Self-Driving Database Management Systems

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, Tieying Zhang
Carnegie Mellon University, *National University of Singapore

ABSTRACT

In the last two decades, both researchers and vendors have built advisory tools to assist database administrators (DBAs) in various aspects of system tuning and physical design. Most of this previous work, however, is incomplete because they still require humans to make the final decisions about any changes to the database and are reactionary measures that fix problems after they occur.

What is needed for a truly "self-driving" database management system (DBMS) is a new architecture that is designed for autonomous operation. This is different than earlier attempts because all aspects of the system are controlled by an integrated planning component that not only optimizes the system for the current workload, but also predicts future workload trends so that the system can prepare itself accordingly. With this, the DBMS can support all of the previous tuning techniques without requiring a human to determine the right way and proper time to deploy them. It also enables new optimizations that are important for modern high-performance DBMSs, but which are not possible today because the complexity of managing these systems has surpassed the abilities of human experts.

This paper presents the architecture of Peloton, the first self-driving DBMS. Peloton's autonomic capabilities are now possible due to algorithmic advancements in deep learning, as well as improvements in hardware and adaptive database architectures.

1. INTRODUCTION

The idea of using a DBMS to remove the burden of data management was one of the original selling points of the relational model and declarative query languages from the 1970s [31]. With this approach, a developer only writes a query that specifies what data they want to access. The DBMS then finds the most efficient way to store and retrieve data, and to safely interleave operations.

Over four decades later, DBMSs are now the critical part of every data-intensive application in all facets of society, business, and science. These systems are also more complicated now with a long and growing list of functionalities. But using existing automated tuning tools is an onerous task, as they require laborious preparation of workload samples, spare hardware to test proposed updates, and above all else intuition into the DBMS's internals. If the DBMS could do these things automatically, then it would remove many of the complications and costs involved with deploying a database [40].

8th Biennial Conference on Innovative Data Systems Research (CIDR '17) January 8-11, 2017 , Chaminade, California, USA.

Much of the previous work on self-tuning systems is focused on standalone tools that target only a single aspect of the database. For example, some tools are able to choose the best logical or physical design of a database [16], such as indexes [30, 17, 58], partitioning schemes [6, 44], data organization [7], or materialized views [5]. Other tools are able to select the tuning parameters for an application [56, 22]. Most of these tools operate in the same way: the DBA provides it with a sample database and workload trace that guides a search process to find an optimal or near-optimal configuration. All of the major DBMS vendors' tools, including Oracle [23, 38], Microsoft [16, 42], and IBM [55, 57], operate in this manner. There is a recent push for integrated components that support adaptive architectures [36], but these again only focus on solving one problem. Likewise, cloud-based systems employ dynamic resource allocation at the service-level [20], but do not tune individual databases.

All of these are insufficient for a completely autonomous system because they are (1) external to the DBMS, (2) reactionary, or (3) not able to take a holistic view that considers more than one problem at a time. That is, they observe the DBMS's behavior from outside of the system and advise the DBA on how to make corrections to fix only one aspect of the problem after it occurs. The tuning tools assume that the human operating them is knowledgeable enough to update the DBMS during a time window when it will have the least impact on applications. The database landscape, however, has changed significantly in the last decade and one cannot assume that a DBMS is deployed by an expert that understands the intricacies of database optimization. But even if these tools were automated such that they could deploy the optimizations on their own, existing DBMS architectures are not designed to support major changes without stressing the system further nor are they able to adapt in anticipation of future bottlenecks.

In this paper, we make the case that self-driving database systems are now achievable. We begin by discussing the key challenges with such a system. We then present the architecture of **Peloton** [1], the first DBMS that is designed for autonomous operation. We conclude with some initial results on using Peloton's integrated deep learning framework for workload forecasting and action deployment.

2. PROBLEM OVERVIEW

The first challenge in a self-driving DBMS is to understand an application's workload. The most basic level is to characterize queries as being for either an OLTP or OLAP application [26]. If the DBMS identifies which of these two workload classes the application belongs to, then it can make decisions about how to optimize the database. For example, if it is OLTP, then the DBMS should store tuples in a row-oriented layout that is optimized for writes. If it is OLAP, then the DBMS should use a column-oriented

	Types	Actions
PHYSICAL	Indexes	`AddIndex`, `DropIndex`, `Rebuild`, `Convert`
	Materialized Views	`AddMatView`, `DropMatView`
	Storage Layout	`Row→Columnar`, `Columnar→Row`, `Compress`
DATA	Location	`MoveUpTier`, `MoveDownTier`, `Migrate`
	Partitioning	`RepartitionTable`, `ReplicateTable`
RUNTIME	Resources	`AddNode`, `RemoveNode`
	Configuration Tuning	`IncrementKnob`, `DecrementKnob`, `SetKnob`
	Query Optimizations	`CostModelTune`, `Compilation`, `Prefetch`

Table 1: Self-Driving Actions – The types of actions that a self-driving DBMS must support for modern database deployments.

layout that is better for read-only queries that access a subset of a table's columns. One way to handle this is to deploy separate DBMSs that are specialized for OLTP and OLAP workloads, and then periodically stream updates between them [54]. But there is an emerging class of applications, known as *hybrid transaction-analytical processing* (HTAP), that cannot split the database across two systems because they execute OLAP queries on data as soon as it is written by OLTP transactions. A better approach is to deploy a single DBMS that supports mixed HTAP workloads. Such a system automatically chooses the proper OLTP or OLAP optimizations for different database segments.

Beyond understanding these access patterns, the DBMS also needs to forecast resource utilization trends. This enables it to predict future demand and deploy optimizations at a time that will have the least impact on performance. The usage patterns of many applications closely follow the diurnal patterns of humans. This is why DBAs schedule updates during off-peak times to avoid service disruptions during normal business hours. Admittedly, there are some workload anomalies that a DBMS can never anticipate (e.g., "viral" content). But these models provide an early warning that enables the DBMS to enact mitigation actions more quickly than what an external monitoring system could support.

Now with these forecast models, the DBMS identifies potential actions that tune and optimize the database for the expected workload. A self-driving DBMS cannot support DBA tasks that require information that is external to the system, such as permissions, data cleaning, and version control. As shown in Table 1, there are three optimization categories that a self-driving DBMS can support. The first are for the database's *physical design*. The next are changes to *data organization*. Finally, the last three affect the DBMS's *runtime behavior*. For each optimization action, the DBMS will need to estimate their potential effect on the database. These estimates not only include what resources the action will consume once it is deployed, but also the resources that the DBMS will use to deploy it.

Even if a system is able to predict the application's workload, choose which action to employ, and determine the best time to enact them, there is still an additional challenge. If the DBMS is not able to apply these optimizations efficiently without incurring large performance degradations, then the system will not be able to adapt to changes quickly. This fact is another reason why autonomous DBMSs have been impossible until now. If the system is only able to apply changes once a week, then it is too difficult for it to plan how to correct itself. Hence, what is needed is a flexible, in-memory DBMS architecture that can incrementally apply optimizations with no perceptible impact to the application during their deployment.

Lastly, an autonomous DBMS has two additional constraints that it has to satisfy to be relevant for today's applications. The first is that the DBMS cannot require developers to rewrite their application to use a proprietary API or provide supplemental information about its behavior. Refactoring code is an expensive process, and most organizations are not willing to do this. The second requirement is that it cannot rely on program analysis tools that only support certain programming environments. This ensures that the DBMS will work with future applications that are not yet invented.

3. SELF-DRIVING ARCHITECTURE

Our research has found that existing DBMSs are too unwieldy for autonomous operation because they often require restarting when changes are made and many of the actions from Table 1 are too slow. Hence, we contend that a new DBMS architecture is the best approach because the integrated self-driving components have a more holistic and finer grained control of the system.

We now describe the architecture of these components in Peloton. There are several aspects of Peloton that make it ideal for this work, as opposed to retrofitting an existing legacy DBMS (e.g., Postgres, MySQL). Foremost is that it uses a variant of multi-version concurrency control that interleaves OLTP transactions and actions without blocking OLAP queries. Another is that it uses an in-memory storage manager with lock-free data structures and flexible layouts that allows for fast execution of HTAP workloads. These design choices have already enabled us to implement support for some optimization actions in Peloton [9].

An overview of the Peloton's self-driving workflow is shown in Figure 1. Other than environment settings (e.g., memory threshold, directory paths), our goal is for Peloton to efficiently operate without any human-provided guidance information. The system automatically learns how to improve the latency of the application's queries and transactions. Latency is the most important metric in a DBMS as it captures all aspects of performance [47, 20]. The rest of this paper assumes that latency is the main optimization objective. Additional constraints can be added for other metrics that are important in distributed environments, such as service costs and energy.

Peloton contains an embedded monitor that follows the system's internal event stream of the executed queries. Each query entry is annotated with its resource utilization. The stream is also periodically punctuated with (1) DBMS/OS telemetry data [20] and (2) the begin/end events for optimization actions [60]. The DBMS then constructs forecast models for the application's expected workload from this monitoring data. It uses these models to identify bottlenecks and other issues (e.g., missing indexes, overloaded nodes), and then selects the best actions. The system executes this action while still processing the application's regular workload and collects new monitoring data to learn how the actions affect its performance.

3.1 Workload Classification

The first component is the DBMS's clusterer that uses unsupervised learning methods to group the application's queries that have similar characteristics [26]. Clustering the workload reduces the number of forecast models that the DBMS maintains, thereby making it easier (and more accurate) to predict an application's behavior. Peloton's initial implementation uses the *DBSCAN* algorithm. This approach has been used to cluster static OLTP workloads [41].

The big question with this clustering is what query features to use. The two types of features are (1) a query's runtime metrics and (2) a query's logical semantics. Although the former enables the DBMS to better cluster similar queries without needing to understand their meaning, they are more sensitive to changes in either the database's contents or its physical design. Similar problems can occur in highly concurrent workloads even if the database does not change [41]. An alternative is to classify queries based on the structure of their logical execution plan (e.g., tables, predicates). Such features are independent from the contents of the database and its physical design. It remains to be seen whether such features produce clusters

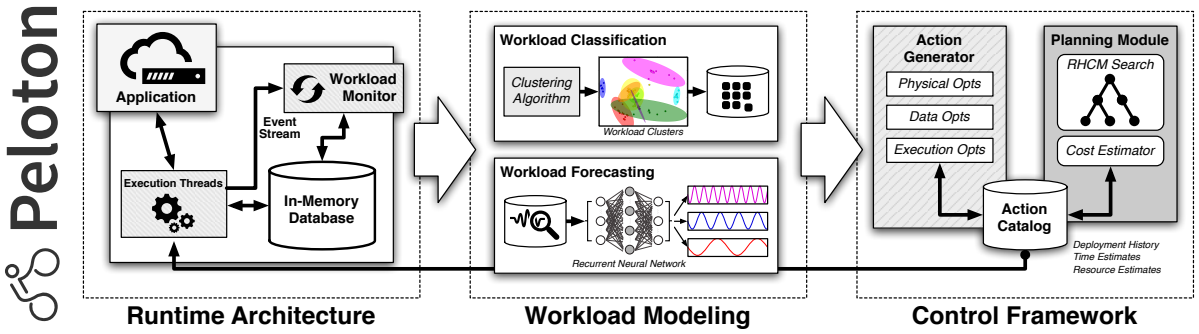


Figure 1: Peloton Self-Driving Architecture – An overview of the forecasting and runtime workflow of Peloton's self-driving components.

that generate good forecasting models, or whether the accuracy of the runtime metrics outweighs the re-training costs. It may turn out that the runtime metrics enable the system to converge to a steady state in less time, and thus the system does not have to retrain its forecasting models that often. Or even if the clustering does change often, hardware-accelerated training enables the system to quickly rebuild its models with minimal overhead.

The next problem is how to determine when the clusters are no longer correct. When this occurs, the DBMS has to re-build its clusters, which could shuffle the groups and require it to re-train all of its forecast models. Peloton uses standard cross validation techniques to determine when the clusters' error rate goes above a threshold. The DBMS can also exploit how queries are affected by actions to decide when to rebuild the clusters.

3.2 Workload Forecasting

The next step is to train forecast models that predict the arrival rate of queries for each workload cluster. With the exception of anomalous hotspots, this forecasting enables the system to identify workload periodicity and data growth trends to prepare for load fluctuations. After the DBMS executes a query, it tags each query with its cluster identifier and then populates a histogram that tracks the number of queries that arrive per cluster within a time period. Peloton uses this data to train the forecast models that estimate the number of queries per cluster that the application will execute in the future. The DBMS also constructs similar models for the other DBMS/OS metrics in the event stream.

Previous attempts at autonomous systems have used the *auto-regressive-moving average model* [8] (ARMA) to predict the workload of web services for autoscaling in the cloud [48]. ARMAs can capture the linear relationships in time-series data, but they often require a human to identify the differencing orders and the numbers of terms in the model. Moreover, the linearity assumption may not be valid for many database workloads because they are affected by exogenous factors.

Recurrent neural networks (RNNs) are an effective method to predict time-series patterns for non-linear systems. A variant of RNNs, called *long short-term memory* (LSTM) [32], allow the networks to learn the periodicity and repeating trends in a time-series data beyond what is possible with regular RNNs. LSTMs contain special blocks that determine whether to retain older information and when to output it into the network. Although RNNs (and deep learning more broadly) are touted as being able to solve many previously intractable problems, research is still needed to discover how to make them viable for self-driving DBMSs.

The accuracy of a RNN is also dependent on the size of its training set data. But tracking every query executed in the DBMS increases the computational cost of model construction. Fortunately, we can exploit the fact that knowing the exact number of queries far into the future is unnecessary. Instead, Peloton maintains multiple RNNs per group that forecast the workload at different time horizons and interval granularities. Although these coarse-grained RNNs are less accurate, they reduce both the training data that the DBMS has to maintain and their prediction costs at runtime. Combining multiple RNNs allows the DBMS to handle immediate problems where accuracy is more important as well as to accommodate longer term planning where the estimates can be broad.

3.3 Action Planning & Execution

The last piece is Peloton's control framework that continuously monitors the system and selects optimization actions to improve the application's performance. This is where the benefit of having a tight coupling of the autonomous components and DBMS architecture is most evident, since it enables the different parts to provide feedback to each other. We also believe that there are opportunities to use reinforcement learning in more parts of the system, including concurrency control [43] and query optimization [10].

Action Generation: The system searches for actions that potentially improves performance. Peloton stores these actions in a catalog along with the history of what happened to the system when it invoked them. This search is guided by the forecasting models so that the system looks for actions that will provide the most benefit. It can also prune redundant actions to reduce the search complexity.

Each action is annotated with the number of CPU cores that the DBMS will use when deploying it. This allows Peloton to use more cores to deploy an action when demand is low but then use fewer cores at other times. Actions that affect the configuration knobs that control the DBMS's resource allocations are defined as delta changes rather than absolute values. Certain actions also have corresponding reversal actions. For example, the reverse of an action to add a new index is to drop that index.

Action Planning: Now with actions in its catalog, the DBMS chooses which one to deploy based on its forecasts, the current database configuration, and objective function (i.e., latency). Control theory offers an effective methodology for tackling this problem [48]. One particular approach, known as the *receding-horizon control model* (RHCM), is used to manage complex systems like self-driving vehicles [45]. The basic idea of RHCM is that at each time epoch, the system estimates the workload for some finite horizon using the forecasts. It then searches for a sequence of actions that minimizes the objective function. But it will only apply the first action in the sequence and then wait for the deployment to complete before repeating the process for the next time epoch. This is why using a high-performance DBMS is critical; if actions complete in minutes, then the system does not have to monitor whether the workload has shifted and decide to abort an in-flight action.

Under RHCM, the planning process is modeled as a tree where each level contains every action that the DBMS can invoke at that

moment. The system explores the tree by estimating the *cost-benefit* of actions and chooses an action sequence with the best outcome. The module may also choose to perform no action at a time epoch. One approach to reduce the complexity of this process by randomly selecting which actions to consider at deeper levels of the search tree, rather than evaluating all possible actions [51]. This sampling is weighted such that the actions that provide the most benefit for the current state of the database and its expected workload are more likely to be considered. It also avoids actions that were recently invoked but where the system later reversed their changes.

An action's *cost* is an estimate of how long it will take to deploy it and how much the DBMS's performance will degrade during this time (if at all). Since many actions will not have been deployed before, it is not always possible to generate this information from previous history. Thus, the system uses analytical models to estimate this cost per action type and then automatically refines them through a feedback mechanism [10]. The *benefit* is the change in the queries' latencies after installing the action. This benefit is derived from the DBMS's internal query planner cost model [15]. It is the summation of the query samples' latency improvements after the action is installed weight by the expected arrival rate of queries as predicted by the forecast models. These forecasts are further weighted by their time horizon such that the immediate (and likely more accurate) models are given greater influence in the final cost-benefit analysis.

In addition to the above cost-benefit analysis, the system also has to estimate how an action will affect Peloton's memory utilization over time. Any action that causes the DBMS to exceed the memory threshold is deemed infeasible and is discarded.

There are obviously subtleties with RHCM that are non-trivial to discern and thus we are currently investigating solutions for them. Most important is how far into the future should the system consider when selecting actions [48, 3]. Using a horizon that is too short will prevent the DBMS from preparing in time for upcoming load spikes, but using a horizon that is too long could make it unable to mitigate sudden problems because the models are too slow. In addition to this, since computing the cost-benefits at each time epoch is expensive, it may be possible to create another deep neural network to approximate them with a value function [51].

Deployment: Peloton supports deploying actions in a non-blocking manner. For example, reorganizing the layout of a table or moving it to a different location does not prevent queries from accessing that table [9]. Some actions, like adding an index, need special consideration so that the DBMS does not incur any false negatives/positives due to data being modified while the action is underway [28].

The DBMS also deals with resource scheduling and contention issues from its integrated machine learning components. Using a separate co-processor or GPU to handle the heavy computation tasks will avoid slowing down the DBMS. Otherwise, the DBMS will have to use a separate machine that is dedicated for all of the forecasting and planning components. This will complicate the system's design and add additional overhead due to coordination.

3.4 Additional Considerations

There are several non-technical challenges that must be overcome for self-driving DBMSs to reach wide-spread adoption. Foremost is the aversion of DBA's from relinquishing control of their databases to an automated system. This is due to years of frustration and mistrust when working with supposed "intelligent" software. To ease the transition, self-driving DBMSs can expose its decision making process in a human-readable format. For example, if it chooses to add an index, it can provide an explanation to the DBA that its models show that the current workload is similar to some point in the past where such an index was helpful.

We also have to support hints from the DBA on whether the system should focus more on optimizing OLTP or OLAP portions of the workload. Similarly, for multi-tenant deployments, the system will need to know whether each database should be tuned equally or whether one database is more important than others.

Lastly, it may be necessary to provide an override mechanism for DBAs. Such human-initiated changes are treated like any other action where the Peloton records its history to determine whether it was beneficial or not. The only difference is that the system is not allowed to reverse it. To prevent the DBA from making bad decisions that are permanent, the DBMS can require the DBA to provide a expiration for the manual actions. If the action was truly a good idea, then the DBMS will keep it. Otherwise, it is free to remove it and choose a better one.

4. PRELIMINARY RESULTS

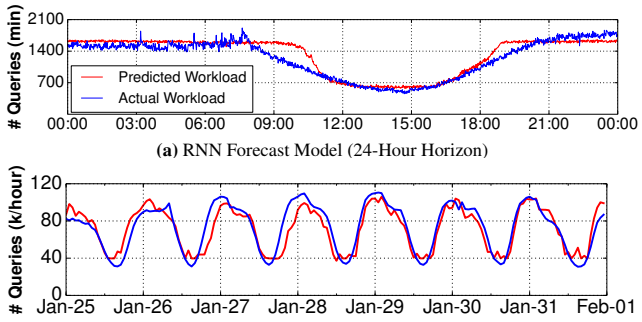
We now provide some early results on Peloton's self-driving capabilities. We have integrated Google TensorFlow [2] in Peloton to perform workload forecasting using the DBMS's internal telemetry data. We trained two RNNs using 52m queries extracted from one month of traffic data of a popular on-line discussion website. For this experiment, we assume that the queries are already clustered correctly using the method described in [41]. We use 75% of the data set (i.e., three weeks) to train the models and then validate them using the remaining 25% data. We apply two stacked LSTM layers on the input, then connect them to a linear regression layer. We use a 10% dropout rate for these layers to avoid over-fitting [53].

The first model predicts the number of queries that will arrive in the next hour at one minute granularity. This model's input is a vector representing the per-minute workload over the past two hours, and the output is a scalar representing the predicted workload an hour later. The second model uses a 24-hour horizon with a one hour granularity. Its input is a vector for the previous day's workload, and the output is a scalar for the predicted workload one day later.

We ran Peloton with TensorFlow on a Nvidia GeForce GTX 980 GPU. The training took 11 and 18 minutes for the first and second RNN, respectively. We observed almost no CPU overhead during this training as all of the computation was performed by the GPU. The graphs in Figures 2a and 2b show that the models are able to predict the workload with an error rate of 11.3% for the 1-hour RNN and 13.2% for the 24-hour RNN. Since we also want to determine the RNNs' runtime overhead, we also measured their storage and computation costs. For the former, each model is approximately 2 MB in size. It takes 2 ms for the DBMS to probe each model for a new prediction, and 5 ms to add a new data point to it. This time is longer because TensorFlow performs back-propagation to update the model's gradients.

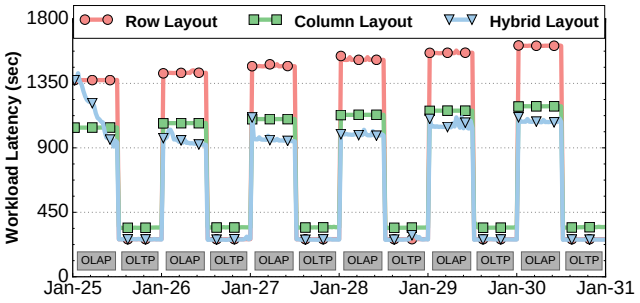
Using these models, we then enable the data optimization actions in Peloton where it migrates tables to different layouts based on the types of queries that access them [9]. Each table's "hot" tuples are stored in a row-oriented layout that is optimized for OLTP, while the other "cold" tuples in that same table are stored in a column-oriented layout that is more amenable to OLAP queries. We use a simulated HTAP workload from the trace that executes OLTP operations during the day and OLAP queries at night. We executed the query sequences in Peloton when autonomic layouts are enabled and compare it against static row- and column-oriented layouts.

The key observation from the performance results in Figure 2c is that Peloton converges over time to a layout that works well for the workload segments. After the first segment, the DBMS migrates the row-oriented tuples to a column-oriented layout, which is ideal for the OLAP queries. The latency drops and matches that of the pure column-oriented system. Next, when the workload shifts to



(a) RNN Forecast Model (24-Hour Horizon)

(b) RNN Forecast Model (7-Day Horizon)



(c) Adaptive Storage Layouts

Figure 2: Preliminary Results – The first graphs in (2a) and (2b) show a visualization of the RNN-generated forecasts versus the actual workload data set for two different time horizons. The second graph in (2c) compares the performance of Peloton when using autonomous hybrid storage layout actions against static layouts for a simulated HTAP application.

OLTP queries, the self-driving system fares better than the static column-oriented one because it performs fewer memory writes.

These early results are promising: (1) RNNs accurately predict the expected arrival rate of queries, (2) hardware-accelerated training has a minor impact on the DBMS's CPU and memory resources, and (3) the system deploys actions without slowing down the application. The next step is to validate our approach using more diverse database workloads and support for additional actions.

5. RELATED WORK

The ability to automatically optimize DBMSs has been studied for decades [12, 18, 16, 59]. As such, there is an extensive corpus of previous work, including both theoretical [14] and applied research [25, 64, 60]. But no existing system incorporates all of these techniques to achieve full operational independence without requiring humans to monitor, manage, and tune the DBMS.

Workload Modeling: Some have used probabilistic Markov models to represent an application's workload states. The approach described in [33, 34] uses them to determine when the workload has changed enough that the DBA needs to update the database's physical design. Others generate Markov models that estimate the next query that an application will execute based on what query it is currently executing so that it can pre-fetch data [50] or apply other optimizations [43]. Similarly, the authors in [24] construct Markov models that predict the next transaction.

The technique proposed in [26] identifies whether a sample workload is either for an OLTP- or OLAP-style application, and then tunes the system's configuration accordingly. The authors in [29] use decision trees to schedule and allocate resources for OLAP queries. Others use similar approaches for extracting query sequences from unlabeled workload traces [62]. DBSeer classifies OLTP transactions based on queries' logical semantics and the number of tuples they access [41]. It then combines analytical and learned statistical models to predict the DBMS's utilization for concurrent workloads.

Most of the previous work on predicting long-term resource demands has been for cloud computing systems. The authors in [48] propose a method based on ARMAs to predict the future resource utilization trends [49, 46] Others use predictive models for tenant placement [19] and load balancing [4] in DBaaS platforms.

Automated Administration: Microsoft's AutoAdmin pioneered the use of leveraging the DBMS's built-in cost models from its query optimizer to estimate the benefits of physical design changes [15]. This avoids a disconnect between what the tool chooses as a good index and what the system uses for query optimization. Configuration tools cannot use the built-in cost models of query optimizers. This is because these models generate estimates on the amount of work to execute a query and are intended to compare alternative execution strategies in a fixed execution environment [52]. All of the major DBMS vendors have their own proprietary tools that vary in the amount of automation that they support [23, 38, 42]. iTuned is a generic tool that continuously makes minor changes to the DBMS configuration whenever the DBMS is not fully utilized [25].

IBM released the DB2 Performance Wizard tool that asks the DBA questions about their application and then provides knob settings based on their answers [39]. It uses models manually created by DB2 engineers and thus may not accurately reflect the actual workload or operating environment. IBM later released a version of DB2 with a self-tuning memory manager that uses heuristics to allocate memory to the DBMS's internal components [55, 57]. Oracle developed a similar system to identify bottlenecks due to misconfiguration [23, 38]. Later versions of Oracle include a tool that estimates the impact of configuration modifications [61, 11]. This approach has also been used with Microsoft's SQL Server [42]. More recently, DBSherlock helps a DBA diagnose problems by comparing regions in performance traces data where the system was slow with regions where it behaved normally [63].

For on-line optimizations, one of the most influential methods is the database cracking technique for MonetDB [36]. This enables the DBMS to incrementally build indexes based on how queries access data [27] and has been extended to where the DBMS builds indexes as a side effect of query processing [28]. The same authors from the cracking method are pursuing modular "self-designing" systems that build on previous RISC-style DBMS proposals [35].

Autonomous DBMSs: The closest attempt to a fully automated DBMS was IBM's proof-of-concept for DB2 [60]. It used existing tools [39] 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 a self-driving DBMS, 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 [37, 21]. Microsoft Azure models resource utilization of DBMS containers from telemetry data and automatically adjusts allocations to meet QoS and budget constraints [20]. There are also controllers for applications to perform black box provisioning in the cloud [13, 3]. The authors in [48] use the same RHCM [45] as Peloton, but they only support a subset of the actions of a self-driving DBMS (see Table 1).

6. CONCLUSION

With the rise of the "Big Data" movement, the demand for an au-

tonomous DBMS is stronger now than it has ever been before. Such a system will remove the human capital impediments of deploying databases of any size and allow organizations to more easily derive the benefits of data-driven decision making applications. This paper outlines the self-driving architecture of the Peloton DBMS. We argued that the ambitious system that we proposed is now possible due to advancements in deep neural networks, improved hardware, and high-performance database architectures.

7. REFERENCES

[1] Peloton Database Management System. http://pelotondb.org.

[2] M. Abadi and *et al.* TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *CoRR*, abs/1603.04467, 2016.

[3] S. Abdelwahed and *et al.* A control-based framework for self-managing distributed computing systems. WOSS'04, pages 3–7.

[4] D. Agrawal and *et al.* Database scalability, elasticity, and autonomy in the cloud. DASFAA, pages 2–15, 2011.

[5] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in SQL databases. VLDB, 2000.

[6] S. Agrawal and *et al.* Integrating vertical and horizontal partitioning into automated physical database design. SIGMOD, 2004.

[7] I. Alagiannis, S. Idreos, and A. Ailamaki. H2o: A hands-free adaptive store. SIGMOD, pages 1103–1114, 2014.

[8] O. D. Anderson. *Time Series Analysis and Forecasting: The Box-Jenkins Approach*. Butterworth & Co Publishers, 1976.

[9] J. Arulraj and *et al.* Bridging the archipelago between row-stores and column-stores for hybrid workloads. SIGMOD, pages 583–598, 2016.

[10] D. Basu and *et al. Cost-Model Oblivious Database Tuning with Reinforcement Learning*, pages 253–268. 2015.

[11] P. Belknap, B. Dageville, K. Dias, and K. Yagoub. Self-tuning for SQL performance in Oracle Database 11g. ICDE, pages 1694–1700, 2009.

[12] P. Bernstein, M. Brodie, S. Ceri, and *et al.* The asilomar report on database research. *SIGMOD record*, 27(4):74–80, 1998.

[13] E. Cecchet, R. Singh, and *et al.* Dolly: Virtualization-driven database provisioning for the cloud. VEE '11, pages 51–62, 2011.

[14] S. Ceri, S. Navathe, and G. Wiederhold. Distribution design of logical database schemas. *IEEE Trans. Softw. Eng.*, 9(4):487–504, 1983.

[15] S. Chaudhuri and V. Narasayya. Autoadmin "what-if" index analysis utility. *SIGMOD Rec.*, 27(2):367–378, 1998.

[16] S. Chaudhuri and V. Narasayya. Self-tuning database systems: a decade of progress. VLDB, pages 3–14, 2007.

[17] S. Chaudhuri and V. R. Narasayya. An efficient cost-driven index selection tool for microsoft SQL server. VLDB, pages 146–155, 1997.

[18] S. Chaudhuri and G. Weikum. Rethinking db system architecture: Towards a self-tuning RISC-style database system. VLDB'00.

[19] C. Curino, E. P. Jones, and *et al.* Workload-aware database monitoring and consolidation. SIGMOD, pages 313–324, 2011.

[20] S. Das, F. Li, and *et al.* Automated demand-driven resource scaling in relational database-as-a-service. SIGMOD, pages 1923–1934, 2016.

[21] S. Das and *et al.* Elastras: An elastic, scalable, and self-managing transactional database for the cloud. *ACM TDS*, 38(1):5:1–5:45, 2013.

[22] B. Debnath, D. Lilja, and M. Mokbel. SARD: A statistical approach for ranking database tuning parameters. ICDEW, pages 11–18, 2008.

[23] K. Dias, M. Ramacher, U. Shaft, V. Venkataramani, and G. Wood. Automatic performance diagnosis and tuning in oracle. CIDR, 2005.

[24] N. Du, X. Ye, and J. Wang. Towards workflow-driven database system workload modeling. DBTest, pages 1–6, 2009.

[25] S. Duan, V. Thummala, and S. Babu. Tuning database configuration parameters with iTuned. *VLDB*, 2:1246–1257, August 2009.

[26] S. Elnaffar, P. Martin, and R. Horman. Automatically classifying database workloads. CIKM, pages 622–624, 2002.

[27] M. R. Frank, E. Omiecinski, and S. B. Navathe. Adaptive and automated index selection in RDBMS. EDBT, pages 277–292, 1992.

[28] G. Graefe and *et al.* Transactional support for adaptive indexing. *VLDB*, 23(2):303–328, 2014.

[29] C. Gupta and *et al.* PQR: Predicting Query Execution Times for Autonomous Workload Management. ICAC, pages 13–22, 2008.

[30] H. Gupta, V. Harinarayan, A. Rajaraman, and J. D. Ullman. Index selection for olap. ICDE, pages 208–219, 1997.

[31] J. M. Hellerstein and M. Stonebraker. What goes around comes around. chapter Transaction Management, pages 2–41. 4th edition, 2005.

[32] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, Nov. 1997.

[33] M. Holze and N. Ritter. Towards workload shift detection and prediction for autonomic databases. In *PIKM*, pages 109–116, 2007.

[34] M. Holze and N. Ritter. Autonomic Databases: Detection of Workload Shifts with n-Gram-Models. In *ADBIS*, pages 127–142, 2008.

[35] S. Idreos. Data systems that are easy to design (SIGMOD Blog). http://wp.sigmod.org/?p=1617, June 2015.

[36] S. Idreos, M. L. Kersten, and S. Manegold. Database cracking. CIDR, pages 68–78, 2007.

[37] J. O. Kephart. Research challenges of autonomic computing. ICSE, pages 15–22, 2005.

[38] S. Kumar. Oracle Database 10g: The self-managing database, Nov. 2003. White Paper.

[39] E. Kwan, S. Lightstone, and *et al.* Automatic configuration for IBM DB2 universal database. Technical report, IBM, jan 2002.

[40] G. Lanfranchi and *et al.* Toward a new landscape of sys. mgmt. in an autonomic computing env. *IBM Syst. J.*, 42(1):119–128, 2003.

[41] B. Mozafari and *et al.* Performance and resource modeling in highly-concurrent oltp workloads. SIGMOD, pages 301–312, 2013.

[42] D. Narayanan, E. Thereska, and A. Ailamaki. Continuous resource monitoring for self-predicting DBMS. MASCOTS'05, pages 239–248.

[43] A. Pavlo and *et al.* On Predictive Modeling for Optimizing Transaction Execution in Parallel OLTP Systems. *VLDB*, 5:85–96, 2011.

[44] J. Rao, C. Zhang, N. Megiddo, and G. Lohman. Automating physical database design in a parallel database. SIGMOD'02, pages 558–569.

[45] J. Richalet and *et al.* Model predictive heuristic control: Applications to industrial processes. *Automatica*, 14(5):413–428, 1978.

[46] F. Rosenthal and W. Lehner. Efficient in-database maintenance of arima models. SSDBM, pages 537–545. 2011.

[47] N. Roy and *et al.* Finding approximate POMDP solutions through belief compression. *J. Artif. Intell. Res. (JAIR)*, 23:1–40, 2005.

[48] N. Roy and *et al.* Efficient autoscaling in the cloud using predictive models for workload forecasting. CLOUD, pages 500–507, 2011.

[49] E. Samaras, M. Shinzuka, and A. Tsurui. ARMA representation of random processes. *J. of Eng. Mechanics*, 111(3):449–461, 1985.

[50] C. Sapia. PROMISE: Predicting Query Behavior to Enable Predictive Caching Strategies for OLAP Systems. DaWaK, pages 224–233, 2000.

[51] D. Silver, A. Huang, and *et al.* Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.

[52] A. A. Soror and *et al.* Automatic virtual machine configuration for database workloads. SIGMOD, pages 953–966, 2008.

[53] N. Srivastava and *et al.* Dropout: A simple way to prevent neural networks from overfitting. *J. ML. Res.*, 15(1):1929–1958, 2014.

[54] M. Stonebraker and U. Cetintemel. "one size fits all": An idea whose time has come and gone. ICDE, pages 2–11, 2005.

[55] A. J. Storm, C. Garcia-Arellano, and *et al.* Adaptive self-tuning memory in DB2. VLDB, pages 1081–1092, 2006.

[56] D. G. Sullivan and *et al.* Using probabilistic reasoning to automate software tuning. SIGMETRICS, pages 404–405, 2004.

[57] W. Tian, P. Martin, and W. Powley. Techniques for automatically sizing multiple buffer pools in DB2. CASCON, pages 294–302, 2003.

[58] G. Valentin, M. Zuliani, and *et al.* DB2 advisor: an optimizer smart enough to recommend its own indexes. ICDE, pages 101–110, 2000.

[59] G. Weikum and *et al.* Self-tuning db technology and info services: From wishful thinking to viable engineering. VLDB'02, pages 20–31.

[60] D. Wiese and *et al.* Autonomic tuning exp.: A frmwk. for best-practice oriented autonomic db tuning. CASCON, pages 3:27–3:41, 2008.

[61] K. Yagoub, P. Belknap, B. Dageville, K. Dias, S. Joshi, and H. Yu. Oracle's sql performance analyzer. *IEEE Data Eng. Bul.*, 31(1), 2008.

[62] Q. Yao, A. An, and X. Huang. Finding and analyzing database user sessions. DASFAA, pages 851–862, 2005.

[63] D. Y. Yoon, N. Niu, and B. Mozafari. Dbsherlock: A performance diagnostic tool for transactional databases. SIGMOD, 2016.

[64] D. C. Zilio, J. Rao, and *et al.* DB2 design advisor: integrated automatic physical database design. VLDB, pages 1087–1097, 2004.