Self-driving database systems: a conceptual approach

Jan Kossmann[1] · Rainer Schlosser[1]

Abstract

Challenges for self-driving database systems, which tune their physical design and configuration autonomously, are manifold: Such systems have to anticipate future workloads, find robust configurations efficiently, and incorporate knowledge gained by previous actions into later decisions. We present a component-based framework for self-driving database systems that enables database integration and development of self-managing functionality with low overhead by relying on separation of concerns. By keeping the components of the framework reusable and exchangeable, experiments are simplified, which promotes further research in that area. Moreover, to optimize multiple mutually dependent features, e.g., index selection and compression configurations, we propose a linear programming (LP) based algorithm to derive an efficient tuning order automatically. Afterwards, we demonstrate the applicability and scalability of our approach with reproducible examples.

Keywords Database systems · Self-driving · Recursive tuning · Workload prediction · Robustness

1 Self-driving database systems

The topic of database systems that change their configuration autonomously came to recent popularity in academia [19, 23, 31] and industry [10, 30]. According to Chaudhuri and Weikum [5], the costs for database personnel are a major factor in the TCO of database systems. These costs can be further increased by a higher complexity of configuration and tuning tasks which is caused, e.g., by non-stable workloads that change over time, a lack of domain knowledge and application context [10] in cloud environments, and more available dependent configuration options [39].

✉ Jan Kossmann
jan.kossmann@hpi.de

Rainer Schlosser
rainer.schlosser@hpi.de

[1] Hasso Plattner Institute, University of Potsdam, Potsdam, Germany

Therefore, systems that are capable of autonomously adjusting their configuration could save costs and lead to more efficient configurations. However, such systems face a multitude of challenges, e.g., finding efficient solutions for configuration problems in a scalable fashion [39], predicting future workloads [23], and learning from past self-management decisions.

The topic also offers interesting questions from a database system integration and architecture perspective since systems were usually not designed with such capabilities in mind. We conducted interviews with industry database architects. These 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, most of the work in this area mainly focuses on single aspects while holistic approaches remain unexplored.

Contribution This paper is an extended version of [18]. The main contributions of [18] are the following. We present the concept of a component-based framework for self-driving database systems, which divides the significant challenge of incorporating self-management capabilities into manageable subproblems (separation of concerns). These subproblems are handled by clearly specified functions and interfaces. Thus, our framework simplifies experimentation and development of self-management techniques by offering reusable and exchangeable components. Further, we propose a linear programming (LP) model to tune multiple features in an optimized recursive order.

Compared to [18], in this paper, we present additional explanations and make the following contributions: First, we discuss different concepts of workload modelling (Sect. 2). Second, we explain robust tuning concepts in more detail (Sect. 3.9). Third, we revised and extended the description of our LP-based tuning approach. In addition, we also added an evaluation of the approach mentioned above with reproducible examples (Sect. 4).

This paper is organized as follows. In Sect. 2, we review related work in this area. Section 3 discusses the integration and design decisions on the basis of the ongoing integration into our research DBMS Hyrise [13]. Ideas on workload anticipation to allow robust optimizations are given in Sects. 3.8, 3.9, and 3.10. Besides, we explain our strategy of how to optimize multiple dependent features, for example, the selection of indexes, compression schemes, and clustering. (Sect. 4). Ideas for future work and final conclusions are given in Sects. 5 and 6.

2 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., [4, 8, 38, 42]), our work takes a holistic approach to the problem proposing a framework to facilitate development and database integration.

Pavlo et al. [31] describe their vision of a self-driving 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. In this work, we focus on the challenge of creating a self-driving DBMS from a system's perspective (i) by dividing problems into smaller subproblems and by (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. [42]. In their work, four features (indexes, partitioning, multi dimensional clustering, and materialized query tables) are considered for tuning 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. In addition, the joint optimization of dependent features is unfeasible because their number is, in general, prohibitively large. We also consider more dimensions to determine dependencies automatically and we do not limit the number of physical design features to tune.

In the remainder of this section, we highlight related work from several areas that are vital to self-driving database systems.

2.1 (Learning) cost models & benefit estimation

Many tuning techniques rely on cost estimations of single queries or complete workloads to estimate the benefit of certain potential configurations. For example, what-if based index selection [4] approaches rely on optimizer cost models. However, recent work of Ding et al. [12] shows that cost estimates for the same query given different plans that reflect different index configurations are often wrong to an extent where a predicted improvement in execution time turned out to be a performance impediment in reality. The authors could improve on that by utilizing machine learning and formulating the aforementioned problem of determining the better plan as a classification task. The authors' findings demonstrate that learning-based approaches are viable to improve the quality of benefit estimations.

In recent years, a couple of *machine learning*-based approaches for cost estimation have been published. These rely on different model types, e.g., gradient boosting regressors [21], support vector machines [1], or neural networks [17, 24, 35]. Marcus et al. [24] present a deep learning approach for query performance prediction. The architecture of their plan-structured neural networks represents the structure of the input query plan whose execution time should be predicted. Their approach offers particularly high prediction accuracies without relying on manually-selected input features which is of special interest for self-managing database systems.

2.2 Workload modeling and prediction

Even though self-managing database systems base their decisions typically on workload predictions, there is no common understanding of how to represent best what a database is processing, i.e., a workload. A detailed discussion and comparison

would go beyond the scope of this paper, but we want to briefly highlight the different alternatives to justify our solution for workload representation.

In [39] the authors see two orthogonal approaches to capture a workload: (i) On runtime level where system performance KPIs, e.g., the CPU utilization, the number of pages read, or the currently open transactions characterize a workload. Alternatively, (ii) on a logical level where the received queries and a representation of the stored data capture the processed workload. Since a self-driving database system is also optimizing its physical design for a particular (predicted) workload, a workload representation must model on which data the system operates as well as in which way and how often the data is accessed. Thus, we see it as insufficient to model the runtime behavior and decided to base tuning decisions on a logical workload representation. However, runtime level information (Sect. 3.1) is helpful to assess the effect of previous decisions.

Logical workload representations can be created with different granularities. (i) The system could store the queries exactly as they are received, as SQL trace [9]. These SQL strings contain information about the accessed relations and attributes. However, since SQL is a declarative language, they lack certain details, e.g., regarding the access paths. (ii) Alternatively, the workload can be captured on a query plan level: these represent more precisely how the data is accessed, which is a valuable input for optimization algorithms.

In addition, the points in time when a specific query was executed have to be stored to preserve effects like seasonality and concurrency.

Furthermore, there is some work on how to specifically represent, model, and store workloads. Besides describing primitives to summarize workloads, Chaudhuri et al. [7] present a schema to summarize SQL workloads. This schema contains information of three categories per statement: (i) syntactic and structural, e.g., (the statement type and SQL string), (ii) plan information, for example, the estimated cost and number of join conditions, and (iii) execution information, for instance, the recorded IO time and memory consumption.

Tran et al. [37] propose a Markov-based approach, which is implemented in the Oracle Database, and that is capable of creating workload models which only represent the workload's main characteristics. Thereby, they avoid over-fitting of their models in order to increase the potential for generalization.

Martin et al. [26] categorize workload models into two different types: (i) exploratory models [14] that are used for analysis and tuning tasks. The authors build such models, for example, by clustering the workload's queries along dimensions as *I/O utilization* or *CPU time consumption*. (ii) Confirmatory workload models indicate whether certain conditions regarding the system or its performance are met. An example for this model type could be the classification of a workload into analytical (OLAP) or transactional (OLTP) based on, e.g., the ratio of queries vs data-modifying statements, the throughput, the number of selected rows, and many more metrics.

In addition to above's solutions for representing a workload, Ma et al. [23] present a forecasting framework called QueryBot 5000 that is capable of predicting arrival rates of queries from historical observations. Their work also builds on a logical workload representation instead of a physical resource-based one. On a high

level, their approach focuses on a three-step approach: in the beginning, SQL query strings are preprocessed, constants are removed, and the formatting is normalized. Afterwards, the queries are clustered to reduce the total number of query templates, thereby making the approach feasible for a large number of templates. Lastly, the actual forecasting is applied. The authors evaluate forecasting techniques of varying complexity, e.g., linear regressions and recurrent neural networks.

2.3 Robust tuning approaches

Approaches for tuning problems in self-managing database systems usually assume that they operate on accurate and reliable input values. For example, in the literature, most index selection algorithms are designed to find the best index configuration for a specific workload while variation and uncertainty considerations of this workload are neglected.

However, it is difficult to perfectly predict, e.g., future workloads since database systems are often subject to unforeseeable events. Therefore, robustness is necessary to enable the application of self-managing approaches in practice [41]. This was also confirmed during the aforementioned interviews with industry database architects.

There are some approaches that specifically incorporate robustness. Boissier et al. [3] determine compression selections base on the system's workload with a focus on robust configurations. The authors take measures to limit the impact of high-frequency queries in order to anticipate potential workload shifts and mitigate the effects of long-running queries.

Tan and Babu [36] demonstrate a robust self-tuning approach for resource management in multi-tenant parallel database systems. The authors note that robustness is achieved by continuously monitoring quantitative metrics and reverting new system configurations in case the mentioned metrics do not dominate previously recorded ones.

Mozafari et al. [29] present a different approach called Cliffguard, which does not focus on tuning a single feature in a robust fashion. They describe a generic framework that employs robust optimization (RO) concepts [2]. Thereby, they add robustness to existing approaches independently of the concrete implementation or underlying database system. Robustness is discussed in more detail in Sect. 3.9.

3 Framework architecture

We present the architecture of our framework for self-managing database systems and the reasoning that influenced its design. 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
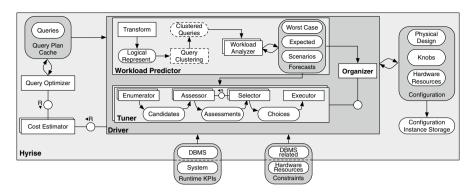
Fig. 1 Architecture diagram of our framework for self-managing database systems, see [18]

components and their interfaces in the following subsections after giving a short general overview of the architecture.

3.1 Overview

A diagram of the proposed framework and an integration into the database system Hyrise is depicted in Fig. 1. 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. Sect. 3.10 for more details. Second, by relying on components with clearly specified interfaces, we simplify the development and experiments of new approaches since components are exchangeable without effort.

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:

– *Workload predictor* The effect of a particular database configuration largely depends on the executed workload. Therefore, a component is required that predicts the upcoming workload based on historical workload data. In this context, the workload predictor is detailed in Sect. 3.8.
– *Tuner* Based on these predictions, the tuner employs a multi-step process to calculate a selection for a certain feature. Section 3.10 contains further information regarding the tuner.
– *Organizer* The *organizer* is orchestrating the whole self-managing processes. It is responsible for starting and stopping tunings during database runtime, enforcing *constraints* and assessing *runtime KPIs*; for more details see Sect. 3.11. Further, it determines in which order features should be tuned (Sect. 4).

Furthermore, the *driver* is the interface for accessing other database or system components that serve as external inputs. The aforementioned components loosely

resemble the MAPE (monitor, analyze, plan, execute) cycle [16] of autonomic systems. A direct one-to-one mapping of the cycle's tasks to our components is not possible since tasks like monitoring are fulfilled by multiple components, the organizer and workload predictor. We describe these inputs in the following paragraphs.

3.2 Query plan cache

Most relational database systems, e.g., SAP HANA [33] and Microsoft SQL Server [27] employ query plan caches. These have typically two purposes: the support of prepared statements and caching of optimized query plans to avoid repeated re-optimizations. We take advantage of the second aspect because information about past workload is necessary for workload-driven optimizations. 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.

3.3 Configurations

The *configuration* of a DBMS is the combination of all of its configurable entities. These *are* 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.

3.4 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. Further, 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.

3.5 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 cost of adjusting the configuration, for example, the utilized CPU time for indexing a number of attributes must be as quantifiable as the processing of a workload given a set of indexes. For example, the computation cost of a query given a particular index (cf. what-if optimization [4, 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 and possibly adaptive cost models are necessary to ensure a maximum of precision of cost predictions which, in turn, enable well-suited database configurations. There are a couple of existing approaches where cost models are created by learning from observed query execution costs, see Sect. 2.1.

3.6 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 disclose when the configuration should be adjusted. For example, when SLAs are constantly violated or performance peaks are detected. Furthermore, these KPIs help to identify phases of low resource utilization that are used to run resource-intensive tunings. Therefore, these are used by the *organizer* to identify favorable points in time for tuning runs.

3.7 Implementation strategies

Our proposed 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 decides 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 [40]. Further, applying new configurations to a whole table is a heavyweight operation. Applying these iteratively to chunks reduces the cost of these operations.

---

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

We see two strategies for implementing self-driving capabilities: as a standalone application outside of the DBMS or as component of the database core [32]. For the first option, a standalone application running outside the database system, the DBMS itself has to provide 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 also introducing overhead by further layers of indirection.

On the other hand, 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. The proposed framework works with both implementation strategies as long as the interfaces to the necessary data are provided.

We decided to implement self-driving capabilities with the plugin infrastructure[2] of Hyrise. Thereby, we 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.

3.8 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 (see Sect. 3.9) support the system in being less sensitive to irregular workload patterns, as seen, e.g., during crises or hypes, and seasonal effects like quarter ending calculations or payroll processing. In this work, we do not present a new technique for workload prediction but describe the functioning and interfaces with other components of our framework.

3.8.1 Workload analyzer

As a first step, the workload predictor accesses information about past workloads from the query plan cache (cf. Sect. 3.1). The information contains which queries were executed and their execution count and cost. By relying on the query plan

---

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

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 [23]) 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 offers the flexibility to hold 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.

3.9 Robustness and reconfiguration costs

Robustness is crucial for large database deployments where workloads are uncertain and not stable. We distinguish the following two aspects of robustness.

First, we refer to the robustness of a system's performance. The goal is to tune the system in a way that workload changes do not seriously affect the system's performance. Instead of optimizing the tuning for an average workload scenario, the tuning should be organized in a way that an acceptable performance is guaranteed for various workload scenarios. Thus, during tuning, common scenarios as well as rarely occurring extreme cases have to be taken into account. To achieve this, not only the expected workload has to be incorporated but also information about the distribution of potential future scenarios, cf. Sect. 3.8. In general, to obtain robust configurations users are willing to sacrifice a certain share of the optimal expected performance.

The second type of robustness refers to the actual installation of configurations. As workloads may change over time, the system's configurations have to be re-optimized regularly. Therefore, it is possible that an updated optimized configuration suggests an entirely different configuration even though the associated performance increase is comparably small. Its installation diminishes the benefit. To avoid such effects *reconfiguration costs* are considered to efficiently balance performance improvements and the installation of configurations to identify minimal-invasive configuration improvements. This has been applied, e.g., for index selection [34] and replication [25].

3.10 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. Sect. 3.1) 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 (see Sect. 4). A tuner, can contain multiple different instances of its subcomponents (Fig. 1). 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.

3.10.1 Enumerator

Enumerators provide a list of *Candidates* to the tuning process. Typically, the candidate set size is one of the main influence factors for 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. [4]) while others consider all available candidates. The framework allows to switch between different enumerators or fall back to restrictive enumerators when necessary. *Candidates* are 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., 0.1 GB to 500.0 GB and the way the value is increased: linearly, e.g., by 0.2 GB with each step or exponentially. Users can either implement enumerators on their own or utilize general ones provided by the system.

3.10.2 Assessor

This component provides assessments of the previously generated candidates. A positive or negative *desirability* indicating its impact on the overall system performance given a forecast scenario is assigned to each candidate. The system assigns different desirability values to the same candidate for different forecast scenarios. Later in the decision process these, possibly differing, desirability values 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 are 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 are incorporated during this step.

3.10.3 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, [34, 38] for index selection with greedy approaches.
- Optimal: Such selectors find optimal configurations (e.g., Dash et al. [11] for index selection or Halfpap et al. [15] for database replication). 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 [28]. Genetic algorithms (e.g., for index selection Kratica et al. [20]) 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: Such selectors are beneficial when acceptable performance in most cases is more important than best performance in the expected case which is likely the case when SLAs are specified [29]. Criteria based on mean-variance optimization, utility functions, value at risk, and worst-case considerations are used in such scenarios.

By strictly relying on the interfaces between components, selectors are exchangeable 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.

3.10.4 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 accesses runtime KPIs to determine favorable points in time for applying the choices.

3.11 Organizer

The whole self-managing process is orchestrated by the organizer. It identifies convenient points in time (e.g., phases of low system resource utilization) 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. The organizer decides on the *order* of tuning processes for these features. More details are given in Sect. 4. 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.

4 Tuning of multiple mutually dependent features

The tuning of multiple features is highly challenging as their inter-dependencies are usually complex and have a significant performance impact [39, 42]. Building an omnipotent model that is capable of determining efficient configurations for all features in a combined fashion is hardly feasible. As the solution space of single features for real-world problems is already substantial, a global model possesses a prohibitively large complexity. Instead, tuning each feature separately is computationally feasible, but also likely to provide a poor performance as feature dependencies are not considered.

4.1 Recursive tuning of single features

The key idea of our approach is to recursively tune single features. As features are mutually dependent the tuning order is crucial. 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 potential features. In such scenarios, only the features with the most significant impact are tuned. Therefore, the overall cost and benefit of the tuning of a specific feature need to be assessed.

Our approach is related to Zilio et al. [42]. They describe a hybrid approach that orders tuning processes by their pairwise dependence: (i) Non-dependent features are tuned one after another in any order, (ii) unidirectional dependent ones are tuned in the most efficient order, and (iii) mutually dependent ones are tuned simultaneously. However, due to (iii) the approach is limited if the number of mutually dependent features and their joint tuning complexity is too large, see also the discussion above and Sect. 2 for more details. Further, the dependencies of features

cannot be assumed to be known in advance. Accurately determining dependencies is of high complexity because it relies on expensive calculations (conducting tunings for all considered features as well as many calls to cost estimators).

In the following, we propose a mechanism to recursively tune all features in a reasonable order by taking their dependencies into account. The basic dependencies are automatically determined. 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. Furthermore, sampling or clustering of queries, as provided by the *workload predictor*, are techniques to reduce the workload size.

In addition, we 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 $d_{A,B} < 1$. In the following, if $d_{A,B} \geq 1$ and A is tuned before B, we say that A and B were tuned in beneficial order. Further, for all combinations of features we can calculate d pairwise. The ratios are used to determine an optimized order to recursively tune all features.

Since we define the dependency of two features A, B by workload cost, efficient and precise ways to estimate these costs are required. Hence, cost estimators, as described in Sect. 3.1 are crucial. In addition, the estimation of workload costs for many combinations and large workloads can become expensive. Decreasing the workload size, e.g., by clustering (cf. Workload Compression [6]) mitigates this problem in exchange for possibly less accuracy.

4.2 Optimization of the tuning order using linear programming

Deriving an optimized order of all features is a highly challenging task as (i) the number of potential orders (permutations) can be large and (ii) a consistent order satisfying all preferred pairwise relations does not have 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)[3] 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 ($x_{A,k} = 0$), $k = 1, \ldots, |S|$, 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 optimize the tuning order, we propose the following integer LP formulation:

$$
\begin{array}{c}
\textit{maximize} \\
x_{A,k}, y_{A,B} \in \{0,1\} \\
A \in S, B \in S\backslash\{A\}, k = 1, \ldots, |S|
\end{array}
\sum_{A \in S, B \in S\backslash\{A\}} y_{A,B} \cdot d_{A,B} \cdot W_{\emptyset}/W_{A,B}
\tag{1}
$$

subject to the feasibility constraints

$$
\sum_{k=1,\ldots,|S|} x_{A,k} = 1, \quad \forall A \in S \quad \textit{and} \quad \sum_{A \in S} x_{A,k} = 1, \quad \forall k = 1, \ldots, |S|
\tag{2}
$$

as well as the coupling constraints

$$
y_{A,B} + y_{B,A} = 1, \quad \forall A \in S, B \in S\backslash\{A\} \quad \textit{and}
\tag{3}
$$

$$
|S| \cdot y_{A,B} \geq \sum_{k=1,\ldots,|S|} k \cdot x_{B,k} - \sum_{k=1,\ldots,|S|} k \cdot x_{A,k}, \quad \forall A \in S, B \in S\backslash\{A\}
\tag{4}
$$

The objective (1) optimizes the sum of the dependence ratios d weighted by the associated impact coefficients $W_{\emptyset}/W_{A,B}$, $A \in S, B \in S\backslash\{A\}$. The first two families of constraints (2) 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 (3)–(4) *uniquely* couple the variables x and y in a *linear* way. The first one, cf. (3), 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 (4) 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. Further, in case certain features are *required* to be tuned in a specific order or in a particular step (for instance, determined by domain knowledge) such additional information is formulated as additional constraints and directly included in the LP. As a result, the complexity of the LP will decrease because there is less freedom for feasible tuning orders.

Robust tuning approaches are also applicable in this framework, see Sect. 3.9. In this case the workload costs W will typically describe a risk-averse criterion, e.g., the expected utility of a certain tuning configuration assuming different potential workloads. Note, the complexity of the LP to determine a suitable order of tuning the features under robustness considerations is *not* affected as the model remains the same and only the inputs W change.

Further, reconfiguration costs should, in general, also be part of W to take the current state of configurations into account, cf. Sect. 3.9. Alternatively, they could also be left out in the workload cost estimations for pairwise tunings to first determine an

Table 1 Tuning order problem input values: relative workload costs $W_{A,B}$ for tuning feature B after feature A, $A \in S$, $B \in S\backslash\{A\}$, for the 3-dimensional example with $S := \{Index, Comp, Clust\}$; no tuning refers to the reference workload costs of $W_{\varnothing} := 100$

A	B		
	Index	Comp	Clust
Index	–	75	89
Comp	68	–	60
Clust	84	66	–

unbiased tuning order and then take them into account when tuning all components recursively.

4.3 Case study: tuning index selection, data compression, and clustering

In this subsection, we highlight the complexity and challenges when multiple features are tuned. For this discussion we focus on the tuning of features that can be tuned automatically in the main memory database system hyrise: indexes [34], data compression [3], and column clustering choices [22].

In such a scenario, tuning decisions are mutually dependent because decisions for one feature influence the performance impact of other features. For data compression the objective is mainly to reduce the memory footprint without affecting the performance too much, the objective for index selection is to improve the overall workload performance while only spending a fixed amount of memory. The main goal of clustering is again to avoid data access but on a more coarse-grained level than for indexes. The stored data is organized in a way that maximizes pruning opportunities during query processing. The above-presented chunk concept of Hyrise enables these optimizations.

To give a more practical example: In general, frequently accessed attributes are candidates for being indexed because the increased memory footprint has the chance to pay off in such cases. On the other hand, infrequently accessed attributes could be heavily compressed to reduce the memory footprint while indexes and clustering could be used in such scenarios to improve performance and avoid unnecessary but costly accesses of compressed data. Conflicting optimization goals, e.g., memory consumption and performance, demonstrate the need to consider feature dependencies.

To optimize the tuning order for $S := \{Index, Comp, Clust\}$, we determine the dependencies of, e.g., compression and indexes, $d_{Comp,Index}$ based on workload processing costs. For example, $W_{Index,Comp}$ is the cost of executing the workload at hand when indexes are selected before the compression configuration is chosen. The workload costs for all feature combinations, cf. S, serve as input for the above-specified LP, which finally provides an optimized tuning order.

Table 1 shows workload processing costs $W_{A,B}$, relative and normalized to $W_{\varnothing} := 100$ for the three features index selection, data compression, and automatic

Table 2 Tuning order solution: selected costs components $d_{A,B} \cdot W_{\varnothing}/W_{A,B}$ for tuning feature B after feature A, $A \in S, B \in S\backslash\{A\}$, for the 3-dimensional example with $S := \{Index, Comp, Clust\}$; unselected terms ($y^*_{A,B} = 0$) are given in brackets

A	B		
	Index	Comp	Clust
Index	–	(1.21)	(1.06)
Comp	1.62	–	1.83
Clust	1.26	(1.38)	–

clustering, $A, B \in S := \{Index, Comp, Clust\}$. The workload processing costs were obtained with Hyrise's TPC-H benchmark executable while the particular configurations were applied. The executable generates and loads the data set before executing each TPC-H query for one minute if not configured differently. For this scenario we used a scale factor of 10. The default (not tuned) configuration does not encode columns, no indexes are created and the data is not clustered or sorted in any particular order. For the case study, we also utilize compression configurations that, e.g., encode attributes which are accessed in the benchmark more heavily. Such attributes are natural index candidates for cost-based index selection algorithms since accessing these attributes is more expensive. This effect cannot be observed if indexes are determined before compression configurations.

While for instance, tuning feature 'Index' before feature 'Comp' reduces the workload costs to $W_{Index,Comp} = 75$, tuning 'Index' *after* 'Comp' appears more beneficial as the costs $W_{Comp,Index} = 68$ are lower than before. This difference indicates that the tuning order matters.

Finally, based on the workload costs in Table 1 the optimal solution of the LP (1)–(4) yields the tuning order: *(i) compression (ii), clustering, and (iii) indexes*. The associated relative workload costs are 51.

The obtained tuning order seems plausible since indexes can mitigate the elevated access costs for heavily compressed data. Furthermore, indexes are of help where data access cannot be avoided by simple data access avoidance techniques like pruning through clustering. The result can be easily reproduced using the provided AMPL program (see Footnote 3) and the input data. In this context, Table 2 summarizes the potential and finally selected cost terms of the weighted objective, cf. (1), that are associated with the tuning order obtained. In the final order of the example, all three pairs of features are tuned in beneficial order. The associated optimal objective value is $1.62 + 1.83 + 1.26 = 4.71$, cf. Table 2.

Finally, for comparison, we applied the three tuning features *(Comp, Clust, Index)* in all 6 possible orders. We obtained the following results for the (relative) workload costs:

  i.   *Comp, Clust, Index*: 51
 ii.   *Comp, Index, Clust*: 60
iii.   *Clust, Comp, Index*: 55

   iv. *Clust, Index, Comp*: 69
   v. *Index, Comp, Clust*: 62
   vi. *Index, Clust, Comp*: 77

For our exemplary case study the results show that the order derived by the LP approach is indeed the best one leading to the lowest workload costs (i.e., 51). The results indicate that proper tuning orders can be determined based on pairwise tuning inputs and the presented LP approach. Note, as the approach is a heuristic, optimal solutions cannot be guaranteed for all scenarios.

4.4 Scalability considerations

In this section, we demonstrate the feasibility and functioning of our solution for larger problem instances containing more features ($|S|$). To illustrate the applicability of our LP approach, we consider a scalable synthetic example with $|S| = K$ features, see Example 1.

Example 1 (Scalable tuning problem) We assume a tuning problem with K potentially dependent features. We consider (randomized) inputs $W_{A,B} \in [0, 100]$ for tuning feature B after feature A, $A \in S, B \in S\backslash\{A\}$, $S := \{1, 2, \ldots, K\}$. The inputs are chosen that two features are independent (i.e., $W_{A,B} = W_{B,A}$) with probability $\pi \in [0, 1]$.

    The source code for the example is available as open source, see Footnote 3. While pairwise dependencies (in total $|S| \cdot (|S| - 1)$) can still be derived for a more significant number of features an *exhaustive* computation of the results of all potential tuning permutations (in total $|S|!$) is intractable.

Table 3 Tuning order problem input for $K = 10$ features: Relative workload costs $W_{A,B}$ for tuning feature B after feature A, $A \in S, B \in S\backslash\{A\}$, $S := \{1, \ldots, 10\}$; no tuning refers to the reference workload costs of $W_\varnothing := 100$; Example 1

A	B									
	1	2	3	4	5	6	7	8	9	10
1	–	46	53	37	43	22	45	55	39	48
2	34	–	29	30	15	30	9	16	5	13
3	53	29	–	67	22	60	21	53	29	69
4	37	34	47	–	24	37	22	23	25	62
5	43	89	22	86	–	66	91	84	47	20
6	22	25	14	37	10	–	7	10	17	34
7	45	9	21	37	22	75	–	57	35	72
8	71	16	53	40	84	10	57	–	43	8
9	39	36	24	23	47	12	26	43	–	18
10	49	13	69	62	20	34	72	34	15	–

Table 4 Tuning order solution for $K = 10$ features: selected costs components $d_{A,B} \cdot W_{\varnothing}/W_{A,B}$ for tuning feature B after feature A, $A \in S, B \in S\backslash\{A\}$, $S := \{1, 2, \ldots, K\}$; unselected terms ($y_{A,B}^* = 0$) are given in brackets; Example 1

A	B									
	1	2	3	4	5	6	7	8	9	10
1	–	(1.61)	1.89	2.70	2.33	4.55	2.22	2.35	2.56	2.13
2	3.98	–	3.45	3.78	39.56	**2.78**	11.11	6.25	144.00	7.69
3	(1.89)	(3.45)	–	(1.05)	(4.55)	(0.39)	(4.76)	(1.89)	(2.85)	(1.45)
4	(2.70)	(2.60)	3.03	–	14.93	2.70	7.64	7.56	**3.68**	1.61
5	(2.33)	(0.19)	4.55	(0.32)	–	(0.23)	(0.27)	(1.19)	(2.13)	(5.00)
6	(4.55)	**(4.80)**	30.61	(2.70)	66.00	–	153.06	(10.00)	(4.15)	(2.94)
7	(2.22)	(11.11)	4.76	(1.61)	18.80	(0.12)	–	(1.75)	(2.12)	(1.39)
8	(1.09)	(6.25)	1.89	(1.44)	1.19	10.00	1.75	–	2.33	53.12
9	(2.56)	(0.39)	5.03	**(4.73)**	2.13	11.81	5.18	(2.33)	–	(4.63)
10	(2.00)	(7.69)	1.45	(1.61)	5.00	2.94	1.39	(0.69)	8.00	–

The two pairs of features that are not tuned in beneficial order are highlighted with bold type

Table 5 Runtimes in seconds of the LP (1)–(4) for different numbers of features K and dependency probabilities π; Example 1

K	$\pi = 0.0$ (s)	$\pi = 0.2$ (s)	$\pi = 0.5$ (s)	$\pi = 0.8$ (s)
3	0.02	0.02	0.01	0.01
6	0.05	0.03	0.02	0.02
10	0.40	0.18	0.14	0.03
15	3.43	3.12	1.01	0.25
20	120.14	72.23	8.48	0.51

Table 3 illustrates the input for a (reproducible) tuning order problem of Example 1 with $K = 10$ features. In this example, the feature dependency probability π is 0.5.

Table 4 illustrates the optimal order (for the example presented in Table 3) revealed by the LP (1)–(4) as well as the summands of the objective. We observe that in the final order all except two pairs of features ((2,6) and (4,9)) are tuned in the beneficial order (according to the input). For these two pairs of features, we observe that opportunity costs of the *wrong* tuning order (as intended) is comparably low, which is guaranteed by the weighting factors used in the objective (1).

In general, as it is not possible to consistently tune each of the $(K \cdot (K - 1)/2)$ pairs in the beneficial order, the LP is designed to realize the beneficial tuning order of *important pairs*, i.e., if their impact on performance is significant or their order strongly matters (cf. (6,7) or (2,9)).

All examples were solved single-threaded on a consumer notebook from 2013 with 20 GB of main memory and an Intel i7 CPU with 2.4 GHz, using the Gurobi solver version 8.1.0. The LP's solving time for the setting of Table 3 is close to zero (0.02 seconds). To assess the scalability of our LP approach further, Table 5

illustrates the solving times for different numbers of features (K). The table also studies the impact of the share of independent feature inputs characterized by the probability π, cf. Example 1.

We observe that the share of pairwise independent features has a significant impact on the LP's runtime. Overall, the results verify that the approach is applicable for determining tuning orders for at least up to 20 strongly dependent features in a reasonable amount of time.

For higher numbers of features LP relaxation approaches such as time limits or the use of optimality gaps could be applied. For instance, using an optimality gap of 10%, for the case $K = 20$ and $\pi = 0.2$ (see Table 5), we obtain a solution with 95% of the optimal objective value in less than 2% of the time (1.3 s).

5 Future work

The presented concepts and ideas open many opportunities for further research. Self-driving systems must be able to precisely assess costs and benefits of preferably each and every operation and action. Thus, the focus of our current work is to incorporate observed execution costs into the current cost models to increase their accuracy.

Robust configurations are especially important for the adoption of self-driving 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 end-to-end evaluation of the presented approach, see Sect. 4, on determining a favorable order to tune multiple features is necessary to better assess the viability and performance implications for large problem instances.

6 Conclusion

The proposed framework describes how to divide the challenge of integrating self-management capabilities in database systems into smaller problems that are tackled by components. We gave a detailed definition of the specific components and their interfaces. Separation of concerns facilitates exchange and reuse of components to streamline experimentation and development. The presented workload predictor is capable of anticipating multiple potential workload scenarios that are incorporated by the respective tuners to allow robust solutions. Moreover, in Sect. 4, we proposed and tested an LP-based approach to identify an efficient order for tuning of mutually dependent features by determining their degree of dependency.

References

1. Akdere, M., Çetintemel, U., Riondato, M., Upfal, E., Zdonik, S.B.: Learning-based query performance modeling and prediction. In: Proceedings of the International Conference on Data Engineering (ICDE), pp. 390–401 (2012)
2. Ben-Tal, A., El Ghaoui, L., Arkadi, N.: Robust Optimization. Vol. 28. Princeton Series in Applied Mathematics. Princeton University Press, Princeton (2009)
3. Boissier, M., Jendruk, M.: Workload-driven and robust selection of compression schemes for column stores. In: Proceedings of the International Conference on Extending Database Technology (EDBT), pp. 674–677 (2019)
4. Chaudhuri, S., Narasayya, V.R.: An efficient cost-driven index selection tool for Microsoft SQL server. In: Proceedings of the International Conference on Very Large Databases (VLDB), pp. 146–155 (1997)
5. Chaudhuri, S., Weikum, G.: Self-management technology in databases. In: Encyclopedia of Database Systems, pp. 2550–2555 (2009)
6. Chaudhuri, S., Kumar G., Ashish, N., Vivek R.: Compressing SQL workloads. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 488–499 (2002)
7. Chaudhuri, S., Ganesan, P., Narasayya, V.R.: Primitives for workload summarization and implications for SQL. In: Proceedings of the International Conference on Very Large Databases (VLDB), pp. 730–741 (2003)
8. Chaudhuri, S., Narasayya, V.R.: Self-tuning database systems: a decade of progress. In: Proceedings of the International Conference on Very Large Data Bases (VLDB), pp. 3–14 (2007)
9. Curino, C., Zhang, Y., Jones, E.P.C., Madden, S.: Schism: aWorkload-driven approach to database replication and partitioning. In: Proceedings of the International Conference on Very Large Databases (VLDB), pp. 48–57 (2010)
10. Das, S., et al.: Automatically indexing millions of databases in Microsoft Azure SQL database. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 666–679 (2019)
11. Dash, D., Polyzotis, N., Ailamaki, A.: CoPhy: a scalable, portable, and interactive index advisor for large workloads. In: Proceedings of the International Conference on Very Large Databases (VLDB), pp. 362–372 (2011)
12. Ding, B., et al.: AI Meets AI: leveraging query executions to improve index recommendations. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 1241–1258 (2019)
13. Dreseler, M., et al.: 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), pp. 313–324 (2019)
14. Elnaffar, S., Patrick M., Horman, R.: Automatically classifying database workloads. In: Proceedings of the International Conference on Information and Knowledge Management (CIKM), pp. 622–624 (2002)
15. Halfpap, S., Schlosser, R.: Workload-driven fragment allocation for partially replicated databases using linear programming. In: Proceedings of the International Conference on Data Engineering (ICDE), pp. 1746–1749 (2019)

16. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Comput. **36**(1), 41–50 (2003)
17. Kipf, A., et al.: Learned cardinalities: estimating correlated joins with deep learning. In: Online Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR) (2019)
18. Kossmann, J., Schlosser, R.: A framework for self-managing database systems. In: Proceedings of the International Conference on Data Engineering ICDE Workshops, pp. 100–106 (2019)
19. Kraska, T., et al.: SageDB: a learned database system. In: Online Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR) (2019)
20. Kratica, J., Ljubic, I., Tosic, D.: A genetic algorithm for the index selection problem. In: Proceedings of the Applications of Evolutionary Computing, EvoWorkshop, pp. 280–290 (2003)
21. Li, J., Christian König, A., Narasayya, V.R., Chaudhuri, S.: Robust estimation of resource consumption for sql queries using statistical techniques. In: Proceedings of the International Conference on Very Large Databases (VLDB), pp. 1555–1566 (2012)
22. Lightstone, S., Bhattacharjee, B.: Automated design of multidimensional clustering tables for relational databases. In: Proceedings of the International Conference on Very Large Databases (VLDB), pp. 1170–1181 (2004)
23. Ma, L., et al.: Query-based workload forecasting for self-driving database management systems. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 631–645 (2018)
24. Marcus, R.C., Papaemmanouil, O.: Plan-structured deep neural network models for query performance prediction. Proc. Int. Conf. Very Large Databases (VLDB) **12**(11), 1733–1746 (2019)
25. Marcus, R., Papaemmanouil, O., Semenova, S., Garber, S.: NashDB: an end-to-end economic method for elastic database fragmentation, replication, and provisioning. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 1253–1267 (2018)
26. Martin, P., Elnaffar, S., Wasserman, T.J.: Workload models for autonomic database management systems. In: Proceedings of the International Conference on Autonomic and Autonomous Systems (ICAS) (2006)
27. Microsoft.: Query processing architecture guide—execution plan caching and reuse. https://docs.microsoft.com/en-US/sql/relational-databases/query-processing-architecture-guide?view=sql-server-ver15#execution-plan-caching-and-reuse (2019)
28. Mitchell, M.: An Introduction to Genetic Algorithms. MIT Press, Cambridge (1998)
29. Mozafari, B., Goh, E.Z.Y., Yoon, D.Y.: Cliff-guard: a principled framework for finding robust database designs. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 1167–1182 (2015)
30. Oracle: World's First "Self-Driving" Database, 2018. https://www.oracle.com/database/autonomous-database/feature.html (2019)
31. Pavlo, A., et al.: Self-driving database management systems. In: Online Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR) (2017)
32. Pavlo, A., et al.: External vs. internal: an essay on machine learning agents for autonomous database management systems. IEEE Data Eng. Bull. **42**(2), 31–45 (2019)
33. SAP: Analyzing SQL execution with the SQL plan cache. 2019. https://help.sap.com/viewer/bed8c14f9f024763b0777aa72b5436f6/2.0.04/en-US/bed20ba0bb57101483ffa333cf3e55c8.html (2019)
34. Schlosser, R., Kossmann, J., Boissier, M.: Efficient scalable multi-attribute index selection using recursive strategies. In: Proceedings of the International Conference on Data Engineering (ICDE), pp. 1238–1249 (2019)
35. Sun, J., Li, G.: An end-to-end learning-based cost estimator. In: Proceedings of the International Conference on Very Large Databases (VLDB), pp. 307–319 (2019)
36. Tan, Z., Babu, S.: Tempo: robust and self-tuning resource management in multi-tenant parallel databases. In: Proceedings of the International Conference on Very Large Databases (VLDB), pp. 720–731 (2016)
37. Tran, Q.T., Morfonios, K., Polyzotis, N.: Oracle workload intelligence. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 1669–1681 (2015)
38. Valentin, G., Zuliani, M., Zilio, D.C., Lohman, G.M., Skelley, A.: DB2 advisor: an optimizer smart enough to recommend its own indexes. In: Proceedings of the International Conference on Data Engineering (ICDE), pp. 101–110 (2000)
39. Van Aken, D., Pavlo, A., Gordon, G.J., Zhang, B.: Automatic database management system tuning through large-scale machine learning. In: Proceedings of the International Conference on Management of Data (SIGMOD), pp. 1009–1024 (2017)

40. Vogelsgesang, A., et al.: Get real: how benchmarks fail to represent the real world. In: Proceedings of the International Workshop on Testing Database Systems, pp. 1:1–1:6 . DBTest@SIGMOD 2018, Houston, TX, USA, June 15, 2018 (2018)
41. Weikum, G., Mönkeberg, A., Hasse, C., Zabback, P.: Self-tuning database technology and information services: from wishful thinking to viable engineering. In: Proceedings of the International Conference on Very Large Databases (VLDB), pp. 20–31 (2002)
42. Zilio, D.C., et al.: DB2 design advisor: integrated automatic physical database design. In: Proceedings of the International Conference on Very Large Data Bases (VLDB), pp. 1087–1097 (2004)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.