www.ijecs.in
*International Journal Of Engineering And Computer Science*
*Volume 9 Issue 08 August 2020, Page No. 25132-25147*
*ISSN: 2319-7242 DOI: 10.18535/ijecs/v9i08.4520*

# Machine Learning for Database Management Systems

**Sai Tanishq N**

Gokaraju Rangaraju Institute of Engineering And Technology

**Abstract:**

Machine Learning (ML) is transforming the world with research breakthroughs that are leading to the progress of every field. We are living in an era of data explosion. This further improves the output as data that can be fed to the models is more than it has ever been. Therefore, prediction algorithms are now capable of solving many of the complex problems that we face by leveraging the power of data. The models are capable of correlating a dataset and its features with an accuracy that humans fail to achieve. Bearing this in mind, this research takes an in-depth look into the of problem- solving potential of ML in the area of Database Management Systems (DBMS). Although ML hallmarks significant scientific milestones, the field is still in its infancy. The limitations of ML models are also studied in this paper.

## 1. Introduction

A computer program is said to learn from experience *E* with respect to some class of tasks *T* and performance measure *P* if its performance at tasks in *T*, as measured by *P*, improves with experience *E* [33]. In order to fix the ideas, it is useful to introduce the machine learning methodology as an alternative to the conventional engineering approach for the design of an algorithmic solution [34]. The goal of machine learning is to program computers to use example data or past experience to solve a given problem. Many successful applications of machine learning exist already, including systems that analyze past sales data to predict customer behavior, optimize robot behavior so that a task can be completed using minimum resources, and extract knowledge from bioinformatics data. [35] Integrating machine

learning into DBMS is an ongoing effort in both academia and industry. The combination of ML and DBMS is attractive because businesses have massive amounts of data residing in their existing DBMS [36]. Furthermore, relational operators can be used to pre-process and denormalize a complex schema conveniently before executing ML tasks [37].

DBMS developers have to use heuristics, restrict the problem space, or even rely on human intervention to solve problems such as query optimization, physical database design optimization, and buffer management. The goal of these heuristics and restrictions is not to find an optimal solution for a particular instance, but to find a solution which has a safe worst-case performance on all cases. Alternatively, ML provides a flexible framework to automatically learn an efficient program to solve these problems for a specific instance without being explicitly programmed by a human developer.

The paper is primarily focused at relational database management systems (RDBMSs) because they remain the most widely used DBMS type. We survey the existing landscape of ML to solve hard programming problems in DBMSs. DBMS is divided into three main sub-components in this paper: 1) Query Parser, 2) Relational Engine, and 3) Execution Engine. After providing some background on each of these subcomponents and different ML methods, several systems that have been proposed using ML are identified. These include systems in Query Parser, Relational Engine, and Execution Engine in Section 3, 4, and 5, respectively. In Section 6, three overarching design decisions are identified that a DBMS developer has

| System | Component | | | | | | | Design Choices | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Query Parser | Relational Engine | | | Execution Engine | | | Integration Mode | Learning Source | ML Model Family | ML Learning Type |
| | | Optimizer | Physical DB Design | Approx. Query Proces. | Knob Tuning | Provisioning/ Scheduling | Data Structures & Algos. | | | | |
| SpeakQL | | | | | | | | External | N/A | DL | N/A |
| Seq2SQL | | | | | | | | External | Queries | DL | RL |
| SQLNet | | | | | | | | External | Queries | DL | Supervised |
| LEO | | | | | | | | Internal | Queries | Classical | Supervised |
| CardLearner | | | | | | | | Internal | Queries | Classical | Supervised |
| Naru | | | | | | | | External | Data | DL | Supervised |
| DeepDB | | | | | | | | External | Data | Classical | Supervised |
| QPPNet | | | | | | | | External | Queries | DL | Supervised |
| SkinnerDB | | | | | | | | Internal | Queries | Classical | RL |
| ReJoin | | | | | | | | Internal | Queries | DL | RL |
| Neo | | | | | | | | Internal | Queries | DL | RL |
| AutoAdmin | | | | | | | | External | Queries | Classical | Supervised |
| QB5000 | | | | | | | | External | Queries | Classical | Supervised |
| DQM | | | | | | | | External | Queries | DL | RL |
| Verditct | | | | | | | | External | Queries | Classical | Supervised |
| DBEst | | | | | | | | External | Data | Classical | Supervised |
| iTuned | | | | | | | | External | Queries | Classical | Unsupervised |
| OtterTune | | | | | | | | External | Queries | Classical | Unsupervised |
| iBTune | | | | | | | | External | Queries | DL | Supervised |
| WiseDB | | | | | | ✓ | | External | N/A | Classical | Supervised |
| Bandit | | | | | | ✓ | | External | Queries | Classical | RL |
| PerfEnforce | | | | | | ✓ | | External | Queries | Classical | RL |
| Learned Index | | | | | | | ✓ | Internal | Data | Classical | Supervised |
| FITing-Tree | | | | | | | ✓ | Internal | Data | Classical | Supervised |
| XIndex | | | | | | | ✓ | Internal | Both | Classical | Supervised |
| NoisePage | | | ✓ | | ✓ | | | Internal | Queries | Both | All |
| SageDB | | ✓ | ✓ | | ✓ | ✓ | ✓ | Internal | Both | Both | All |

***Figure 1: Systems surveyed in this paper, DBMS component they optimize, and the design choices they make (DL: Deep Learning, RL: Reinforcement Learning).***

to make when incorporating ML into a DBMS. An abstract summary of where the surveyed systems fall in this taxonomy is presented in Figure 1. In Section 7, we identify three open challenges to ML: 1) improving robustness, 2) re-thinking the DBMS architecture, and 3) exploiting transfer learning.

## 2. Background

We provide some background on the database management system internals and machine learning methods to help understand the rest of this survey.

### 2.1 Database Management Systems

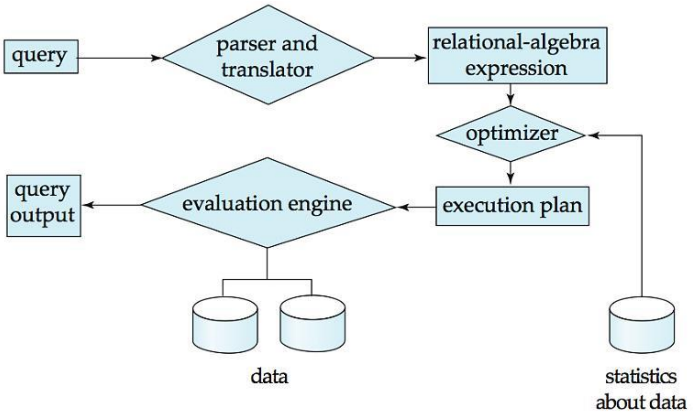


***Figure 2: Steps in query processing***

We identify three main components of a DBMS: 1) Query Parser, 2) Relational Engine, and 3) Execution Engine. To give a brief introduction to each of these components, we next explain the journey of a typical SQL query.

**Example Query.** *Alice is a data analyst at an online e-commerce website and she wants to find the top 10 products in terms of sales revenue. She writes a SQL query using the SQL client in her laptop and submits it to the DMBS for execution.*

**Query Parser.** At the DBMS, the SQL query will be first intercepted by the query parser. The query parser will verify the query is 1) free from syntax and semantic errors, 2) verify the user is authorized to execute the query, and 3) convert the query into the internal format used by the system.

**Relational Engine.** The parsed query is then sent to the relational engine which outputs an optimal query evaluation plan (QEP) for the given query. It does so

by searching the space of possible query evaluation plans and estimating the cost of each option. For this task, it relies on the metadata information and statistics about the data in the DBMS, which has to

be generated beforehand. QEP dictates the order of execution of each relational operator and which physical operator implementation to be used.

**Execution Engine.** Finally, the QEP is sent to the execution engine for execution, which is responsible for managing all the low-level system resources such as memory buffers and thread pools, accessing the data and indices, and coordinating the execution of the query either on a single machine or on multiple machines. It also handles concurrency control and failure recovery of the DBMS. The output generated by executing the query is sent back to Alice.

**Database Administrator.** Being complex software systems, DBMSs require a significant amount of tuning to achieve good performance for a particular use case. To worsen the situation, the set of default configurations in the DBMS are often obsolete and do not match the resource availability of modern hardware (e.g., MySQL default buffer pool size is 128MB!). Thus, in Alice's use case, the DBMS has to be tuned to the schema of the database and the set of widely used queries by the analysts in her company. Furthermore, for query optimization the relevant data statistics have to be generated beforehand and kept up to date. Typically, these tasks are performed by a database administrator who has specialized knowledge about the internals of the DBMS.

### 2.2 Machine Learning Methods

For the purpose of this paper, we divide machine learning into two major model families: 1) deep learning and 2) classical machine learning.

**Deep Learning.** Deep learning (DL) [1] is the name given to the family of ML models that are composed of artificial neural networks (ANNs). ANNs are inspired by the structure and function of the human brain. They learn a hierarchy of parametric features using layers of various types (e.g., fully connected, ReLU). All parameters are trained using a technique called back-propagation. Training a deep learning

model incurs massive costs: they typically need many GPUs for reasonable runtimes, huge labeled datasets, and complex hyper-parameter tuning. Recently, DL methods have been able to produce superior accuracy results outperforming other ML methods on hard tasks like computer vision and natural language processing. In some cases, they have even surpassed the human-level accuracy.

**Classical Machine Learning.** Despite many successes using DL-based ML methods, in many applications that involve working with structured data, ML model families like generalized linear models, decision tree models, and Bayesian models are widely used. Typically, these model families are collectively referred to as *classical machine learning*, a term coined to contrast them with DL models. Unlike DL models, which can all be trained using back-propagation method, each sub-family in classical ML uses different statistical learning foundations and learning methods. Furthermore, their characteristics are also significantly different among different sub-families.

For the above two ML model families, we further identify three different learning paradigms: 1) supervised learning, 2) unsupervised learning, and 3) reinforcement learning.

**Supervised Learning.** In supervised learning, training data consists of a set of input (also called features) and output pairs. The goal of the ML model is to learn a prediction function that takes in unseen inputs and predicts an output value such that the discrepancy between the predicted value and the actual value corresponding to the unseen input is minimized. Supervised learning is applicable in settings where there is a direct observable mapping between input and output, a large amount of training data available, and we are confident that the training data covers the entire data distribution.

**Unsupervised Learning.** In unsupervised learning, the training data contains only the input and no explicit output. The purpose of the ML model is to learn a function that can discern the latent structure of the inputs. Given an unseen test input, the trained unsupervised ML model can be used to predict the structural properties of the input. Popular

applications of unsupervised learning include probability density function estimation and data clustering.

**Reinforcement Learning.** The goal of reinforcement learning methods is to learn a function that takes in an input state and generates an action that will maximize the overall cumulative reward (not the immediate reward as in supervised learning). Unlike supervised learning methods, they

do not make any assumptions about the data generating process and also do not require a training set of state-action (input-output) pairs to be presented. State-action pairs are generated as the model interacts with the environment and models use an explore-exploit paradigm to learn while being used to make predictions at the same time. Reinforcement learning techniques are useful in settings where the reward of action is not directly observable and we want the model to try different things and pick the best option. However, as there is no initial assumption on the data generating process (e.g., through training data), in some problems reinforcement learning method will fail or take a very long time to learn.

## 3. Query Parser

The task of the query parsing sub-component in a DBMS is to check whether a given SQL query in text format is free from syntax and grammar errors and if so, translate it into a relational algebra expression, which is an already solved problem. Hence, most of the new work on query parsing focuses on supporting query modalities beyond text and relaxing the grammar of SQL to support natural language-based querying. Recent advancements in natural language processing using deep learning techniques provide a promising opportunity to achieve these goals.

### 3.1 Expanding the Query Modalities

#### 3.1.1 Speaking SQL Queries

SpeakQL [2] provides a speech-driven querying interface, which can be used to query data by speaking out a SQL query instead of typing it. It uses an off the shelf automatic speech recognition engine to compile a spoken SQL query into text

form and use knowledge about the schema of the underlying data to refine the structure and the literals of the query. SpeakQL is able to capture complex SQL queries but puts significant cognitive load on the user when dictating such queries.

#### 3.1.2 Speaking Natural Language Queries

Seq2SQL [3] and SQLNet [4] are two systems that focus on compiling natural language queries into SQL. Seq2SQL uses a large dataset (n=84,000) of manually annotated natural language-SQL pairs and trains a deep reinforcement learning model to compile natural language queries into SQL. SQLNet uses the same training dataset and uses a neural machine translation approach. While these systems have shown some early promising results for supporting natural language queries over single table data, their accuracy significantly suffers for complex queries that involve joins over multiple tables.

## 4. ML for Relational Engine

Relational Engine is one of the most important components in a database management system, that has been extensively studied for the last 40 years. We identify three different sub-areas of ML applications in the relational engine: ML for (1) query optimization, (2) physical database design automation, and (3) approximate query processing. Next, we discuss some of the most prominent works in each of these sub-areas.

### 4.1 Query Optimization

The goal of query optimization is to transform an input relational algebra expression into an optimal query evaluation plan (QEP). To achieve this, traditional query optimizers perform a search over the space of potential QEPs and pick the one with the least total cost. However, estimating the total cost is a complex task and it is often approximated by estimating the total size of the intermediate tuples generated during query evaluation. This is called the *cardinality estimation* problem, which remains still an unsolved problem despite advancements over many decades. Database optimizers often make assumptions such as uniformity, independence, and the principle of inclusion [5] to perform cardinality

estimation. These assumptions often do not hold in real data and optimizers tend to under- or over-estimate the query cost and pick sub-optimal query evaluation plans, which can be worse by orders of magnitude. We found that the use of ML for query optimization can be broadly divided into two major approaches. The first approach is to train ML models for cardinality estimation and integrate them with the existing search strategies in the optimizer. The second approach is to completely replace the traditional query optimizer by using ML to generate the QEP, end-to-end.

#### 4.1.1 Cardinality Estimation

We identify three different methods for learning ML models for cardinality estimation.

**Predicate Level Models.** LEO [6] is one of the

very first systems developed to use ML techniques to improve the cardinality estimation. It uses the optimizer estimated cardinalities and the cardinalities observed during execution to learn predicate level linear models to predict the correct output cardinalities. However, not accounting for QEP structure and using the optimizer estimated cardinality as the only feature prevents LEO from achieving high accuracies.

**Sub-graph Level Models.** CardLearner [7] extends the idea of LEO and builds ML models to predict cardinalities of commonly occurring query templates instead of each predicate. A template is a family of queries with the same structure but varying parameters and inputs. Instead of training one single model for all commonly occurring query templates, it trains separate ML models for each template. CardLerarner uses several hand-engineered features including metadata features, input cardinalities of all input datasets, and features associated with operators. Operating on query templates enables CardLearner to capture the semantics of the QEP and correlation in data and yield better results compared to predicate level approaches such as LEO. However, its accuracy degrades when faced with new queries and queries which have unseen sub- graphs.

**Graph Level Models.** One could also train ML models to predict the output cardinality of entire QEPs. QPPNet uses a novel neural network architecture that matches the QEP structure. It is composed of stacking sub-neural-modules corresponding to each operator in the QEP which takes in hand-engineered features as well as output from other sub- neural-modules. As a result, it can learn the correlation in data, relationships between operator features, and plan structure to predict the QEP cost more accurately. However, one of the limitations of replacing just the cardinality estimation component in the optimizer is that the model will have zero knowledge about the plans that were never generated by the optimizer, which limits the optimizer's ability to generate new and better plans.

Thus, instead of learning from query workloads, Naru [8], and DeepDB [9] try to solve the cardinality estimation problem by modeling the joint probability distribution of the data. Naru uses deep autoregressive models and combines it with a novel Monte Carlo integration scheme to efficiently support range and wildcard queries. DeepDB uses Relational Sum-Product Networks (RSPNs), a variant of a probabilistic graphical model, to model the joint probability distribution.

It is important to mention that some of the above systems (e.g., LEO in IBM DB2 and CardLearner in Microsoft SCOPE) have been (or are being) used in enterprise systems.

**Query Evaluation Plan Generation** The QEP search strategy in the optimizer closely resembles the reinforcement learning (RL) methods in machine learning. Hence, several systems have tried to replace the entire QEP generation process using RL instead of using ML models to augment an existing optimizer. This ability to learn from the feedback from the chosen QEPs enables RL models to avoid choosing the same bad QEP over and over again. We identify two different methods of using RL for generating QEPs end-to-end.

**Intra-Query Classical RL.** SkinnerDB [10] proposes an intra-query regret-bounded RL learning strategy to find the optimal join ordering for a QEP.

Instead of learning from past query executions, it uses UTC algorithm [11] to learn from the current query execution to optimize the remaining of the current query. While this approach incurs some overheads due to cold starting for every query, the overall overheads remain negligible as it can avoid catastrophic join order choices.

**Deep RL.** ReJoin [12] and Neo [13] use recent advancements in deep reinforcement learning to generate optimal QEPs. ReJoin uses the existing cost model of the optimizer to learn a policy network that can outperform the optimizer search strategy after more online training. Neo uses the observed latency of QEP executions to learn a value network to predict the latency of any new QEP. To reduce learning time and avoid choosing and evaluating prohibitively expensive join orders, Neo bootstraps the value network using latencies observed for the QEPs generated by a traditional query optimizer.

While the above systems have shown some early promising results on the ability of ML techniques to replace the traditional query optimizers, a vast number of important challenges remain open to enable practical adoption. For example, these

systems make simplifying assumptions on the grammar of supported queries, do not support physical operator selection, and in some cases assumes the availability of customized execution engines (e.g., SkinnerDB [10]).

### 4.2 Physical Database Design Automation

One of the most important properties of DBMSs is physical data independence. This allows changing the physical structure of the database without requiring to change the user queries. However, the physical database design significantly affects the performance and has to be tuned for a specific use case. Physical database design choices include creating indices, selecting materialized views, and selecting data partitioning. Traditionally, this has been the responsibility of database administration personnel and they have used heuristics and human judgment to perform these tasks. Several systems have used ML techniques that learn from workload

patterns either to develop decision support systems for database administrators or to automate the process. We identify two different methods for integrating ML for physical database design automation.

#### 4.2.1 Reactive Systems

SQL Server AutoAdmin [14] system is one of the very first systems to adopt ML techniques for the physical database design process. It adopts a reactive strategy where it takes in a historical query work-load and searches for a configuration that minimizes the cost of execution of the workload and recommends that to the database administrator. The chosen configuration dictates which indices and materialized-views to be created and which data partitioning scheme to be used. The search strategy uses a variant of frequent itemsets mining techniques to efficiently explore the enormous search space generated by a large number of possibilities. The search strategy also requires estimating the cost of a new configuration without actually executing the workload. To achieve this, AutoAdmin extends the query optimizer's cost model to support what-if queries which can assume the presence of a selected set of configurations (either hypothetical or materialized) and ignore the presence of other configurations. However, due to the well-known limitations of the optimizer's cost model, a configuration that the optimizer thinks is better than others can be worse when implemented. In a followup work [15], AutoAdmin uses ML models trained on past QEP execution experiences to obtain confidence in the selected configuration before making the actual change. It does so by formulating a classification problem to predict whether the new configuration will be better than the current configuration.

#### 4.2.2 Proactive Systems

Configurations chosen by reactive systems like AutoAdmin may be sub-optimal for the workload in the near future. On the contrary systems like QB5000 [16] and DQM [17] takes a proactive strategy for physical database design that completely automates the process without any intervention of a human. QB5000 trains ML

models to predict the query workload to the future and uses that to select the best set of indices. DQM trains a deep RL model to learn a policy to opportunistically select and evict materialized views subject to storage constraints, that will have the most benefit into the future. While these systems have shown promising initial results, much work is still needed to improve the robustness before incorporating them into enterprise systems.

### 4.3 Approximate Query Processing

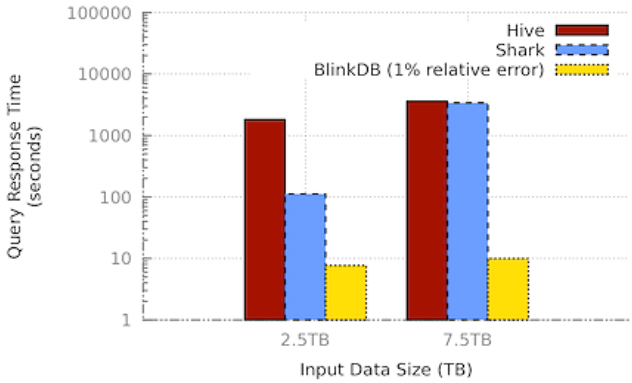


***Figure 3: Input Data Size vs Query Response Time graph***

While the ever-growing volumes of data enable us to glean unprecedented insights, the associated high computational and resource costs often become a bottleneck. Approximate query processing (AQP) techniques try to mitigate this issue by generating approximate answers to the original query at a fraction of the time and cost of the original query execution. The conventional approach to AQP is to use query time data sampling and data statistics to

answer queries. After the query is executed, the work done for that query is never reused. However, every new query execution reveals some new information about the data and executing more and more queries over time enables us to refine that information even further. Thus, machine learning techniques provide an interesting opportunity to learn from past query executions and use that learning to approximately answer future queries. We identify two different methods of using ML for AQP.

**4.3.1 Augmenting Existing AQP Components**
Verdict [18] is one of the first systems to apply this technique in the context of AQP. It uses a data structure called *Query Synopsis* to store the past query results and uses it to refine the approximate answer generated by the system for new queries. Thus, Verdict can reduce the runtime required for an approximate query for specified error bound or reduce the error bound for an approximate query with a specified time budget. This is achieved by modeling a multivariate normal distribution model using the principle of maximum entropy.

**4.3.2 Replacing Existing AQP Components**
Instead of learning from past queries and augmenting sampling-based methods with ML methods at execution time, DBEst [19] takes a pure ML-based approach for answering AQP queries. It samples data corresponding to each predicate attribute and group by attribute values and trains regression and density models. At execution time it uses the corresponding models and performs integration over those models to generate AQP result. Similarly, DeepDB [9] also uses ML methods to learn the joint probability distribution of the data, which can be used to answer AQP queries. The type of the ML models used by DeepDB is called Relational Sum-Product Networks (RSPN), which belongs probabilistic graphical model family. Given a relational database and the correlations between columns, DeepDB trains RSPNs for the tables and use the RSPN ensemble to answer SQL queries by compiling them into inference procedures over the RSPNs.

## 5. ML for Execution Engine

In this section, we identify several prominent works that use ML methods to either augment or replace critical components in a database execution engine. We categorize them into three major sub-areas: ML for 1) knob tuning, 2) scheduling and resource provisioning, and 3) data structures and algorithms.

### 5.1 Knob Tuning

The performance of the database execution engine is highly dependant on the chosen values of the tunable knobs which control nearly all aspects of the runtime

Operations. For example, these knobs control aspects such as how much memory to be used for caching data versus transaction log buffer, how often the data to be written to the disk, and things like query execution parallelism. Similar to physical data- base design, finding a good knob configuration for a target query workload is generally the responsibility of database administration personnel, for which they either use common sense heuristics and/or a trial and error procedure. Alternatively, one could use ML techniques to automatically find an optimal knob configuration for a target query workload.

iTuned [20] approaches this problem by executing a series of carefully-planned experiments and picking the configuration which minimizes the overall workload time. Given a target query workload iTuned uses Latin hypercube sampling to pick an initial set of configurations and execute them to obtain the execution time. The results are then modeled using a Gaussian process representation to pick the next configuration to be executed. This process continues until a good enough configuration is found. It also uses several techniques to reduce the overall tuning time including early elimination of configurations with insignificant improvements, executing parallel experiments, and compressing the query workload.

Similarly, OtterTune [21] also executes a series of experiments chosen by modeling a Gaussian representation process. However, instead of starting with a sampled set of initial configurations, it maps the current workload to a similar previous workload which has been already tuned and uses that to pick the next experiment configuration. Leveraging the learning from previous workloads helps OtterTune to finish the tuning process much faster than iTuned. Workload mapping is achieved through a workload characterization step which combines

factor analysis (over metrics obtained through database monitoring tools) and k-means clustering. It also prunes irrelevant knob configurations ranked using Lasso feature selection and incrementally increase the number of tuned configuration based on their importance as tuning progresses.

iBTune [22] is a system for reducing buffer pool sizes of cloud OLTP database instances to reclaim memory while conforming to the service level agreements (SLA) on query response time. It iteratively uses data from other instances with similar workloads to choose a target buffer pool size using large deviation analysis. But before making the change, it uses a pairwise deep neural network to predict the new response time and proceeds only if the predicted value is within the SLA. One of the major bottlenecks for ML-based knob tuning methods is not having access to low-level/sub-component level system performance metrics. While database systems do provide metrics, they often happen to be aggregated at the entire system level and are less informative to tune sub-component level knobs. Another bottleneck is the inability to change system configuration values without complete system restarts. Database systems are not designed to be tuned by iteratively executing multiple experiments. But, ML-based methods require on the fly configuration testing which incurs significant overheads.

### 5.2 Resource Provisioning and Scheduling

Execution engines have to make several planning decisions to meet the service level objectives imposed by users as they execute QEPs. This includes resource *provisioning*: deciding how many machines to be used for execution and/or *scheduling*: deciding which QEP to be executed on which worker and in which order. The emergence of cloud computing environments makes these planning decisions even more important because of their pay- as-you-go model. The existing approach taken by many database systems is to use human-engineered rules or heuristics-based approaches, which are often too rigid and slow to respond to the rapid and dynamic query workloads. Alternatively, ML provides an opportunity to automatically learn these planning decisions using past query workloads. We identify two different settings of applying ML for resource provisioning and scheduling.

#### 5.2.1 Provisioning and Scheduling for Batch Processing

WiSeDB [23] is a workload management service for cloud databases, which holistically addresses both the resource provisioning and scheduling problems. It takes in a query workload, the runtime of each query on each machine type, a user-defined target performance goal (e.g., average/max latency, percentile-based metrics), and recommends a set of strategies and their associated costs to the user. Under certain assumptions, solving the original planning problem can be reduced to the bin packing problem which is an NP-hard problem. WiSeDB samples a large number of small workloads from the original workload and solves them using a brute force graph search algorithm. It then extracts decisions and features from each of the decisions made by the graph algorithm and trains a decision tree model. Finally, this decision tree model is used to generate the planning strategies and their associated costs for the original work-load. By using a learning-based adaptive strategy, WiSeDB can outperform many heuristic-based techniques with little training overhead. One of the major limitations of WiSeDB is that it requires the user to provide cost estimates for each query, and accurately estimating the cost of previously unseen queries is an open challenge.

#### 5.2.2 Provisioning and Scheduling for Online Processing

Similar to WiSeDB, Bandit [24] is another system for solving the provisioning and scheduling problem of cloud database systems which focuses on online scheduling rather than batch scheduling. It models the planning problem as a multi-tiered contextual multi-arm bandit problem (CMAB), a well-known reinforcement learning technique. The tiers in the CMAB corresponds to the different VM types and they are organized in the descending order of their capacity/cost. The goal of the CMAB is to reduce the overall cost, which can be both monetary cost or cost due to not meeting a deadline. When a new query arrives, starting from the first

VM of the first tier the CMAB model iteratively makes one of three choices,

1) Assign the query to the current machine,

2) Send the query to next machine of the current tier,

and

3) Send the query to the next tier. As the context, it uses features from both the query and the current state of the VM. Hence, it can implicitly estimate the cost of a query as part of the learning problem and support previously unseen queries. PerfEnforce [25] also uses ML techniques based on reinforcement learning and multi-layer perceptrons to automatically scale the size of a data processing cluster to meet the user-defined target performance goal.

While the above systems have shown promising results on the ability to replace provisioning and scheduling components using ML, there has not been much adoption in real-world systems, and there is a reluctance among systems developers to put scalability decisions in the hands of machine learning algorithms. One of the major reasons for this is the inability of ML techniques to provide bounds on the worst-case scenario. But as ML techniques get more robust, it can be expected that existing heuristics-based planning modules will get replaced by learned components.

### 5.3 Data Structures and Algorithms

Every operation that is executed by the execution engine heavily relies on core data structures such as index structures and algorithms such as sorting. These data structures and algorithms do not make any assumptions on the distribution of the data and give guarantees on the worst-case performance. However, in some cases, if we know certain properties about the distribution of the underlying data, it is possible to come up with data structures and algorithms that can yield superior performance. ML provides a flexible framework for learning the empirical data distribution and a recent line of research has shown the possibility of using ML models to replace core data structures and algorithms in DBMSs.

#### 5.3.1 Learned B-Tree Indices

Learned Index [26] is the first system to propose the idea of using ML models to replace core data structures and algorithms. It focussed mainly on replacing B-Tree indices. Given a key, what a B-Tree index essentially does is finding its position on a sorted list using a series of tree traversals. B-Trees also give a guarantee on the error on the selected position: the maximum error is bounded by the page size. In this sense, a B-Tree index is a model that captures the cumulative distribution function (CDF) of the key values. Thus, it is possible to replace the B-Tree index with an ML model that is trained to capture the CDF of the keys. The error guarantee of the ML model can be found during training, which is the maximum training error for any key. Interestingly, unlike other ML applications, the objective here is to minimize the training error and not the generalization error. The advantage of replacing a B-Tree index using an ML model is that it reduces both the lookup time and memory footprint of the index.

FITing-Tree [27] is another system that replaces B-Tree indices using learned ML models. It uses a set of piece-wise disjoint linear functions to approximate the CDF distribution of the keys and uses a conventional B-Tree index to map a key to the correct linear function. Using linear functions helps FITing-Tree achieve fast look-ups and also support inserts, which was one of the major limitations of the Learned Index system. It also provides a tunable knob to make the index optimize either for faster look-ups or smaller memory footprint.

XIndex [28] is a concurrent learned index that supports read, write, and update operations. The architecture of XIndex is similar to that of FITing-Tree. However, the root node in XIndex also uses a linear model, unlike the B-Tree index in the FITing-Tree. For enabling concurrent updates it uses a delta index and periodically runs a two-phase compaction scheme to merge and copy the delta index with the main index structure. Concurrency during this compaction stage is enabled through classical concurrency constructs such as the read-copy-update

barrier. In addition to this, XIndex also adapts its structure at runtime to optimize for the query distribution. During learning, both Learned Index and FITing-Tree try to minimize the worst-case error. In practice, most query workloads are highly skewed and their runtime performance will be determined by the performance on a small set of hotkeys. XIndex monitors the observed error during runtime and tries to split the regions that observe high error into multiple models to reduce the error.

### 5.3.2 Other Learned Data Structures and Algorithms

Learned Index system also proposed methods to replace several other DBMS data structures and algorithms using learned ML models such as learned hash maps, sorting, and bloom filters. One of the main issues with hash map structures is getting hash collisions which increases the latency of retrievals. An ML model that captures the CDF can be used to replace the hash function and thus reduce the number of hash collisions. The same CDF model can be used to sort the data more efficiently by first ordering the data in nearly sorted order and then using insertion/bubble sort. Bloom filters can be replaced by training a classification model for which the decision threshold is chosen such that the false-negative rate is zero. While these systems have shown initial promising results for the feasibility of replacing core data structures and algorithms using ML models, much work is still needed to make them available for enterprise DBMSs.

## 6. Design Choices

We discuss three main overarching design choices that one has to make when integrating ML components into DBMSs:

1) integration mode, 2) learning source and 3) choice of ML paradigm.

### 6.1 Integration Mode

We found that there are two main engineering approaches for integrating ML components into DBMSs: external vs. internal integration [29].

### 6.1.1 External Integration

Modern DBMSs are complex systems and they allow human database administrators to control the query execution performance by (1) optimizing the physical database design, (2) providing query optimization hints, (3) knob tuning, and (4) resource provisioning. They also provide information about the system such as resource usage, query traces, and performance metrics. Under this context, the focus of externally integrated ML components is to provide recommendations to human database administrators or replace them and automatically perform the tasks using the standard configuration endpoints provided by the DBMS.

An enterprise-grade DBMS typically requires decades of highly advanced software development efforts and thus there is huge resistance among DBMS developers to integrate new components that require significant architectural changes. External integration keeps the ML components outside the critical path of a DBMS and still provides a value. For this reason, most of the systems surveyed in this paper fall into this category (see Figure 1) and some have even been successfully adopted by several enterprise DBMSs.

However, external integration also faces several limitations. First developing multiple external components that operate on different sub-problems may lead to interference among the decisions taken by those systems. For example, assume an external query optimization component that hints a specific query plan to DBMS assuming the absence of a particular index. At the same time assume there is another physical database design component that decides to create this index which renders the chosen evaluation plan become sub-optimal. Avoiding this kind of interference requires co-ordination among different components, which is difficult to implement in external components. Second, external components for knob tuning and resource provisioning take an iterative approach where they try out several different settings before picking the best option. Existing DBMSs are not optimized for

such rapid experimentation and hence require system downtimes or restarts for the configurations to take effect. This significantly increases the time required for knob tuning by an ML component. Finally, the system information metrics provided by the DBMSs are primarily intended to be consumed by human database administrators for diagnosing performance bottlenecks. Thus, they can be too high- level for ML components to learn from.

### 6.1.2 Internal Integration

Internal integration of ML components tries to mitigate much of the above-mentioned limitations by changing the DBMS architecture to treat ML components as first-class components. As a result, ML components get more access to the low-level information and more fine-grained control to the DBMS. Coherence among the decisions taken by multiple ML components inside a DBMS can be achieved by having a centralized coordinator that takes suggested actions from different ML components and execute them only if they don't

interfere with other decisions. However, internal integration requires tight coupling between the components inside a DBMS and can pose query execution performance degradation when training the ML models. Hence, they are mostly applicable to new data system developments that are being developed from scratch (also called greenfield systems).

NoisePage [30] is one such green field system for in- memory hybrid transactional and analytic processing, which can automatically optimize query execution without a human database administrator. It focusses on knob tuning, resource provisioning, and physical database design optimization. It also has a modular architecture optimized for efficient offline training data collection by ML components. Training data for each module (e.g., transaction manager) can be obtained in isolation without the need of going through the entire DBMS execution path. These offline collected data is then combined with the data collected through online query execution to learn ML models. The ML pipeline in NoisePage has three main phases: 1) modeling, 2) planning, 3) deployment. In the modeling stage, it

builds models to predict the future query workload and models to predict the behavior of system components under different configuration values. In the planning stage, it uses reinforcement learning to pick actions based on the models trained in the modeling phase, instead of interacting with the actual system. Finally, in the deployment phase, the chosen actions are applied and the observed performance metrics are later fed back to modeling and planning models to improve their performance.

SageDB [31] is another system that proposes a novel DBMS architecture that uses ML models combined with program synthesis techniques to generate internal system components like data structures and algorithms. To balance the training time vs. accuracy it proposes using multiple ML models each specialized for a particular task. ML models in SageDB are optimized to capture the empirical data distribution of the data and not optimized for the ability to generalize to unseen data. These synthesized systems components are used for optimizing data access (e.g., indices), query optimization (e.g., cardinality estimation), and query execution (e.g., sorting).

### 6.2 Learning Source

The main goal of using ML for DBMS components is to improve the performance of the system for future query work-loads. Thus, one way for adopting ML methods is to learn from past or current (in the case of reinforcement learning) queries. But the performance of the queries is dependant on the state of the underlying data in the DBMS. Hence, in some cases, it is also possible to achieve the same goal by learning directly from the data.

#### 6.2.1 Learning from Queries

As shown in Figure 1, learning from query workloads is the most widely used approach for integrating ML into DBMS across all components. Learning from queries enables the ML models to learn a narrow-scoped problem which is much easier to model/learn and hence improve the overall performance of the system.

However, this approach faces three main challenges. First, collecting training data for this approach can

be expensive as each query needs to be executed on large databases. Second, it does not generalize well for unseen workload queries and causes significant performance degradation at execution time. Third, changes in the workload patterns or underlying data require capturing new training data and expensive retraining which can cause system downtime.

#### 6.2.2 Learning from Data

More recently several systems have been proposed that learn from DB-resident data, instead of query workloads. These systems train models to learn the empirical data distribution of the data and use them to improve DBMS performance. For example, DBEst [19] and DeepDB [9] uses probability distribution models to answer AQP queries. Naru [8] and DeepDB [9] use joint probability distribution models in the relational engine to optimize cardinality estimates. Empirical data distribution models are also the main building block in learned data structures and algorithms such as Learned Index [26], FITing-Tree [27], and XIndex [28].

ML models trained using data can be reused despite changes in the workload pattern and are also more robust to small changes in the data. More importantly, DeepDB [9] has shown that the same

probability distribution model can be used in multiple tasks such as AQP and cardinality estimation, reducing the total training time required. However, accurately capturing the joint probability distribution of a relational dataset with multiple tables is a complex learning task that requires models with high learning capacity and training time. Furthermore, not all DBMS components can be purely learned from data (e.g., execution engine knob tuning).

### 6.2.3 Hybrid Methods

While most of the existing systems fall into one of the above two approaches, it would be interesting to explore the possibility of combining both approaches. Some early work in this regard has been proposed in the XIndex [28] system. XIndex is a learned index structure that replaces B-tree indices. It does so by learning the empirical

cumulative distribution function of the keys of the data. During training, the goal is to minimize the maximum error made by the model for predicting the position of a key. But the performance of a particular workload will be dominated by the errors made by the model on the keys that are frequently used in the workload. Hence, XIndex performs the second phase of learning where it dynamically further minimizes the error on the most frequently used keys.

## 6.3 Choice of the ML Paradigm

We cover the use of different ML model and learning paradigms for integrating ML into DBMSs.

### 6.3.1 ML Model Family

We observed two prominent families of ML models: 1) deep learning and 2) classical ML.

**Deep Learning.** It should be noted that much of the recent renaissance in applying ML methods for DBMS internals, and also systems in general, has to be credited to the recent advancements in deep learning. Deep learning models have high model capacities and hence can learn highly complex data distributions. They have shown superior performance in hard tasks such as in natural language processing (NLP). As a result, the same deep NLP models have been used in systems like SpeakQL [2], SeqSQL [3], and SQLNet [4] to provide spoken and/or natural language interfaces for DBMSs. Naru [8] uses a transformer-based deep learning model to learn the joint probability distribution of the data for cardinality estimation. Neo [13], DQM [17], and iBTune [22] also use deep learning models for query optimization (QEP generation), physical database design (materialization optimization), and knob tuning (buffer size tuning), respectively.

However, the high model capacity of deep learning models and their ability to learn highly accurate models come at a cost. First, these models require significantly large amounts of training data without which the models will start to overfit. In many DBMS components, collecting large amounts of training data is expensive as the queries have to be executed potentially on large databases. Second,

deep learning models are highly compute-intensive which can require few hours to a few days of training even when using expensive hardware accelerators such as GPUs. This can cause degradation of DBMS query execution performance. Furthermore, typically deep learning inference times are in few hundreds of milliseconds and do not match with the performance requirements of the DBMS components such as cardinality estimators and indices which have to be in the order of few milliseconds. Finally, the explainability/debuggability of deep learning models is still an active area of research and there is minimal understanding of the internal workings of them. As a result, while deep learning-based methods have shown promising accuracy results, they are not yet widely integrated into enterprise DBMSs due to the above open challenges.

**Classical ML.** We found that classical ML-based methods are widely used to integrate ML into DBMS components and some of them (e.g., LEO [6], CardLearner [7], AutoAdmin [14], and VerdictDB [18]) have been even integrated in the enterprise systems. Classical ML methods overcome much of the limitations of deep learning methods: they are less compute-intensive to train, require much less training data, have faster inference times, and generate much easy to explain predictions. However, most classical ML models are known to have fewer model capacities and have less predictive power compared to deep learning models. Thus, their accuracy can be lower.

### 6.3.2 Learning Paradigm

We explain the use of three learning paradigms that systems have used to integrate ML into DBMSs: 1) supervised learning, 2) reinforcement learning, and 3) unsupervised learning.

**Supervised Learning.** Out of the systems we surveyed supervised learning approach is the most widely used learning paradigm (see Figure 1). The training data required for supervised learning is collected either beforehand or continuously during query execution. Seq2SQL [3] and SQLNet [4] use a large manually generated dataset of SQL and natural language query pairs. Most other systems

(e.g., CardLearner [7], QPPNet [32], Naru [8], AutoAdmin [14, 15], and Learned Index [26]) preform initial training from previously collected training data and then perform periodic retraining as new data becomes available or the query workload or the data in the DBMS significantly change. In some cases, collecting training data can be expensive as it requires executing a large number of queries potentially on large databases. However, the prevalence of cloud databases has mitigated these issues as cloud operators have access to large amounts of query execution traces from many tenants.

**Reinforcement Learning.** Reinforcement learning (RL) methods are particularly applicable when a system component has to make a series of decisions and the reward of each decision is not directly observable. Thus, RL methods have been used in tasks including QEP generation (e.g., SkinnerDB [10], ReJoin [12], Neo [13]), materialization optimization (e.g., DQM [17]) and scheduling (e.g., Bandit [24]). RL methods perform on the job training and collect data as the DBMS execute queries. Initially, they may generate worse results as the models have not converged yet. To overcome this, one could use a bootstrapping strategy called *learn by demonstration* where the existing DBMS component is used to generate initial training data for the RL model to bootstrap. After this initial training, the RL model will continue to learn on the job and become better than the existing DBMS component. For example, Neo [13] showed the feasibility of building an RL-based learned query optimizer which surpasses the PostgreSQL DBMS query optimizer, even though the RL model was initially bootstrapped using it.

**Unsupervised Learning.** We found that unsupervised learning techniques are widely used for DBMS knob tuning in systems like iTuned [20], OtterTune [21], and QB5000 [16]. One such popular technique is to reduce the number of different queries by performing clustering based on query templates. While DBMS may encounter a large number of different queries, most of them are different parameterizations of the same query template. Thus, by reducing the queries into query

templates the complexity for the ML model can be significantly reduced. iTuned [20] and OtterTune [21] also use Gaussian mixture models, another unsupervised learning method, to model the DBMS performance corresponding to different systems configurations settings.

## 7. Open Challenges

Integrating ML methods into DBMS components has proven to optimize average system performance. Some systems have already integrated ML into enterprise DBMSs [6, 7, 18]. However, the field is still in its infancy and requires solving many open challenges to realize the full potential. Next, we identify three such major open challenges:

### 7.1 Improving Robustness

While ML methods improve the average query execution performance, they can make mispredictions that are significantly off and lend the system become unrobust. On the contrary, traditional software components are designed to minimize the worst-case performance cost. Worst-case performance guarantees are a crucial aspect of software systems as one single fault can have ripple effects and make the entire system unusable eventually (e.g., the evaluation time difference between a good QEP and bad QEP can be orders of magnitude big). Understanding the worst-case behavior of ML-driven software components is still an untouched area and it is possible that coming up with tight theoretical guarantees is a very hard problem.

Another approach to solving the same problem would be to integrate adaptive query execution strategies with ML-driven components. This requires observing the outcome of the decisions taken by ML-driven components and dynamically adjusting them when a performance degradation is detected. Some initial work on this regard is

proposed in SkinnerDB [10]. SkinnerDB finds the best join ordering for a query by using an intra-query RL method that switches between different orders before it finds the optimal one. It also provides worst-case performance guarantees for this method. However, this space is still very open and much

work is needed to make ML-driven DBMS components more robust.

### 7.2 Rethinking the DBMS Architecture

When people designed DBMS architectures several decades ago, enabling autonomous control was not a design goal. As a result, when one integrates ML into DBMS components, they have to face several fundamental architectural limitations. For example, the separation of concerns such that the relational engine making all the intelligent decisions and the execution engine passively executing them no longer holds. Decisions taken by the relational engine when compiling the QEP may turn out to be wrong when executing it. Thus, the relational engine should be able to observe the performance of a QEP as it executes and refine the decisions as new in- formation becomes available. Such an approach will require tight-coupling between the relational and execution engines with feedback-loops.

Also, integrating ML into DBMS components requires the ability to easily experiment and having access to fine-grained system information. In current DBMSs, especially with external integration, it is very hard to profile and generate training data for a specific component without invoking the full QEP execution path, which is costly. Furthermore, the level of system information exposed by the DBMS is very coarse-grained. They are intended to be consumed by humans for debugging purposes and are too high-level for ML model training. Some of these limitations have been already identified and are being actively worked on [30].

Exploiting Transfer Learning

There is limited success in learning transferable knowledge that can be reused in multiple different settings. For example, most of the ML models used in existing DBMS components perform poorly when there is a deviation in the query work-load or a change in the system context or data. The situation is even worse when they are applied to a new DBMS instance or it is not possible to apply to a new instance at all. This significantly increases the cost of training and maintaining ML models and faces

problems like cold-start and the need to continuously retrain to keep up with the changes.

*Transfer Learning* is a technique that can be applied to overcome this limitation, which is popular in other fields such as computer vision and natural language processing. Instead of training separate models for different tasks from scratch, transfer learning enables us to reuse a master model and fine-tune it to the task at hand using limited resources (e.g., compute power and training data). This master model is trained on a very large dataset so that it can learn most of the relevant information for any related task. For example, *ImageNet* is a popular computer vision transfer learning dataset that has over 1 million hand-labeled images. Identifying and curating such a dataset for DBMSs require solving several open challenges. For example, one has to select a common representation format that can capture the data schema, data statistics, query structure, and hardware properties. Collecting such a large dataset is also a challenge. However, the migration of DBMSs into the cloud provides a unique opportunity to centrally collect all the relevant information.

## 8. Conclusion

Combining machine learning (ML) tasks with database management systems (DBMS) is an active research field and there have been many efforts exploring this both in research and industry. This combination is attractive because businesses have massive amounts of data in their existing DBMS and there is a high potential for using ML to extract valuable information from it [38]. In addition, the rich relational operators provided by the DBMS can be used conveniently to denormalize a complex schema for the purposes of ML tasks [39]. Although the field of machine learning is scaling heights, it is ridden with several limitations as well. The challenge for machine learning is to recover the discipline's original breadth of vision and its audacity to develop learning mechanisms that cover the full range of abilities observed in humans—who remain our only example of truly intelligent systems [40].

**References**


[1.] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT press, 2016.

[2.] Vraj Shah, Side Li, Arun Kumar, and Lawrence Saul. SpeakQL: Towards Speech-driven Multimodal Querying of Structured Data.

[3.] Victor Zhong, Caiming Xiong, and Richard Socher. Seq2sql: Generating Structured Queries from Natural Language Using Reinforcement Learning. *arXiv preprint arXiv:1709.00103*, 2017.

[4.] Xiaojun Xu, Chang Liu, and Dawn Song. SQLNet: Generating Structured Queries from Natural Language without Reinforcement Learning. *arXiv preprint arXiv:1711.04436*, 2017.

[5.] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. Query Optimization Through the Looking Glass, and What We Found Running the Join Order Benchmark. *The VLDB Journal*, 27(5):643–668, 2018.

[6.] Michael Stillger, Guy M Lohman, Volker Markl, and Mokhtar Kandil. LEO-DB2's learning optimizer. In *VLDB*, volume 1, pages 19–28, 2001.

[7.] Chenggang Wu, Alekh Jindal, Saeed Amizadeh, Hiren Patel, Wangchao Le, Shi Qiao, and Sriram Rao. Towards a Learning Optimizer for Shared Clouds. *Proceedings of the VLDB Endowment*, 12(3):210–222, 2018.

[8.] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Xi Chen, Pieter Abbeel, Joseph M Hellerstein, Sanjay Krishnan, and Ion Stoica. Deep Unsupervised Cardinality Estimation. *Proceedings of the VLDB Endowment*, 13(3):279–292, 2019.

[9.] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. DeepDB: Learn from Data, not from Queries! *Proceedings of the VLDB Endowment*, 13(7):992– 1005, 2020.

[10.] Immanuel Trummer, Junxiong Wang, Deepak Maram, Samuel Moseley, Saehan Jo, and Joseph Antonakakis. SkinnerDB: Regret-Bounded Query Evaluation via Reinforcement Learning. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1153–1170, 2019.

[11.] Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo Planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

[12.] Ryan Marcus and Olga Papaemmanouil. Deep Reinforcement Learning for Join Order Enumeration. In *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, pages 1–4, 2018.

[13.] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. Neo: A Learned Query Optimizer. *Proceedings of the VLDB Endowment*, 12(11):1705–1718, 2019.

[14.] Surajit Chaudhuri and Vivek Narasayya. Self-Tuning Database Systems: A Decade of Progress. In *Proceedings of the 33rd international conference on Very large data bases*, pages 3–14, 2007.

[15.] Bailu Ding, Sudipto Das, Ryan Marcus, Wentao Wu, Surajit Chaudhuri, and Vivek R Narasayya. AI meets AI: Leveraging Query Executions to Improve Index Recommendations. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1241–1258, 2019.

[16.] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J Gordon. Query-based Workload Forecasting for Self-Driving Database Management Systems. In Proceedings of the 2018 International Conference on Management of Data, pages 631–645, 2018.

[17.] Xi Liang, Aaron J Elmore, and Sanjay Krishnan. Opportunistic View Materialization with Deep Reinforcement Learning. arXiv preprint arXiv:1903.01363, 2019.

[18.] Yongjoo Park, Ahmad Shahab Tajik, Michael Cafarella, and Barzan Mozafari. Database Learning: Toward a Database that Becomes.

Smarter Every Time. In Proceedings of the 2017 ACM International Conference on Management of Data, pages 587–602, 2017.

[19.] Qingzhi Ma and Peter Triantafillou. DBEst: Revisiting Approximate Query Processing Engines with Machine Learning Models. In Proceedings of the 2019 International Conference on Management of Data, pages 1553–1570, 2019.

[20.] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. Tuning Database Configuration Parameters with iTuned. Proceedings of the VLDB Endowment, 2(1):1246–1257, 2009.

[21.] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. Automatic Database Management System Tuning through Large-Scale Machine Learning. In Proceedings of the 2017 ACM International Conference on Management of Data, pages 1009–1024, 2017.

[22.] Jian Tan, Tieying Zhang, Feifei Li, Jie Chen, Qixing Zheng, Ping Zhang, Honglin Qiao, Yue Shi, Wei Cao, and Rui Zhang. iBTune: Individualized Buffer Tuning for Large-Scale Cloud Databases. Proceedings of the VLDB Endowment, 12(10):1221–1234, 2019.

[23.] Ryan Marcus and Olga Papaemmanouil. WiSeDB: A Learning-Based Workload Management Advisor for Cloud Databases. Proc. VLDB Endow., 9(10):780–791, June 2016.

[24.] Ryan Marcus and Olga Papaemmanouil. Releasing Cloud Databases for the Chains of Performance Prediction Models. In CIDR, 2017.

[25.] perfenforce demonstration: Data analytics with performance guarantees.

[26.] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. The Case for Learned Index Structures. In Proceedings of the 2018 International Conference on Management of Data, pages 489–504, 2018.

[27.] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. FITing-Tree: A Data-aware Index Structure. In Proceedings of the 2019 International Conference on Management of Data, pages 1189–1206, 2019.

[28.] Chuzhe Tang, Youyun Wang, Zhiyuan Dong, Gansen Hu, Zhaoguo Wang, Minjie Wang, and Haibo Chen. XIndex: A Scalable Learned Index for Multicore Data Storage. In Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 308–320, 2020.

[29.] Andrew Pavlo, Matthew Butrovich, Ananya Joshi, Lin Ma, Prashanth Menon, Dana Van Aken, Lisa Lee, and Ruslan Salakhutdinov. External vs. Internal: An Essay on Machine Learning Agents for Autonomous Database Management Systems. IEEE Data Engineering, 11:1910–1913, 2019.

[30.] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C Mowry, Matthew Perron, Ian Quah, et al. Self-Driving Database Management Systems. In CIDR, volume 4, page 1, 2017.

[31.] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H Chi, Jialin Ding, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. Sagedb: A Learned Database System. 2019.

[32.] Ryan Marcus and Olga Papaemmanouil. Plan-Structured Deep Neural Network Models for Query Performance Prediction. Proceedings of the VLDB Endowment, 12(11):1733–1746, 2019.

[33.] Mitchell, T. (1997). Machine Learning. McGraw Hill. p. 2. ISBN 978-0-07-042807-2.

[34.] O. Simeone, "A Very Brief Introduction to Machine Learning With Applications to Communication Systems," in IEEE Transactions on Cognitive Communications and Networking, vol. 4, no. 4, pp. 648-664, Dec. 2018, doi: 10.1109/TCCN.2018.2881442.

[35.] Alpaydin, Ethem (2010). Introduction to Machine Learning. London: The MIT Press. ISBN 978-0-262-01243-0. Retrieved 1 August 2020.

[36.] K. Kara, K. Eguro, C. Zhang, and G. Alonso. ColumnML: Column-Store Machine Learning with On-the-Fly Data Transformation. PVLDB, 12(4):348–361, 2018.

[37.] A. Kumar, J. Naughton, and J. M. Patel. Learning generalized linear models over normalized data. In Proceedings of the 2015 ACM SIGMOD International 360 Conference on Management of Data, pages 1969–1984. ACM, 2015.

[38.] Alonso, G., Istvan, Z., Kara, K., Owaida, M. and Sidler, D., 2019. doppioDB 1.0: Machine Learning inside a Relational Engine. IEEE Data Eng. Bull., 42(2), pp.19-31

[39.] A. Kumar, J. Naughton, and J. M. Patel, "Learning Generalized Linear Models Over Normalized Data," in SIGMOD'15.

[40.] Langley, Pat (2011). "The changing science of machine learning". Machine Learning. 82 (3): 275–279. doi:10.1007/s10994-011-5242-y.

.