Evaluating Database Self-Tuning Strategies in a Common Extensible Framework

Rafael Pereira de Oliveira[1], **Sergio Lifschitz**[1], **Marcos Kalinowski**[1], **Marx Viana**[1], **Carlos Lucena**[1], **Marcos Antonio Vaz Salles**[2]

[1]Pontifical Catholic University of Rio de Janeiro (PUC-Rio)
[2]University of Copenhagen

Abstract. *Database automatic tuning tools are an essential class of database applications for database administrators (DBAs) and researchers. These self-management systems involve recurring and ubiquitous tasks, such as data extraction for workload acquisition and more specific features that depend on the tuning strategy, such as the specification of tuning action types and heuristics. Given the variety of approaches and implementations, it would be desirable to evaluate existing database self-tuning strategies, particularly recent and new heuristics, in a standard testbed. In this paper, we propose a reuse-oriented framework approach towards assessing and comparing automatic relational database tuning strategies. We employ our framework to instantiate three customized automated database tuning tools extended from our framework kernel, employing strategies using combinations of different tuning actions (indexes, partial indexes, and materialized views) for various RDBMS. Finally, we evaluate the effectiveness of these tools using a known database benchmark. Our results show that the framework enabled instantiating useful self-tuning tools with a low effort by just extending well-defined framework hot-spots. Hence, sharing it with the database research community may facilitate evolving state of the art on self-tunning strategies by enabling to evaluate different strategies on different RDBMS, serving as a common and extensible testbed.*

1. Introduction

We may find many different self-tuning systems available to run, externally coupled, or integrated within a DBMS. There is a need for evaluating their effectiveness and efficiency due to the variety of automatic tuning tools, mainly for relational DBMSs. Either we might want to check for existing features that are mandatory for a given application, or we would like to characterize the pros and cons of implemented heuristics when compared with new ones proposed in the literature. These provide the best-expected results for the database system performance.

There exist specific tuning tasks that might have automatic support in a given RDBMS, but not others. For example, it is possible to tune database parameters to server hardware with PostgreSQL [PGTune 2019] automatically. Still, it is hard to find tools for automating a variety of physical database design tasks (e.g., creating materialized views or partial indices). It is infeasible for the DBA to, promptly, test and evaluate all possible physical design tuning actions for a given workload, as even index selection is NP-hard [Piatetsky-Shapiro 1983]. Nevertheless, tools to aid in this task exist for other RDBMS [Bruno 2012].

Software reuse is one of the significant goals of software engineering research [Frakes and Kang 2005]. It aims to reduce development and maintenance efforts and, at the same time, improve software quality due to the use of already designed software units, implemented, validated, and tested [Peters and Pedrycz 2000] [Sommerville 2011]. We observe that database automatic tuning tools share a core of common tasks, even though they might suggest different tuning action types. All self-tuning systems rely on basic tasks such as workload capture, data extraction, SQL parsing, or DBMS catalog access. Hence, this research area could take advantage of software reuse. However, reuse opportunities to support the comparison and evaluation of database self-tuning tools have not yet been systematically explored.

Aiming at taking a step towards bridging this gap, this paper describes how we may instantiate a reuse-oriented framework to support the evaluation of self-tuning tools for relational database systems. This framework reduces the effort of building a specialized database application by identifying a typical common architecture and enabling customization through well-defined framework hot-spots that may implement specific tuning strategies.

In pursuit of this aim, this paper makes the following contributions:

1. A suggested list of reuse-oriented requirements for database self-tuning tools where we discuss possible modeling options and justify our choices;
2. A component-based architecture of a framework to create self-tuning applications for relational databases with its corresponding open-source implementation;
3. Three customized automatic database tuning tools extended from our framework kernel, employing strategies using combinations of different tuning actions (indexes, partial indexes, and materialized views) for different RDBMS;
4. An evaluation of how effective our self-tuning tools are and how much they can increase the throughput of a well-known benchmark.

Based on our results, we conclude that the framework enabled instantiating useful self-tuning tools empolying different strategies, which significantly reduced query execution costs on different RDMS, with low effort. Hence, we believe that it can serve as a valuable common and extensible testbed for further evolving self-tuning strategies.

Our framework is open-source and is available at *https://github.com/hidden-for-double-blind-review* All data, queries, and selected tuning actions, besides results obtained, are available at *hidden-for-double-blind-review* for experimental reproducibility.

The remainder of this paper is organized as follows. Section 2 presents concepts about software reuse and database tuning. In Section 3, we propose a list of reuse-oriented requirements for database self-tuning tools. We discuss possible modeling options and justify our choices. It also puts forward our software framework to support building database self-tuning tools. Section 4 depicts the evaluation process and experimental results for database self-tuning tools built using our framework. Section 5 describes the related work and discusses database self-tuning tools. Finally, Section 6 brings our concluding remarks.

2. Background

Software reuse is the use of existing software or software knowledge to develop new software. Reusable assets can be either reusable software or software knowledge. It aims

to reduce development and maintenance costs and improve software quality due to the use of already designed, implemented, validated and tested software units [Sommerville 2011]. Reusable software **components** are self-contained, clearly identifiable artifacts that describe or perform specific functions and have clear interfaces, appropriate documentation, and a defined reuse status [Sametinger 1997]. Components are ideally known as software pieces that can be reused in many software systems [Alvaro et al. 2007].

A **framework** is a reusable application that can be specialized to produce custom applications. Frameworks aim at reusing software units divided according to the requirements of the final software [Fayad et al. 1999]. Typical characteristics of software frameworks include a reusable, semi-complete application that can be specialized to produce custom applications or can act as an application generator directly related to a specific domain [Fayad and Schmidt 1997]. The points of the flexibility of a framework are called **hot-spots**, which are typically abstract classes or methods that must be implemented. If one wants to generate an application based on a framework, it is necessary to implement a specific code for each hot-spot. Some features of the framework are not mutable and constitute the framework kernel, also called **frozen-spots**.

Database tuning is the process of making a database application run more quickly. This usually means higher throughput, and it may also mean lower response time for some applications [Shasha and Bonnet 2002]. The process of database tuning is an activity where the Database Administrator (DBA) monitors the database, plans possible tuning actions, evaluates and executes them on the database system, and, again, monitors the database to observe the impact on the workload. **Self-tuning** tools automate this process by delegating steps performed by the DBA to autonomic mechanisms able to follow algorithms and heuristics to monitor and execute tuning actions.

Performance is the main optimization goal of the tuning activity. The literature considers different optimization goals, such as availability, fault tolerance, and resource consumption. In this paper, we present a framework to support developing self-tuning tools that can target any of these optimization opportunities.

Several reasons motivate database self-tuning systems: (i) the increasing complexity of multi-tenant monolithic applications and services; (ii) the growing complexity of RDBMS's administration and tuning, which provides hundreds of tuning knobs; (iii) the high cost of ownership for RDBMS-based solution, where hiring experts in system tuning and management is dominant [Chaudhuri and Weikum 2005].

There are many tuning action types defined in the literature. In this work, we focus on the manipulation of three access structures that may achieve better performance for database systems: indexes, partial indexes, and materialized views. We will explore the action of creating these structures with a closer look into multiple DBMSs, exploring our reuse-oriented framework solution.

3. Requirements and Framework Architecture Overview

There is a comprehensive amount of literature surrounding automatic tuning tools for database systems. They involve cost models, algorithms and heuristics, strategies about how to collect and use the database workload, and so on. Due to the different features of these tools, we present requirements for automatic tuning tools that are adherent to our reuse-oriented approach and benchmarking framework.

Non-intrusive approach: There are self-tuning tools that change the RDBMS source code to generate and evaluate tuning actions. They have the advantage of making more efficient predictions about tuning actions effectiveness. On the other hand, this intrusive approach depends on the availability of the RDBMS source code. Even in an open-source RDBMS the tuning tools need to be revised to each new software release. All this makes maintenance costly. non-intrusive method for the creation and evaluation of tuning actions can provide more reuse opportunities, and decrease the maintenance cost. Furthermore, decoupling the tool is necessary to use the same self-tuning tool for multiple RDBMSs.

Online workload analysis: Database tuning tools use two main approaches: In the offline method, the DBA identifies and collects parts of the workload that are representative and provides them as input to the tuning tool. Alternatively, in the online approach, a tuning tool automatically tracks the RDBMS and evaluates which tuning actions should be executed for the current workload [Chen et al. 2010]. We chose the online strategy as a requirement by the automatic feature desired in a self-tuning application. The online approach presents two potential features which can be employed from a reuse perspective: (i) automating of the workload collection, a costly task, leaving the DBA free to perform other activities and save time; and (ii) allowing the tool to adapt the physical design of the database according to workload trends and possible seasonal variations.

Handling different RDBMSs: Aiming large-scale reuse, we consider that the framework should be able to deploy self-tuning tools that can handle different RDBMS. We propose multiple cost models for each self-tuning tool. The main idea is to create and evaluate tuning actions in a specific manner with different cost models for each RDBMS.

Canonical cost model: In order to allow working with multiple RDBMSs, we decided to create and use an external and independent canonical cost model. It has the following advantages: (i) It can be customized by both hardware setup and RDBMS brand; (ii) The tools are independent of RDBMS versions; and (iii) it can be applied to any open-source RDBMS, as well as to proprietary RDBMS. Another point that support our decision to use canonical cost models is to be able to adjust costs for different hardware setups.

Inclusion of new tuning action types: Many proposals for tuning actions types can be found in the literature: creating and maintaining access structures such as indexes and materialized views, logical and physical database partitioning, query rewriting, adjusting RDBMS parameters and others [Bruno 2012] [Shasha and Bonnet 2002], [Chaudhuri and Narasayya 2007]. It is not feasible to model all these tuning actions, and new strategies may appear. Thus, the inclusion of new self-tuning action types becomes a relevant requirement. In fact, to develop a framework that allows further integration of new tuning action strategies with a low-impact on the proposed architecture represents an opportunity.

Inclusion of new tuning heuristics: For a tool to propose a tuning action (e.g., query rewriting), it usually needs to perform two general steps: (i) generate alternative tuning actions and (ii) evaluate the effectiveness and efficiency of these proposed actions. There are a significant number of heuristics developed for both steps (i) and (ii). Following the same argument for allowing including new tuning action types, we decided to also enable the inclusion of new heuristics for both steps. The self-tuning tools developed using our framework will be able to execute those heuristics independently and concurrently. As a result, the heuristics can be ranked according to their effectiveness and efficiency, helping

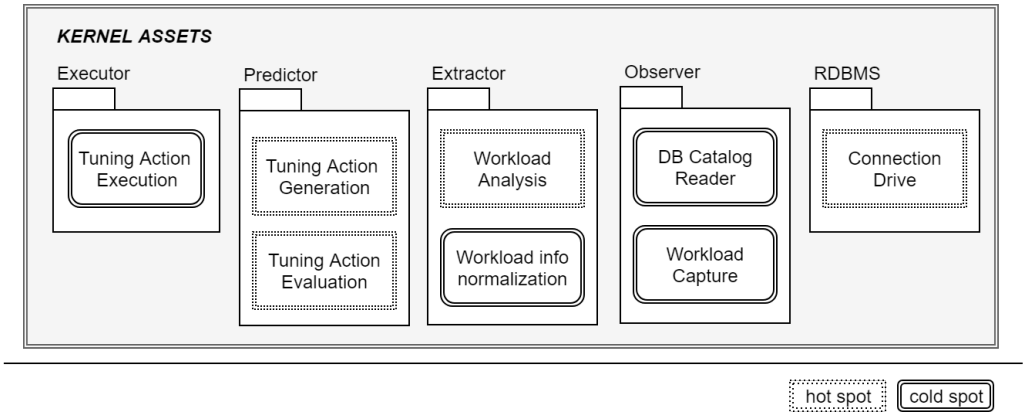


Figure 1. Database Self-Tuning Tool Framework Components.

the DBA to choose the best tuning strategy for different scenarios.

Our framework architecture is focused on large-scale reuse to produce relational database self-tuning tools. It was designed to be able to keep track of workload changes to update the database configuration automatically or semi-automatically Furthermore, we have designed mechanisms for the inclusion of (i) multiple RDBMSs, (ii) new tuning action types, and (iii) new heuristics for tuning actions, enabling their proper evaluation. The main benefits of developing a component-based framework originate from the modularity, reusability and extensibility features [Fayad et al. 1999]. Hence, our framework explores these characteristics to satisfy those relevant requirements. Figure 1 shows the main components of our domain framework.

There are five components: RDBMS, Observer, Extractor, Predictor, Executor.

- The **RDBMS** component is responsible for encapsulating all specifications related to communication between the RDBMS and the framework. It has a hot-spot called *Connection Drive* that contains the implementation to connect and run queries over the database.
- The **Observer** component is responsible for capturing the database workload and statistics from the catalog. It has two frozen-spots: the *DB Catalog Reader* and *Workload Capture*. The *Workload Capture* runs continuously and has the purpose of monitoring and capturing the executed SQL commands and their execution plans (workload). The *Catalog Reader* frozen-spot gets the database statistics catalog and provides information about the database schema.
- The **Extractor** component fetches information from the captured workload that tuning heuristics use to propose and evaluate tuning actions. The *Workload Analysis* is a hot-spot that gives independence to instances of the framework for workload data mining.
- The **Predictor** component generates tuning actions based on the captured workload and each modeled self-tuning tool. It also analyzes these actions and defines which ones will bring the most significant benefit. The *Tuning Action Generation* hot-spot generates tuning actions using algorithms defined in the literature and implemented in the component. The Predictor also has the *Tuning Action Evaluation* hot-spot that identifies the subset of tuning actions that maximize the benefits and respects the computational resource limits.
- Finally, the **Executor** Component is responsible for executing tuning actions that

were considered useful for the workload by the Predictor component. The tuning actions can be performed when the DBA considers it appropriate, e.g. when the system is idle.

Further details about our framework specification are available at [HIDDEN]. Anyhow, the main focus of this research work involves its straightforward application, the evaluation of its efficacy and to present some practical results considering actual DBMSs.

4. Evaluating Database Self-Tuning Strategies in a Common Extensible Framework

Our proposal involves using our software framework to support the development and evaluation of customized self-tuning strategies for relational databases. Therefore, we conducted the following two steps:

1. we instantiated self-tuning tools employing different tuning actions and for different RDBMS to (a) evaluate the feasibility of using the framework for such purpose and (b) use this experience as a basis to discuss the effort; and
2. We tested these tools in terms of their effectiveness on different RDBMS to see whether they would be useful (e.g., increase database throughput).

4.1. The framework instantiation

There are two main ways to extend our framework: to include new tuning action types, and to support new RDBMSs:

Tools for different database tuning action types - we extended our framework into three self-tuning tools for distinct access structures:

1. **Materialized views (MV)**: The suggested MVs are inferred from statistical analyses of the workload and the information about the database schema. Due to the characteristics of MVs, this tool is suitable for workloads that have analytical queries because, typically, MVs are considered good alternatives to SQL commands that have aggregations and summarizations [Shasha and Bonnet 2002].
2. **Indexes**: The index selection is considered one of the most important actions of the database tuning task, and it can bring significant benefit to the performance of database management systems [Bonnet and Shasha 2009]. This benefit occurs by reducing the number of pages that need to be copied from the disk to the main memory during an SQL command execution.
3. **Partial indexes**: This tuning action uses a similar access structure of the previous one for indexes. These two types, when combined, bring significant benefits to the execution of the workload. The index and partial index tools are complementary to each other which favors them when they run concurrently, and the gain in performance is greater than when the strategies are executed independently. [Stonebraker 1989]

These tools were initially developed to perform automatic tuning using an individual cost model for the PostgreSQL, supporting its SQL specificities.

Tools for different RDBMS - Once we tested the extensions for tuning actions, the resulting tools were extended (based on the framework's hot-spots) for two other

RDBMSs: Oracle and SQL Server. This extension is significant because details about the SQL execution plan are specific for each RDBMS, and they are used by the cost models to predict the quality (or not) for tuning actions. Different execution plans need different information extraction functions.

For all extensions performed, this validation helped to refine the framework's model iteratively. Besides the evaluation, it helped both validate and justify the hot-spots and move some other functions to the framework's core.

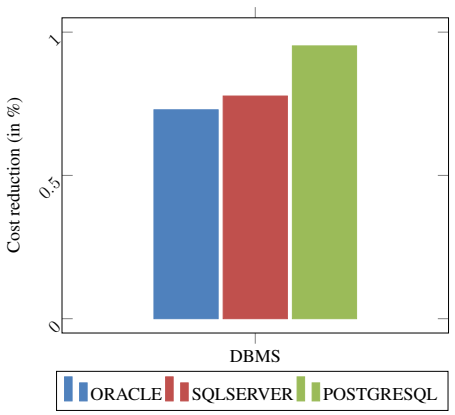
4.2. Experimental evaluation using the instantiated tools

Our framework helps to develop self-tuning tools that can carry out database tuning actions automatically according to their specialization using the chosen RDBMS. Nevertheless, are they useful?

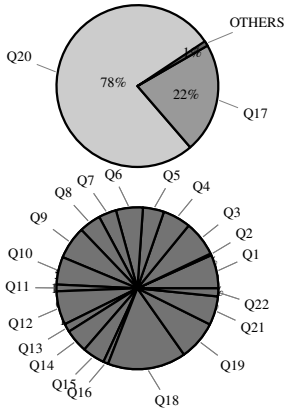
Due to space constraints, it is not feasible to present a complete evaluation for each strategy on each RDBMS or to provide in-depth discussions on the tuning strategies. Instead, we selected different tunning strategies and different RDBMS to evaluate self-tuning tools derived from our framework in different scenarios. The strategy for the PostgresSQL tunning tool used indexes, partial indexes, and MV. The strategy for the Oracle tunning tool used indexes and MV. Finally, the strategy for the SQL Server tunning tool used only indexes.

We tested our instantiated tools using the benchmark TPC-H [Transaction Processing Performance Council (TPC) 2018], which is one of the most frequently used database benchmarks for analytical workloads.

For what follows, we used the PostgreSQL 10, Oracle 12c, and SQL Server 2017. The machine configuration included an Intel I7 3.40GHz CPU, 16GB of RAM, and 2TB of HDD. We tested our tools using all the 22 TPC-H query templates on a database with a size of 10GB. For each query template, we instantiated 30 distinct variations. Therefore, our workload had 660 queries. In this context, each query has a distinct execution cost, i.e., queries created based on the same template have different restrictive attributes in the *where* clause.



(a) Workload cost reduction for different DBMSs.



(b) TPC-H workload cost distribution on PostgreSQL with 10GB.

Figure 2

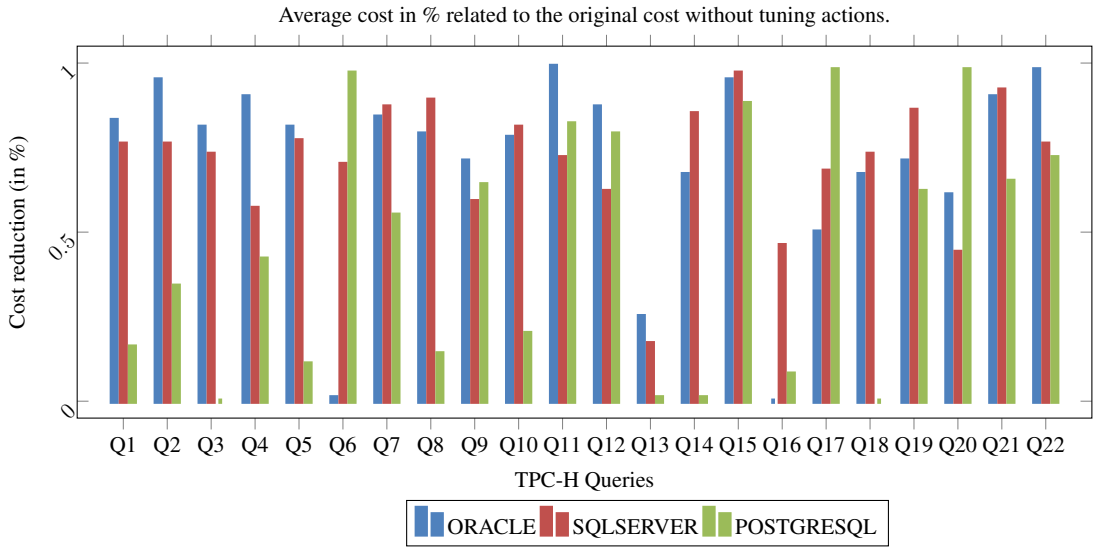


Figure 3. Cost reduction by query template for different RDBMS

Figure 2(a) shows the cost reduction to randomly execute the 660 queries. Given that different RDBMS have different query cost units, we present the reduction percentage of each RDBMS considering its own cost units, when compared to executing the workload on that RDBMS without any tuning actions.

It is possible to see (Figure 2(a)) that in all scenarios, the self-tuning tools produced significant cost reductions. The tunning strategies employed for the commercial (and more robust) RDBMS, reduced the original costs up to 3/4. The strategy used for the open-source RDBMS PostgreSQL, achieved an improvement of more than 95%. We believe that this huge improvement for PostgreSQL was due to the combined use of MVs, indexes and partial indexes. In order to further explain these improvement, it is necessary to describe a bit the TPC-H query costs.

The query execution costs for instances from the same template manage to have the same order of magnitude (e.g., all instances of Q1 have almost the same cost). Therefore, the most critical differences occur between the group of queries: first Q17 and Q20, and secondly, the other queries (Figure 2(b)). We can see that the queries based on templates Q17 and Q20 correspond to 99% of the workload execution cost, while the sum of all other queries cost represents only 1%.

Figure 3 shows that our three self-tuning tools improved most of the queries of the workload. Between queries with low cost, the results are good for most of them, besides some queries with small cost reductions.

However, huge performance improvements were obtained when improving the most expensive queries of the TPC-H benchmark: Q17 and Q20 (Figure 3). The impressive cost reduction obtained for the PostgreSQL tunning tool may be explained by the use of materialized views, indexes, and partial indexes together to answer these queries. Materialized views are data structures that precompute query answers to use them later. The answer was just ready on the materialized view structures when the RDBMSs executed a query based on templates Q17 or Q20.

It is worth mentioning that the creation of materialized views can have a high cost

once it computes all possible answers to a query template. The algorithms implemented in our tool did not create materialized views for all query templates, but only for the top-k most expensive queries. The discussion about these algorithms is out of scope here, but they are well described in [Agrawal et al. 2006] [Shasha and Bonnet 2002] [Zhou et al. 2008] [Shirkova 2011] [Halevy 2001].

Overall, the experimental tests showed that the self-tuning tools instantiated through our framework could produce effective and useful tuning results, employing different tuning strategies on different RDBMS. They could execute simultaneously and improve a well-known database benchmark workload.

The main evidence found was that our reuse strategy did not restrict the capacity of our self-tuning tools. Even shared features of the framework kernel maintained their specific characteristics isolated and encapsulated. These results support the claim that our framework represents a valuable contribution to the community, enabling to generate and evaluate useful self-tuning tools employing different strategies with less effort by employing a software reuse strategy.

5. Related Work

Database self-tuning tools are, by nature, intrinsically complex. There are many specialized and vendor-specific tools, such as for workload monitoring [Curino et al. 2011] [Oracle 2017] [SQL Server Profile 2017], and others that monitor and perform tuning actions in automatic or semi-automatic ways [Aken et al. 2017].

Among the ones that perform automatic database tuning, they can be specialized in different types of tuning actions including indexes [Ameri 2016] [Tran et al. 2015], partial indexes [Stonebraker 1989], materialized views [Halevy 2001] [Chirkova and Yang 2012] [Baralis et al. 1997] and RDBMS parameters [Duan et al. 2009]. They can also perform more than one tuning action type on the same job [Ameri 2016] [Belknap et al. 2009] [Kimura et al. 2010] [de Oliveira et al. 2019] [Almeida et al. 2019]. Such tools typically implement specific strategies and, therefore, require significant software design and implementation efforts. However, they offer no path towards reuse of these efforts nor a systematized platform to compare tuning strategies.

The work of Van et al. [Aken et al. 2017] presents a new strategy based on machine learning models to test several knob configurations and propose them to DBAs. By the nature of machine learning models, we can consider this work with some level of flexibility to add new tuning action types. However, the authors do not discuss reuse and extensibility of tuning strategies, since their primary goal is to find the best knob configuration. This flexibility is more a consequence of the machine learning approach than the proposal of their work.

Regarding self-tuning tools using multiple tuning action types, the work of Pavlo et al. [Pavlo et al. 2017] presents a considerable number of distinct tuning action types as indexes, materialized views, partitioning, knob configuration, query optimization and features about database migration and resource elasticity. Still, their work concerns a new self-driven database management system. Even though they present strategies to include further tuning actions, they can not be used for other RDBMSs.

The work by Almeida et al [de Oliveira et al. 2019] [Almeida et al. 2019] presents

a framework to support the DBA on choices concerning tuning activities. Their research work includes the proposal of an ontology for database tuning that enables a formal approach for decisions and inferences. It permits the inclusion of new tuning types through the extension of that ontology. It might be one of the closest related works found in the literature. However, the primary goal to use the ontology is to offer transparency and confidence in the available tuning alternatives concerning the possible RDBMS scenarios through a concrete justification for the made decisions. Although a well-defined way to include new tuning practices is presented, a concrete and extensible tooling to compare self-tuning strategies across multiple DBMSs is not available, in contrast to our work.

6. Concluding Remarks

Given the variety of approaches and implementations of self-tuning tools, it would be desirable to evaluate existing database self-tuning strategies, particularly recent and new heuristics, in a standard testbed.

In this paper, we presented a reuse-oriented framework approach towards assessing and comparing automatic relational database tuning strategies. We used our framework to instantiate three customized automated database tuning tools, employing strategies using combinations of different tuning actions for various RDBMS.

We then evaluated these tools using a known database benchmark. Results showed that the framework enabled instantiating useful customized self-tuning tools, which provided significant improvements concerning query execution cost reduction.

The main benefits of sharing this framework with the database research community, with respect to evaluating different tuning strategies, are twofold:

1. instead of developing a complete tool, using the framework only specific hotspots need to be extended, concerning the tunning strategies to be compared and adapting the tools to the RDBMS of interest; and According to our results,
2. provides a common testbed that does not restrict the capacity of self-tuning tools and enables fair experimental comparisons by sharing main common features that are not related to the experimental factor of interest (i.e., the tuning strategy and the adaptation to the RDBMS).

The developed framework is open-source [HIDDEN] and can be further evolved by the scientific community to create and evaluate customized database self-tuning tools.

References

Agrawal, S., Chu, E., and Narasayya, V. (2006). Automatic physical design tuning: Workload as a sequence. In *Proc. ACM SIGMOD*, pages 683–694.

Aken, D. V., Pavlo, A., Gordon, G. J., and Zhang, B. (2017). Automatic database management system tuning through large-scale machine learning. In *Procs. ACM SIGMOD*, pages 1009–1024.

Almeida, A. C., Campos, M. L. M., Baião, F. A., Lifschitz, S., de Oliveira, R. P., and Schwabe, D. (2019). An ontological perspective for database tuning heuristics. In *Conceptual Modeling - 38th Intl Conf. ER, Proceedings*, pages 240–254.

Alvaro, A., Garcia, V. C., Federal, U., and Pernambuco, R. D. (2007). *C.R.U.I.S.E: Component Reuse in Software Engineering*. Number May 2014. C.E.S.A.R. Books.

Ameri, P. (2016). On a self-tuning index recommendation approach for databases. In *IEEE 32nd Int. Conf. on Data Engineering Workshops (ICDEW)*, pages 201–205.

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

Belknap, P., Dageville, B., Dias, K., and Yagoub, K. (2009). Self-tuning for SQL performance in oracle database 11g. *Procs. IEEE ICDE*, pages 1694–1700.

Bonnet, P. and Shasha, D. E. (2009). Index Tuning. *Encyclopedia of Database Systems*, pages 1433–1435.

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

Chaudhuri, S. and Narasayya, V. (2007). Self-tuning database systems: A decade of progress. In *Proc. of VLDB*, VLDB '07, pages 3–14, Vienna, Austria. VLDB.

Chaudhuri, S. and Weikum, G. (2005). Foundations of automated database tuning. In *Procs. ACM SIGMOD*, 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 Trans. on Database Systems*, 35(3):1–51.

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

Curino, C., Jones, E. P. C., Madden, S., and Balakrishnan, H. (2011). Workload-aware database monitoring and consolidation. In *Procs. ACM SIGMOD*, pages 313–324.

de Oliveira, R. P., Baião, F. A., Almeida, A. C., Schwabe, D., and Lifschitz, S. (2019). Outer-tuning: an integration of rules, ontology and RDBMS. In *Procs SBSI*, pages 60:1–60:8.

Duan, S., Thummala, V., and Babu, S. (2009). Tuning database configuration parameters with ituned. *Proc. VLDB*, 2:1246–1257.

Fayad, M. and Schmidt, D. C. (1997). Object-oriented application frameworks. *Communications of the ACM*, 40(10):32–38.

Fayad, M., Schmidt, D. C., and Johnson, R. E. (1999). *Building application frameworks: object-oriented foundations of framework design*. Wiley.

Frakes, W. and Kang, K. (2005). Software reuse research: status and future. *IEEE T. on Software Engineering*, 31(7):529–536.

Halevy, A. Y. (2001). Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294.

Kimura, H., Huo, G., Rasin, A., Madden, S., and Zdonik, S. B. (2010). Coradd: Correlation aware database designer for materialized views and indexes. *Proc. VLDB*, 3(1-2):1103–1113.

Oracle, C. (2017). Oracle Enterprise Manager Cloud Control. http://www.oracle.com/technetwork/oem/enterprise-manager/.

Pavlo, A., Angulo, G., Arulraj, J., Lin, H., Lin, J., Ma, L., Menon, P., Mowry, T. C., Perron, M., Quah, I., Santurkar, S., Tomasic, A., Toor, S., Aken, D. V., Wang, Z., Wu,

Y., Xian, R., and Zhang, T. (2017). Self-driving database management systems. In *Proc. CIDR*, pages 8–13.

Peters, J. F. and Pedrycz, W. (2000). *Software engineering: an engineering approach*. Worldwide S. Comp. Science. New Jersey.

PGTune (2019). PGTune. http://pgfoundry.org/projects/pgtune/.

Piatetsky-Shapiro, G. (1983). The Optimal Selection of Secondary Indices is NP-complete. *SIGMOD Rec.*, 13(2):72–75.

Sametinger, J. (1997). *Software Engineering with Reusable Components*. Springer-Verlag Berlin Heidelberg.

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

Shirkova, R. (2011). Materialized Views. *Foundations and Trends in Databases*, 4(4):295–405.

Sommerville, I. (2011). *Software Engineering*. Pearson Education, New York.

SQL Server Profile, M. (2017). SQL Server Profiler. https://docs.microsoft.com/en-us/sql/tools/sql-server-profiler/.

Stonebraker, M. (1989). The case for partial indexes. *SIGMOD Rec.*, 18(4):4–11.

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

Transaction Processing Performance Council (TPC) (2018). TPC-H. http://www.tpc.org/tpch/.

Zhou, L., Xu, M., Shi, Q., and Hao, Z. (2008). Research on Materialized Views Technology in Data Warehouse. *2008 IEEE Int. Symp. on Knowledge Acquisition and Modeling Workshop*, pages 1030–1035.