A Framework for Self-Managing Database Systems

Jan Kossmann
Hasso Plattner Institute
Potsdam, Germany
jan.kossmann@hpi.de

Rainer Schlosser
Hasso Plattner Institute
Potsdam, Germany
rainer.schlosser@hpi.de

***Abstract*—Database systems that autonomously manage their configuration and physical database design face numerous challenges: They need to anticipate future workloads, find satisfactory and robust configurations efficiently, and learn from recent actions. We describe a component-based framework for self-managed database systems to facilitate development and database integration with low overhead by relying on a clear separation of concerns. Our framework results in exchangeable and reusable components, which simplify experiments and promote further research. Furthermore, we propose an LP-based algorithm to find an efficient order to tune multiple dependent features in a recursive way.**

I. Self-Managing Database Systems

The idea of database systems that autonomously adjust their configuration is almost as old as the idea of relational database systems itself. For example, the first work on the topic of self-adaptive index selection dates back to 1976 by Hammer and Chang [1]. However, the topic of self-managing database systems gained recent popularity [2]–[4]. Studies show that the spending on database personnel is a key factor in the TCO of database systems [5]. A higher complexity of this task, e.g., caused by volatile workloads, a lack of domain knowledge and application context [6] in cloud scenarios and more available dependent configuration options further increases this number.

These aspects demand systems that support autonomous adjustments of their configuration. Such systems face a multitude of challenges, e.g., finding efficient solutions for configuration problems in a scalable fashion [7], predicting future workloads [3], and learning from past self-management decisions.

The topic is also interesting from an architecture and integration point of view. Database systems are usually not designed with self-managing capabilities in mind. Interviews that we conducted with industry database architects showed that low overhead, a maximum of 1% of additional runtime introduced by such capabilities, and minimally invasive changes to the architecture are mandatory. However, many papers focus on tuning while holistic approaches for multiple features or database integration remain unexplored.

Contribution: Therefore, we present a component-based framework for self-managing database systems. Our framework divides the significant challenge of incorporating self-management capabilities into manageable subproblems (separation of concerns). Components with clearly specified functions and interfaces handle these subproblems. Our framework simplifies experimentation and development of self-management techniques by offering reusable and exchangeable components. Further, we discuss integration and design decisions on the basis of the ongoing integration into our research DBMS Hyrise [8] in Section II. Besides, we explain our strategy of how to optimize multiple dependent features, for example, the selection of indexes, compression schemes and the decision on data placement simultaneously (Section III). Ideas on workload anticipation to allow robust optimizations are given in Section II-C and Section II-D. Related work in this area is examined in Section IV. Section VI concludes our work after a presentation of future work in Section V.

II. Framework Architecture

In this section, we present the architecture of our framework for self-managing database systems and the reasoning that influenced its design. For this work, we substantially extend our previous ideas of such an architecture [9]. The framework recursively divides common challenges in the context of self-managing database systems into smaller subproblems that are handled by exchangeable components. Thereby, we achieve a clear separation of concerns which simplifies the development of such systems. Also, the framework offers interfaces to access data that is provided by common database entities, e.g., cost models and the query plan cache. We detail the involved components and their interfaces in the following subsections after giving a short general overview of the architecture.

A. Overview

Figure 1 depicts a diagram of the proposed framework and an integration into the database system Hyrise. We decided to divide the system into components, each handling smaller sub-problems for a couple of reasons. First, we recognized that when tuning different features often similar or related subproblems are solved. By making components reusable and shareable, we avoid redundancy, cf. Section II-D for more details. Second, by relying on components with clearly specified interfaces, we simplify the development and experiments of new approaches since components can be exchanged effortlessly.

The *driver* is the central entity encapsulating all the other components that are responsible for adding self-management capabilities. It consists of three key components which are further divided into the following sub-components:

- *Workload Predictor*: The effect of a particular database configuration largely depends on the executed workload.

 DOI: 10.1109/ICDEW.2019.00-27

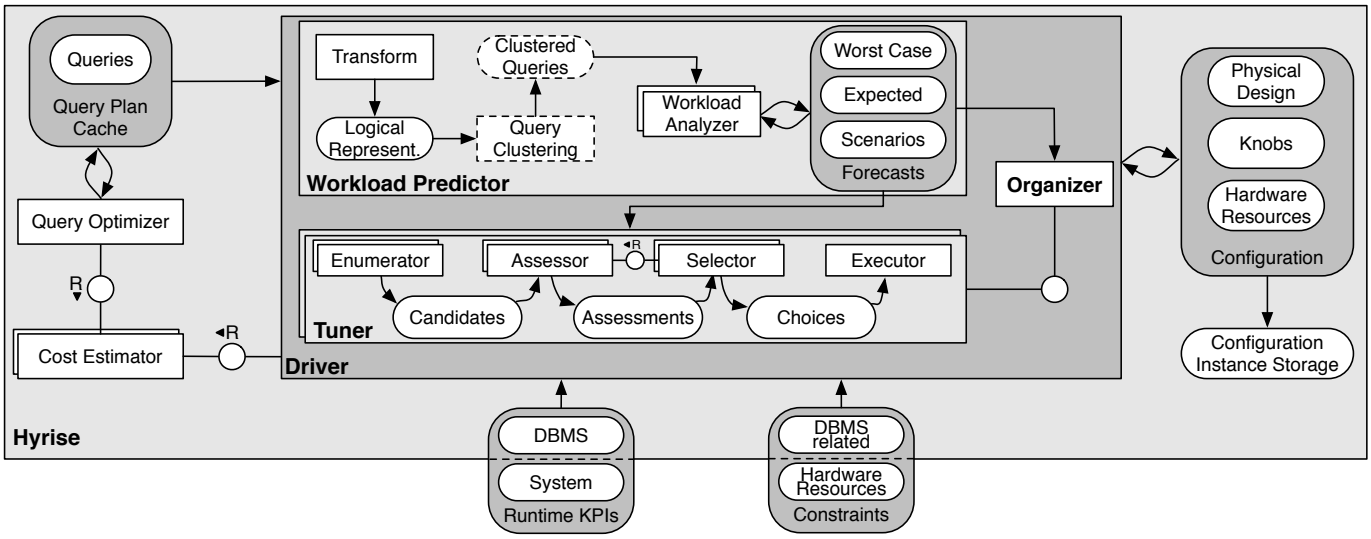

Fig. 1. Architecture diagram of our framework for self-managing database systems.

Therefore, a component is required that predicts the upcoming workload based on historical workload data. The workload predictor is detailed in Section II-C.

- *Tuner*: Based on these predictions, the tuner employs a multi-step process to calculate a selection for a certain feature. Section II-D contains further information regarding the tuner.
- *Organizer*: The *organizer* is orchestrating the whole self-managing processes. It is responsible for starting and stopping tunings, enforcing *constraints* and assessing *runtime KPIs*. Further, it determines in which order features should be tuned (cf. Section III). Section II-E gives more details on the *organizer*.

Furthermore, the *driver* is the interface for accessing other database or system components that serve as external inputs. We describe these inputs in the following paragraphs.

a) Query Plan Cache: Most relational database systems, e.g., Microsoft SQL Server [10] and SAP HANA [11] employ query plan caches. These have typically two purposes: the support of prepared statements and caching of optimized query plans to avoid optimizing the same SQL queries repeatedly. For our work, the latter aspect is of importance because workload-driven optimizations need information about workloads that were executed in the past. In addition to query plans, information such as the execution time and the number of executions of the queries is stored and used by the *workload predictor* to generate forecasts of future workloads.

b) Configurations: The *configuration* of a DBMS is the combination of all of its configurable entities. These can be categorized into features regarding the physical database design, the knob configuration of the database, or hardware resources that are available to the system. The selection of indexes, a partitioning scheme, or data placement are examples for physical design features while the buffer pool size or the number of available threads are typical examples for knobs. A particular configuration is called *configuration instance*. When the configuration is adjusted, former configuration instances are stored. This storing is central to establish a feedback loop for past decisions by enabling the assessment of the impact of past tuning decisions.

c) Constraints: Constraints are DBMS-related or result from the available hardware resources. Examples for the first case are user-defined service level agreements (SLAs) or limitations of the memory utilized for indexes. Other entities, as management software in cloud scenarios or applications itself, could also set these constraints.

Hardware resource constraints limit the available options during the tuning process from a physical perspective. A configuration instance that requires more memory than actually available on the system should not be considered. Both types of constraints could conflict. In such cases, available hardware resources overwrite externally specified ones.

d) Cost Estimators: Cost estimation is a crucial part of self-managing database systems. To determine efficient configurations, different options must be compared. Therefore, cost estimation must be involved at every stage of the tuning process. It is required to quantify the impact of all decisions the system may take. To also make these decisions and actions of the system comparable across different features, cost must be estimated in the same unit, for instance, runtime. The costs of changes to the configuration, e.g., the runtime of applying a specific compression to an attribute must be as quantifiable as the processing of workloads. For example, the computation cost of a query given a particular index (cf. what-if optimization [12]) must be determinable to enable the proposed workload-driven approach.

Simple logical cost models are not capable of representing the interplay of, e.g., data types, encodings, and coprocessors in their cost estimations. We argue that hardware-dependent cost models are necessary to ensure a maximum of precision

of cost predictions which, in turn, enable well-suited database configurations. The proposed cost models can be created adaptively by learning from observed query execution costs. At database system start, a minimal set of queries is run to create training data for a specialized cost model. During further database operation more data points are collected, thus enabling more specialized models that ensure precise cost predictions. Cost models can be obtained by applying simple linear regressions [13] or more advanced approaches like gradient boosting regressors or neural networks [14].

e) Runtime KPIs: We classify runtime KPIs as DBMS or system specific. Examples for typical DBMS KPIs are query response times or the number of aborted transactions. On the other hand, system KPIs are mostly comprised of hardware metrics: CPU utilization, memory usage, or cache misses. The use cases of runtime KPIs are manifold. First, they are necessary for determining the impact of adjusted configurations, e.g., how did a certain index decision influence the average query response time? Second, runtime KPIs can disclose when the configuration should be adjusted. For example, when SLAs are constantly violated or performance peaks are detected. Furthermore, these KPIs can help to identify phases of low resource utilization that can be used to run resource-intensive tunings. Therefore, these are used by the *organizer* to identify favorable points in time for tuning runs.

B. Implementation Strategies

The aforementioned architecture is currently under implementation in our research database system Hyrise[1], which is categorized as a relational in-memory database system and tables are stored in column-major format. Every table is implicitly partitioned into chunks of a certain size. Decisions about, e.g., compression, indexes, or data distribution in NUMA systems are all taken on a per-chunk instead of a per-table basis. This chunking increases the flexibility in the context of self-managing database systems since decisions are possible for fractions of the data of an attribute. For example, the system can decide to create indexes only on the frequently accessed and most beneficial chunks to save memory. This approach is especially useful for skewed data which is often found in real-world systems [15]. Further, applying new configurations to a whole table is a heavyweight operation. Applying these iteratively to chunks reduces the cost of these operations.

There are mainly two different options of how to implement self-management capabilities: inside the database core or outside of the database system as a standalone application. For the first option, the database core functionality could be extended by integrating self-management with the database source code. This extension introduces tight coupling between the self-management system and the database core which would, in turn, complicate the development process because every developer has to be aware of and understand the self-managing system. On the other hand, for a standalone application running outside the database system, the DBMS itself has to provide powerful interfaces to adjust the configuration from outside, access KPIs and other entities which are usually not publicly accessible, e.g., the cost estimators. Providing these interfaces would induce additional development efforts while at the same time introducing overhead by further layers of indirection. The proposed framework works with both implementation strategies as long as the interfaces to the necessary data are provided.

We decided to implement self-management capabilities with the plugin infrastructure[2] of Hyrise. Thereby, we can combine the strengths of the aforementioned approaches. The plugin interfaces offer direct access to database core methods without implementation or performance overhead. In addition, it avoids tight coupling of the development of database core and self-management functionality. Plugins are dynamic libraries which are loaded during database runtime. The development of plugins is identical to the development of the database core, but plugin code is not compiled with the database system itself. Thus, the database system remains independent.

C. Workload Predictor

The workload predictor is responsible for creating forecasts about future workloads. Such predictions are indispensable for self-managing database systems. The configurations itself (determined by the *tuner*) as well as the points in time when the process of deciding on configurations should be triggered (by the *Organizer*) are based on these predictions. Robust predictions support the system in being less sensitive to irregular workload patterns.

As a first step, the workload predictor accesses information about past workloads from the query plan cache (cf. Section II-A). The information contains which queries were executed and their execution count and cost. By relying on the query plan cache, no further overhead is added during query execution time and the database system's architecture remains unchanged. The prediction itself is a multistep process. First, depending on how the query plan cache stores information about past queries, these are transformed into an abstract logical representation of query templates to remove unnecessary information. The second step is an optional query clustering (e.g., similar to [3]) for large and diverse workloads. Here, similar queries can be combined to reduce the number of queries that have to be processed in the following and, in the end, reduce the time necessary for predictions and tunings. Lastly, a *workload analyzer* calculates a forecast of future workloads. The system can consist of multiple workload analyzer instances that each employ different methods to create forecasts, e.g., based on expert knowledge, latest scenarios (seasonal time intervals) as well as simple linear regressions, time series analysis (cf. ARIMA), or more expensive recurrent neural networks.

In this context, we recall, that robustness is a crucial component for self-managing database systems. We refer to

[1]Source Code available at: https://github.com/hyrise/hyrise

[2]Example plugin available at: https://git.io/HyriseExamplePlugin

robustness as how workload changes affect a system's performance. Robust configurations do not aim to provide the best performance for the expected scenario but aim for acceptable performance for most scenarios so that small workload changes do not have a large impact on the performance. Hence, not only the expected workload should be incorporated but also information about the distribution of potential future scenarios to allow the computation of robust configurations.

D. Tuner

Tuners are components that take workload forecasts and cost estimations as input and deliver configurations for features as output. There is one tuner instance per feature, e.g., a tuner for index selection and another tuner for determining efficient partitioning schemes. Tuning relies heavily on accurate cost estimations (cf. Section II-A) to determine configurations. Without sufficiently precise estimations different configuration options cannot be compared. We specified tuning as a multi-step process where each step is mapped to a subcomponent to enable reuse across the tuning of multiple features (cf. Section III). Figure 1 indicates that multiple instances for the tuning subcomponents exist. Thereby, different techniques can be employed and compared based on the outcome of former tuning runs or (time) constraints. In the following, we detail the involved subcomponents.

a) Enumerator: An enumerator is responsible for providing a list of *Candidates* to the tuning process. The size of the candidate set is typically a significant contributor to the execution time of optimization algorithms. Hence, providing a variety of enumeration algorithms is advisable to be able to influence the runtime. Some enumeration algorithms restrict the candidate set based on heuristics (cf. [12]) while others consider all available candidates. The framework allows to switch between different enumerators or fall back to restrictive enumerators when necessary. *Candidates* can be of various forms to represent different types, i.e., physical design features or knobs. For discrete problems, for example for index selection, candidates would be a set of lists (to support multi-attribute indexes) of attributes. For continuous problems, e.g., the decision about the buffer pool size candidates are specified by providing the start and the end of a range, e.g., 1.0 GB to 100.0 GB and the smallest available intervals to pick in this range, e.g., 0.5 GB. Users can either implement enumerators on their own or utilize general ones provided by the system.

b) Assessor: This component provides an assessment of the previously generated candidates. Each candidate is assigned a positive or negative *desirability* indicating its impact on the overall system performance given a forecast scenario. The system assigns different desirabilities to the same candidate for different forecast scenarios. Later in the decision process these, possibly differing, desirabilities are utilized for robustness considerations. Besides, the assessor assigns an associated *confidence*, describing the certainty of the assessment, and a *cost* to each assessment. The cost component is twofold: it consists of permanent costs (e.g., the memory consumption of an index) and one-time costs for applying the configuration (e.g., the cost of constructing an index). The sum of all these one-time costs are so-called *reconfiguration costs*. These are of importance in the following scenario: The tuner might find a new improved configuration that suggests to completely change the current one even though the associated performance increase is comparably small. To avoid such effects reconfiguration costs can be used to balance performance improvements and reconfigurations to identify minimally invasive changes. Thus, accurate cost models are indispensable for precise and fast assessments.

Again, the system can contain different assessors that reflect the use of different cost models, e.g., simple logical, physical or what-if optimizer-based models. Choosing an assessor is a trade-off between accuracy and runtime. Learnings from past decisions, i.e., the effect of specific configurations on runtime KPIs can be incorporated during this step.

c) Selector: A selector chooses candidates based on the previous assessments and specified constraints, e.g., a memory budget for indexes. As in the previous steps, there are multiple selectors available, each following a different strategy. For selection, a third component is added to the trade-off of finding optimized solutions or achieving low computation times for the optimization: robustness of the chosen candidates. We consider the following classes of selectors (including existing approaches for the tuning of specific features) to be interesting for self-managing database systems:

- Greedy: The greedy selector chooses candidates based on the desirability per cost. Choosing the candidates with the highest ratio first and proceeding until the constraint is violated. The strength of the greedy selector is its short runtime. For example, [16], [17] for index selection or [18] for data tiering with greedy approaches.
- Optimal: As the name implies, optimal selectors find optimal configurations (e.g., Dash et al. [19] for an optimal index selection). This selector is usually based on off-the-shelf solvers that are heavily optimized for such a task. Optimal selectors might lead to long runtimes.
- Genetic: These algorithms are based on the biological principles of mutation, selection, and crossover [20]. Genetic algorithms (e.g., for index selection Kratica et al. [21]) can be applied when the search space is too large to find optimal solutions. They usually find close-to-optimal solutions in relatively short amounts of time.
- Robust and risk-averse: Selectors that act risk-averse are a good choice for scenarios in which stable performance in most cases is preferred over best performance in the expected case. (cf. [22]). Criteria based on mean-variance optimization, utility functions, value at risk, and worst-case considerations can be used.

By strictly relying on the interfaces between components, selectors can be exchanged and shared between features. Selectors can also request re-assessments of certain candidates from the assessors. This is useful to reflect changed circumstances or incorporate interaction between candidates.

d) Executor: The executor takes care of applying the choices that were selected previously. There are different application strategies regarding order, point in time and sequential or parallel application. The executor can access runtime KPIs to determine favorable points in time for applying the choices.

E. Organizer

The organizer is responsible for orchestrating the whole self-managing process. It identifies convenient points in time for tuning by constantly monitoring runtime KPIs and taking workload forecasts into account. The organizer also decides whether changes observed in workload forecasts are significant enough to justify possibly expensive tunings. This decision relies, upon other terms, on the difference of the current workload cost and the estimated workload cost for the forecasted workload given the current configuration.

Furthermore, self-managing database systems manage the configuration of multiple features, e.g., the selection of indexes and compression schemes for attributes. The organizer decides on the *order* of tuning processes for these features. More details are given in Section III. In the future, the organizer could also, based on the workload forecast, decide to only tune the subset of features which is expected to yield the largest benefits to avoid wasting resources on unprofitable tunings.

III. Combined Tuning of Multiple Features

In this section, we discuss ideas and highlight challenges regarding the tuning of multiple features. There are generally two approaches of tuning multiple features: An iterative approach tuning one feature at a time or an integrated approach that relies on an omnipotent model that is capable of determining efficient configurations for all features in a combined fashion. The first approach does not consider dependencies at all. The solution space of single features for real-world problems is already substantial. Consequently, the second method results in prohibitively large complexity. Therefore, building a single model for many features is unfeasible.

A. Recursive Tuning of Subproblems

A third approach is to recursively tune single features. The order is of importance since features are not independent of each other. For example, depending on the chosen compression scheme the impact of indexing a particular attribute might be affected. Furthermore, resource constraints might prohibit the tuning of all features. In some cases, only the features with the most significant impact can be tuned. Therefore, the overall cost and benefit of the tuning of a specific feature need to be considered to determine a tuning order.

Zilio et al. [23] describe a hybrid approach that orders tuning processes by their pairwise dependence: Non-dependent features are tuned one after another in any order, unidirectional dependent ones are tuned in the most efficient order, and mutually dependent ones are tuned in a combined fashion. More details and a differentiation are given in Section IV.

Accurately determining dependencies can be of high complexity because it relies on expensive calculations: conducting the actual tunings for all features as well as many calls to cost estimators. However, the complexity can be reduced by reducing the workload size with sampling or working on clustered queries as provided by the *workload predictor*.

We propose a mechanism to recursively tune all features in a reasonable order while taking their dependencies into account. These basic dependencies are automatically determined. We see multiple interesting dimensions to consider when determining the dependence of different features. The authors of the aforementioned approach consider the feature B strongly dependent on the feature A when a change in the selection of A often causes a change in the selection of B. We believe that the impact on performance regarding, e.g., workload cost is an important additional factor in determining dependence.

In the following, we are going to lay out the concept of our solution before we discuss the applicability and complexity. The *organizer* retrieves the expected workload forecast from the *workload predictor*. Based on this forecast the cost ($W_\varnothing$) of executing the expected workload without any optimization is determined using the *cost estimators*. This cost serves as a reference for future considerations. Afterward, a separate tuning run is conducted for each single feature A and the cost for the execution of the expected workload W_A is determined. The ratios $W_\varnothing / W_A$ provide a simple way of assessing the *impact* of the tuning of each feature (while not considering any dependencies). Note, considering the costs of the respective tunings allows a heuristic-based ranking of *impact per cost* which can be utilized when resources do not suffice for tuning all features.

In addition, we can determine whether the order in which two features A and B are optimized is of importance. We first optimize feature A followed by feature B and determine the workload cost: $W_{A,B}$. We repeat the same for $W_{B,A}$. A dependence ratio $d_{A,B} := \frac{W_{B,A}}{W_{A,B}}$ close to 1 indicates that the order of optimizing A and B is less important. A value of $d_{A,B} > 1$ indicates that A should be optimized *before* B and the other way around if $d_{A,B} < 1$.

Further, we can calculate d pairwise for all combinations of features. The ratios can be used to determine an optimized order to tune all features recursively.

Since we define the dependency of two features A, B by workload cost, efficient and precise ways to estimate these costs are required. Therefore, cost estimators, as described in Section II-A, are of high importance. In addition, the estimation of workload costs for many combinations and large workloads can become expensive. Decreasing the workload size, for example, by clustering (cf. Workload Compression [24]) can mitigate this problem in exchange for possibly less accuracy.

B. LP-Based Order Optimization

Deriving an optimal order of all features is a highly challenging task as (i) the number of permutations can be large and (ii) a consistent order satisfying all preferred pairwise relations cannot be assumed to exist.

Based on the values $d_{A,B}$ the preferable order, as well as its importance, can be quantified for all pairs of features A and B.

To determine an optimized tuning order of features we propose the following integer linear programming (LP) approach. By the family of binary variables $x_{A,k}$ we denote whether feature $A \in S$ is tuned in *step* k ($x_{A,k} = 1$) or not ($x_{A,k} = 0$), $k = 1, ..., |S|$, where S is the set of features. The second family of binary variables $y_{A,B}$ expresses whether feature $A \in S$ is tuned *before* feature $B \in S \backslash \{A\}$ (i.e., $y_{A,B} = 1$) or not ($y_{A,B} = 0$). To determine an optimized tuning order, we use the integer LP formulation as follows.

$$\underset{\substack{x_{A,k}, y_{A,B} \in \{0,1\} \\ A \in S, B \in S \backslash \{A\}, k = 1, ..., |S|}}{maximize} \quad \sum_{A,B \in S} y_{A,B} \cdot d_{A,B} \cdot W_{\varnothing} / W_{A,B}$$

subject to

$$\sum_{k=1,...,|S|} x_{A,k} = 1 \quad , A \in S$$

$$\sum_{A \in S} x_{A,k} = 1 \quad , k = 1, ..., |S|$$

and the coupling constraints, $A \in S, B \in S \backslash \{A\}$,

$$y_{A,B} + y_{B,A} = 1$$

$$|S| \cdot y_{A,B} \geq \sum_{k=1,...,|S|} k \cdot x_{B,k} - \sum_{k=1,...,|S|} k \cdot x_{A,k}$$

The objective optimizes the sum of the dependence ratios d weighted by the associated impact coefficients $W_{\varnothing}/W_{A,B}$, $A \in S, B \in S \backslash \{A\}$. The first two families of constraints guarantee an admissible permutation order of all features: (i) each feature is assigned to exactly one tuning step and (ii) each tuning step is associated to exactly one feature.

The last two families of constraints uniquely couple the variables x and y in a *linear* way. The first one guarantees that exactly one of the variables $y_{A,B}$ and $y_{B,A}$ is equal to one (for all pairs of features $A \in S, B \in S \backslash \{A\}$). The last constraint works as follows: If a feature B is supposed to be tuned *after* a feature A then the right-hand side of the inequality is *positive* and, hence, $y_{A,B}$ has to be equal to one.

The number of variables and constraints is $2 \cdot |S|^2 - |S|$ and $2 \cdot |S|^2$, respectively. The integer LP can be solved using off-the-shelf solvers. Our LP approach is (i) viable, (ii) allows the consideration of many features, and (iii) effectively accounts for mutual dependencies when tuning multiple features.

IV. Related Work

The field of database systems that autonomously adjust their configuration regained popularity. In contrast to earlier solutions for commercial database systems (e.g., [12], [16], [23], [25]), our work takes a holistic approach to the problem proposing a framework to facilitate development and database integration.

Pavlo et al. [2] describe their vision of a self-managing database that autonomously adjusts the configuration of multiple features. Further, they discuss the integration and architecture of such a system in their database Peloton. In their work, the authors sketch the system's architecture on a higher level and do not discuss the joint optimization of multiple arbitrary features. Our work sets its focus on (i) dividing the challenge of creating a self-managing DBMS into small subproblems and (ii) giving a more detailed specification of the components that handle these subproblems. Thereby, contrary to aforementioned work, also challenges from a software engineering and development point of view are considered.

As a third contribution, we present an approach to efficiently combine the tuning of multiple features which goes beyond the idea of Zilio et al. [23]. The authors assume that knowledge about dependencies of features is given. In their work, four problems are considered and the dependencies are defined manually. We argue that dependencies are challenging to be manually determined with volatile workloads, varying hardware and an increasing number of features to tune. Further, their approach is limited as, in general, the number of dependent features which have to be jointly optimized can be too large. We also consider more dimensions to determine dependencies automatically and we do not limit the number of physical design features to tune.

V. Future Work

The presented concepts and ideas open many opportunities for further research. Self-managing systems must be able to precisely assess costs and benefits of preferably each and every operation and action. Therefore, we currently work on an approach to adaptive cost estimation where costs for the processing of every operation are logged during database operation. Subsequently, this data is used to generate updated accurate cost models from time to time.

Robustness is especially important for the adoption of self-managing DBMSs in practice. If the performance of such systems degrades as soon as the actual workload deviates from the expected workload, customers will not adopt these systems. Thus, we incorporated support for different forecast scenarios in the *workload predictor* and see their application and evaluation as an important area for further research.

Lastly, a thorough evaluation of the presented ideas on determining a favorable order to tune multiple features in Section III is necessary to assess the viability and performance implications for large problem instances.

VI. Conclusion

Our framework divides the challenge of integrating self-management capabilities in database systems into smaller problems that are tackled by components. We detailed the specific components and interfaces between them. The separation of concerns enables reuse and exchange of components as well as simplification of development and experimentation. The presented workload predictor is capable of predicting multiple workload scenarios that are incorporated by the respective tuners to allow robust solutions. Furthermore, in Section III, we discussed an LP-based approach to find an efficient order for the recursive tuning of mutually dependent features.

References

[1] M. Hammer and A. Chan, "Index selection in a self-adaptive data base management system," in *Proceedings of the International Conference on Management of Data (SIGMOD)*, 1976, pp. 1–8.

[2] A. Pavlo *et al.*, "Self-driving database management systems," in *Online Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2017.

[3] L. Ma, D. V. Aken, A. Hefny, G. Mezerhane, A. Pavlo, and G. J. Gordon, "Query-based workload forecasting for self-driving database management systems," in *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2018, pp. 631–645.

[4] T. Kraska *et al.*, "Sagedb: A learned database system," in *Online Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2019.

[5] S. Chaudhuri and G. Weikum, "Self-management technology in databases," in *Encyclopedia of Database Systems*, 2009, pp. 2550–2555.

[6] S. Das *et al.*, "Automatically Indexing Millions of Databases in Microsoft Azure SQL Database," in *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2019.

[7] D. V. Aken, A. Pavlo, G. J. Gordon, and B. Zhang, "Automatic database management system tuning through large-scale machine learning," in *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2017, pp. 1009–1024.

[8] M. Dreseler, J. Kossmann, M. Boissier, S. Klauck, M. Uflacker, and H. Plattner, "Hyrise Re-engineered: An Extensible Database System for Research in Relational In-Memory Data Management," in *Proceedings of the International Conference on Extending Database Technology (EDBT)*, 2019.

[9] J. Kossmann, "Self-Driving: From General Purpose to Specialized DBMSs," in *Proceedings of the VLDB PhD Workshop co-located with the 44th International Conference on Very Large Databases (VLDB)*, 2018.

[10] Microsoft. (2018). Query Processing Architecture Guide - Execution Plan Caching and Reuse, [Online]. Available: https://docs.microsoft.com/en-US/sql/relational-databases/query-processing-architecture-guide?view=sql-server-2017#execution-plan-caching-and-reuse (visited on 02/20/2019).

[11] SAP. (2018). Analyzing SQL Execution with the SQL Plan Cache, [Online]. Available: https://help.sap.com/viewer/bed8c14f9f024763b0777aa72b5436f6/2.0.03/en-US/bed20ba0bb57101483ffa333cf3e55c8.html (visited on 02/20/2019).

[12] S. Chaudhuri and V. R. Narasayya, "An efficient cost-driven index selection tool for microsoft SQL server," in *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 1997, pp. 146–155.

[13] Q. Zhu and P.-A. Larson, "Building regression cost models for multidatabase systems," in *Proceedings of the International Conference on Parallel and Distributed Information Systems*, ser. DIS '96, IEEE Computer Society, 1996, pp. 220–231.

[14] A. Kipf, T. Kipf, B. Radke, V. Leis, P. A. Boncz, and A. Kemper, "Learned cardinalities: Estimating correlated joins with deep learning," in *Online Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2019.

[15] A. Vogelsgesang *et al.*, "Get real: How benchmarks fail to represent the real world," in *Proceedings of the International Workshop on Testing Database Systems, DBTest@SIGMOD 2018, Houston, TX, USA, June 15, 2018*, 2018, 1:1–1:6.

[16] G. Valentin, M. Zuliani, D. C. Zilio, G. M. Lohman, and A. Skelley, "DB2 advisor: An optimizer smart enough to recommend its own indexes," in *Proceedings of the International Conference on Data Engineering (ICDE)*, 2000, pp. 101–110.

[17] R. Schlosser, J. Kossmann, and M. Boissier, "Efficient scalable multi-attribute index selection using recursive strategies," in *Proceedings of the International Conference on Data Engineering (ICDE)*, 2019.

[18] M. Boissier, R. Schlosser, and M. Uflacker, "Hybrid data layouts for tiered HTAP databases with pareto-optimal data placements," in *Proceedings of the International Conference on Data Engineering (ICDE)*, 2018, pp. 209–220.

[19] D. Dash, N. Polyzotis, and A. Ailamaki, "CoPhy: A Scalable, Portable, and Interactive Index Advisor for Large Workloads," *PVLDB*, vol. 4, no. 6, pp. 362–372, 2011.

[20] M. Mitchell, *An introduction to genetic algorithms*. MIT Press, 1998, ISBN: 978-0-262-63185-3.

[21] J. Kratica, I. Ljubic, and D. Tosic, "A genetic algorithm for the index selection problem," in *Proceedings of the Applications of Evolutionary Computing, EvoWorkshop*, 2003, pp. 280–290.

[22] B. Mozafari, E. Z. Y. Goh, and D. Y. Yoon, "Cliffguard: A principled framework for finding robust database designs," in *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2015, pp. 1167–1182.

[23] D. C. Zilio *et al.*, "DB2 design advisor: Integrated automatic physical database design," in *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2004, pp. 1087–1097.

[24] S. Chaudhuri, A. K. Gupta, and V. R. Narasayya, "Compressing SQL workloads," in *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2002, pp. 488–499.

[25] S. Chaudhuri and V. R. Narasayya, "Self-tuning database systems: A decade of progress," in *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2007, pp. 3–14.