Autonomic Combination and Selection of Tuning Actions

Rafael Pereira de Oliveira[1], Fernanda Baião[2], Javam Machado[3], Ana Carolina Almeida[4,5], Sergio Lifschitz[6]

[1]Aditum, [2]Departamento de Engenharia Industrial, PUC-Rio,
[3]Departamento de Computação, UFC,
[4]Instituto de Matemática e Estatística, UERJ, [5]University of Copenhagen,
[6]Departamento de Informática, PUC-Rio

rafael.oliveira@aditum.com.br, fbaiao@puc-rio.br,

javam.machado@dc.ufc.br, ana.almeida@ime.uerj.br, sergio@inf.puc-rio.br

Abstract. *Combining database tuning actions has neither a precise formulation nor a formal approach to solving it. It is a complex and relevant problem in database research, both for the DBA manual solutions and automatic ones using specialized software. This work proposes an automated method for generating and selecting combined tuning solutions for relational databases. It addresses how to mix solutions while still preserving both technological constraints and available computational resources. The results show that our technique can produce more efficient combined solutions than independent local solutions.*

1. Introduction

Database tuning is a task that involves tuning parameters and maintaining structures to improve database performance. In general, the database administrators (DBAs) identify some abnormal behavior in the database and interfere with some actions. As a given action can interfere with the workload, this work starts from the hypothesis that combining fine-tuning activities through a global analysis of the workload may generate a more significant benefit than individual actions selected locally.

The research problem identified in this work is that combining fine-tuning actions in relational databases does not yet have a systematic method independent of the techniques involved. The question of how to combine tuning strategies remains to be investigated. The literature has no comprehensive approach for generating combined actions during a single fine-tuning task. The general objective of this paper is to propose a method of independent, systematic, and automatic combination of techniques for fine-tuning actions in relational databases. The test scenarios demonstrate that the variety of strategies brings efficiency and effectiveness to the tuning process.

The remainder of this paper is organized as follows. Next, Section 2 presents the fundamental concepts and discusses related literature; in Section 3, we detail our proposed method. In Section 4, we demonstrate some evaluations performed and discuss a beneficial form of combination. We make final considerations in Section 5.

2. Background and Related Work

Database tuning is the process of adjusting the physical database schema, rewriting queries, and refining DBMS parameters to improve performance [Shasha and Bonnet

2002]. Concerning the physical database schema, DBAs may select and create access structures (e.g., indexes), determine the partitioning of persistent database objects, force table denormalization, among other actions. A DBA may count on several tools to help achieve his goals (e.g. [Bruno 2012, Shasha and Bonnet 2002]).

Our focus is on relational databases and the following access structures, possibly combined into a single tuning task: indexes, partial indexes, and materialized views. A *complete* index **(IND)** is the traditional index present in any DBMSs. Partial indexes **(PIN)** allow defining a subset of the tuples through a conditional expression [Stonebraker 1989]. Indexes improve performances by reducing both the number of logical reads and the processing cost. We add the term *complete* to distinguish regular indexes from partial ones. A complete index can be primary or secondary. Materialized View **(MV)** is a view that has its result stored for later use. This strategy may bring a significant gain, as the query does not need to recalculate its result during execution. However the benefits are affected by MVs maintenance costs. Changes in base tables incur in mandatory updates in the related MVs. Therefore we must carefully choose which MV bring the greatest benefit and the lowest possible costs, for a given workload [Chirkova and Yang 2012].

Branch-and-bound (BnB) is widely used in combinatorial optimization problems and is a method that may help us in the selection and combination of database tuning actions. This method is based on the idea of developing an intelligent enumeration of actions to the optimal solution of a problem. BnB can generate solutions for complex problems by reducing the search space, especially when it is unfeasible to test all solutions [Lawler and Wood 1966]. The BnB algorithm has four main components: *(i) a branching rule* that defines the way a state (or a node) originates its next-level nodes; *(ii) a selection rule* that indicates the order (e.g., depth, width) in which active states will be expanded. *(iii) an elimination rule* that is the criterion (e.g. lower-bound) adopted to discard active states. And finally, *(iv) an end condition* when all expansion attempts have been constructed.

To achieve both the independence of tuning strategies and deal with the impact of their combinations, the **software agents** abstraction is promising. A software agent is a computer system capable of autonomous actions [Kwon et al. 2011]. The choice to use this approach is due to the agents' characteristics of autonomy and intelligence. The agent is able to follow its goals and adapt to changes in the environment automatically. The degree of autonomy is defined by the software engineer during agent behavior modeling, and is sensitive to the domain in which this solution is applied, for example, database tuning (e.g. [Elfayoumy and Patel 1999, Mrozek et al. 2014]).

The literature on fine-tuning relational databases is extensive, with many algorithms and heuristics already proposed. However, only few are directly related to our work, considering combined or integrated solutions. The work in [Agrawal et al. 2000] presents a two-phase method for automatically generating, selecting, and combining fine-tuning actions involving indexes and materialized views. In the first phase, activities of each type are developed independently, and in the second phase, the generated actions are filtered according to one of their selection policies (MVFIRST and INDFIRST). MVFIRST selects the materialized views that maximize benefits according to their cost model and fit into the available storage space. Then the best indexes are chosen according to the remaining disk space. INDFIRST differs from MVFIRST in the order of selection of techniques, selecting indexes before materialized views.

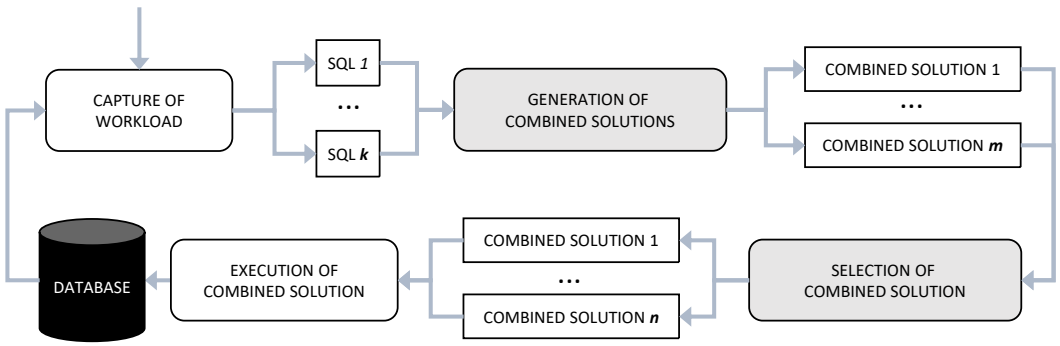

Figure 1. Steps for Combining and Selecting Automatically Fine Tuning Actions.

Kimura et al. [Kimura et al. 2010] also explore the combination of indexes and materialized views in the CORADD framework, which correlates attributes to recommend materialized views and indexes. The proposed tool evaluates the workload and finds a set of materialized views and indexes to be created in combination according to a storage limit dedicated for fine-tuning actions. The authors use the same strategy during the generation phase as [Agrawal et al. 2000]. In the selection phase, they group the workload SQL commands through the *k-means* [Hartigan and Wong 1979] algorithm and, for each cluster, select the best materialized views followed by a set of indexes.

The authors in [de Oliveira et al. 2019] propose a tool that encompasses an extensible knowledge base (KB) in which it is possible to add new types of tuning actions and specify heuristics for their combination. The fine-tuning KB defines all the concepts involved in the fine tuning actions, and formal axioms representing the heuristics for generating and selecting materialized views and indexes independently. The KB also specify a theoretical way to generate fine tuning suggestions combining different action types.

We propose an automatic method for combining a broad spectrum of fine-tuning actions for relational databases, while still respecting the limits of available computational resources. This differs from the above-mentioned works as they propose specific and fixed combinations for two types of fine-tuning techniques and do not present an algorithm for generating the combined actions. In addition, in contrast to our proposal, they do not provide a systematic method for exploring the fine-tuning actions, since their combination is only considered in the selection phase.

3. Methods and Strategies

We propose a comprehensive combination method that follows the classic automatic cycle [Bruno 2012] of collecting workload information, generating fine-tuning actions, evaluating them, and executing them in the database. The new strategies of *generation* and *selection* of combined actions are the main innovations concerning the state-of-the-art. Our proposed method has four steps and is illustrated in Figure 1:

1. **Workload capture**: The workload consists of the data stored in the database and a set of SQL commands. The method's execution flow starts with capturing the workload, which selects those representative elements.
2. **Generation** of combined solutions: this step uses the k elements selected in the previous step and suggest m combined solutions. Note that the number m of combined solutions may differ from the number of k SQL statements.

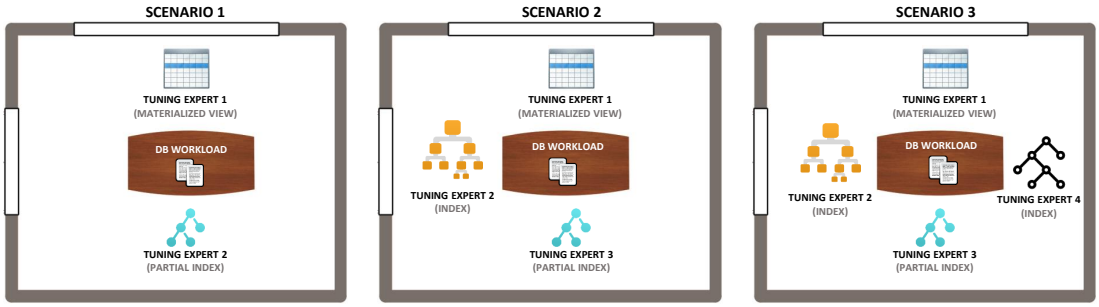

Figure 2. Possible scenarios of tuning expert committees.

3. **Selection** of combined solutions: here we filter the generated m solutions according to computational constraints. As a result, we list n combined solutions that may reduce the execution cost for the considered workload.
4. **Execution** of combined solutions: this last step is executing the n combined solutions filtered previously. As this execution can interfere with the performance and database availability in general, DBAs usually perform them at times of low demand, such as night shifts or maintenance slots.

3.1. Automatic Generation of Fine-Tuning Combined Solutions

There are different fine-tuning techniques in the literature and, for each of them, multiple generation methods for each type of action. It is possible to rewrite a query to generate materialized views using different methods (e.g. [Agrawal et al. 2000, Talebian and Kareem 2010, Vijay Kumar et al. 2010, Vijay Kumar and Ghoshal 2009, Vijay Kumar and Kumar 2013, Vijay Kumar and Kumar 2012, Baralis et al. 1997]), and there are several ways to propose (complete) indexes [Schnaitter et al. 2006,Bellatreche et al. 2013,Chaudhuri and Weikum 2006, Chen et al. 2010, Tran et al. 2015].

A tuning **expert** is an abstraction of a unique algorithm for generating combined solutions from a specific technique. For example, an algorithm capable of rewriting a SQL query or generating one or more materialized views. There may be two or more specialists of the same technique (e.g., MVs) as long as they use different algorithms.

An **Experts Committee** is a group of specialists who agree to generate solutions during a database system fine-tuning task. The inspiration came from the natural behavior of a possible group of human experts. A DBA typically specializes in a subset of tuning solutions. When in a group, they need to collaborate and negotiate so that they consider alternatives offered by other experts through an ubiquitous analysis of the problem. So, we propose the following premises for the formation of this committee:

1. The committee must have two or more experts so that the combination process is possible.
2. Theoretically, there is no maximum number of specialists, but there is a practical limit regarding the computational costs to perform the task.
3. There is no fixed expert pool: we may add new elements without needing to modify the combination method.
4. There is no execution order among experts.
5. There is only one instance of the same expert.

Figure 2 illustrates possible scenarios for the combination of expert committees that respect the proposed assumptions. *Scenario 1*: Two experts (minimum number) come together in a "virtual meeting room" to plan global fine-tuning solutions. One is specialized in materialized views and the other in partial indices. *Scenario 2*: there are three specialists, one from each specialty (IND, PIN, MV), at least one from each tuning type; Finally, *Scenario 3* illustrates a case where there are multiple specialists and more than one specialist of the same type of action (e.g. complete index) where each element represents a different generation algorithm (Experts 3 and 4).

It is essential to emphasize the complexity of generating a fine-tuning action (both local and global). It is also worth mentioning that finding the optimal solution for some particular techniques is, in practice, unfeasible. Selecting an appropriate set of MVs that minimize the total response time for a given workload is an NP-Hard problem [Agrawal et al. 2000], as well as the index selection problem [Chaudhuri et al. 2004].

Our approach needs to combine multiple actions that are already complex. The expert committee is an abstraction that breaks the problem into parts, solving the problem in a *divide-and-conquer* based approach. We propose here not only a theoretical method for the combination of individual tuning actions but, also, a strategy that can be implemented and is viable in practice. The actual implementation of these complex algorithms in a single integrated block of software would be very difficult, if not unfeasible.

The proposed strategy tries to combine different types of actions through the representation of experts as autonomous parts of an intelligent software system. In our case, we have instantiated them as software agents. However, the strategy of dividing the problem into independent parts suffers from the classic *Control Problem* studied in the Artificial Intelligence (AI) domain [Hayes-Roth 1985]. To solve a problem in a particular domain, a system with autonomous units performs a series of actions to reach the solution. Each activity is triggered by previously generated data or solutions and applies some source of domain knowledge to develop or modify the current solution. Several of these actions may be possible at each point in the solution process.

Therefore the control problem actually is: *among the potential actions, which one should a system with autonomous units (agents) perform at each point in the problem-solving process?* [Hayes-Roth 1985].

The problem of control is fundamental to all cognitive processes and intelligent systems. When solving it, a system decides, implicitly or explicitly, what problems it will try to solve, what knowledge it will bring, and which problem-solving methods and strategies will be applied. It also decides how it will evaluate alternative problem solutions, know when specific problems will be solved, and under what circumstances it will stop running for selected issues or sub-problems.

3.2. Combined-solutions Generation Algorithm in Action

Combinations can be relatively complex to generate. We present here the Combined-solutions Generation (CSG) algorithm (Algorithm 1), which is further detailed in [Oliveira 2019].

The CSG algorithm prevents solutions that do not go towards the optimal solution from expanding and, consequently, a more significant reduction in the search space.

The GSC has the following input data:

- **sql**: an SQL command captured from the workload for which the DBA wants to decrease the execution cost;
- **experts**: a list of independent agents to generate combined solutions capable of decreasing the executing cost of the SQL command;
- **cost_model**: to evaluate the combined solutions, generating predictions such as execution cost and benefits. Each RDBMS may have a specific cost model for different tuning action types.

The output of GSC is:

- **selected_solutions**: the set of combined solutions that minimize the cost of executing the given SQL command.

Algorithm 1: Combined-solutions Generation (CSG)

```
   input : sql, experts, cost_model.
   output: best_solution_found
 1 begin
 2     Q ← {sql};
 3     best_solution_found ← ∅;
 4     best_C ← {sql};
 5     best_cost ← ∞;
 6     while P ∈ Q do
 7         current_cost ← execution cost of sql using P calculated by
             cost_model;
 8         if current_cost < best_cost then
 9             best_cost ← current_cost;
10             S ← all combinations generated by the experts from P;
11             if S is a valid combination then
12                 for C ∈ S do
13                     current_cost ← execution cost of sql using C calculated
                         by the cost_model;
14                     if current_cost < best_cost then
15                         best_cost ← current_cost;
16                         best_C ← C;
17                     end
18                 end
19             end
20             if best_C ∉ best_solution_found then
21                 best_solution_found ← best_solution_found ∪ {best_C};
22             end
23             if best_C ∉ Q then
24                 Q ← Q ∪ {best_C};
25             end
26         end
27     end
28     return best_solution_found
29 end
```

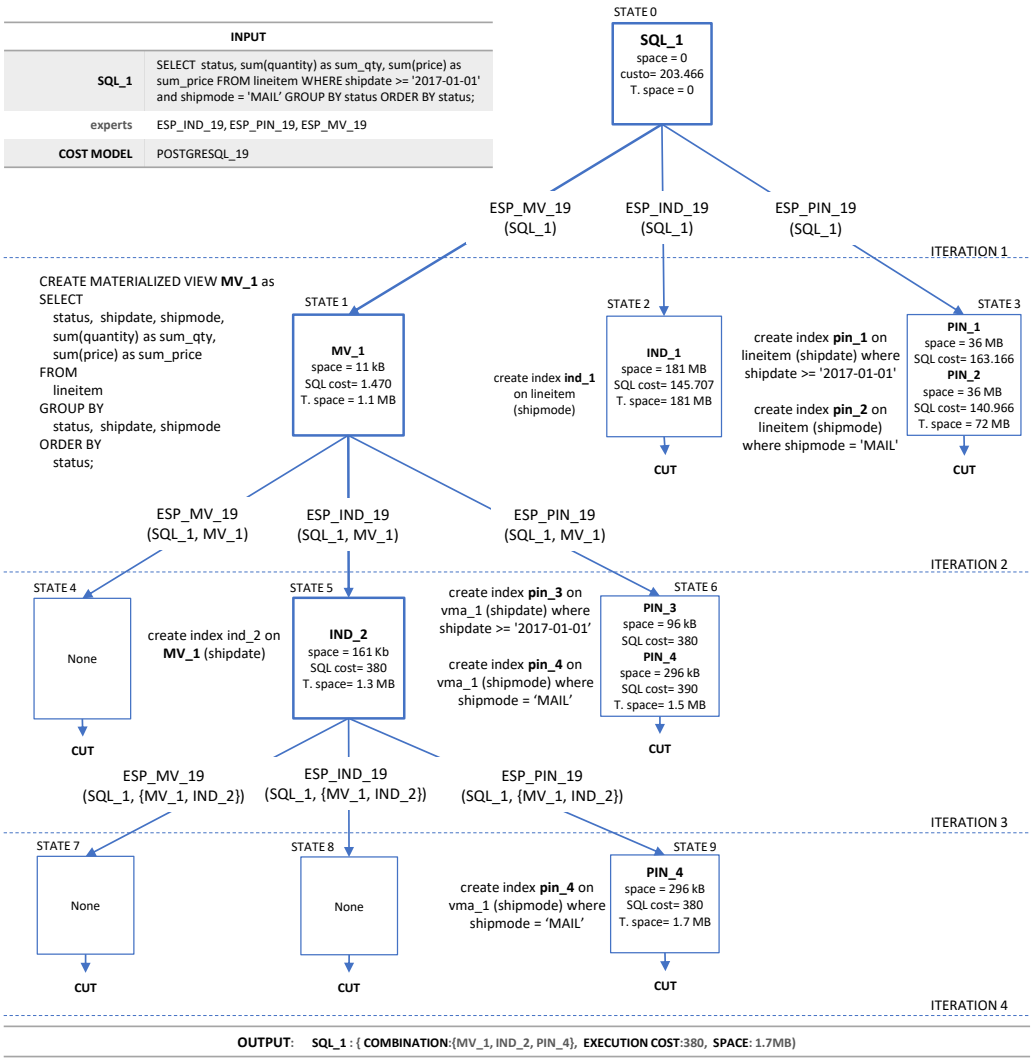

Figure 3. CSG Algorithm in action

Figure 3 illustrates a possible execution of the CSG algorithm, with a simple but realistic example, describing the algorithm step by step. The iterations are detailed at each stage with actual values from an execution extracted during the experiments using the PostgreSQL DBMS. The algorithm receives the following variables as its *input*:

1. **SQL_1**: a query-type SQL command that summarizes the quantity of items sold *(sum(quantity) as sum_qty)* and the value *(sum(price)* as *sum_price*) from the table *lineitem* that was dispatched on a date equal to or greater than *'01/01/2017' (shipdate* $\geq$ *'2017-01-01')*, shipping mode *(shipmode = 'MAIL')*, grouped and ordered by item status *(GROUP BY status ORDER BY status)*;
2. **ESP_IND_19, ESP_PIN_19, ESP_MV_19**: three experts, one for indices, another in partial indices, and another for materialized views.
3. **POSTGRESQL_19**: a model for predicting the costs of access structures with respect to creation and persistency, and the corresponding benefit to the workload.

First iteration: The SQL command generates the state *STATE 0* which has an execution cost of 203,466 in PostgreSQL [The PostgreSQL Global Development Group 2019] cost units and the computational resources considered by the cost model (in this

case, secondary memory space). Note that *STATE 0* represents the command execution without any fine-tuning action. This state can also be a final solution if no other step lowers its execution cost.

Second iteration: The algorithm applies the BnB expansion rule. Each of the experts listed as input generates combined solutions from the active state *STATE 0*. This results in the creation of STATE 1 by the expert *ESP_MV_19* who proposes the action *MV_1* ; *STATE 2* by the expert *ESP_IND_19* with the action *IND_1* ; and *STATE 3* by the expert *ESP_PIN_19* who suggests two actions: *PIN_1* and *PIN_2* .

The second iteration shows the first combination step, where each expert creates a sub-tree with their solution as the tree's root to be expanded in the following iterations. In this example, we do not select the states *STATE 2*, and *STATE 3* as active states since the pruning rule of the CSG algorithm only determines the best solution at each iteration; in this case *STATE 1* . It is worth mentioning that if we applied the relaxed pruning rule of the CSGR algorithm, both *STATE 2* and *STATE 3* would be expanded, generating a more significant number of states but, consequently, testing a greater number of possible combinations.

Third iteration: The only active state *STATE 1* is expanded to the states *STATE 4* , *STATE 5* and *STATE 6* . The state *STATE 4* has no fine-tuning action, as the expert *ESP_MV_19* proposed the solution of *STATE 1* in the previous iteration. *ESP_IND_19* proposes *STATE 5* with the creation of an index *IND_2* on the materialized view *MV_1* , and *ESP_PIN_19* proposes *STATE 6* with the creation of partial indices *PIN_3* and *PIN_4* on *MV_1* .

In this iteration, *STATE 5* is selected as the active state because it has the lowest cost. Note that *STATE 6* shows a solution with two partial indices *PIN_3* and *PIN_4* for *MV_1* where the lowest cost of the solution (*PIN_3*) has the same predicted cost as state *STATE 5* (*IND_2*), but *STATE 5* was executed first by the breadth-first selection rule. Therefore, *STATE 5* remains the active state of the iteration.

The fact that costs and benefits are predictions motivated the study of the relaxed version of the pruning rule and the CSGR algorithm. A full index can be more generic than a partial index and benefit from a more significant number of queries. However, if the evaluated command has a high frequency in the workload, we would have missed a suitable solution such as *STATE 6* and all its later variations. In some cases where the difference between the solutions is tiny, the pruning rule of the CSG algorithm may be restrictive and not expand states that would possibly bring more significant benefits to the evaluated query.

Fourth iteration: The expert *ESP_MV_19* tries to expand to *STATE 7* but is unable to propose any action, as he has already contributed to this combined solution in *STATE 1*. The same is true for the expert *ESP_IND_19* who contributed to the solution in *STATE 5*. *ESP_PIN_19*, the only one that has not yet contributed to this thread, then suggests the partial index *PIN_4*.

In the third iteration the expert *ESP_PIN_19* suggested the *PIN_3* and *PIN_4* for the *MV_1*, but in the iteration 4 the shipdate column already has an index proposal (*IND_2*). So *PIN_3* would be an invalid solution for the current state of the combination, so only *PIN_4* is suggested.

Stop State: After expansion by all experts in iteration 4, no state is selected as an active state since the solution proposed in *STATE 6* did not decrease the cost of executing the analyzed SQL command. Therefore, the execution is terminated by the stop condition in which all possible expansions have been tested, and the active state queue is empty.

4. Evaluation

We carried out experiments to evaluate the effectiveness and efficiency of the proposed global fine-tuning method in different test scenarios and different DBMSs. Due to lack of space, we present the results only on a single DBMS (PostgreSQL).

4.1. Experiments Specification

Setup. The server consisted of a 3.60GHz Intel Core I7-7700 processor, 32GB RAM, 1TB Hard Disk (HDD) with Windows Server 2016 64-bit. To avoid competition for resources between the DBMS and the fine-tuning tool, an independent client machine was used to run the tool. Although the client hardware did not influence the results since the query processing and relevant statistics are from the server, for reproducibility purposes it had an Intel I7-7820 2.9GHz processor, 12GB of RAM, 512GB HDD, and Windows 10 64bit.

Metric. Performance was compared using the query execution cost unit of each DBMS optimizer. Cost unit is an arbitrary value for each DBMS, typically dependent of: i) the cost of reading the disk pages needed to execute the query, and ii) the CPU cost of processing data in main memory.

Benchmark. We chose TPC Benchmark™ H (TPC-H) [1], which illustrates a decision support system that examines a large volume of data, executes highly complex queries, and provides answers to critical business questions.

Given the complexity of the queries, we considered 10GB a sufficient data volume to demonstrate the effectiveness and efficiency of the method, while allowing for the execution of a large number of tests. Thus, we generated a 10GB database, which grew to 22GB after inserting data and creating primary and foreign keys. Since different instantiations from the same query model may have different execution costs due to different constraint values in the attributes of the *where* clause, the workload consisted of 30 instantiations of each of the 22 models provided by TPC-H, totaling 660 queries.

It is worth noticing in our experiment specification that the workload from the TPC benchmark is highly unbalanced among the TPC-H query models. The 60 queries generated from Q17 (37%) and Q20 (62%) models are the most costly and correspond to 99% of the total workload.

Scenarios. We performed tests using the following scenarios:

1. ***ORIGINAL***: Original costs of all queries (no fine-tuning actions). Database with the primary indexes, primary and foreign keys suggested by the benchmark.
2. ***IND***: only the complete index specialist is enabled.
3. ***PIN***: only the partial index specialist is enabled.
4. ***MV***: only the materialized view specialist is enabled.

[1] http://www.tpc.org/

5. ***IND+PIN***: only the complete and partial indexes specialists are enabled.
6. ***IND+PIN+MV***: only the complete and partial indexes and materialized view specialists are enabled.
7. ***(MV+IND)***: only the materialized view and complete indexes specialists are enabled. Only combined solutions that have materialized views indexed by complete indexes are considered, indexes are not created on structures other than MVs.
8. ***(MV+PIN)***: only the materialized view and partial indexes specialists are enabled. Only combined solutions that have materialized views indexed by partial indexes are considered, indexes are not created on structures other than MVs.
9. ***(MV+IND+PIN)***: only materialized view, complete indexes, and partial indexes specialists are enabled. Only combined solutions that have materialized views indexed by complete and/or partial indexes are considered, indexes are not created on structures other than MVs.
10. ***IND + PIN + (MVs+IND+PIN)***: All specialists are considered, and all valid combined solutions are considered during the generation and selection processes.

Methodology. To evaluate each scenario, the following five steps were performed (except the ORIGINAL, which executes only the **(v)** step): **(i)** the specialist(s) selected for the fine-tuning task is/are activated; **(ii)** The 660 queries are executed in random order; **(iii)** The specialists observe the workload, generate and select the combined solutions; **(iv)** The selected combined solutions are executed on the database; **(v)** The 660 queries are executed in random order and the costs of each execution are collected.

4.2. Results and Discussions

This subsection presents the results using PostgreSQL in all the scenarios. We present an overview comparing the different scenarios, then discuss a solution with combined tuning strategies.

Figure 4 shows that all combinations of specialists provided benefits to the workload. In scenarios with one enabled specialist (IND, PIN, MV) the best performance was achieved by the complete index specialist (IND), with total execution cost equivalent to 1% of the ORIGINAL scenario (i.e. a benefit of 99%). The second best was the partial index specialist (PIN) with 37% of the ORIGINAL cost, followed by MV with 73%.

The right part of Figure 4 zooms in the top-3 best scenarios shown in its left part. As the possibilities of combinations increases, the results improve (lower cost). The complete index specialist is the most efficient, and combining its solutions with the ones from the other two specialists benefits the workload even more. It was evidenced that the best performance was obtained in the IND+PIN+(MV+IND+PIN) scenario - the one that accepts all types of combinations.

An interesting combination of tuning strategies generated from our proposed method benefited the instance 10 of query 4 (Q4), and is detailed in Figures 5 and 6.

According to TPC-H, Q4 aims to determine the efficiency of the order system simulated by the benchmark, filtering out orders that were not delivered late (*l_commitdate* $<$ *l_receiptdate*). The execution plan of this query without any tuning action can be seen in Figure 8.

Note that the most costly operation is a full scan on *lineitem*, the largest table in

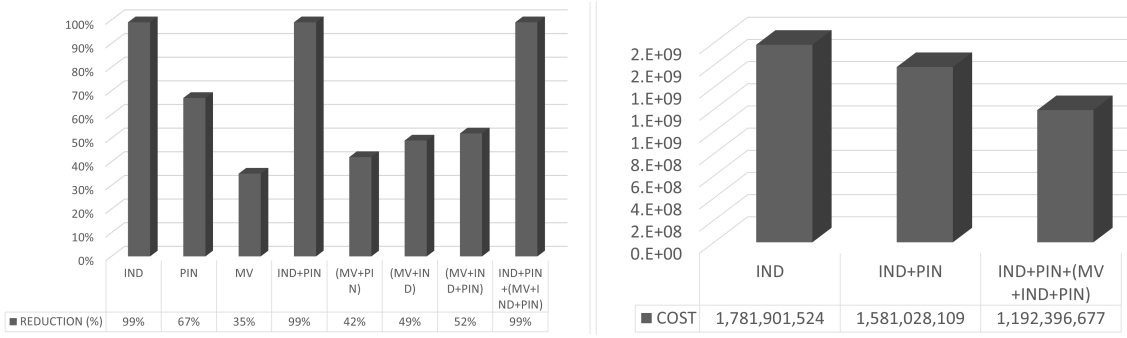

Figure 4. a) Execution cost for each scenario relative to the original cost, and, b) Comparison of the cost of execution between the best tested scenarios

```
/* TPC_H Q4_instance_10 */
SELECT o_orderpriority, count(*) AS
order_count
FROM orders
WHERE o_orderdate >= date '1998-07-25'
AND o_orderdate < date '1998-07-25' +
interval '3 month'
AND EXISTS
    (SELECT *
     FROM lineitem
     WHERE l_orderkey = o_orderkey
     AND l_commitdate < l_receiptdate )
GROUP BY o_orderpriority
ORDER BY o_orderpriority;
```

Figure 5. Example of Query Q4.

```
/* Combined Solution for TPC_H Q4_instance_10 */
/* ACTION 1 */
CREATE materialized VIEW MV_TAP_Q4 AS
SELECT * FROM lineitem WHERE l_commitdate <
l_receiptdate;

/* ACTION 2 */
CREATE INDEX ID_TAP_S_N914717080 ON
mv_tap_q4(l_orderkey);

/* ACTION 3 */
CREATE INDEX PID_TAP_N285721920 ON orders
(o_orderdate)
WHERE orders.o_orderdate >= '1998-07-25'
AND   orders.o_orderdate  < '1998-10-25';
```

Figure 6. Combined Solution for TPC_H Q4_instance_10

the database (67% of its total size), followed by another full scan on the *orders* table (the 2nd largest table with 18%).

The combined candidate solution generated from our method for *Q4_instance_10* (Figure 5) has three actions. ACTION is a materialized view (*MV_TAP_Q4*) for the Q4 subquery that filters the items delivered on time from the *lineitem* table. Note that this materialized view may benefit any Q4 instance. ACTION 2 suggests the creation of an index for the materialized view proposed in ACTION 1 on the *l_orderkey* column. This column is used as a filter by the outer query, and therefore, an opportunity to obtain tuples of the MV without the need for a full scan through a complete index. Finally, ACTION 3 suggests the creation of a specific partial index for *Q4_instance_10* where a partial index indexes the column *o_orderdate* used by the outer query to filter the table *orders*.

The execution plan of *Q4_instance_10* after the execution of the previous combined solution can be seen in Figure 7. The query optimizer replaced the two full scans of the original plan with scans using the indexes proposed by the combined solution.

It is important to highlight that the proposed method suggested tuning actions for both the outer query and the subquery of Q4. In general, existing approaches in the literature have difficulties in dealing with subqueries.

Furthermore, the strategy of traversing the search space by brute force allows different specialists to test a large number of permutations between their actions in a systematic way. If, on the one hand, the cost of testing this number of combinations can be

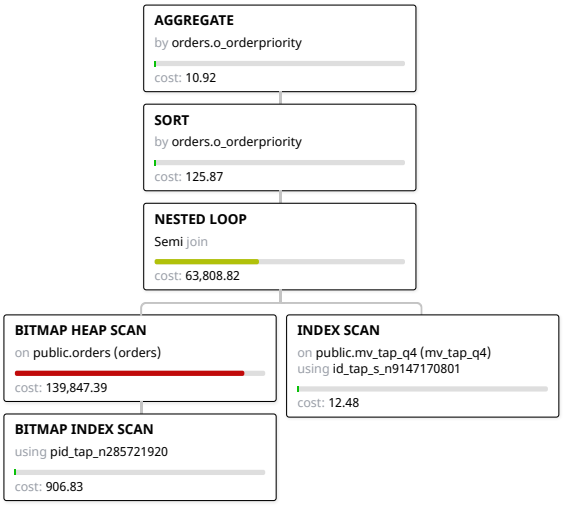

Figure 7. Execution Plan of *Q4_instance_10* - IND+PIN+(MV+IND+PIN) Scenario.

Figure 8. Execution Plan of *Q4_instancia_10* - ORIGINAL Scenario

expensive and needs to be controlled, on the other hand, it allows complex solutions that perhaps a human DBA would have difficulties in planning and testing.

5. Conclusions

In this research work we propose a technique-independent fine-tuning action combination method capable of generating combined actions among different techniques traversing the search space and applying constraints due to the complexity of testing all alternatives; and select through a global strategy the set of solutions combinations that minimize the cost of running the workload and respect the limits of available computing resources.

As future work we plan to insert updates statements in the workload and consequently adapt the cost model. New specialists may be added too with new database tuning strategies, such as: cluster indexes, query rewritten, horizontal partitioning among others.

References

Agrawal, S., Chaudhuri, S., and Narasayya, V. R. (2000). Automated selection of materialized views and indexes in SQL databases. In *Procs VLDB Conf*, pages 496–505.

Baralis, E., Paraboschi, S., and Teniente, E. (1997). Materialized views selection in a multidimensional database. *Procs VLDB Conference*, pages 156–165.

Bellatreche, L., Boukhalfa, K., and Mohania, M. (2013). Pruning Search Space of Physical Database Design. *Procs DEXA Conference*, 63(8):479–488.

Bruno, N. (2012). *Automated Physical Database Design and Tuning*. CRC Press.

Chaudhuri, S., Datar, M., and Narasayya, V. (2004). Index selection for databases: A hardness study and a principled heuristic solution. *IEEE TKDE*, 16(11):1313–1323.

Chaudhuri, S. and Weikum, G. (2006). Foundations of automated database tuning. *Procs SIGMOD Conference*, pages 964–965.

Chen, S., Nascimento, M. A., Ooi, B. C., and Tan, K.-L. (2010). Continuous online index tuning in moving object databases. *ACM TODS*, 35(3):1–51.

Chirkova, R. and Yang, J. (2012). Materialized views. *Fnd. Trends in DBs*, 4(4):295–405.

de Oliveira, R. P., Baião, F., Almeida, A. C., Schwabe, D., and Lifschitz, S. (2019). Outertuning: an integration of rules, ontology and RDBMS. In *Procs. Brazilian Symposium on Information Systems SBSI*, pages 1–8. ACM.

Elfayoumy, S. and Patel, J. (1999). Database performance monitoring and tuning using intelligent agent assistants. In *Procs Intl Conf Artificial Intelligence*, pages 331–335.

Hartigan, J. A. and Wong, M. A. (1979). Algorithm AS 136: A K-Means clustering algorithm. *Applied Statistics*, 28(1):100–108.

Hayes-Roth, B. (1985). A blackboard architecture for control. *AI*, 26(3):251–321.

Kimura, H., Huo, G., Rasin, A., Madden, S., and Zdonik, S. B. (2010). CORADD: Correlation aware DB designer mat. views and indexes. *PVLDB*, 3(1-2):1103–1113.

Kwon, O., Im, G. P., and Lee, K. C. (2011). An agent-based web service approach for supply chain collaboration. *Scientia Iranica*, 18(6):1545–1552.

Lawler, A. E. L. and Wood, D. E. (1966). Branch-And-Bound Methods : A Survey Published. *Operations Research*, 14(4):699–719.

Mrozek, D., Malysiak-Mrozek, B., Mikolajczyk, J., and Kozielski, S. (2014). Database Under Pressure – Testing Performance of Database Systems Using Universal Multi-Agent Platform. *Man-Machine Interactions 3*, pages 631–641.

Oliveira, R. P. d. (2019). *Automatic Combination and Selection of Tuning Actions (in portuguese)*. Phd thesis, Pontifícia Universidade Católica do Rio de Janeiro.

Schnaitter, K., Abiteboul, S., Milo, T., and Polyzotis, N. (2006). COLT: Continuous On-line Tuning. *Procs SIGMOD Conference*, pages 1–23.

Shasha, D. and Bonnet, P. (2002). *Database Tuning: Principles, Experiments, and Troubleshooting Techniques*. Elsevier Science.

Stonebraker, M. (1989). The Case for Partial Indexes. *SIGMOD Conf*, 18(4):4–11.

Talebian, S. H. and Kareem, S. A. (2010). A lexicographic ordering genetic algorithm for solving multi-objective view selection problem. *Procs ICCRD Conf*, pages 110–115.

The PostgreSQL Global Development Group (2019). Explain PostgreSQL.

Tran, Q. T., Jimenez, I., Wang, R., Polyzotis, N., and Ailamaki, A. (2015). Rita: An index-tuning advisor for replicated databases. *Procs SSDBM Conf*, pages 22:1–22:12.

Vijay Kumar, T. and Ghoshal, A. (2009). A reduced lattice greedy algorithm for selecting materialized views. *Information Systems, Technology and Management*, 31:6–18.

Vijay Kumar, T. and Kumar, S. (2012). Materialized view selection using genetic algorithm. *Contemporary Computing*, 306:225–237.

Vijay Kumar, T. and Kumar, S. (2013). Materialized view selection using iterative improvement. *Advances in Computing and Information Technology*, 178:205–213.

Vijay Kumar, T. V., Haider, M., and Kumar, S. (2010). Proposing Candidate Views for Materialization. *Information Systems, Technology and Management*, 54:89–98.