status_cd = 'READY' and job_id = 101 and submission_id = 2716
141	select step_id, command_line from steps where status_cd = 'READY' and job_id = 101 and submission_id = 2719

Figure 3. Oracle's 10g2 SQL Access Advisor

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 $k_{1,1}, k_{1,3}$.

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.

IV. GROUPED QUERIES APPROACH

In this paper we focus on related queries group and because of this relation and 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 could 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 commercial tools will have to be used and results compared.

For previous examples, we suggested 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 will involve creating the weighted list that will include all the index candidate query-related columns and their number of occurrence in the examined queries block:

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

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 subsets 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. 4.

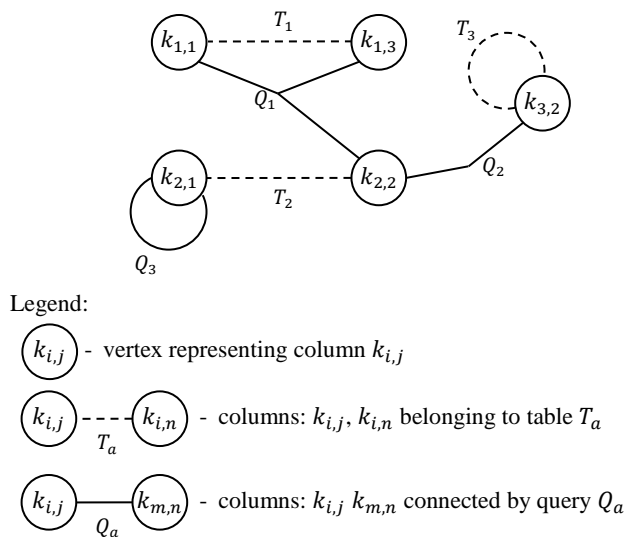


Figure 4. Hypergraph for considered set of queries Q

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.

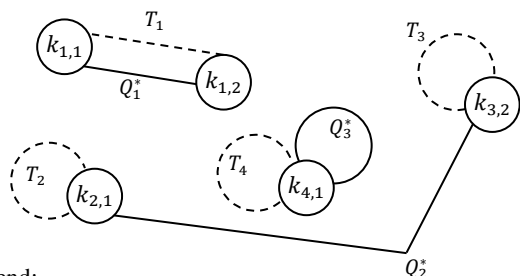
As a counterexample, given is a group of three database queries $Q^* = \{Q_1^*, Q_2^*, Q_3^*\}$:

Q_1^* : *SELECT * FROM T₁, T₂ WHERE k_{1,1} > k_{1,2},*
 Q_2^* : *SELECT * FROM T₂, T₃ WHERE k_{2,1} = k_{3,2},*
 Q_3^* : *SELECT * FROM T₄ WHERE k_{4,1} > [const].*

Example of a hypergraph for considered queries Q^* is presented on Fig. 5. 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:

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



Legend:

- $(k_{i,j})$ - vertex representing column $k_{i,j}$
- $(k_{i,j}) \text{ --- } (k_{i,n})$ - columns: $k_{i,j}$ $k_{i,n}$ belonging to table T_a
- $(k_{i,j}) \text{ --- } (k_{m,n})$ - columns: $k_{i,j}$ $k_{m,n}$ connected by query Q_a^*

Figure 5. Hypergraph for considered set of queries Q^*

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^* .

V. CONCLUSION

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 shows that there is a need for finding an automatic index selection mechanism with grouped queries-oriented rather than a classic (single query) approach. 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 that grouped queries indexes set are more effective than individual queries indexes because queries Q_1, Q_2, Q_3 satisfy the relation condition (Table 1).

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 considered 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.

TABLE I. CLASSIC AND GROUPED QUERIES APPROACH FOR CORRELATED DATABASE QUERIES

<p><i>Database queries:</i></p> <p>Q_1: <i>SELECT * FROM T₁, T₂ WHERE k_{1,1} < k_{2,2} AND k_{1,3}=[const];</i></p> <p>Q_2: <i>SELECT * FROM T₂, T₃ WHERE k_{2,2} = k_{3,2};</i></p> <p>Q_3: <i>SELECT * FROM T₂ WHERE k_{2,1} > [const];</i></p>	<p><i>Classic approach:</i></p> <p>CREATE INDEX k1_col1_idx ON T₁(k_{1,1}); CREATE INDEX k1_col3_idx ON T₁(k_{1,3}); CREATE INDEX k2_col1_idx ON T₂(k_{2,1}); CREATE INDEX k2_col2_idx ON T₂(k_{2,2}); CREATE INDEX k3_col2_idx ON T₃(k_{3,2});</p> <hr/> <p><i>Grouped queries approach:</i></p> <p>CREATE INDEX k2_col1_col2_idx ON T₂(k_{2,1}, k_{2,2});</p>
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

VI. FUTURE WORK

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 reasonableness 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] S. Agrawal, S. Chaudhuri, L. Kollar, A. Marathe, V. Narasayya, and M. Syamala, "Database Tuning Advisor for Microsoft SQL Server 2005". In Proceedings of the 30th International Conference on Very Large Databases, 2004.
- [2] T. Back, "Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms", Oxford University Press Oxford, UK, 1996.
- [3] N. Bruno and S. Chaudhuri, "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] S. Chaudhuri and V. Narasayya, "An efficient Cost-Driven Index Selection Tool for MS SQL Server", Very Large Data Bases Endowment Inc, 1997.
- [5] D. Comers, "The Ubiquitous B-Tree", Computing Surveys 11 (2), doi:10.1145/356770.356776, pp. 123-137.
- [6] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zait, and M. Ziauddin, "Automatic SQL Tuning in Oracle 10g". In Proceedings of the 30th International Conference on Very Large Databases, 2004.
- [7] C. Dawes, B. Bryla, J. Johnson, and M Weishan, "OCA Oracle 10g Administration I", Sybex, 2005, pp.173.
- [8] S. Finkelstein, M. Schkolnick, and P. Tiberio, "Physical database design for relational databases", ACM Trans. Database Syst. 13(1), (1988), pp.91-128.
- [9] M. Frank and M. Omiecinski, "Adaptive and Automated Index Selection in RDBMS", Proceedings of EDBT, 1992.
- [10] H. Gupta, V. Harinarayan, A. Rajaraman, and J. D. Ullman, "Index Selection for OLAP", In Proceedings of the Internatoinal Conference on Data Engineering, Birmingham, U.K., April 1997, p. 208-219.
- [11] D. Knuth, "The Art of Computer Programming", vol. 3, Sorting and Searching. Addison- Wesley, Reading, Mass., 1973.
- [12] D. Knuth, "Sorting and Searching, The Art of Computer Programming", Volume 3 (Second ed.), Addison-Wesley.
- [13] P. Kołaczowski and H. Rybiński, "Automatic Index Selection in RDBMS by Exploring Query Execution Plan Space", Studies in Computational Intelligence, vol. 223, Springer, 2009, pp.3-24
- [14] J. Kratica, I. Ljubic, and D. Tomic, "A Genetic Algorithm for the Index Selection Problem", EvoWorkshops'03 Proceedings of the 2003 international conference on Applications of evolutionary computing, 2003.
- [15] P.L. Lehman, "Efficient locking for concurrent operations on B-trees", ACM Transactions on Database Systems (TODS), Volume 6 Issue 4, Dec. 1981, pp.650-670.
- [16] Y. Maggie, L. Ip, L. V. Saxton, and Vijay V. Raghavan, "On the Selection of an Optimal Set of Indexes", IEEE Transactions on Software Engineering, 9(2), March 1983, p.135-143.
- [17] M. Schkolnick, "The Optimal Selection of Indices for Files", Information Systems, V.1, 1975.
- [18] K. Schnaitter, "On-line Index Selection for Physical Database Tuning", ProQuest, UMI Dissertation Publishing, 2011.
- [19] S. Tatham, "Counted B-Trees", <http://www.chiark.greenend.org.uk/~sgtatham/algorithms/cbtree.html>, 11.02.2013.
- [20] H. Wedekind, "On the selection of access paths in a data base system. In Data Base Management", J.W. Klimbie and K.L. Koffeman, Eds. North-Holland, Amsterdam, 1974, pp. 385-397.