

# Make Your Database System Dream of Electric Sheep: Towards Self-Driving Operation

Andrew Pavlo, Matthew Butrovich, Lin Ma  
Prashanth Menon, Wan Shen Lim, Dana Van Aken, William Zhang  
Carnegie Mellon University  
pavlo@cs.cmu.edu

## ABSTRACT

Database management systems (DBMSs) are notoriously difficult to deploy and administer. Self-driving DBMSs seek to remove these impediments by managing themselves automatically. Despite decades of DBMS auto-tuning research, a truly autonomous, self-driving DBMS is yet to come. But recent advancements in artificial intelligence and machine learning (ML) have moved this goal closer.

Given this, we present a system implementation treatise towards achieving a self-driving DBMS. We first provide an overview of the NoisePage self-driving DBMS that uses ML to predict the DBMS's behavior and optimize itself without human support or guidance. The system's architecture has three main ML-based components: (1) workload forecasting, (2) behavior modeling, and (3) action planning. We then describe the system design principles to facilitate holistic autonomous operations. Such prescripts reduce the complexity of the problem, thereby enabling a DBMS to converge to a better and more stable configuration more quickly.

## PVLDB Reference Format:

Andrew Pavlo, Matthew Butrovich, Lin Ma and Prashanth Menon, Wan Shen Lim, Dana Van Aken, William Zhang. Make Your Database System Dream of Electric Sheep: Towards Self-Driving Operation. PVLDB, 14(12): 3211 - 3221, 2021.  
doi:10.14778/3476311.3476411

## 1 INTRODUCTION

Much of the previous work on automated DBMSs has focused on standalone tuning tools that target a single problem. For example, some tools choose the best logical or physical design of a database, such as indexes [15, 29, 30, 69], partitioning schemes [5, 52, 55, 58, 60, 79], data organization [7], or materialized views [4]. Other tools select the tuning parameters for an application [6, 12, 26, 38, 70, 77]. Most of these tools operate in the same way: the DBA provides a sample database and workload trace that guides the tool's search process to find a configuration that optimizes a single aspect of the system (e.g., what index to build). The major vendors' tools, including Oracle [25, 36], Microsoft [14, 51], and IBM [66, 68], operate in this manner. There is a recent trend for integrated components that support adaptive architectures [8, 31], but these again only solve one problem at a time. Cloud database vendors employ automated

resource management tools at the service-level [23] or provide managed versions of their previous recommendation tools [2, 22].

Although these previous efforts are influential, they are insufficient for a completely autonomous DBMS because they only solve part the problem. That is, they are only able to identify potential *actions* that may improve the DBMS's performance (e.g., which index to add). They are unable, however, to infer which ones to apply and when to apply them because they do not predict workload trends or account for deployment costs [43]. Thus, they rely on a knowledgeable human DBA to update the DBMS during a time window when it will have the least impact on applications. They are also unable to learn which actions under what conditions provide the most benefit and then apply that knowledge to new situations [44]. This need for a human expert contributes to the high cost of ownership for DBMS software and the difficulty in supporting complex applications.

What is needed is a *self-driving* DBMS that predicts an application's needs and then automatically chooses actions that modify all system aspects holistically [56]. The DBMS learns how it responds to each action it applies and reuses such knowledge in different scenarios. With this knowledge, a self-driving DBMS can potentially support most management tasks without requiring a human to determine the proper way and time to deploy them.

The goal of a self-driving DBMS is to configure, manage, and optimize itself automatically as the database and its workload evolve over time. The core idea that guides the DBMS's decision-making is a human-selected *objective function*. An objective function could be either performance metrics (e.g., throughput, latency, availability) or deployment costs (e.g., hardware, cloud resources). This is akin to a human telling a self-driving car their desired destination. The DBMS must also operate within human-specified constraints, such as cost budgets or service-level objectives (SLOs).

The way that a self-driving DBMS improves its objective function is by deploying *actions* that it deems will help the application workload's execution. These actions control three aspects of the system: (1) *physical design*, (2) *knob configuration*, and (3) *hardware resources*. The first are changes to the database's physical representation and data structures (e.g., indexes). The second action type are optimizations that affect the DBMS's runtime behavior through its configuration knobs. These knobs can target individual client sessions or the entire system. Lastly, the resource actions change the hardware resources of the DBMS (e.g., instance type, number of machines); these assume that the DBMS is deployed in an elastic/cloud environment where additional resources are readily available.

In this paper, we provide an overview of our ongoing research towards achieving a true self-driving DBMS. We begin with a discussion of the different levels of automation that a DBMS can support.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing [info@vldb.org](mailto:info@vldb.org). Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.  
Proceedings of the VLDB Endowment, Vol. 14, No. 12 ISSN 2150-8097.  
doi:10.14778/3476311.3476411

We then describe the self-driving architecture of the **NoisePage** DBMS [1]. The design of NoisePage’s planning components are inspired from self-driving vehicles to facilitate autonomous operation [39]. This system serves as our research vehicle for exploring new ML methods to solve complex action planning problems [64].

But even the most sophisticated ML methods are impotent unless system developers design the DBMS to support autonomous operations. Thus, we also present software engineering design principles that we believe are necessary in a self-driving DBMS. These come from our experiences developing ML-based tuning tools for existing systems [6, 58, 70, 76] and new autonomous architectures [1, 8, 43, 44, 56, 59]. Such principles are for developers who are working on either existing DBMSs or building one from scratch. Furthermore, they are independent of the ML methods or the system implementation. The latter includes the storage organization (disk vs. memory), system architecture (single-node vs. distributed), or workload (OLTP vs. OLAP).

## 2 SELF-DRIVING DATABASES

There is currently no standard definition of what it means for a DBMS to be “self-driving”. Furthermore, there is confusion about how self-driving systems relate to self-adaptive [30], self-tuning [14], and self-managing [36] DBMSs. Achieving full autonomy in a DBMS is years away. There are, however, intermediate levels where a DBMS removes control from humans and takes over more management responsibilities on its own. The amount of autonomy of a self-driving DBMS plays a central role in its design and the user experience. Thus, we propose the following taxonomy on the levels of autonomy that a DBMS can provide. We organize this in increasing levels of autonomy with decreasing user interaction and control. An overview of these levels is shown in Table 1.

**Level #0:** At the lowest level of autonomy, the system provides an interface for the user to operate in a *manual* fashion. In other words, the DBMS only does exactly what the human instructs it.

**Level #1:** The next level of autonomy provides *assistant* tools that recommend improved configurations to the user for some DBMS sub-systems. The user has to (1) select which recommendations to apply, (2) decide when to apply them, and (3) monitor the DBMS’ behavior afterward. These tools often require the user to prepare sample workloads and/or deploy a second database copy to evaluate new recommendations [13]. This level’s examples include self-tuning tools that propose knob configuration [6, 26, 67, 70] or index selection [15, 80] based on workload traces.

**Level #2:** This level of autonomy assumes a minimal control system that collaborates with the user to configure some DBMS sub-systems. The initiative for making decisions is *mixed* between the DBMS and the user. Such autonomy exposes complex problems to the user since the user may issue changes that conflict with the DBMS’s control component. One example of this type of system was a prototype for IBM DB2 from 2008 [74]. This system used existing tools [37] in an external controller that triggered a change or notify a DBA whenever a resource threshold was surpassed (e.g., buffer pool hit ratio). It still required a human to select optimizations and to occasionally restart the DBMS.

Level	Name	Description
0	Manual	No autonomy.
1	Assistant	Recommendations to assist user.
2	Mixed	Simultaneous user and system control.
3	Local	Self-contained components.
4	Directed	Semi-autonomous with direction from user.
5	Self-Driving	Fully autonomous without direction.

**Table 1: Levels of Autonomy** – A classification of the levels of capabilities of autonomy in DBMSs.

**Level #3:** The next level consists of *local* autonomy where each sub-system can adapt without human guidance, but there is no higher-level coordination between them or long-term planning. The user decides to enable/disable these components’ autonomy. For example, some DBMSs support automatic memory allocations (e.g., Oracle [36], IBM DB2 [66]) or automatic creation/drop of indexes (e.g., Microsoft’s auto-indexing in Azure [22, 42]). We also believe that Oracle’s autonomous database-as-a-service offerings are this level’s examples [2]. The service requires users to select whether their application is a transactional or analytical workload. It then uses Oracle’s existing assistance tools (Level #1) to handle common tuning activities, but they are managed in isolation.

**Level #4:** At the *directed* level, the system manages all its sub-systems and the user only provides high-level direction for global configuration issues. Such direction includes hints about future workload needs so that the DBMS can prepare for long-term decisions (e.g., capacity planning). For example, the user may select actions to begin scaling the DBMS’s deployment to prepare for an upcoming surge in traffic. We also anticipate that the DBMS still requires session-specific hints from humans at this level. The DBMS is intelligent enough to recognize when to ask a human for help.

**Level #5:** Lastly, at the highest level of autonomy, the DBMS is completely independent (i.e., *self-driving*) because it coordinates across all sub-systems holistically without direction from the user. It accounts for future workloads in its decision-making and supports all tuning activities that do not require an external value judgment. For each planned or deployed action, the DBMS also provides human-understandable explanations about why it made the decision to deploy that action. For example, the DBMS reports its estimations on the action’s completion time, resource consumption, and benefits to the system performance. Whether a self-driving DBMS should still allow humans to configure aspects of the system is up for debate. How the DBMS’s planning components would respond to such human input is a difficult and unsolved problem.

## 3 NOISEPAGE ARCHITECTURE

We now describe NoisePage’s self-driving architecture that aims to achieve the highest level of DBMS autonomy (Level #5). We use the analogy of self-driving cars to explain our system’s design. At a high-level, a self-driving car consists of (1) a perception system, (2) mobility models, and (3) a decision-making system [54]. The *perception system* observes the road condition and predicts the future state, such as other vehicles’ moving trajectories. The *mobility models* estimate the vehicle’s behavior under a control action in relevant operating conditions. Lastly, the *decision-making system*

**Figure 1: NoisePage Architecture** – NoisePage’s self-driving architecture consists of an on-line workload forecasting component, an on-line behavior modeling component, and an on-line action planning component.

selects actions to achieve the driving objectives using its perception data and model estimates.

As illustrated in Figure 1, NoisePage’s self-driving architecture consists of three similar components: (1) workload forecasting, (2) behavior models, and (3) action planning [57]. NoisePage’s on-line *workload forecasting* component predicts the application’s future queries over multiple time horizons based on their past arrival rates [43]. NoisePage then uses these forecasts with its *behavior models* generated on-line to predict the cost and benefit of deploying actions without expensive exploratory testing [44]. With this cost/benefit estimates, the DBMS’s on-line *action planning* component selects action(s) that it believes will improve the system’s target objective function (e.g., latency, throughput). NoisePage then automatically applies these actions, monitors its behavior, and repeats the process. All of this occurs without any human intervention.

We now discuss these three components in more detail:

**Workload Forecasting:** Forecasting allows the DBMS to prepare itself for future workloads, much like a self-driving car predicts the road condition up ahead using LIDAR and cameras. If the DBMS only considers the past workload, it will be unable to prepare itself in time for upcoming changes in the application’s workload that require actions with long deployment time. Applying actions without knowing the future workload can also increase resource contention if DBMS applies them during inopportune times.

NoisePage uses workload traces and database statistics to generate forecast models [43]. When NoisePage receives the workload from the clients, it stores the workload’s query arrival rates and parameter samples in aggregated intervals (e.g., per minute) in internal tables. It periodically sends this data to its training components process to build the forecast models. These models predict the future arrival rates for each query in the aggregated intervals over different horizons (e.g., one hour, one day, one week). NoisePage also uses query templating, arrival-rate-pattern clustering, and an ensemble of time-series forecasting methods to efficiently train these models and accurately predict various workload patterns.

**Behavior Modeling:** Similar to a self-driving vehicle that uses models to estimate the effect of turning the steering wheel, a self-driving DBMS also uses *behavior models* to estimate and explain how a potential action changes the system’s performance. This

is challenging because DBMSs are complex software that may require a high-dimensional model, which may require a large amount of training data and make debugging more difficult. Concurrent operations in multi-core environments further complicate DBMS behavior modeling.

To address these issues, NoisePage decomposes the DBMS’s into small and independent *operating units* (OUs) that it then models separately [44]. Each OU model represents some internal task in the system (e.g., scanning a table, building an index). The OU decomposition means that each model is low-dimensional and only requires a moderate amount of training data. NoisePage also uses an additional *interference model* that captures the resource competition among concurrent OUs. It uses a set of specialized on-line runners that sufficiently exercise each OU to generate training data for the OU models. It selects the best ML algorithm for each model over a wide range of popular ML algorithms using cross-validation.

**Action Planning:** A self-driving DBMS needs to decide when to apply which actions to optimize the objective given the workload forecast and model estimation. To make a DBMS completely autonomous, this planning step must (1) account for both the current and future workloads to address problems before they occur, (2) satisfy all system constraints (e.g., maximum memory consumption), and (3) provide explanations for the past and future planned actions, which are important for examination, debugging, and auditing. NoisePage uses a standard scheme in control theory, called *receding horizon control* (RHC) [48], to ensure these requirements. Although RHC allows a system to perform multi-stage planning, solving RHC’s continuous, discrete, and constrained optimization problem is computationally expensive. Thus, NoisePage employs a Monte Carlo tree search (MCTS) planning method to improve RHC’s efficiency. MCTS explores a number of randomized sequences of actions up to the planning horizon and selects the best sequence. It can potentially balance the trade-off between the planning cost and quality. Previous research on AI systems have successfully used MCTS, such as AlphaGo [64].

In the next sections, we discuss the design principles for building such a self-driving DBMS. As this is an active research area, there are many unanswered problems. We will not cover the ML methods to solve them. Instead, our focus is on how to build a DBMS that is

E O		A M D		A E	
<b>Workload History (§4.1)</b> Maintain a history of the queries and transactions with their execution context, runtime behavior, plan hints, and result.		<b>Configuration Knobs (§5.1)</b> Mark which knobs are untunable. Expose a knob's accepted value range or options. Mark whether a knob can be tuned per session. Provide hints on how to increment/decrement each knob based on its current value. Include meta-data tags about their DBMS sub-system and hardware resource.		<b>No Downtime (§6.1)</b> Do <u>not</u> block queries or require the DBMS to restart before an action takes effect.	
<b>Runtime Metrics (§4.2)</b> Include meta-data tags about metrics' corresponding DBMS sub-system and hardware resource. Support variable metric collections rates for different parts of the DBMS. Expose metrics for tunable sub-components. Always use the same unit of measurement for related metrics.		<b>Dependencies (§5.2)</b> Do <u>not</u> use special knob values to indicate whether a feature is enabled/disabled. Do <u>not</u> allow effects of an action to be implicitly controlled by another action. Support explicit reversal actions that undo the effects of another action.		<b>No Self-Managed Components (§6.2)</b> Do <u>not</u> include separate sub-systems that are automatically managed separately from the DBMS.	
<b>Hardware Capabilities (§4.3)</b> Periodically measure hardware performance and align with collected metrics time-series data. Collect training data on different hardware configurations in elastic environments.		<b>Deployment History (§5.3)</b> Maintain a log of every action deployment to track outcomes over time.		<b>Observable Deployment Costs (§6.3)</b> Only deploy one action at a time. Provide a notification API to push alerts when an action deployment starts and stops.	
				<b>Aborted Actions (§6.4)</b> Reject actions that will put DBMS in an invalid state before starting their deployment. Do <u>not</u> allow actions to cause unexpected behaviors that are observable by clients.	
				<b>Adjustable Deployment Resources (§6.5)</b> Support variable resource limits for actions.	

**Table 2: Design Principles Summary** – An overview of the design principles categories to support self-driving operation in a DBMS.

amenable to autonomous planning and operation. Table 2 provides a summary of these principles that we present in Sections 4 to 6.

There are two overarching themes in our discussion. Foremost is on how to expose useful information about the DBMS. This includes both the application's workload and metrics about the DBMS's internals, as well as how to control its behavior. The next issue is related to how the DBMS deploys actions. Ideally each action completes quickly and with little impact on performance, but this is not always possible. Both of these issues are important for enabling a DBMS to manage itself without wild swings in performance. Achieving this is difficult because DBMS deployments are fluid and the solution space for tuning them is vast. Hence, these design principles are often ways to reduce the number of choices that the DBMS has to consider, as well as to make it easier for it to explore configurations and learn about its environment.

## 4 ENVIRONMENT OBSERVATIONS

Most DBMSs are good at collecting information about their environment, such as the workload (Section 4.1) and internal runtime metrics (Section 4.2). The problem is that there is an overabundance of data that makes it difficult to separate signals from the noise [70]. Furthermore, many DBMSs also do not expose information about their hardware to enable reusing training data across operating environments (Section 4.3). A self-driving DBMS must maintain a history of this information instead of only keeping the latest snapshot; retaining older data provides context about the previous state of the system that it uses for learning. We discuss how to remedy these issues in this section.

### 4.1 Workload History

A self-driving DBMS selects actions based on what its forecast models predict the system will need in the future. The challenge with such forecasting is that the environment is dynamic (i.e., the workload, database physical design, and knob configuration change over time). Even if these factors are static, the size of the database could also grow or shrink. This variability means that forecasting low-level metrics during query execution, such as CPU utilization

or tuples read/written, will lead to unstable models as these will change as the DBMS evolves its configuration.

A better approach is to predict the arrival rate of queries and then extrapolate their expected resource utilization [43]. To do this, a self-driving DBMS records the *workload history* of the transactions and queries that it executes. Each entry in the history contains the logical operation invoked (e.g., SQL) along with its execution context. This context includes client meta-data, query plan hints, and session-specific configuration knobs (Section 5.1). The system also tracks each query's completion status, such as its execution time and whether its parent transaction aborts. Recording this additional information ensures that the DBMS can properly approximate how the queries execute in its predictions.

The naïve approach to collect this history is for the DBMS to track every query that it executes. Such a strategy, however, will consume a sizable amount of storage space for large applications and incur significant runtime overhead. Furthermore, many of the entries of the history will be redundant (e.g., the same query with different constants), which may lead to overfitting the ML models. The two approaches for reducing this overhead are (1) sampling and (2) aggregation. With the former, the DBMS should sample at the transaction-level to ensure every query in a transaction is included in the history.

The other approach for reducing the history is to aggregate the traces. One common technique is to extract the logical structure of the SQL query by removing the literals and then combine queries with the same logical structure. There are also more advanced techniques beyond logical structure extraction. For example, Microsoft proposed compressing a workload by searching for redundant SQL statements with an application-specific distance function [13]. An alternative approach that is explicitly designed for self-driving DBMSs is to compress the workload using the similarities between the temporal patterns of queries [43].

### 4.2 Runtime Metrics

A DBMS's *metrics* are performance counters that record the activities of its internal runtime components. Engineers add these metrics

to enable DBAs and monitoring tools to observe the system's behavior and diagnose problems. The DBMSs also use them internally to trigger maintenance operations, such as garbage collection in MVCC systems and compaction in LSM systems. A self-driving DBMS trains models from metrics that estimate the cost/benefit of actions under varying conditions. Metrics also guide the system to propose/prune candidate actions.

There are two categories of metrics. The first are *accumulating* metrics that count the number of events that have occurred since some point of time. For example, the DBMS can record the number of pages read from disk since it started. The other category are *aggregation* metrics that record the average number of events over a time window. As we now describe, exposing metrics needs careful consideration so that ML algorithms have the proper data:

**Meta-Data Tags:** Each metric should record what sub-system it collects data from in a structured meta-data tag. They should match action tags that modify those sub-systems. Similarly, if a metric measures some aspect of a hardware resource (e.g., CPU utilization, disk I/O), then it should also include a tag that records this information. These tags serve two purposes. First, they remove the need for the DBMS to train separate models to identify which metrics are affected by which actions. The DBMS still has to learn how actions affect these metrics and how they impact the objective function, as these will be different per workload and can change over time. The second benefit of tagging is that the system can use them to produce human-understandable explanations for its decisions. Such information is important for instilling confidence in DBAs that a self-driving DBMS is operating correctly.

**Variable Fidelity:** Related to meta-data tags is the ability for the DBMS to dynamically adjust the frequency at which it collects metric data. The DBMS may want to temporarily enable fine-grained metric data for one part of the DBMS without having to enable it for the entire system. Such increased fidelity does not mean that the DBMS enables data collection for counters that normally are dormant. Rather the DBMS collects more samples of its fixed set of metrics. Increasing the number of samples produces more data for parts of the system where it has low confidence in its models or where it is exhibiting unexpected behavior as it deploys actions.

**Sub-Components:** Some DBMSs support separate configurations for individual sub-components. This can be either changes to database objects (e.g., tables, indexes) or knobs that dynamically create new dependent knobs. For the former, some DBMSs support customized storage policies for each table. As an example of dynamic knobs, one can create multiple buffer pools in IBM DB2, each of which the DBA can tune separately. It is imperative to expose metrics about each individual component. Otherwise it is more difficult for the DBMS to infer how an action has affected that component. It is possible for algorithms to infer the effect a sub-component has on the DBMS's objective function, but it has to hold all other variables constant (e.g., workload, hardware resources).

### 4.3 Hardware Capabilities

Including hardware profiles with a DBMS's metrics enables the ML components to reuse training data across deployments [77].

Without this, the system has to collect a new training corpus for each new deployment or upgrade even if it is the same application. If the DBMS only has the old training data without the hardware context, then its models are unable to distinguish whether unexpected changes in performance (even if they are positive) are due to its actions or because the hardware is different. This is important in cloud environments where the performance differences between instance types are not uniform for all resources. For example, Amazon provides large memory instances that use older CPUs with fewer cores compared to other instances with less memory but newer CPUs with more cores. If queries are slower after migrating the database to this larger memory instance, then the DBMS must determine whether this is because of a recently applied action or because the CPU is slower.

A hardware profile is a combination of its specifications and measured performance. For example, the CPU's specification would include the number of cores. The performance measurements are synthetic benchmarks that target each hardware resource (e.g., fio for disk, bogomips for CPU). Some DBMSs already use microbenchmarks for enhancing the query optimizer's internal cost model [40]. The system will want to run these periodically since performance varies over time, especially in cloud environments. It will also categorize the measurements for each hardware component using the same tags as the metrics described above.

One additional challenge is how to support resource scaling in elastic environments where the capabilities of new machines are unknown. For example, if the DBMS wants to migrate the database to a faster machine, then it needs to estimate how much it will affect the objective function before it decides whether to move. To do this, the system's planning components need training data about the available hardware. This is again where the metrics' meta-data tags are useful: since hardware changes are multi-dimensional (e.g., memory, CPU, disk), the tags enable the system to identify what resource(s) are under/over-utilized so that it can select the proper resource to shrink/expand. This is a good example of how cloud-based DBaaS vendors will have an advantage over organizations that run a self-driving DBMS themselves, since vendors have access to more deployments on a myriad of configurations.

## 5 ACTION META-DATA

In this section, we discuss the importance of exposing meta-data about actions. One could implement this information by hard-coding the meta-data into the tuning algorithms. We contend, however, that it is better software engineering to maintain this information as first-class concepts along with the actions inside of the DBMS. This ensures that the meta-data (more) accurately reflects the DBMS's internals, since the same developers implementing the actions will also provide the meta-data.

### 5.1 Configuration Knobs

A DBMS's configuration knobs control aspects of its runtime operations. The three categories of knobs are (1) *resources*, (2) *policies*, and (3) *locations*. Knobs in the first category specify how much of a resource the system uses for a task. These can be either for fixed components (e.g., the number of garbage collection threads) or for dynamic activities (e.g., the amount of memory to use per

query). Policy configuration knobs control how the DBMS behaves for certain tasks. For example, a knob can control whether or not the DBMS flushes the write-ahead log to disk when a transaction commits. Lastly, the location knobs specify where the DBMS finds resources that it needs (e.g., file paths) and how it interacts with the outside world (e.g., network port number).

We now describe the knob meta-data that a DBMS must expose for its ML algorithms.

**Untunable Knobs:** Any knob that requires human knowledge to make a value judgment about the right decision should not be exposed to the autonomous components. They should be clearly marked as untunable so that the system does not modify them. There are obvious cases, like location knobs that define file paths, where the system will not function if they are set incorrectly.

Setting other policy knobs incorrectly may not cause the system to become inoperable, but more subtly affect the database's correctness or safety. The most common example of this that we found is whether to require the DBMS to flush a transaction's log records to durable disk before it is committed. If the ML algorithms discover that changing this knob improves the objective function, then they will likely make that change. But this means that the DBMS could potentially lose data for recently committed transactions if there is a crash. Such a trade-off may be appropriate for some applications where losing the last few milliseconds of transactions is not a problem (e.g., recording website clicks), but for other applications such as a loss is impermissible (e.g., financial transactions). The DBMS is unable to know what the right choice is for an application because it requires a human to decide what is allowed in their organization.

**Value Ranges:** Constrained value ranges for knobs (min/max for numerical values or enums for categorical values) are crucial to a safe automated tool. Without this information, the algorithm can crash the system outright through incorrect values. Even worse, the system could continue to operate under the assumption that the previous action was successful and is responsible for the current database state, leading to byzantine failures. Having the DBMS expose this information obviates the need for developers to hard-code this information into the ML algorithms.

**Scope:** When the DBMS changes a knob, every client should see that change immediately or soon after (e.g., the next query). We are not aware of any knob change that should not be immediately observable by all sessions, even if that client is in the middle of executing a transaction. To the best of our knowledge, this has no adverse effects on the training data that the system collects.

Some DBMSs also allow clients to override global values for knobs and configure them for their session. For these knobs, the DBMS should denote in its catalog whether a knob supports per-session changes. Supporting automatic tuning for per-session knobs increases the complexity of the problem because it requires the DBMS to construct additional models that (1) predict the execution patterns of clients and (2) classify new connections based on their behavior. Hence, we believe that the first self-driving DBMSs are unlikely to provide per-session tuning.

**Tuning Deltas:** System developers design most knobs to allow a DBA to set them to arbitrary values. For example, one can set the

value for the knob that controls a DBMS's buffer pool size to any number of bytes between some lower bound and the total amount of memory on the machine. This flexibility comes at a cost in that it increases the solution space for each knob. When there is such a large number of choices for a single knob, then there will be a large range of values where their effect on the objective function is unknown. This means that the ML components need more training data before they are able to converge to good configurations. In the case of the knob that controls the buffer pool's memory size, the DBMS may already have data about the system's behavior when the size is 1.0 GB, but not when it is set to 1.01 GB. There is unlikely to be a difference in performance between these two settings.

To help reduce the complexity of this solution space, a self-driving DBMS should provide hints on how to increment a knob for given ranges. That is, instead of the DBMS deploying an action that sets an exact value for a knob, the action increments or decrements the knob by a fixed amount. An additional advantage of using deltas is they potentially provide smoother transition between configurations and avoid large performance oscillations.

To further reduce the number of the DBMS's potential configuration states, these delta actions should also vary the magnitude of the change based on a knob's current value. For example, for buffer pool sizes less than 1 GB the delta amount is 10 MB, whereas the delta amount is 1 GB for sizes less than 1 TB. The insight here is that the difference between smaller deltas are greater on machines with small amounts of RAM, whereas the impact is trivial for large memory machines.

**Meta-Data Tags:** Each knob should include meta-data about what part of the system it affects. The planning components can pair these tags with the metric tags (Section 4.2) to provide hints about which actions to prioritize given the system's current state.

## 5.2 Dependencies

The DBMS should also track how actions interact with each other. Such dependencies exist when one action enables/disables a feature in the system while another action controls that feature's behavior. Unless the DBMS deploys the first action, the second action will have no effect. This dependency information enables the DBMS to avoid selecting an action that modifies the dependent component unless their parent is set. Again, this reduces the number of actions that the DBMS has to consider. As we describe below, there are some dependencies that make it difficult to model how the system reacts to changes.

**No Special Knob Values:** Tracking dependencies removes the need for ad-hoc handling of knobs that use a special value (e.g., 0, -1) to do something different than what the knob normally does, such as disabling a feature. The DBMS instead should provide a separate boolean knob to control it. This knob is marked as untunable and then the system only examines the dependent knobs if the first knob is enabled.

There are other knobs with special values that do not affect correctness, but rather make modeling the DBMS's behavior more difficult. For example, a special value of "0" may allow for the DBMS to assign its own value for the knob, either to some default or expected optimal value. The planning components may never

explore such a setting for this knob (even though it may be optimal) because its models will predict that performance degrades as the knob's value converges to zero. Or they will think that a lower value is better even though the opposite is true, which drives the optimization search in the wrong direction. We again contend that a self-driving DBMS should never use such knobs.

**No Hidden Dependencies:** Similar to special values, actions should not have hidden dependencies. That is, the behavior of one part of the DBMS should not depend on the configuration of another subsystem. Postgres provides an illustrative example of this problem. The DBMS provides a knob that controls the maximum amount of memory that the system's garbage collector is allowed to use<sup>1</sup>. If the action sets this knob to a special value (i.e., -1), then the actual knob value at runtime is the value of another knob<sup>2</sup>. One only discovers their implicit dependency from reading the documentation.

In addition to removing the special value for the knob in this example, it is better that the two knobs are decoupled. Again, such dependencies greatly increase the complexity of modeling the system's behavior under different conditions because the planning components must consider two operations modes (i.e., one where they are linked, and one where they are separate). If there are truly scenarios where the two knobs should be equivalent, then the system will discover that on its own with enough training data.

**Dynamic Actions:** More complex dependencies are (1) actions that enable other actions or (2) actions that the DBMS only exposes when it creates an object (e.g., table-specific knobs). The DBMS must provide meta-data that indicates what knobs are created or enabled by an action's deployment.

The easiest dynamic actions to support in a self-driving DBMS are "reversal" actions that undo the modifications of a previously deployed action. All actions except for knobs and resource actions need to have reversal actions. For example, the reversal of a physical design action that creates an index is to drop that index. Knob configuration and resource scaling actions do not need them if their changes are defined as deltas rather than discrete values.

To understand the issues with non-reversal dynamic actions, consider the example of an action that adds an index to a table. After the DBMS deploys this action, then it exposes (1) actions to modify that index's knobs and (2) actions that install hints in query plans to force them to use that index. One could refactor the latter actions into a generic form to remove this dependency. That is, rather than the action installing hints for a specific index, it instead chooses the most recent index created. This is likely the better approach, since which query plans to target will evolve over time. But dealing with the first group of actions that tune a new system component or database object is more challenging.

### 5.3 Deployment History

The DBMS needs to maintain the history of each action deployment. This record is more than the timestamps of when the DBMS starts/finishes an action. It also includes internal metrics from the DBMS's sub-systems, as well as a representation of the DBMS's state at the moment of the deployment. This state is the DBMS's

knobs, physical design, hardware, query plan hints, in addition to an succinct encoding of the current workload and database contents.

The deployment history provides several benefits. Foremost is that it provides a log to help the planning algorithms reason about how the system may have reached its current state. This is important for initiating actions in response to performance degradation or SLO violations. Lastly, the planning components can extrapolate a human-readable explanation of why the action was chosen from this encoding. This improves transparency by allowing DBAs to examine the system's reasoning and behavior.

## 6 ACTION ENGINEERING

A self-driving DBMS uses the environment data and action meta-data discussed in the previous sections to select actions. There are two requirements that the DBMS must ensure when deploying actions. Foremost is to minimize the cost of deploying an action on the objective function. This means no downtime, but also no wild oscillations in performance. Lastly, actions should not cause applications to observe any effects of an action's deployment other than changes in the DBMS's objective function. Such effects include incorrect query results, lost/corrupted data, and client disconnections.

We now discuss how the DBMS can support efficient and informative action deployments.

### 6.1 No Downtime

The most important thing for a self-driving DBMS is the ability to apply changes without periods of unavailability. This downtime could be either blocking query execution until an action completes (e.g., halting queries from accessing a table while the DBMS builds a index) or having to restart the DBMS before a change can take effect. Taking the system offline to apply changes makes planning more difficult because a human has to instruct the DBMS when it is allowed to restart itself.

We are not aware of any system modification that a self-driving DBMS should be allowed to automatically tune that requires such downtime. We believe that this limitation is entirely due to engineering factors and not some fundamental scientific reason. Certainly there are heavy-weight modifications that degrade performance during deployment (e.g., compaction), but the system can factor this in its estimates.

If the DBMS is unable to deploy actions without downtime, then it must include the expected downtime in their cost estimates. It will also need to distinguish between blocking downtime versus restarting. Restarting the DBMS is particularly pernicious because some actions may require additional work on restart, which makes it unavailable for longer periods. These restart times can also depend on the current DBMS's configuration. For example, suppose one is changing MySQL log file size<sup>3</sup> to 5 GB. If the previous setting for this knob was 10 GB, then the DBMS will compact the log file upon restart. But if the previous setting was 1 GB, then the DBMS does not need to do anything. If restarting is unavoidable, then the DBMS could provide hints about its expected recovery time based on what work it needs to perform and its hardware capabilities.

<sup>1</sup>PostgreSQL Knob – AUTOVACUUM-WORK-MEM

<sup>2</sup>PostgreSQL Knob – MAINTENANCE-WORK-MEM

<sup>3</sup>MySQL Knob – INNODB-LOG-FILE-SIZE

Another problem with downtime is that the DBMS may also have to ask a human for permission to restart the system to avoid downtime during peak hours. This complicates planning because the algorithms must include the date and time in its estimates to decide whether it is allowed to choose actions that require restarting.

## 6.2 No Self-Managing Components

Some commercial DBMS vendors introduced “self-managing” sub-systems in recent years that operate without the need for humans to tune them. For example, Oracle provides a memory allocator [19] that uses heuristics to automatically determine the memory allocations for its components (e.g., buffer pools, query caches). Some cloud-based DBaaS vendors, like Microsoft SQL Azure, provide tools that automatically install table indexes [22]. Each of these sub-systems maintain their own models and then independently decide when to deploy an action.

The issue with separate self-managing sub-systems is that they introduce externalities that are difficult to capture in the ML models if the system is trying to do holistic planning. Consider the scenario of Oracle’s self-managing memory feature. Suppose the allocator initially assigns a small amount of memory for query result caching and a large amount to the buffer pool. The DBMS then chooses to build an index because memory pressure in the buffer pool is low. But then the allocator decides on its own to increase the result cache size and decrease the buffer pool size. With this change, there is now less memory available to store the index and data pages, thereby increasing the amount of disk I/O during query execution. Thus, the index that the DBMS just added is now a bad choice because of another change in the system that it does not control. To avoid this problem, the DBMS has to include the possibility of allocator configuration changes in its models, which increases their dimensionality and complexity.

## 6.3 Observable Deployment Costs

In addition to estimating how an action can improve the objective function if it is deployed, the DBMS’s behavior models also estimate the cost of deploying each action [44]. This cost includes (1) the amount of resources that the DBMS uses during deployment (e.g., CPU, memory), (2) changes to the objective function, and (3) the estimated elapsed time for the deployment. The system assigns each action invocation a unique identifier that allows it to track what caused its configuration to change. Every sub-system participating in deploying an action records the amount of physical resources that it uses during the operation. The DBMS then aggregates this information together after the action completes and stores it in its history. The accumulated resource usage is a combination of DBMS (e.g., latches held) or OS (e.g., CPU wait time) measurements. Since the DBMS does not execute actions as frequently as queries, the overhead from collecting this data for every action is negligible.

The DBMS should only deploy one action at a time so that the training data accurately reflects their execution costs. It is unknown, however, how to automatically determine the best time that the DBMS should wait before deploying the next action. With longer waits, the more confident it can be about the current configuration. The shorter the wait, the harder it is to identify what action is causing a problem.

The DBMS must also provide a notification API for the planning components to identify when an action deployment starts and finishes. Without this, it is more difficult to determine whether a performance degradation is due to the cost of deploying the action (e.g., resizing a log file on disk) or because it was a bad choice.

## 6.4 Aborted Actions

Two overarching principles for handling aborted actions is that the DBMS (1) cannot allow partial actions and (2) cannot retain observation data from its deployment. An example of the former is an action that reduces the amount of memory allocated for a sub-system by 100 MB, but for whatever reason the system can only reduce it by 50 MB. The DBMS cannot allow this because then the internal models will not accurately reflect the state of the system [20]. The partial action might also be equivalent to another action, but then the system would incorrectly attribute the training data collected to the original action. Similarly, the DBMS must discard data collected during rejected and failed actions. This is because it is difficult to track how far along the DBMS’s sub-systems may gotten in the deployment before the failure.

We next discuss how the DBMS should handle scenarios that cause it to abort an action during its deployment.

**Rejections:** The DBMS may attempt to deploy an action that it is unable to complete. For example, suppose that the DBMS’s configuration assign one thread to its background maintenance pool (e.g., garbage collection). The system then choose an action that decrements the number of threads in this pool. This action is invalid because the pool must have at least one thread. The DBMS should therefore reject the action and record that it was not able to comply with the request. Ideally, the DBMS should also provide additional information, such as whether the action was rejected because of a short-term issue (e.g., no idle threads) or whether it is a more permanent problem (e.g., not enough memory).

**Failures:** An action failing to complete is different than a rejected action because the action would otherwise have completed successfully. But just the same, the training data that the DBMS collects during its deployment is “tainted” and thus it must be discarded. Such failures could occur due to logistical issues (e.g., a human drops an index right before the DBMS attempts to drop it) or hardware failures. The latter is more difficult to handle when there are transient failures (e.g., the network goes down for a brief period of time during an action).

## 6.5 Adjustable Deployment Resources

Lastly, the DBMS must support executing the same action with varying resource usage levels. Such resources are based on the system’s hardware components: (1) number of CPU threads, (2) amount of memory, (3) disk bandwidth, and (4) network bandwidth. This allows the DBMS to choose more aggressive deployment strategies based on deadlines or the current load. For example, the DBMS could build an index with only one thread during the day when demand is high to avoid interfering too much with the application’s queries. But at night when the DBMS has more idle cycles, it can use eight threads to construct the index more quickly. This index could target the current workload or part of the system’s preparation



for the next day’s workload. In the case of the latter scenario, the DBMS could also run simulations to determine whether adding that index was the correct decision.

Selecting how much of a resource to let an action use is difficult, as the DBMS must be careful to not violate SLO constraints. Another complexity is that these resource allocations are sometimes the upper bound of how much the DBMS is allowed to use and not what it actually uses. For example, even though an action could run on four threads does not mean that the DBMS uses four during deployment. This makes it difficult to accurately predict total resource usage. Hence, we anticipate that the first self-driving DBMSs will use fixed allocations to simplify this problem.

## 7 RELATED WORK

To the best of our knowledge, there has never been a fully autonomous DBMS. In the early 2000s, there were several groundbreaking solutions that moved towards this goal [14], most notably Microsoft’s SQL Server AutoAdmin [16] and IBM’s DB2 Database Advisor [60, 65, 80]. Others proposed RISC-style DBMS architectures made up of inter-operable components that make it easier to reason about and therefore control the overall system [73]. Teradata’s Active System Management provides tools for DBAs to write rules for automatic admission control. But over a decade later, all of this research has been relegated to standalone tools that assist DBAs and not supplant them.

The closest attempt to a fully automated DBMS was IBM’s proof-of-concept for DB2 from 2008 [74]. This system used existing tools [37] in an external controller and monitor that triggered a change whenever a resource threshold was surpassed (e.g., the number of deadlocks). This prototype still required a human DBA to select tuning optimizations and to occasionally restart the DBMS. And unlike self-driving DBMSs, it could only react to problems after they occur because the system lacked forecasting.

Automation is more common in cloud computing platforms because of their scale and complexity [21, 33]. Microsoft appears to be again leading research in this area. Their Azure service models resource utilization of DBMS containers from internal telemetry data and automatically adjusts allocations to meet QoS and budget constraints [23]. There are also controllers for applications to perform black box provisioning in the cloud [3, 11, 61, 62]. Oracle announced their own “self-driving” DBaaS in 2017 [2]; although there is little public information about its implementation, our understanding from their developers is that they are running their previous tuning tools in a managed environment.

More recently, there have been attempts to use deep learning [72] combined with reinforcement learning (RL) to assist with specific parts of the DBMS’s runtime architecture. For example, researchers have proposed RL models for query optimization [45, 46, 53], index tuning [9, 63], cardinality estimation [32, 41], partitioning [27], and general tuning choices [71]. Other forms of deep learning have also been applied to cardinality estimation [35] and query performance prediction [47]. Alternative models include graph embeddings [78] and analytical models [50]. These approaches are each limited to a single decision problem in a static operating environment and do not consider long-term forecasts in their models.

A line of research in the software engineering community aims to develop general purpose frameworks that enable self-adaptation capabilities in arbitrary software [17, 24, 34]. Such frameworks act as external controllers of a target system by executing actions using information that it collects with probes. One of the most influential projects is the Rainbow framework [28]. It supports the reuse of adaptation strategies and infrastructure across systems by the separation of a generic adaptation infrastructure from system-specific knowledge. Software developers use a language, called Stitch [18], to represent the adaptive strategies and express the business objectives. The framework then selects a strategy that has optimal utility in a given context. There are also works on proactive self-adaptation [49], self-protection in antagonistic environments [75], and maintaining robustness [10]. The goal of such frameworks is to reuse the adaption infrastructure across different software systems. A human, however, still specifies strategies for their systems using their domain knowledge. This includes which actions the system can take under what conditions, and the expected cost/benefit of such actions on multiple dimensions.

## 8 CONCLUSION

Self-driving DBMSs will enable organizations to deploy database applications that are more complex than what is possible today, and at a lower hardware and personnel costs. Achieving full autonomy (i.e., Level 5 from Section 2) has two tracts of research: (1) novel ML approaches for value and policy functions and (2) novel DBMS architectures that are amenable to autonomous control. These efforts are symbiotic; one cannot make strides in one without an understanding of the other. Thus, this paper presented design principles based on our experiences in the NoisePage DBMS.

One final point that we would like to make is that we believe that self-driving DBMSs will not supplant DBAs. We instead envision autonomous systems will relieve them from the burdens of arduous low-level tuning and allow them to pursue higher minded tasks, such as database design and development.

## ACKNOWLEDGMENTS

This work was supported (in part) by the National Science Foundation (IIS-1846158, III-1423210, DGE-1252522), Google Research Grants, and the Alfred P. Sloan Research Fellowship program.

VGI tI EtYyXN-YSBcZXRyYXl I ZCBNZQo=

## REFERENCES

- [1] 2021. NoisePage – Database Management System Project. <https://noise.page>.
- [2] 2021. Self-Driving Database | Autonomous Database Oracle 19c. <https://oracle.com/database/autonomous-database/>.
- [3] Sherif Abdelwahed, Nagarajan Kandasamy, and Sandeep Neema. 2004. A Control-based Framework for Self-managing Distributed Computing Systems. In *Proceedings of the 1st ACM SIGSOFT Workshop on Self-managed Systems (WOSS '04)*. 3–7.
- [4] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. 2000. Automated Selection of Materialized Views and Indexes in SQL Databases. In *VLDB*.
- [5] Sanjay Agrawal, Vivek Narasayya, and Beverly Yang. 2004. Integrating Vertical and Horizontal Partitioning into Automated Physical Database Design. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD '04)*. New York, NY, USA, 359–370.
- [6] Dana Van Aken, Dongsheng Yang, Sebastien Brillard, Ari Fiorino, Bohan Zhang, Christian Billian, and Andrew Pavlo. 2021. An Inquiry into Machine Learning-based Automatic Configuration Tuning Services on Real-World Database Management Systems. *Proceedings of the VLDB Endowment* 14, 7 (2021), 1241–1253.

- [7] Ioannis Alagiannis, Stratos Idreos, and Anastasia Ailamaki. 2014. H2O: A Hands-free Adaptive Store (*SIGMOD '14*). 1103–1114.
- [8] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. 2016. Bridging the Archipelago Between Row-Stores and Column-Stores for Hybrid Workloads (*SIGMOD '16*). 583–598.
- [9] Debabrata Basu, Qian Lin, Weidong Chen, Hoang Tam Vo, Zihong Yuan, Pierre Senellart, and Stéphane Bressan. 2015. *Cost-Model Oblivious Database Tuning with Reinforcement Learning*. 253–268.
- [10] Javier Camara, Rogerio de Lemos, Nuno Laranjeiro, Rafael Ventura, and Marco Vieira. 2017. Robustness-Driven Resilience Evaluation of Self-Adaptive Software Systems. *IEEE Transactions on Dependable and Secure Computing* 14, 1 (2017), 50–64.
- [11] Emmanuel Cecchet, Rahul Singh, Upendra Sharma, and Prashant Shenoy. 2011. Dolly: Virtualization-driven Database Provisioning for the Cloud (*VEE '11*). 51–62.
- [12] Stefano Cereda, Stefano Valladares, Paolo Cremonesi, and Stefano Doni. 2021. CGPTuner: a Contextual Gaussian Process Bandit Approach for the Automatic Tuning of IT Configurations Under Varying Workload Conditions. *Proc. VLDB Endow.* 14, 8 (2021), 1401–1413.
- [13] Surajit Chaudhuri, Ashish Kumar Gupta, and Vivek Narasayya. 2002. Compressing SQL Workloads. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD '02)*. 488–499.
- [14] Surajit Chaudhuri and Vivek Narasayya. 2007. Self-tuning database systems: a decade of progress. In *VLDB (Vienna, Austria)*. 3–14.
- [15] Surajit Chaudhuri and Vivek R. Narasayya. 1997. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *VLDB*. 146–155.
- [16] Surajit Chaudhuri and Gerhard Weikum. 2000. Rethinking Database System Architecture: Towards a Self-Tuning RISC-Style Database System. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB '00)*. 1–10.
- [17] Betty H. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, Je-Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Marzò Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Je-Magee, Marin Litoiu, Sam Malek, Ra-aela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. 2009. Software Engineering for Self-Adaptive Systems: A Research Roadmap. In *Software Engineering for Self-Adaptive Systems*. Springer-Verlag, 1–26.
- [18] Shang-Wen Cheng and David Garlan. 2012. Stitch: A language for architecture-based self-adaptation. *Journal of Systems and Software* 85, 12 (2012), 2860–2875.
- [19] Benoît Dageville and Mohamed Zait. 2002. SQL Memory Management in Oracle9i. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB '02)*. 962–973.
- [20] Wenyuan Dai, Qiang Yang, Gui-Rong Xue, and Yong Yu. 2007. Boosting for Transfer Learning. In *Proceedings of the 24th International Conference on Machine Learning (ICML '07)*. 193–200.
- [21] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2013. ElasTraS: An Elastic, Scalable, and Self-managing Transactional Database for the Cloud. *ACM Trans. Database Syst.* 38, 1, Article 5 (April 2013), 5:1–5:45 pages.
- [22] Sudipto Das, Miroslav Grbic, Igor Ilıc, Isidora Jovandic, Andrija Jovanovic, Vivek R. Narasayya, Miodrag Radulovic, Maja Stikic, Gaoxiang Xu, and Surajit Chaudhuri. 2019. Automatically Indexing Millions of Databases in Microsoft Azure SQL Database. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. 1–14.
- [23] Sudipto Das, Feng Li, Vivek R. Narasayya, and Arnd Christian König. 2016. Automated Demand-Driven Resource Scaling in Relational Database-as-a-Service. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. 1923–1934.
- [24] Rogério de Lemos, Holger Giese, Hausi A. Müller, Mary Shaw, Jesper Andersson, Marin Litoiu, Bradley Schmerl, Gabriel Tamura, Norha M Villegas, Thomas Vogel, et al. 2013. Software engineering for self-adaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems II*. Springer, 1–32.
- [25] Karl Dias, Mark Ramacher, Uri Shaft, Venkateshwaran Venkataramani, and Graham Wood. 2005. Automatic Performance Diagnosis and Tuning in Oracle. In *CIDR*.
- [26] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. 2009. Tuning database configuration parameters with iTuned. *VLDB* 2 (August 2009), 1246–1257.
- [27] Gabriel Campero Durand, Marcus Pinnecke, Rufat Piriyev, Mahmoud Mohsen, David Broneske, Gunter Saake, Maya S. Sekeran, Fabián Rodríguez, and Laxmi Balami. 2018. GridFormation: Towards Self-Driven Online Data Partitioning Using Reinforcement Learning (*aiDM'18*). Article 1, 7 pages.
- [28] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. 2004. Rainbow: Architecture-based self adaptation with reusable infrastructure. *Computer Science Department* (2004), 668.
- [29] Himanshu Gupta, Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. 1997. Index Selection for OLAP. In *ICDE*. 208–219.
- [30] Michael Hammer and Arvola Chan. 1976. Index selection in a self-adaptive database management system. In *SIGMOD*. 1–8.
- [31] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2007. Database Cracking. In *CIDR*. 68–78.
- [32] Oleg Ivanov and Sergey Bartunov. 2017. Adaptive Cardinality Estimation. *CoRR* abs/1711.08330 (2017). <http://arxiv.org/abs/1711.08330>
- [33] Jeffrey O. Kephart. 2005. Research Challenges of Autonomic Computing (*ICSE '05*). 15–22.
- [34] Jeffrey O. Kephart and David M. Chess. 2003. The vision of autonomic computing. *Computer* 1 (2003), 41–50.
- [35] Andreas Kipf, Dimitri Vorona, Jonas Müller, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, Thomas Neumann, and Alfons Kemper. 2019. Estimating cardinalities with deep sketches. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. 1937–1940.
- [36] Sushil Kumar. 2003. Oracle Database 10g: The Self-Managing Database. White Paper.
- [37] Eva Kwan, Sam Lightstone, Adam Storm, and Leanne Wu. 2002. *Automatic Configuration for IBM DB2 Universal Database*. Technical Report. IBM.
- [38] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. Qtune: A query-aware database tuning system with deep reinforcement learning. In *VLDB*. 2118–2130.
- [39] Shih-Chieh Lin, Yunqi Zhang, Chang-Hong Hsu, Matt Skach, Md E. Haque, Lingjia Tang, and Jason Mars. 2018. The Architectural Implications of Autonomous Driving: Constraints and Acceleration. In *ASPLOS*. 751–766.
- [40] Sherry Listgarten and Marie-Anne Neimat. 1996. Modelling Costs for a MM-DBMS. In *Proceedings of the International Workshop on Real-Time Databases, Issues and Applications (RTDB)*. 72–78.
- [41] Henry Liu, Mingbin Xu, Ziting Yu, Vincent Corvinelli, and Calisto Zuzarte. 2015. Cardinality Estimation Using Neural Networks (*ASCASCON '15*). 53–59.
- [42] Lin Ma, Bailu Ding, Sudipto Das, and Adith Swaminathan. 2020. Active learning for ML enhanced database systems. In *Proceedings of the 2020 International Conference on Management of Data (SIGMOD '20)*. 175–191.
- [43] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J. Gordon. 2018. Query-based Workload Forecasting for Self-Driving Database Management Systems. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. 631–645.
- [44] Lin Ma, William Zhang, Jie Jiao, Wuwen Wang, Matthew Butrovich, Wan Shen Lim, Prashanth Menon, and Andrew Pavlo. 2021. MB2: Decomposed Behavior Modeling for Self-Driving Database Management Systems. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21)*. 1248–1261.
- [45] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making Learned Query Optimization Practical. In *SIGMOD '21: International Conference on Management of Data*. 1275–1288.
- [46] Ryan Marcus and Olga Papaemmanouil. 2018. Deep Reinforcement Learning for Join Order Enumeration. In *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management (aiDM'18)*. Article 3, 4 pages.
- [47] Ryan Marcus and Olga Papaemmanouil. 2019. Plan-Structured Deep Neural Network Models for Query Performance Prediction. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1733–1746.
- [48] Jacob Mattingley, Yang Wang, and Stephen Boyd. 2011. Receding horizon control. *IEEE Control Systems Magazine* 31, 3 (2011), 52–65.
- [49] Gabriel A. Moreno, Javier Cizmar, David Garlan, and Bradley Schmerl. 2015. Proactive self-adaptation under uncertainty: a probabilistic model checking approach. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*. ACM, 1–12.
- [50] Barzan Mozafari, Carlo Curino, Alekh Jindal, and Samuel Madden. 2013. Performance and resource modeling in highly-concurrent OLTP workloads. In *Proceedings of the 2013 International Conference on Management of Data (SIGMOD '13)*. ACM, 301–312.
- [51] Dushyanth Narayanan, Eno Thereska, and Anastasia Ailamaki. 2005. Continuous Resource Monitoring for Self-predicting DBMS. In *MASCOTS*. 239–248.
- [52] Rimma Nehme and Nicolas Bruno. 2011. Automated partitioning design in parallel database systems. In *SIGMOD (SIGMOD)*. 1137–1148.
- [53] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S. Sathiya Keerthi. 2018. Learning State Representations for Query Optimization with Deep Reinforcement Learning (*DEEM'18*). Article 4, 4 pages.
- [54] Brian Paden, Michal Cizmar, Sze Zheng Yong, Dmitry S. Yershov, and Emilio Frazzoli. 2016. A Survey of Motion Planning and Control Techniques for Self-Driving Urban Vehicles. *IEEE Trans. on Intelligent Vehicles* (2016), 33–55.
- [55] Stratos Papadomanolakis and Anastasia Ailamaki. 2004. Autopart: Automating schema design for large scientific databases using data partitioning. In *SSDBM*.
- [56] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C. Mowry, Matthew Perron, Ian Quah, Siddharth Santurkar, Anthony Tomasic, Skye Toor, Dana Van Aken, Ziqi Wang, Yingjun Wu, Ran Xian, and Tieying Zhang. 2017. Self-Driving Database Management Systems. In *Conference on Innovative Data Systems Research (CIDR '17)*.
- [57] Andrew Pavlo, Matthew Butrovich, Ananya Joshi, Lin Ma, Prashanth Menon, Dana Van Aken, Lisa Lee, and Ruslan Salakhutdinov. 2019. External vs. Internal: An Essay on Machine Learning Agents for Autonomous Database Management Systems. *IEEE Data Engineering Bulletin* (2019), 32–46.

- [58] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. 2012. Skew-Aware Automatic Database Partitioning in Shared-Nothing, Parallel OLTP Systems. In *SIGMOD*.
- [59] Andrew Pavlo, Evan P.C. Jones, and Stan Zdonik. 2011. On Predictive Modeling for Optimizing Transaction Execution in Parallel OLTP Systems. *Proceedings of the VLDB Endowment* 5 (October 2011), 85–96. Issue 2.
- [60] Jun Rao, Chun Zhang, Nimrod Megiddo, and Guy Lohman. 2002. Automating physical database design in a parallel database. In *SIGMOD*. 558–569.
- [61] J. Rogers, O. Papaemmanouil, and U. Cetintemel. 2010. A generic auto-provisioning framework for cloud databases. In *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*. 63–68.
- [62] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. 2011. Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting. In *Proceedings of the 2011 IEEE 4th International Conference on Cloud Computing (CLOUD '11)*. 500–507.
- [63] Ankur Sharma, Felix Martin Schuhknecht, and Jens Dittrich. 2018. The Case for Automatic Database Administration using Deep Reinforcement Learning. *CoRR* abs/1801.05643 (2018). <http://arxiv.org/abs/1801.05643>
- [64] David Silver, Aja Huang, Chris J. Maddison, et al. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529, 7587 (2016), 484–489.
- [65] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. 2001. LEO - DB2's LEarning Optimizer. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01)*. 19–28.
- [66] Adam J. Storm, Christian Garcia-Arellano, Sam S. Lightstone, Yixin Diao, and M. Surendra. 2006. Adaptive Self-tuning Memory in DB2. In *VLDB*. 1081–1092.
- [67] David G. Sullivan, Margo I. Seltzer, and Avi Pfeffer. 2004. Using Probabilistic Reasoning to Automate Software Tuning (*SIGMETRICS*). 404–405.
- [68] Wenhui Tian, Pat Martin, and Wendy Powley. 2003. Techniques for Automatically Sizing Multiple Buffer Pools in DB2. In *CASCON*. 294–302.
- [69] G. Valentin, M. Zuliani, D.C. Zilio, G. Lohman, and A. Skelley. 2000. DB2 advisor: an optimizer smart enough to recommend its own indexes. In *ICDE*. 101–110.
- [70] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. 1009–1024.
- [71] Junxiong Wang, Immanuel Trummer, and Debabrota Basu. 2021. Demonstrating UDO: A Unified Approach for Optimizing Transaction Code, Physical Design, and System Parameters via Reinforcement Learning. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21)*. 2794–2797.
- [72] Wei Wang, Meihui Zhang, Gang Chen, H. V. Jagadish, Beng Chin Ooi, and Kian-Lee Tan. 2016. Database Meets Deep Learning: Challenges and Opportunities. *SIGMOD Rec.* 45, 2 (Sept. 2016), 17–22.
- [73] Gerhard Weikum, Axel Moenkeberg, Christof Hasse, and Peter Zabback. 2002. Self-tuning Database Technology and Information Services: From Wishful Thinking to Viable Engineering. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB '02)*. 20–31.
- [74] David Wiese, Gennadi Rabinovitch, Michael Reichert, and Stephan Arenswald. 2008. Autonomic Tuning Expert: A Framework for Best-practice Oriented Autonomic Database Tuning (*CASCON '08*). 27–41.
- [75] Eric Yuan, Sam Malek, Bradley Schmerl, David Garlan, and Jeannette Gennari. 2013. Architecture-based self-protecting software systems. In *Proceedings of the 9th international ACM Sigsoft conference on Quality of software architectures*. ACM, 33–42.
- [76] Bohan Zhang, Dana Van Aken, Justin Wang, Tao Dai, Shuli Jiang, Jacky Lao, Siyuan Sheng, Andrew Pavlo, and Geoffrey J. Gordon. 2018. A Demonstration of the OtterTune Automatic Database Management System Tuning Service. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1910–1913.
- [77] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jia Shu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. 2019. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. ACM, 415–432.
- [78] Xuanhe Zhou, Ji Sun, Guoliang Li, and Jianhua Feng. 2020. Query Performance Prediction for Concurrent Queries using Graph Embedding. *Proceedings of the VLDB Endowment* 13, 9 (2020), 1416–1428.
- [79] Daniel C. Zilio, Anant Jhingran, and Sriram Padmanabhan. 1994. *Partitioning Key Selection for Shared-Nothing Parallel Database System*. Technical Report 87739. IBM Research.
- [80] Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy M. Lohman, Adam J. Storm, Christian Garcia-Arellano, and Scott Fadden. 2004. DB2 Design Advisor: Integrated Automatic Physical Database Design. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases*. 1087–1097.