Automatic Index Selection in RDBMS by Exploring Query Execution Plan Space [⋆]

Piotr Kołaczkowski, Henryk Rybiński

Abstract A novel approach to solving Index Selection Problem (ISP) is presented. In contrast to other known ISP approaches, our method searches the space of possible query execution plans, instead of searching the space of index configurations. An evolutionary algorithm is used for searching. The solution is obtained indirectly as the set of indexes used by the best query execution plans. The method has important features over other known algorithms: (1) it converges to the optimal solution, unlike greedy heuristics, which for performance reasons tend to reduce the space of candidate solutions, possibly discarding optimal solutions; (2) though the search space is huge and grows exponentially with the size of the input workload, searching the space of the query plans allows to direct more computational power to the most costly plans, thus yielding very fast convergence to "good enough" solutions; and (3) the costly reoptimization of the workload is not needed for calculating the objective function, so several thousands of candidates can be checked in a second. The algorithm was tested for large synthetic and real-world SQL workloads to evaluate the performace and scalability.

1 Introduction

Relational Database Management Systems (RDBMS) have been continuously developed for more than three decades now and became very complex. To administer them, much experience and knowledge is required. The costs of employing professional database administrators are often much higher than the costs of database

Piotr Kołaczkowski
Warsaw University of Technology, e-mail: pkolaczk@ii.pw.edu.pl

Henryk Rybiński
Warsaw University of Technology, e-mail: hrb@ii.pw.edu.pl

⋆ The work has been granted by Polish Ministry of Education (grant No 3T11C 002 29)

software licensing [12]. The total administration costs are especially large for large databases containing hundreds of tables and executing millions of queries a day. Recently we observe a high demand on solutions reducing these costs. Especially intelligent, automatic tools for solving complex administration problems are very helpful. One of such complex problems is performance tuning. In this paper we consider the aspect of proper index selection, which often significantly affects the overall database application performance. The importance of proper index selection increases with the size of a database.

Indexes are used by the RDBMS to accelerate query processing. Below we illustrate some cases, where they are especially useful:

- A query fetches only a small fraction of tuples stored in the database, determined by predicates with small selectivity:

```
SELECT * FROM person WHERE person_id = 12345;
SELECT * FROM person WHERE age > 100;
```

- Tuples should be returned in the same order as an order defined by an index. For example if a B+ tree index on the column `birth_date` is present, it can be potentially used for executing the query:

```
SELECT * FROM person ORDER BY birth_date LIMIT 10;
```

  Using indexes for ordering tuples may also be sane when the query requires sorting as a preparatory step, e.g. before joining relations, or grouping and aggregating tuples.
- A query requires joining two relations - a small one with a huge one. Then it may benefit from an index on the primary or foreign key of the latter one:

```
SELECT * FROM person p
  JOIN departement d ON (p.dept_id = d.dept_id)
  WHERE person_id = 12345;
```

- A query processes a subset of columns that is totally contained in the index, so that accessing the table can be avoided. Index-only scans are usually much faster than table scans, because indexes are usually smaller than tables. Besides, the physical order of tuples is rarely the same as the order of tuples in the index, so avoiding the table access also reduces large amount of costly random block fetches.
- A query is best handled by some dedicated kind of index, e.g. it contains a full-text-search, spacial search, skyline computation etc. These cases require some extensions to the SQL and are not covered by this paper, though the presented methods can be easily extended to support these cases.

There can be usually more than one execution plan available for a single query. These different plans may use different indexes. Each plan may use more than one index at a time, but also a single index may be used in several plans. The physical ordering of rows in the table often affects gains the application has from using a given index, so some database systems enable creation of the so called *correlated* or *clustered* indexes, which force the table rows to have the same ordering as the ordering of the index. There can be at most one clustered index per table.

One should note, however, that indexes induce an extra maintenance cost, as their existence may slow down inserting, deleting and updating the data. Additionally, the chosen indexes require some storage space, which may be limited. Therefore one can pose the following problem:

Find such an index set, that minimizes the cost of processing of the input workload and that requires less storage space than the specified limit.

The problem is known in the literature as *Index Selection Problem* (ISP). According to [10] 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 a way that a significant amount of storage space remains free even if the limit were totally used. Therefore, slightly exceeding the storage space limit is usually acceptable if only this provides strong performance improvements.

The ISP problem resembles the well known *Knapsack Problem* (KP). However, it is more complex, because the performance gain of a candidate index depends on the other indexes in the index configuration, while in the traditional KP the values and weights of items are independent of each other. Additionally, the process of calculating the total gain of the given candidate index configuration is computationally costly. Actually, given a candidate index configuration, it is necessary to calculate the optimal plan and its cost for each query in the input workload. To cope with the complexity, most researchers concentrated on efficient greedy heuristics that select a small subset of possibly useful index candidates in the first phase and then use some kind of heuristic search strategy to find a good index configuration within the specified space limit in the second phase [3, 8, 21, 24, 25, 26]. The essential disadvantage of this two-phase process is that by aggressive pruning the result space beforehand, one may remove the optimal solution and possibly also some other good solutions from the searched space. It is illustrated in Fig. 1, where the optimal solution denoted by the black triangle is located outside the searched space. To this end, we concentrate in the paper on finding a method that converges to the global optimum in a continuous, one-phase process.

Let us note that the index configuration being the global optimal solution of ISP determines the optimal set of the query execution plans, obtained for the input query workload. Also, the globally optimal set of plans determines the optimal index configuration. Thus we can formulate a *dual problem to ISP*:

Among all possible sets of query execution plans for the given workload, find the one that minimizes the sum of query processing times plus the sum of maintenance times of used indexes and utilizes the indexes of a total size that does not exceed the specified storage space limit.

The method proposed in the paper solves the dual ISP problem, and gets the final index configuration from its result set of query plans. Because the search space is extremely large even for a small number of queries, an evolution strategy is used to drive the search. The approach is illustrated in Fig. 2. Here, the optimizing process consists in searching the space of plan vectors (by means of an evolution strategy). Each candidate plan vector has the relevant index configuration directly assigned. To this end, in addition to a good global query plan, a set of picked indexes is returned as a result. Our algorithm proposes also clustered indexes and takes index maintenance costs into account.

Evolutionary strategies and genetic algorithms were recognized as feasible to solving ISP [11]. It was also shown that they can be superior to some other heuristic search strategies for this application [18]. However, the previous works concentrated on searching the index space, rather than the plan space, and this disallowed to focus more on the optimization of costly query plans. We believe that focusing on the most costly plans is crucial for achieving high performance of the index selection tool and makes it possible to solve larger problems. The evolutionary strategy approach has also an advantage over other heuristic search strategies, such as simulated annealing [15], tabu-search [13] or hill-climbing [23], by the fact that the fitness function and the problem and solution spaces may dynamically change while the problem is being solved and, if only the mutation intensity is high enough, the population will follow the changes and converge to the new solution without the need to restart

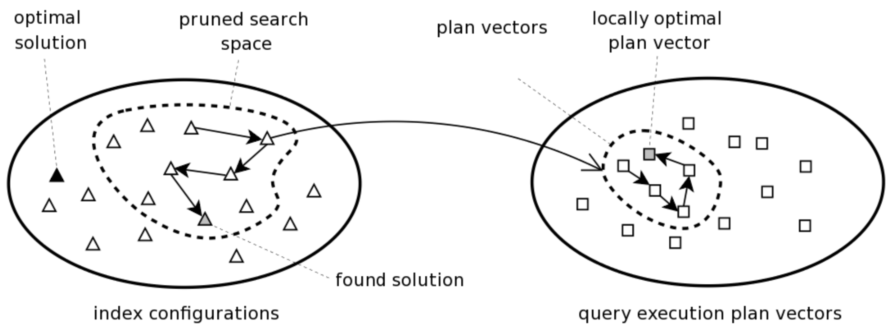

Fig. 1 Heuristic search of the index configuration space

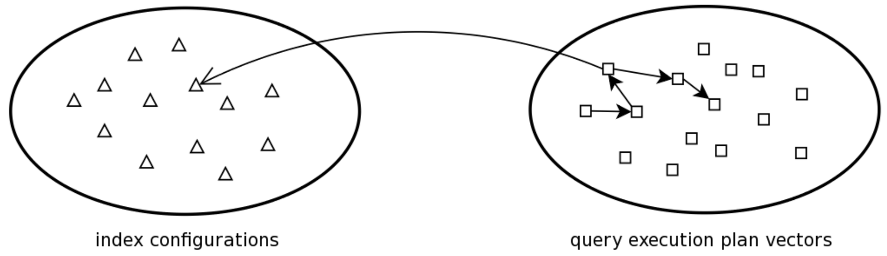

Fig. 2 Heuristic search of the query execution plan space

the whole process. This makes the evolutionary approach ideal for autonomic, self-tuning database systems.

The method was inspired by our observations of professional database administrators. We were wondering, how they tune their database systems. At first, they usually investigate the most costly queries. These are the queries that require a long time for each execution and/or are executed very frequently. Concentrating on these queries can lead to significant performance gains with only a little effort on the optimization, as usually the number of costly queries is small relative to the number of all queries. Then, they display a plan for each query by using tools that come with the database management system (usually called `EXPLAIN` or similar), and try to figure out, why the plan is bad and how it can be improved. When they know, how the query should be optimally processed, they create all needed structures like indexes or materialized views and eventually give some hints to the query optimizer to make it choose their plan. Of course, sometimes the administrator's best plan is not the optimal one, or the database query planner does not do what it is expected to, so for complicated queries some trial and error method is used. However, less experienced admistrators tend to jump right to the trial-and-error phase, creating various possibly useful structures and checking whether they improve the query execution times or not.

The contributions of this paper are the following. First we describe a novel algorithm for solving the ISP in the query plan space. Second, we provide a theoretical proof of convergence of the method to the optimal solution and analyze its other properties. Third we demonstrate the results of experiments showing that the algorithm works correctly and achieves higher performance and better results than the state-of-art commercial index advisor.

2 Related Work

There are many papers that refer to the problem of finding an optimal configuration of indexes. The research on ISP has been conducted since mid seventies of the previous century. In order to reduce the solution search space, already in [7] some simple heuristics have been proposed. Several papers [2, 9, 14, 29] recognize ISP as a variation of a well-known Knapsack Problem and propose some heuristic methods to solve it. However, none of them uses a query plan optimizer to estimate gains of the selected indexes nor to propose index candidates.

In [10] a method is described that suggests various index configurations, and uses an R-system query plan optimizer to evaluate their gains. The candidate configurations are generated by heuristic rules, based on the SQL queries in the workload, so the whole solution is quite complex. The authors of the Index Tuning Wizard for MS SQL Server [8] took a very similar approach, but they used simpler heuristics for the index candidate selection. They start from the indexes on all indexable columns used in each query, and later prune this set by choosing only the best index configuration for each query.

The idea of employing the query plan optimizer was further extended in [26]. In this method, the query plan optimizer not only evaluates the gains of possible index configurations, but also generates the best index configuration for each query. This technique reduces the number of required calls to the query plan optimizer. Only one call is required to get the best index configuration for a given query. The final solution is evaluated by transforming the ISP problem to the Knapsack Problem and applying a simple heuristic. Our solution is somehow based on this idea. However, our optimizer does it for many queries in parallel, and it is not limited to selecting locally optimal index configurations for one query.

In some approaches [4, 5, 6, 17, 18] the authors concentrate on the ISP problem itself, independently of its database environment. These methods assume the gains of indexes or their configurations as explicitly given, and treat queries and indexes as set elements, without diving into their structure. Optimal solutions for such defined synthetic problems involving several thousands of queries and candidate indexes have been reported. However, none of these papers analyzes the problem of generating the candidate index configurations or evaluating their gains for a real-world application.

Some local search heuristics have been applied to solving a variation of ISP where the workload and data can change over time [24, 25]. Candidate indices are proposed by the query optimizer as in [26]. Each candidate index is given a rank based on the total performance gain it causes. The performance gain is estimated by analyzing selectivity of the query predicates and by the query plan optimizer. Old queries influence the total index gain less than the recent ones. Given a ranking of indexes, the result index set is chosen by heuristics that solves a Knapsack Problem.

The algorithms from [8, 26] usually solve some variations of the Knapsack Problem by starting from an empty index set and adding candidate indexes to it, until the space limit is exceeded. Another approach has been presented in [3]. Here, the optimization process starts from the set of indexes and materialized views that generate the highest total performance gain for the `SELECT` queries in the workload. Then, the elements are removed from this set or merged together as long as the space constraint is violated. The method uses the query optimizer to evaluate the gains of the candidate structures. Some heuristics for reducing the number of calls to the query optimizer are discussed. These heuristics were further improved in [21] in such a way that they do not incur query cost misestimations while requiring 1.3–4 times less optimizer calls than the original method [3] thanks to reusing some parts of the locally optimal plans. The improvement can be used by any ”what-if” ISP solver, allowing it potentially to examine much more candidate index configurations, and yielding better final results.

In the papers [4, 17, 20] the ISP problem is formulated as an *Integer Linear Programming* problem. The authors use an exact algorithm based on branch-and-bound with various kinds of constraint relaxation to solve it. Unlike the heuristic approach, this class of methods is able to give an exact upper bound on how far the actual solution is from the optimum. The performed experiments have shown that these methods yield results with acceptable error for the ISP instances of practical size very quickly, though finding the exact optimum usually requires many hours of computation. However, the ILP approach requires additionally running a preparatory step for a careful candidate index set selection, just as the previously mentioned heuristics ([4, 5, 6, 17, 18, 24, 25, 26]). Feeding all the numerous possible index configurations to the ILP solver is impractical, as it could easily create ILP instances having much too many variables and constraints to be solvable in a reasonable amount of time [20]. On the other hand, pruning this set too much may cause good solutions to be missed. The accurate candidate index selection itself is a complex problem and some work has been done in the context of the OLAP workloads [27], but to our best knowledge it remains open for the general case.

3 Formal Problem Definition

The database system processes a set of n tasks $Q = \{q_1, q_2, \ldots, q_n\}$. Each task can select, insert, update or delete data. For simplicity, whenever we use word *query*, we also mean the tasks that modify data. By D we denote a set of all possible indexes that can be created for a given database, whereas certain D_x, $D_x \subseteq D$, will be called index configuration. Each query can be executed according to its *query execution plan*. The query execution plan consists of instructions for the *query executor*, along with a list of indexes which are to be used while performing the query. The formalism below describes this mechanism.

By P_Q we denote a space of all possible query execution plans that can be generated for the queries from Q, assuming any index from D is available. As in our optimization problem Q is well defined and does not change, in the sequel we simply write P. Let $P(q) \subseteq P$ be a set of all possible query execution plans for the query q, $P(q) = \{p_1, p_2, ..., p_m\}$ and $P(q, D_x)$ be a set of plans valid for the D_x index configuration.

Given a configuration D_x and a query $q \in Q$, the query planner generates a plan according to the function $\pi : Q \times 2^D \to P$. An ideal planner would optimize the execution of q and provide the plan $p^*_{q,D_x} = \pi(q, D_x)$, $p^*_{q,D_x} \in P(q, D_x)$, which minimizes the execution cost of the query q. Each plan p has an associated *execution cost* $\text{cost}(p, D_x)$. Let us note that for a plan p there is a minimal subset of indexes $D_I(p)$, which are indispensable for this plan. Obviously $D_x \supseteq D_I(p)$. The index maintenance costs are included in the cost calculations, so the cost of a given plan may depend on the other indexes, not used by the plan itself. The objective is to find such a subset D_x of D that minimizes the total cost of the query executions:

$$\min_{D_x \subseteq D} \sum_{q \in Q} w_q \text{cost}(p^*_q, D_x) \tag{1}$$

$$\text{size}(D_x) \leq s_{\max} \tag{2}$$

where $\text{size}(D_x)$ is the total size of the used indexes, $s_{\max}$ is the size limit and w_q is a weight of query q in the input workload, usually related to the frequency of that query.

4 The Algorithm

For searching the space of index configurations we use a standard steady-state evolution strategy with a constant population size, a constant mutation rate, and neither elitist subpopulation, nor crossover.

Individual Representation and Fitness Function

According to the cost function (1), a naïve approach would consist in directly seeking for a subset $D_x \subseteq D$, such that the value of the total cost is minimal. However, this would involve calling the query optimizer for every individual index configuration D_x possibly many times, which is a very costly operation. Actually, many calls would be performed only to find out that the new index configuration does not change the best plans, even though only a small subset of queries actually needs to be reoptimized at every D_x change. Note, that the number of various possible index configurations used by a single query grows exponentially with the number of query predicates and joins. This problem was thoroughly discussed in [3].

In our approach, each individual in the population is a vector of query execution plans $v = [v_1, v_2, \ldots, v_n]$ where $v_i \in P(q_i)$ for $i = 1 \ldots n$. Each plan can use any indexes from the set D. We minimize a function:

$$\min_v \sum_{i=1}^{n} [\mathrm{cost}(v_i, D_x) + g(D_x)], \tag{3}$$

where $g(D_x)$ is a penalty for exceeding the size limit $s_{\max}$:

$$g(D_x) = \max\{0, \alpha[s(D_x) - s_{\max}]\} \tag{4}$$

The value of α can be set large enough to make exceeding the size limit non-profitable. The set D_x is a function of the vector v:

$$D_x(v) = \bigcup_{i=1}^{n} D_I(v_i) \tag{5}$$

Note that the set of possible v values is a superset of the set of all possible plan configurations established during the optimization process of the cost function (1), because v may contain locally suboptimal query plans.

The indexes used by v are also kept as a part of the representation of an individual. Each index has an associated reference counter so that it is easy to tell, when the index is not used any more and can be safely discarded.

Evolution Process

Now we can describe in more detail the evolution process. The optimization is performed as follows:

1. a population is initialized with a single plan vector containing optimal plans for the existing set of indexes in the database;
2. in each iteration:
   a. a random individual is selected with a probability determined by its fitness;
   b. the reproduced individual is mutated;
   c. the mutated individual is added to the population;
   d. if the maximum size of the population is exceeded, a random individual is removed from the population.

To reduce the overhead of copying large query execution plans and index metadata on each individual reproduction phase, the plans and indexes are lazily copied. The actual copying is done only for small parts of the vector that is mutated.

Selection

For the selection phase we use a simple tournament selection. The selection pressure can be adjusted by one parameter $e \in \mathbf{R}^+$ that influences the tournament size and the probability of worse individual being selected over a better one. The exact procedure is performed as follows:

1. Let $e' := e$.
2. Select an individual v randomly with a uniform probability distribution.
3. While $e' > 0$, repeat:
   a. Select an individual v' randomly with a uniform probability distribution.
   b. If v' is better than v, assign $v := v'$ with probability $e' - \lceil e' - 1 \rceil$.
   c. Assign $e' := e' - 1$.
4. Return v.

For $e = 0$ we get a uniform selection. For $e = 0.5$ we get a 2-way tournament, where the best individual has 50% chances to win. For $e = 1$ we get a classical 2-way tournament where the best individual always wins. The larger e, the larger is the selection pressure.

Mutation

The basic mutation of an individual v consists of applying some small *atomic transformations* to a randomly chosen subset of plans in v. Every transformation guarantees that the transformed plan is semantically equivalent to the original one. If this cannot be satisfied, it fails and a transformation of another class is chosen. There is a fixed number of atomic transformation classes.

- The *join reordering* transformation changes the execution order of two random adjacent join operators. The transformation may fail for some coincidence of joins of different types (inner, left outer, right outer, full).
- The *join algorithm change* transformation chooses a different algorithm for joining two relations. So far, our experimental optimizer handles a few well known join algorithms: a *block nested loops join*, an *index nested loops join*, *sort merge join* and a *hash join*. None of these algorithms is the best one in all possible situations, so selecting a proper one is essential in creating a good plan. For the index nested loops join, a new index may be introduced as well as some index introduced earlier for another plan can be reused. The transformation cannot introduce an index identical to any of the indexes used by the other plans.
- The *table access change* modifies leaves of the plan by choosing one of the following relation access methods: full table scan, full index scan and range index scan. This transformation may introduce new indexes, as well as, may choose any index already used by some other plan, and eventually extend it with more columns. The column set of the index is selected from the columns in the query predicates, columns used in the `GROUP BY` or `ORDER BY` clauses, or all columns used in the plan.

A *mutation of the plan* consists of at least one successful atomic transformation. All the transformation classes are allotted with the same probability. The number of successful single plan transformations k_t performed per mutation is a random number, generated with a probability distribution, such that numbers near 1 are the most probable. This makes most of the mutations small and enables the algorithm to perform *hill climbing*. However, it is essential that sometimes the mutation is large, in order to avoid premature convergence to a locally optimal solution and to ensure the whole search space is explored. This we achieve by using the positive part of a Gaussian distribution:

$$K(\sigma) = \lfloor |N(0, \sigma^2)| + 1 \rfloor \tag{6}$$

The number of plans k_p mutated in each individual in one iteration of the evolutionary algorithm is calculated from the same distribution (6). The algorithm was insensitive to the exact setting of σ.

The plans to be mutated can be selected in one of the following ways:

- The probabilities of selecting each of the plans are equal. We will call it *uniform selection*.
- The probability of selecting a plan is proportional to its cost. We will call it *proportional selection*. This type of selection makes mutating costly plans more frequent than the cheap ones and greatly increases the convergence rate of the algorithm. However, in some rare cases a very cheap plan may be indirectly responsible for a high total cost due to requirement of the index with a high maintenance cost caused by frequent updates. In such cases, the uniform plan selection may be better.

- Uniform or proportional selection is chosen with probability of 0.5. We will call it *mixed selection*. This seems to have advantages of both the uniform and proportional selection. The costly plans are mutated much more often than the cheap ones, while each of the cheap plans is guaranteed a mutation rate only 2 times lower than it is for the uniform plan selection.

After completing the mutation phase, a cost of the new plan is estimated. Due to the fact that mutations are often local, there is usually no point in recalculating the costs of most of the nodes in the query plan tree. The transformation operators mark only changed parts of the trees and all their ascendants to be recalculated. For instance, changing the table access method in one of the leaves of the plan does not cause changing the costs of other leaves. This partial plan caching technique is somehow similar to that used in [21], but may be more aggressive due to the fact that the result plan need not be optimal. For large plans, e.g., as the ones from the TPC-H benchmark [28], the partial plan caching improves the performance of the mutation step by 3 to 10 times. In the future, this technique can be improved to also avoid recalculating the ascendants of the changed parts in most cases.

Replacement

When the population reaches a defined size, the mutated individual replaces a randomly chosen individual from the population. The first version of the algorithm made uniformly a choice of individuals to be replaced. However, this often led to replacing some very good individuals with bad ones, and the algorithm converged very slowly. Therefore for the selection of individuals to kill, we use the same tournament selection scheme as we use in the selection of individuals for mutating, though ”in the opposite direction” – this time the worse individual has higher chances to be chosen. Thus, the good individuals are very unlikely to be removed from the population pool, but still the probability of removing them is slightly greater than zero.

Clustered Indexes

The basic method of plan transformations can advise only non-clustered indexes. The transformations do not introduce new clustered indexes. It would be tempting to allow them to do so, but this could result in a possibility of having more than one clustered index on the same table, because the transformations are performed independently of each other. It is possible to add additional checking code for each mutation to avoid this, but we decided to make the process of introducing clustered indexes as a separate mutation, so that its rate could be controlled separately.

We extend the mutation procedure with a possibility to *mutate indexes* directly by means of changing the types of used indexes D_x, so that the restriction for at most one clustered index per table always holds. The complete mutation procedure is as follows:

1. Generate a random number k_t using the formula (6).
2. Repeat k_t times:
   a. with probability $(1-\mu)$ mutate a random plan in v;
   b. with probability μ change a random index in D_x from non-clustered to clustered or the other way around. If needed, change one index from clustered to non-clustered, so that there is at most one clustered index per table. Recalculate the costs of all plans that use the changed indexes.

The μ parameter is usually a small positive number. We used in our experiments $\mu = 0.05$. The higher value of μ may cause extra overheads of the algorithm, because the index type change requires calculating costs of many plans, so it significantly increases the average computational cost of a single iteration. One index may be used by several queries. Thus, modifying one index type requires more CPU cycles than modifying a single plan. Besides, performing index mutations too often does not decrease the total number of iterations that must be executed to achieve good results.

Redundant Indexes Problem

In some cases the proposed algorithm does not converge to the good solution quickly, and often it proposes overlapping, redundant indexes. Consider m semi-identical queries in the workload in the sense they are querying one table and having 2 predicates A_1 and A_2 with the same selectivity. There may be two optimal two-column indexes for each query, differing only with the order of columns in the index key. Each index alone would reduce the cost of each query by the same amount. The optimal solution should contain only one such index, assuming the considered indexes are not useful for any other query in the workload. Unfortunately the random nature of the algorithm makes it unlikely that the same optimal plan for all m queries is chosen, especially if m is large. Probably approximately half of the plans would use the index on (A_1, A_2) and the other half – the index on (A_2, A_1). Note that changing a single query plan to use one index over another does not change the fitness of the solution, and the objective function forms a plateau here. Without a special handling of this situation, the solution might contain both of the indexes and the maintenance cost would be higher than actually required.

To this end we introduce yet another type of individual mutation that removes a random index from the solution by transforming a set of plans that use it. The removal of index usage is performed by the standard plan transformations: (1) the join algorithm change to replace index nested loops joins, and (2) the table access change to change the index used in the index scans. This time the transformations are not random – they are given an exact tree node to operate on. If there exists an alternative index in D_x, which gives similar benefits to the plans as the removed index, it is used. There is no need to perform this kind of mutation as frequently as the other types of mutations, due to the fact that redundant indexes are not created very often. We set the probability of triggering this type of mutation in each iteration to 0.01, and it turns out to be sufficient.

5 Analysis

The presented algorithm resembles the *evolution strategies*(ES), as presented in [1]. It was proven that the ES algorithm with a mutation operator using Gaussian distribution guarantees finding the optimum of a *regular* optimization problem. However, the algorithm in [1] represents each solution as a vector of real numbers, but the domain of our problem is non-linear and each solution is represented by a vector of trees. Thus, our optimization problem does not meet the requirements to be regular, as stated in [1], and the convergence theorem cannot be applied.

Fortunately, the domain of our problem can be represented by a graph, where each vertex represents a single query plan vector and edges connect solutions (states) that can be directly transformed into each other by applying a single atomic transformation (as described in Section 4). Each mutation step in our ES can be viewed as a random walk on this graph. Due to the fact that the following holds:

- the probability of each transformation is stationary and greater than 0,
- the current solution depends only on the preceding solution and transformation chosen,
- the graph is finite and connected,

the process forms a finite-state Markov chain [19]. Additionally, each transformation has a corresponding co-transformation that brings the individual back to its original state. Thus, there are no absorbing or transient states in the system, and the Markov chain is irreducible (but not necessarily ergodic). This guarantees that by performing enough state transitions, each possible solution will be reached. Due to the fact, that the number of transformations in each mutation step given by the equation (6) is unlimited, each possible solution can be reached in a single mutation step of the ES, which guarantees finding the optimal solution after some (sufficiently large) number of iterations. Note, that for this to hold, the atomic transformations must be able to create any feasible solution. For instance, if the optimal solution required scanning an index with 10 columns, and there was no transformation that could introduce such 10-column index, the assumption that the solution graph is connected would not be met and the method would fail to find the optimum plan.

The average and pessimistic time complexity to find the global optimum is exponential with respect to the number of joins, predicates and used columns in the queries. Unless the complexity classes **P** = **NP**, this property is true for any ISP solver claiming global convergence. However, as shown further in the Section 6, the ES is capable of finding good solutions in a very short time even for large and complex ISP instances.

The time complexity of each iteration of the presented ES is as follows:

- the selection step requires selecting a constant number of random plans; if plans are stored in an array with random access, the complexity of the selection step is $O(1)$;
- the reproduction step requires copying the plan vector and indexes; even though it is lazily-copied, the complexity of this step is $O(n+l)$, where l is the number of indexes in the individual;
- the mutation step requires selecting some number of plans in the plan vector to be mutated and then actually performing the mutations on them; the average number of selected plans is constant; the complexity of each mutation is proportional to the number of atomic transformations applied to the plan and to the complexity of each transformation; the average number of atomic transformations is constant, while the complexity of the transformation is proportional to the number of nodes plus the number of predicates in the query plan tree; assuming the query plans are of limited size, the complexity of the mutation step is $O(1)$;
- the replacement (recombination) step is similar to the selection step and has the complexity $O(1)$.

The asymptotic complexity of the single ES iteration is $O(n+l)$, which means the algorithm will get slower the more queries in the workload are given and the more indexes are selected. However, for the typical number of queries and plans, the reproduction step would be fast enough not to become a bottleneck, as making a copy of an individual is a relatively simple operation. The most complex and computationally costly step is the mutation, requiring not only modifying some plans, but also estimating the costs of changed plan tree nodes.

The memory requirements for the algorithm are quite low in comparison to the algorithms that require the index configuration candidate selection step, as no numerous candidates need to be remembered. Actually, the memory is used for storing the population, i.e. the query plans and the selected indexes. Thus the required memory size is proportional to the number of queries in the input workload, final result size, and the user defined maximal number of individuals in the population.

6 Implementation and Experiments

Our first attempt at implementing our algorithm was to modify an existing query planner found in a widely used open-source RDBMS such as PostgreSQL or MySQL, so that it could use our approach to select indexes. However, at this time their planners lacked important features, like support for covering indexes or proper query rewrite engine. These features are essential for achieving high performance in complex analytical benchmarks. Even though these features could be added, still a lot of further effort would be required to implement the query execution plan mutation procedure in a foreign environment, probably by modifying large parts of the existing code. Partial query execution plan caching, as proposed in this paper in order to increase the performance of the index selection tool, would probably complicate things even more. Thus, we have implemented our algorithm as a standalone tool, separated from any RDBMS index selection tool, possibly duplicating some work already done in the existing query planners. To minimize the effort, we have decided to reuse as much concepts and code from the PostgreSQL database system as possible, including the formulas for cardinality and cost estimation, and directly accessing metadata gathered by the PostgreSQL's `ANALYZE` tool. PostgreSQL was chosen because it seemed to provide the most advanced query planner among the open-source solutions available. Moreover, we had access to its large and advanced commercial applications.

In order to be able to perform experiments using synthetic benchmarks, we have added missing support for index-only-scans, and improved the query rewrite engine, as well as, the PostgreSQL query execution cost estimation model, so that now it better matches the one of commercial database systems, in particular IBM DB/2.

The tool was entirely implemented in the Java programming language, with help of the ANTLR [22] parser generator for creating the SQL parser, JDBC API to import the PostgreSQL's metadata and the SWING library to create an easy to use user interface. The application with source code will be freely available for academic research.

The experiments were divided into two groups. The first group of experiments consisted in running the tool for a very small synthetic workloads and simple database schemas, to check if the optimal index recommendations are found. The results were compared with the results obtained from the DB/2 index advisor [26]. The second group of experiments consisted in running the tool for two different real-world transactional workloads, as well as, a selected set of TPC-H queries. Below we describe the results in more detail.

6.1 Simple Workloads

A single table *table* with 3 integer columns $c1, c2, c3$ and a workload consisting of 2 queries were sufficient to show the most interesting properties of the algorithm. Every column in this test had the same width of 4 bytes, and evenly distributed values in range $[1, 1000000]$. There were 1 million rows in the table.

One of the tests was to check, how frequent updates of one column might affect the index recommendations given by the algorithms tested. It consisted of two workloads. The first workload consisted of only one query q_A with the weight $w_A = 1$:

```
SELECT c1, c2 FROM table WHERE c1 < 1000 AND c2 < 2000;
```

The storage space limit was set enough high not to affect the result. The obvious recommendation given both by our tool and the DB/2 advisor was a covering index on (c1, c2). Adding an update statement q_B with a very large weight $w_B = 10000$:

```
UPDATE table SET c1 = <constant> WHERE c3 = 10;
```

changed the recommendations. Both index advisors recommended the index on the column (c3), to accelerate q_B. However, the DB/2 advisor did not provide any index to accelerate the query q_A, while our tool recommended a clustered index on the column c2. Such index is still much better than no index at all, and does not impose the index maintenance cost that would be required for the index on (c1, c2) due to the frequent updates q_B. Note, that such recommendation for q_A would be suboptimal if q_B did not exist in the workload or had a smaller weight. As the initial index candidate sets generated by the methods [3, 8, 26] are based on the indexes picked by the query planner for each of the queries separately, the recommendations given by these algorithms cannot be optimal for the whole workload in this case. Actually, the recommendation given by our tool is much better than the one of the DB/2 advisor, even according to the original query plan cost model, which is used by the DB/2 query planner.

A similar behavior was observed in the case of a single `Select` statement, and a certain storage size limit. Let us consider a query:

```
SELECT c1, c2 FROM table WHERE c1 = <constant>;
```

With a relaxed storage limit, both advisors recommended creating a covering index on $(c1, c2)$, so that it can be used to quickly locate tuples matching the query predicate. Then we set the storage limit slightly below the total size of the database with the recommended index, and repeated the optimizing procedure. This time our tool recommended a smaller, clustered index on the single column c1, while the DB/2 advisor did not recommend any index. Obviously the fitness of both solutions differed by a factor of more than 500, because the DB/2 recommendation forces the query planner to choose a costly full table scan, which could have been easily avoided.

Another test has shown the value of index merging technique, originally introduced in [3], and used in our approach by the table access change transformation to create new index candidates. Consider two queries:

```
SELECT avg(c1) FROM table;
SELECT avg(c2) FROM table;
```

Due to the fact that answering these queries requires reading every tuple of the table, and assuming the database storage is row-based, the only possibility to accelerate the execution of these queries is to create covering indexes, so that one avoids reading not needed columns. Obviously, with no storage space limit 2 separate indexes, one on the column $c1$, and the other one on $c2$ constitute the optimal solution. This recommendation was given by the tested index advisors. However, if the storage space limit was reduced, so that one of the recommended indexes did not fit, the optimal solution would be an index on $(c1, c2)$ or $(c2, c1)$. This correct solution was given by our tool, in contrast to the solution given by the DB/2 advisor. The algorithm in [3] obviously should also find the optimal solution in this simple case.

6.2 Complex Workloads

For the experiments, we used two real-world server side transactional applications: a commercial multiplayer network game with 100,000 users and 0.7 GB of data in 108 tables (further referred to as MG) and a smaller mobile web application of one of the Polish telecom operators, using 33 tables (further referred to as WA). MG executed much more update statements than WA, which was mostly read-only (see Tab. 1). The two applications already had some indexes created by experts. MG was a mature application running in production for over 3 years, while WA was in its beta stage, and has been optimized very roughly by the application designers without looking at the workload. There existed indexes for primary keys and for some most often executed queries. Each type of measurement that we performed, was repeated 20 times, and the best, average, and the worst results were recorded.

Table 1 Characteristic of the workloads

	Share [%]	
Statement type	MG	WA
Single-table `SELECT`	66.95	78.73
Multi-table `SELECT`	18.93	16.14
Aggregate `SELECT`	2.84	3.30
`INSERT`	0.92	1.83
`UPDATE`	12.71	0.98
`DELETE`	0.49	2.31

The automatic index selector was used to select indexes for the copies of the databases in two different situations:

- hot start: after leaving the database ”as is”, without dropping any indexes
- cold start: after dropping all the indexes

Each workload consisted of 50000 SQL queries recorded from the database system log files. Because it took too long to estimate the cost of such a large workload

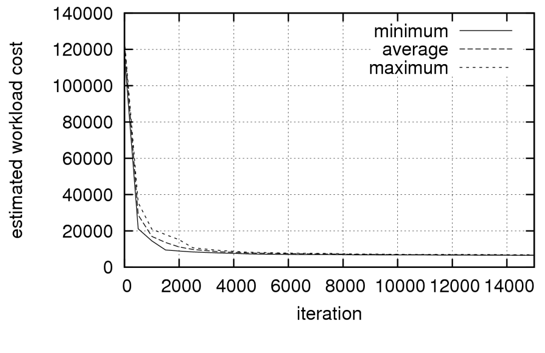

Fig. 3 Final workload cost as a function of number of iterations for MG database for cold start

and then to automatically select indexes, both workloads were compressed by a method presented in [16]. The final numbers of queries after compression were 289 for MG, and 62 for WA.

The algorithm was insensitive to a wide range of settings such as the population size and environment pressure. We have obtained good results for the populations of the size 10, as well as 80. Also setting the environment pressure e between 1.0 and 2.0 did not significantly affect the quality of the results and the speed of convergence. We present below the results for $n = 20$ and $e = 1.5$. The size limit $s_{\max}$ was set to 150% of the database size without indexes, but the limit was never reached during the experiments. The size of the database with the materialized solution index configuration never exceeded 130% of the original database size.

We have observed a very fast convergence of the estimated workload cost. For the cold start, the first 500 iterations reduced the workload cost by over 80% for MG (Fig. 3). In the next 5,000 iterations, the workload cost dropped by 10% of the original workload cost, which was already very close to the cost obtained at the end of the experiment, after 150,000 iterations. For the hot start, the initial workload cost was much lower, so the performance improvement achieved by the index selection was not as large as for the cold start (Fig. 4), but again most of the improvement was achieved by the first few hundred iterations. For MG, our implementation did about 450 iterations per second on average on a single core AMD Athlon 3700+ processor. The test program was executed by the Sun Java 6 virtual machine with

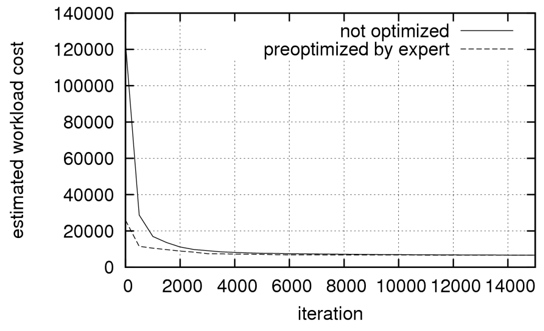

Fig. 4 Average convergence curves for hot and cold start

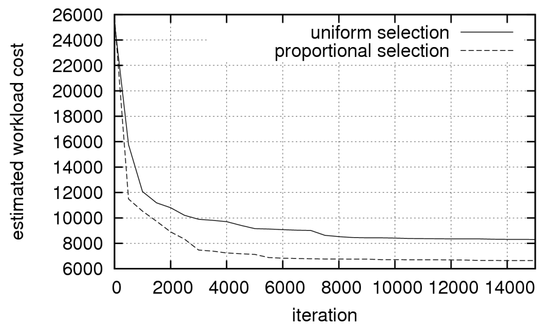

Fig. 5 Influence of the plan selection type on the average convergence speed

250 MB of heap space. It was possible to use a 64 MB heap, but the performance was slightly lower – about 280 iterations per second. The compressed workload of WA was smaller, and our index selection tool was able to run over 3100 iterations per second. Moreover, it was possible to get ”good enough” index configurations in less time than it took the optimizer to calculate the minimum cost of the whole workload for the one, fixed index configuration, which involved generating an optimal plan for each query in the workload.

We have also checked how the plan selection algorithm affects the performance of the method. As a reference, we have implemented the uniform plan selection algorithm. The proportional selection gave much better average results over the uniform selection (Fig. 5). When using the uniform selection, sometimes running even 100,000 iterations did not produce optimal results. The best recorded convergence curve for the uniform selection was also much worse than that for the proportional selection (Fig. 6). Therefore we conclude the proper plan selection is crucial to achieving fast and stable convergence.

Additionally, to test if our approach is feasible also for analytical workloads, we used 20 different queries from the standard TPC-H database benchmark [28] and a sample TPC-H database generated with a factor of 0.1 (about 100 MB). For this experiment, the database contained only the primary key indexes at the start. In contrast to the workloads MG and WA, this test was performed both by the DB/2 advisor, and by our implementation of the presented ES algorithm. Obviously, due to the fact that the ES implementation is based on the cost model of PostgreSQL, it would be the best to compare it to a reference index advisor for PostgreSQL. Unfortunately, there is no such index advisor for any recent PostgreSQL version yet.

As for the MA and MG workloads, we measured the maximal, minimal and average convergence rate for 20 runs of our tool, 20000 iterations each (Fig. 7). The solution converged slower than in the case of the transactional workloads, but good results were obtained in less than 6000 iterations. The performance was about 1500 iterations per second, and it took about 10 seconds to give almost optimal solution. This was similar to the times required for running the DB/2 advisor, which were between 5 and 12 seconds. We predict the performance of our tool, measured as

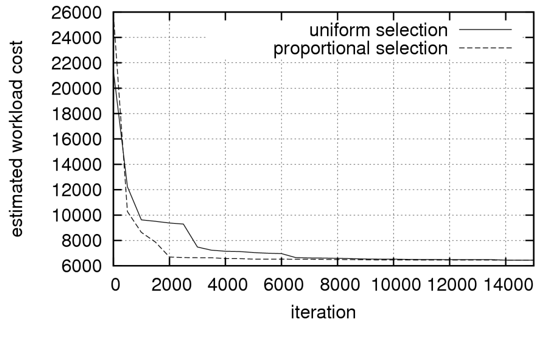

Fig. 6 Influence of the plan selection type on the maximum convergence speed

a rate of executed iterations, can be much better, because no extensive code optimizations has been made yet and a lot of possibilities of improvement has not been tried. Due to the fact that (1) the query execution plans for the TPC-H workload are often large and contain many joins, (2) the plan transformations are often local, more aggresive caching of join cardinality and cost calculations can bring possibly essential benefits. The convergence rate in this experiment was rather insensitive to the exact settings of the parameters. In particular, the solution converged well for the population size set to 10 or 100, and the environment pressure set to 0.5 or 1.8.

The TPC-H workload required much more space for indexes than MG and WA. Without the storage space limit, the DB/2 advisor recommended indexes having a total size of over 200 MB, while our tool — of about 250MB. Reducing the storage size limit for the recommended indexes increased the final estimated costs of workload for both solutions (Fig. 8). The recommended index sets and their estimated benefits were similar, but not exactly the same in both solutions. . The observed differences were presumably caused by the query plan cost models that did not *exactly* match. Materializing the indexes recommended by our tool for the storage size limit of 40%, applied for the DB/2 database, improved the estimated workload cost by over 55%, but obviously not as much as the original DB/2 recommendation. The same situation was observed when applied the DB/2 recommendation to our cost model — the value of the objective function was better for our solution than for the DB/2 advisor's one.

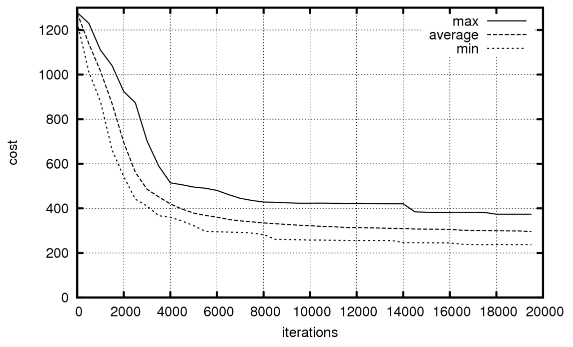

Fig. 7 Convergence curves for the TPC-H workload

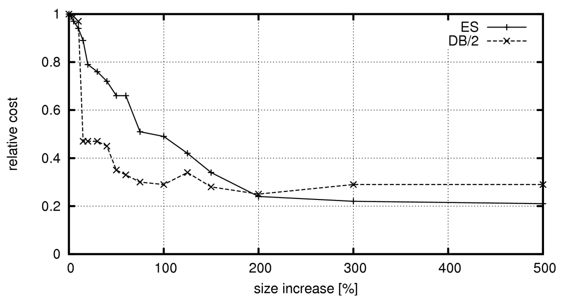

Fig. 8 Estimated costs of TPC-H workload as a function of the storage space limit

7 Conclusions

We have presented a novel approach to solving the index selection problem. The main idea of the presented algorithm was that instead of seeking the index space we suggest to search the space of query execution plans. In order to avoid preliminary pruning we have decided to apply an evolutionary strategy. The experiments proved acceptable performance and good reliability of the developed algorithms for real-world index selection problems for medium-size relational databases and also feasibility of the approach for complex analytical workloads. We proved theoretically that the method converges to the global optimum. We also presented examples where our algorithm actually gives better solutions than the solutions given by the state-of-the-art heuristics which explore the index configuration space. The method does not require any phase for preparatory candidate index selection. In addition it shows very low sensitivity to parameter settings, therefore it can be especially useful in the implementations of self-tuning, autonomic database systems.

References

1. Back, T., Hoffmeister, F., Schwefel, H.P.: A survey of evolution strategies. In: Proceedings of the Fourth International Conference on Genetic Algorithms, pp. 2–9. Morgan Kaufmann (1991)
2. Barcucci, E., Pinzani, R., Sprugnoli, R.: Optimal selection of secondary indexes. IEEE Trans. Softw. Eng. **16**(1), 32–38 (1990). DOI http://dx.doi.org/10.1109/32.44361
3. Bruno, N., Chaudhuri, S.: Automatic physical database tuning: a relaxation-based approach. In: SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data, pp. 227–238. ACM, New York, NY, USA (2005). DOI http://doi.acm.org/10.1145/1066157.1066184
4. Caprara, A., Fischetti, M., Maio, D.: Exact and approximate algorithms for the index selection problem in physical database design. IEEE Trans. on Knowl. and Data Eng. **7**(6), 955–967 (1995). DOI http://dx.doi.org/10.1109/69.476501
5. Caprara, A., González, J.J.S.: Separating lifted odd-hole inequalities to solve the index selection problem. Discrete Appl. Math. **92**(2-3), 111–134 (1999). DOI http://dx.doi.org/10.1016/S0166-218X(99)00050-5
6. Caprara, A., Salazar, J.: A branch-and-cut algorithm for a generalization of the uncapacitated facility location problem. TOP **4**(1), 135–163 (1996). URL citeseer.ist.psu.edu/caprara95branchcut.html
7. Chan, A.Y.: Index selection in a self-adaptive relational data base management system. Tech. rep., Cambridge, MA, USA (1976)
8. Chaudhuri, S., Narasayya, V.R.: An efficient cost-driven index selection tool for Microsoft SQL Server. In: VLDB '97: Proceedings of the 23rd International Conference on Very Large Data Bases, pp. 146–155. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1997)
9. Choenni, S., Blanken, H.M., Chang, T.: Index selection in relational databases. In: International Conference on Computing and Information, pp. 491–496 (1993). URL citeseer.ist.psu.edu/choenni93index.html
10. Finkelstein, S., Schkolnick, M., Tiberio, P.: Physical database design for relational databases. ACM Trans. Database Syst. **13**(1), 91–128 (1988). DOI http://doi.acm.org/10.1145/42201.42205

11. Fotouhi, F., Galarce, C.E.: Genetic algorithms and the search for optimal database index selection. In: Proceedings of the The First Great Lakes Computer Science Conference on Computing in the 90's, pp. 249–255. Springer-Verlag, London, UK (1991)
12. Ganek, A.G., Corbi, T.A.: The dawning of the autonomic computing era. IBM Syst. J. **42**(1), 5–18 (2003)
13. Glover, F.: Tabu search – part i. ORSA Journal on Computing **1**(3), 190–206 (1989)
14. Ip, M.Y.L., Saxton, L.V., Raghavan, V.V.: On the selection of an optimal set of indexes. IEEE Trans. Softw. Eng. **9**(2), 135–143 (1983). DOI http://dx.doi.org/10.1109/TSE.1983.236458
15. Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by simulated annealing. Science **220**, 671–680 (1983)
16. Kołaczkowski, P.: Compressing very large database workloads for continuous online index selection. In: S.S. Bhowmick, J. Küng, R. Wagner (eds.) DEXA, *Lecture Notes in Computer Science*, vol. 5181, pp. 791–799. Springer (2008)
17. Kormilitsin, M., Chirkova, R., Fathi, Y., Stallman, M.: Plan-based view and index selection for query-performance improvement. Tech. Rep. 18, NC State University, Dept. of Computer Science (2008)
18. Kratica, J., Ljubić, I., Tošić, D.: A genetic algorithm for the index selection problem (2003). URL citeseer.ist.psu.edu/568873.html
19. Meyn, S.P., Tweedie, R.: Markov Chains and Stochastic Stability. Springer Verlag (1993)
20. Papadomanolakis, S., Ailamaki, A.: An integer linear programming approach to database design. In: Workshop on Self-Managing Database Systems (2007)
21. Papadomanolakis, S., Dash, D., Ailamaki, A.: Efficient use of the query optimizer for automated physical design. In: VLDB '07: Proceedings of the 33rd international conference on Very large data bases, pp. 1093–1104. VLDB Endowment (2007)
22. Parr, T.: ANTLRv3: Another tool for language recognition (2003–2008). URL http://www.antlr.org/
23. Russell, S.J., Norvig, P.: Artificial Intelligence: A Modern Approach. Pearson Education (2003). URL http://portal.acm.org/citation.cfm?id=773294
24. Sattler, K.U., Schallehn, E., Geist, I.: Autonomous query-driven index tuning. In: IDEAS '04: Proceedings of the International Database Engineering and Applications Symposium (IDEAS'04), pp. 439–448. IEEE Computer Society, Washington, DC, USA (2004). DOI http://dx.doi.org/10.1109/IDEAS.2004.15
25. Schnaitter, K., Abiteboul, S., Milo, T., Polyzotis, N.: Colt: continuous on-line tuning. In: SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data, pp. 793–795. ACM Press, New York, NY, USA (2006). DOI http://doi.acm.org/10.1145/1142473.1142592
26. Skelley, A.: DB2 advisor: An optimizer smart enough to recommend its own indexes. In: ICDE '00: Proceedings of the 16th International Conference on Data Engineering, p. 101. IEEE Computer Society, Washington, DC, USA (2000)
27. Talebi, Z.A., Chirkova, R., Fathi, Y., Stallmann, M.: Exact and inexact methods for selecting views and indexes for olap performance improvement. In: EDBT '08: Proceedings of the 11th international conference on Extending database technology, pp. 311–322. ACM, New York, NY, USA (2008). DOI http://doi.acm.org/10.1145/1353343.1353383
28. Transaction Performance Council: The TPC-H decision support benchmark (2001–2008). URL http://www.tpc.org/tpch/
29. Whang, K.Y.: Index selection in relational databases. In: FODO, pp. 487–500 (1985)