TuningChef: an approach for choosing the best cost-benefit database tuning actions

Victor Augusto Lima Lins de Souza[1], **Sergio Lifschitz**[1]

[1] Departamento de Informática – PUC-Rio

{vlins,sergio}@inf.puc-rio.br

Abstract. *While many research works propose a way to list a set of fine-tuning options for a given workload, only a few offer a way to help the DBA make better decisions when encountering a set of available options, especially when taking his possibilities into consideration. We propose and develop a step-by-step decision process. Given a set of fine-tuning options, we recommend the most cost-benefit subset. Enough context for the DBA accompanies the recommendation to understand its reasoning, with the possibility of letting the user build his own subset and check the expected impact. We show some experimental results on actual database systems that further explain our approach and solution.*

1. Introduction and Related Work

Although vastly researched, a complete (automatic) fine-tuning process for Relational Database Systems (RDBMS) is still highly dependent on specialized users such as Database Administrators (DBA). One must be able to analyze the database workload, list as many fine-tuning actions that might have a positive impact as possible, and then make a cost-benefit analysis on them to decide which are worthy of being implemented.

This ongoing research work proposes a way of helping the DBA make more reasonable decisions and even automate them when deciding which fine-tuning actions are worth it, given an initial set of options and a workload. We do this by presenting enough context to understand the reasoning behind each decision made our recommendation.

Many researchers have proposed ways to increase relational database system's performance in different ways. Some focus on database configuration options, while others suggest index or materialized views that might benefit a given workload.

OtterTune [Aken et al. 2017], CDBTune+ [Zhang et al. 2021] and DB-BERT [Trummer 2022] are examples of works focused on parameter tuning with applied Machine Learning techniques. OtterTune uses supervised and non-supervised machine learning methods to determine near-optional configuration parameters for PostgreSQL. In contrast, CDBTune+ uses Deep Reinforcement Learning (DRL), and DB-BERT uses natural language processing (NLP) to provide near-optimal configurations for multiple RDBMSs. Some authors focus on creating access methods, like indexes and materialized views. We may cite those making use of inference through ontology, like OnDBTuning [Perciliano et al. 2021] and OuterTuning [Oliveira 2015]. Others suggest access structures using machine learning techniques [Ding et al. 2019].

All those works have in common is that they focus on generating a specific subgroup of fine-tuning strategies options and do not compare nor combine those options with others, or even against ones that the DBA might find beneficial. They also do not

provide enough context to justify why one option might be better than another. It is up to the DBA to see the expected cost-benefit of his own selected set of options.

2. TuningChef Proposal

The RDBMS fine-tuning process can be divided into two main steps: (1) listing the available options and (2) choosing the best ones. During this research, we are focusing on the latter, the decision step.

For a DBA to decide on the best fine-tuning options in a given set, one needs to evaluate their cost and benefit in a given workload. Here the cost can be, for example, using more disk space for increased performance (better query execution times).

Let us consider that for making decisions for each available fine-tuning option and an RDBMS workload, one must be able to check how much an option might impact each workload query and its trade-off/cost. *TuningChef* proposes a solution for the decision step, enabling even a mix of multiple tuning actions.

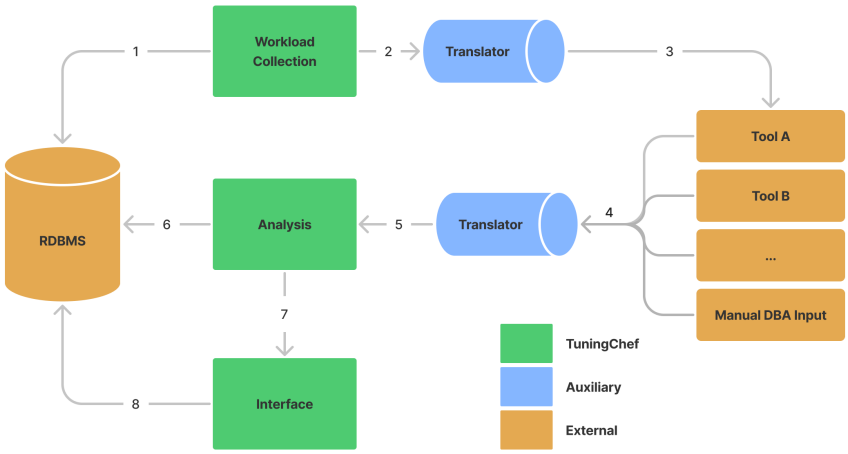

Figure 1. Proposed architecture

Our tool must be extensible and able to run independently. Therefore, we have designed the architecture (Figure 1): green blocks represent *TuningChef* components that run separately from one another. Orange ones are external systems that can be plugged in *TuningChef*, while blue ones are auxiliary modules that bind everything together.

In order to make backed-up decisions, collecting workload statistics is crucial. For this reason, workload collection is considered a part of *TuningChef*. This module is responsible for collecting executed queries and, for each of them, their total and mean execution time, total times executed, and parameter distribution. Once collected (step 1), this workload information can be used by other tools by translating it (steps 2 and 3).

The analysis module (Figure 1) does all the decision work. For this reason, it needs a set of fine-tuning options. Those can come from many different sources and be translated for a format that *TuningChef* accepts (steps 4 and 5) through a YAML file. The cost of each option is calculated and compared to its expected impact on the workload [Souza 2022]. If lower than the gain, it is considered beneficial.

The interface module is responsible for showing the recommended decisions with enough context to understand its reasoning (step 7). It also lets the administrator make his

own decisions and see the expected impact and trade-offs. Putting the DBA in the loop let him use future knowledge, like the expected workload changes, into consideration when choosing the tuning options.

3. Implementation and Experimental Results

Initial work makes use of hypothetical structures for simple, partial and compound indexes, and materialized views to simulate their impact on a given workload. Other tuning actions, like parameter changes, can be added through new heuristics.

Hypothetical indexes are available natively in SQLServer but can be simulated or included in other RDBMSs through extensions. During this work, we focus on PostgreSQL and the HypoPG extension. This decision was made due to existing research that shows the benefit of using this extension [Kossmann et al. 2020, Schlosser and Halfpap 2020] when analyzing possible gains of an index. The costs of an index are also taken into consideration, both for creating and maintaining it.

```
1  select sum(l_extendedprice + (1 - l_discount)) as revenue,
2         l_orderkey, o_orderdate, o_shippriority
3    from customer, orders, lineitem
4   where c_mktsegment = 'BUILDING'
5     and c_custkey = o_custkey and l_orderkey = o_orderkey
6     and o_orderdate < date '1995-03-15'
7     and l_shipdate > date '1995-03-15'
8  group by l_orderkey, o_orderdate, o_shippriority
9  order by revenue desc, o_orderdate;
```

(a): Example query

```
1  create materialized view mv_revenue as
2  select sum(l_extendedprice + (1 - l_discount)) as revenue,
3         l_orderkey, o_orderdate, o_shippriority,
4         c_mktsegment, l_shipdate
5    from customer, orders, lineitem
6   where c_custkey = o_custkey and l_orderkey = o_orderkey
7  group by l_orderkey, o_orderdate, o_shippriority,
8           c_mktsegment, l_shipdate
9  order by revenue desc, o_orderdate;
```

(b): Example materialized view

Figure 2. Materialize view example

A query rewrite using a Common Table Expression (*CTE*) is applied for **Hypothetical Materialized Views**. To exemplify how this is done, let us look at the example in Figure 2, with **a** being the original query and **b** a materialized view proposed for it. Firstly we identify which conditions exists in **a** but not in **b**, them being the lines 4, 6 and 7 of **a**. Then, we create a *CTE* with **b** query and apply a *SELECT* in it with the aforementioned extracted conditions, which gives us the following result:

```
1  with mv_revenue as materialized (
2      select sum(l_extendedprice + (1 - l_discount)) as revenue,
3             l_orderkey, o_orderdate, o_shippriority,
4             c_mktsegment, l_shipdate
5    from customer, orders, lineitem
6   where c_custkey = o_custkey and l_orderkey = o_orderkey
7  group by l_orderkey, o_orderdate, o_shippriority,
8           c_mktsegment, l_shipdate
9  order by revenue desc, o_orderdate)
10 select *
11 from mv_revenue
12 where c_mktsegment = 'BUILDING'
13   and o_orderdate < date '1995-03-15'
14   and l_shipdate > date '1995-03-15';
```

Figure 3. Resulting CTE

Extracting Figure 3 query execution plan using *EXPLAIN* clause gives us the expected cost of creating a temporary table (*CTE* cost), which is the same as executing the proposed materialized view query, and the expected cost of running a query on the materialized view (lines 10-14) if it was created. With a solution for creating the hypothetical

structures at hand, we can now evaluate their impact on a given workload. For this, a recursive algorithm developed, which is able to find beneficial combinations for the given tuning options. To optimize the search space, *Branch and Bound* is applied to restrict how deep our combination tree should be, based on [Oliveira 2019]. This optimization helps us reduce the search space by avoiding paths with incompatible fine-tuning options.

As a result of the proposed algorithm, we have a tree where each node represents a tuning action (in our case, an index or materialized view creation) with statistics like its creation and maintenance cost and impact on each workload query. With this information, we can recommend the tuning path with the highest cost-benefit and show the DBA the reasoning behind it while allowing the administrator to build his custom path and see its expected impact. All this information is available to the DBA through an interface.

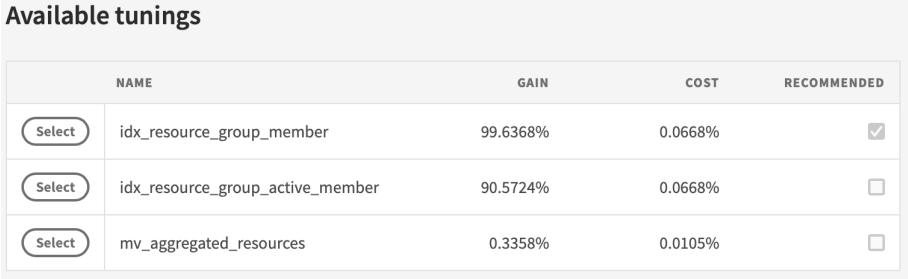

Figure 4. Available Tuning Actions

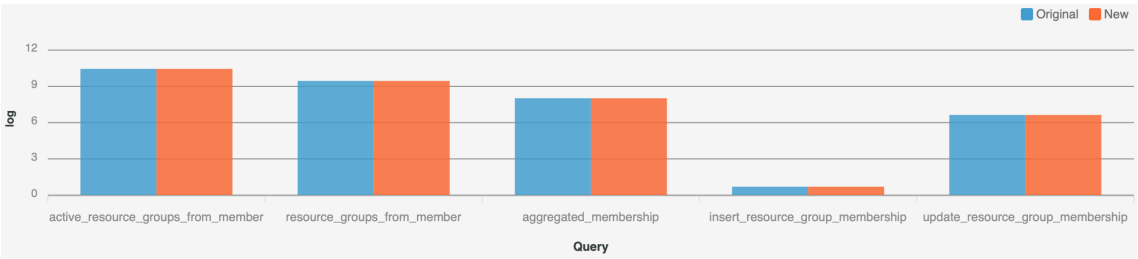

Figure 5. Queries Cost Distribution

Figures 4 and 5 show the *TuningChef* main interface. Figure 4 lists the available tuning options, together with its possible gain, cost, and if it is a recommendation. Gain and cost represent a % of the total collected workload cost. Figure 5 shows the original cost of each query (orange bars) and the expected cost (blue bars) if the selected tuning actions were applied. In the screenshots, the values are the same as none were selected.

Each of the tuning options can be selected to show its details (see Figure 6). The DBA can see the action's expected impact on each workload query. The blue bars are the current cost of each query (they might be different from the original cost if any previous actions were selected) and the expected new cost if the detailed action is applied. The raw *SQL* command and some additional details are also provided.

If the recommended tuning actions are selected, we can see the final result in Figure 7. The DBA can see the expected impact of the recommended path (*idx_resource_group_member* and *mv_aggregated_resources*) on each of the workload queries, and the total expected gain and cost as a percentage of the total workload cost.

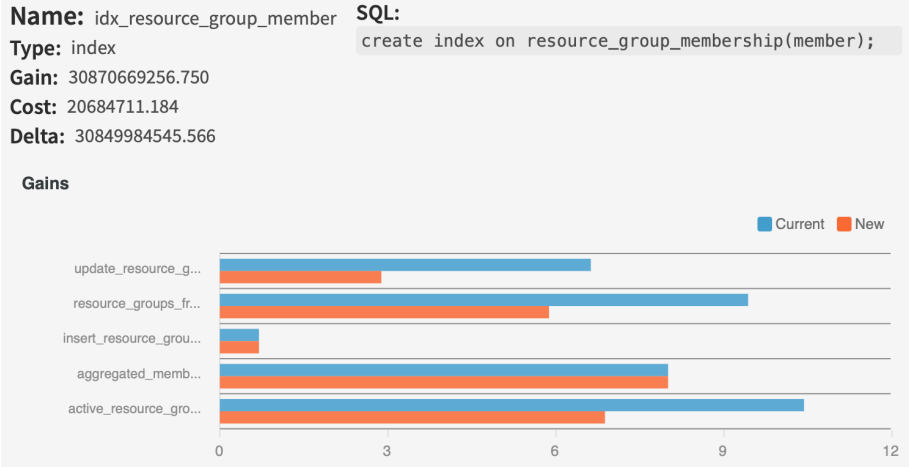

Figure 6. Tuning Details

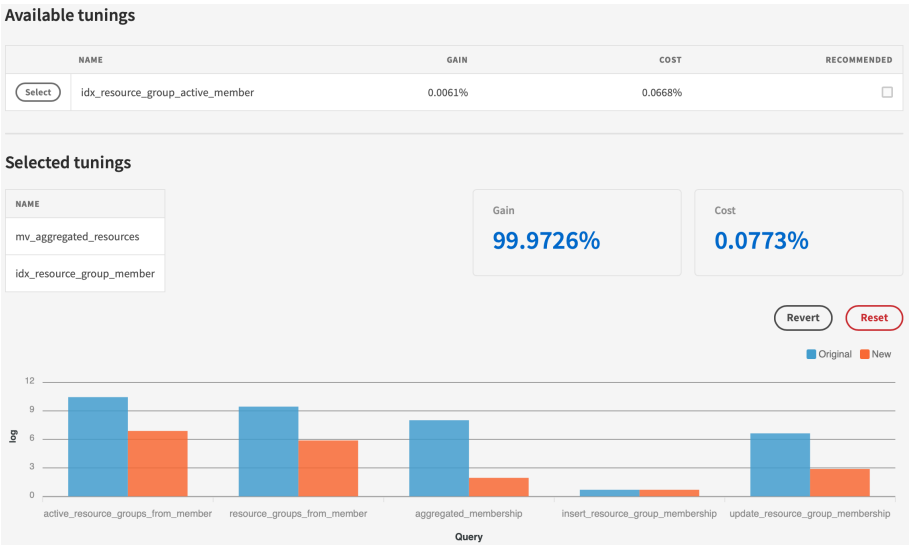

Figure 7. Recommended tunings selected

One tuning action is still on the list, but it was not recommended because its cost is higher than its gain. When clicked, the administrator can see the non-recommended tuning details as shown in Figure 8. This tuning is a partial index that would only impact one query by a small amount (the chart results are shown with *log* value). This happens because a previously selected index was enough to improve the query performance. Because the DBA might know some information that *TuningChef* does not know, like an expected increase of query usage, he can still choose to apply the non-recommended action.

4. Conclusions

In this ongoing work a way to support the decision step of a fine-tunning process is proposed. This is done though the use of hypothetical access structures and a algorithm able to combine and analyze the impact and costs of each of them. As an extra contribution, a new way of estimating materialized view impacts is proposed. For future work, *TuningChef* we expected to analyze other fine-tuning techniques, like data partitioning.

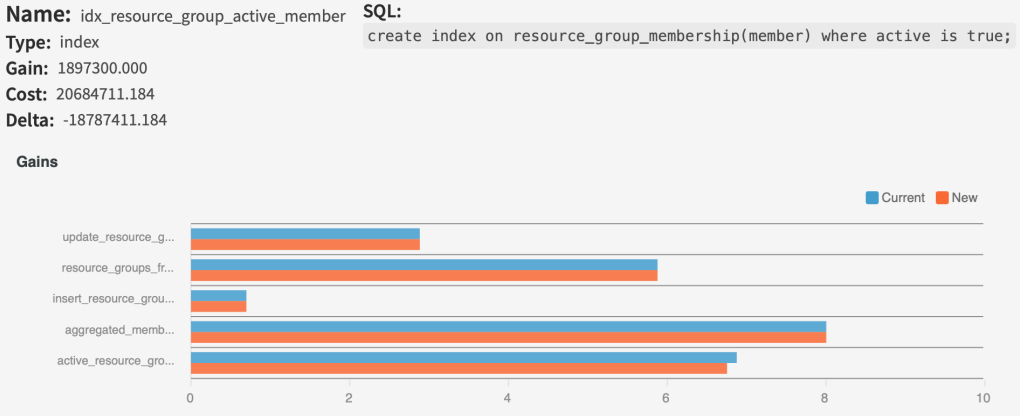

Figure 8. Not recommended tuning details

References

Aken, D. V., Pavlo, A., Gordon, G. J., and Zhang, B. (2017). Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1009–1024. ACM.

Ding, B., Das, S., Marcus, R., Wu, W., Chaudhuri, S., and Narasayya, V. R. (2019). AI meets AI: Leveraging query executions to improve index recommendations. In *Procs Intl Conf on Management of Data*, pages 1241–1258, Amsterdam Netherlands.

Kossmann, J., Halfpap, S., Jankrift, M., and Schlosser, R. (2020). Magic mirror in my hand, which is the best in the land? an experimental evaluation of index selection algorithms. *Proc. VLDB Endow.*, 13(12):2382–2395.

Oliveira, R. (2015). Ontology-based fine tuning: the case of materialized views (in portuguese). Master's thesis, PUC-Rio.

Oliveira, R. (2019). *Automatic Selection and Combination of Tuning Actions (in portuguese)*. Phd, PUC-Rio.

Perciliano, L., dos V. Santos, Baião, F., Haeusler, E. H., Lifschitz, S., and Almeida, A. C. (2021). Inferencing relational database tuning actions with ondbtuning ontology. In *Anais do XXXVI Simp. Bras. de Bancos de Dados (SBBD)*, pages 157–168.

Schlosser, R. and Halfpap, S. (2020). A decomposition approach for risk-averse index selection. In *32nd Intl Conf Scientific and Statistical Database Management*.

Souza, V. (2022). TuningChef: an approach for choosing best cost-benefit tuning actions (in portuguese). Msc, PUC-Rio.

Trummer, I. (2022). DB-BERT: A database tuning tool that reads the manual. In *Procs Intl Conf on Management of Data*, pages 190–203.

Zhang, J., Zhou, K., Li, G., Liu, Y., Xie, M., Cheng, B., and Xing, J. (2021). CDBTune+: An efficient deep reinforcement learning-based automatic cloud database tuning system. *The VLDB Journal*, 30(6):959–987.