Database Meets AI: A Survey

Xuanhe Zhou, Chengliang Chai, Guoliang Li, Ji Sun

Abstract—Database and Artificial Intelligence (AI) can benefit from each other. On one hand, AI can make database more intelligent (AI4DB). For example, traditional empirical database optimization techniques (e.g., cost estimation, join order selection, knob tuning, index and view selection) cannot meet the high-performance requirement for large-scale database instances, various applications and diversified users, especially on the cloud. Fortunately, learning-based techniques can alleviate this problem. On the other hand, database techniques can optimize AI models (DB4AI). For example, AI is hard to deploy in real applications, because it requires developers to write complex codes and train complicated models. Database techniques can be used to reduce the complexity of using AI models, accelerate AI algorithms and provide AI capability inside databases. Thus both DB4AI and AI4DB have been extensively studied recently. In this paper, we review existing studies on AI4DB and DB4AI. For AI4DB, we review the techniques on learning-based configuration tuning, optimizer, index/view advisor, and security. For DB4AI, we review AI-oriented declarative language, AI-oriented data governance, training acceleration, and inference acceleration. Finally, we provide research challenges and future directions.

Index Terms—Database, Artificial Intelligence, DB4AI, AI4DB

1 Introduction

Artificial intelligence (AI) and database (DB) have been extensively studied over the last five decades. First, database systems have been widely used in many applications, because databases are easy to use by providing user-friendly declarative query paradigms and encapsulating complicated query optimization functions. Second, AI has recently made breakthroughs due to three driving forces: large-scale data, new algorithms and high computing power. Moreover, AI and database can benefit from each other. On one hand, AI can make database more intelligent (AI4DB). For example, traditional empirical database optimization techniques (e.g., cost estimation, join order selection, knob tuning, index and view selection) cannot meet the high-performance requirement for large-scale database instances, various applications and diversified users, especially on the cloud. Fortunately, learning-based techniques can alleviate this problem. For instance, deep learning can improve the quality of cost estimation and deep reinforcement learning can be used to tune database knobs. On the other hand, database techniques can optimize AI models (DB4AI). AI is hard to deploy in real applications, because it requires developers to write complex codes and train complicated models. Database techniques can be used to reduce the complexity of using AI models, accelerate AI algorithms and provide AI capability inside databases. Thus both DB4AI and AI4DB have been extensively studied recently.

1.1 AI for DB

Traditional database design is based on empirical methodologies and specifications, and requires human involvement (e.g., DBAs) to tune and maintain the databases [77], [17]. AI techniques are used to alleviate these limitations – exploring more design space than humans and replacing heuristics to address hard problems. We categorize existing techniques of using AI to optimize DB as below.

- *Xuanhe Zhou, Chengliang Chai, Guoliang Li, Ji Sun were with the Department of Computer Science, Tsinghua University, Beijing, China. Corresponding author: Guoliang Li, Chengliang Chai.*

Learning-based Database Configuration. (1) Knob tuning. Databases have hundreds of knobs and it requires DBAs to tune the knobs so as to adapt to different scenarios. Obviously, DBAs are not scalable to millions of database instances on cloud databases. Recently the database community attempts to utilize learning-based techniques [3], [76], [154] to automatically tune the knobs, which can explore more knob combination space and recommend high-quality knob values, thus achieving better results than DBAs. (2) Index/View advisor. Database indexes and views are fairly crucial to achieve high performance. However, traditional databases rely highly on DBAs to build and maintain indexes and views. As there are a huge number of column/table combinations, it is expensive to recommend and build appropriate indexes/views. Recently, there are some learning-based works that automatically recommend and maintain the indexes and views. (3) SQL Rewriter. Many SQL programmers cannot write high-quality SQLs and it requires to rewrite the SQL queries to improve the performance. For example, the nested queries will be rewritten into join queries to enable SQL optimization. Existing methods employ rule-based strategies, which uses some predefined rules to rewrite SQL queries. However, these rule-based methods rely on high-quality rules and cannot be scale to a large number of rules. Thus, deep reinforcing learning can be used to judiciously select the appropriate rules and apply the rules in a good order.

Learning-based Database Optimization. (1) Cardinality/-Cost Estimation. Database optimizer relies on cost and cardinality estimation to select an optimized plan, but traditional techniques cannot effectively capture the correlations between different columns/tables and thus cannot provide high-quality estimation. Recently deep learning based techniques [63], [107] are proposed to estimate the cost and cardinality which can achieve better results, by using deep neural networks to capture the correlations. (2) Join order selection. A SQL query may have millions, even billions of possible plans and it is very important to efficiently

find a good plan. Traditional database optimizers cannot find good plans for dozens of tables, because it is rather expensive to explore the huge plan space. Thus there are some deep reinforcement learning based methods to automatically select good plans. (3) End-to-end Optimizer. A full-fledged optimizer not only replies on cost/cardinality estimation and join order, but also requires to consider indexes and views, and it is important to design an end-to-end optimizer. Learning-based optimizers [97], [143] use deep neural networks to optimize SQL queries.

Learning-based Database Design. Traditional database is designed by database architects based on their experiences, but database architects can only explore a limited number of possible design spaces. Recently some learning-based self-design techniques are proposed. (1) Learned indexes [65] are proposed for not only reducing the index size but also improving the indexing performance. (2) Learned data structure design [53]. Different data structures may be suit for different environments (e.g., different hardware, different read-/write applications) and it is hard to design an appropriate structure for every scenario. Data structure alchemy [53] is proposed to create a data inference engine for different data structures, which can be used to recommend and design data structures. (3) Learning-based Transaction Management. Traditional transaction management techniques focus on transaction protocols, e.g., OCC, PCC, MVCC, 2PC. Recently, some studies try to utilize AI techniques to predict the transactions and schedule the transactions [89], [124]. They learn from existing data patterns, efficiently predict future workload trend and effectively schedule them by balancing the conflict rates and concurrency.

Learning-based Database Monitoring. Database monitoring can capture database runtime metrics, e.g., read/write latency, CPU/memory usage, and thus can remind administrators when anomalies happen (e.g., performance slowdown and database attacks). However, traditional monitoring methods rely on database administrators to monitor most database activities and report the problems, which is incomplete and inefficient. Therefore, machine learning based techniques are proposed to optimize database monitoring [61], [40], which determine when and how to monitor which database metrics.

Learning-based Database Security. Traditional database security techniques, e.g., data masking and auditing, rely on user-defined rules, but cannot automatically detect the unknown security vulnerabilities. Thus AI based algorithms are proposed to discover sensitive data, detect anomaly [83], conduct access control [39], and avoid SQL injection [132], [153]. (1) Sensitive data discovery is to automatically identify sensitive data using machine learning. (2) Anomaly detection is to monitor database activities and detect vulnerabilities. (3) Access control is to avoid data leak by automatically estimate different data access actions. (4) SQL injection is to mine user behavior and identify SQL injection attacks with deep learning.

1.2 DB for AI

Although AI can address many real-world problems, there is no widely deployed AI system that can be used in different fields as popular as DBMS, because existing AI systems have poor replicability and are hard to be used by ordinary users. To address this problem, database techniques can be used to lower the barrier of using AI.

Declarative Query Paradigm. SQL is relatively easy to be used and widely accepted in database systems. However, SQL lacks some complex processing patterns (e.g., iterative training) compared with other high-level machine learning languages. Fortunately, SQL can be extended to support AI models [118], and we can also design user-friendly tools to support AI models in SQL statements [33].

Data Governance. Data quality is important for machine learning, and data governance can improve the data quality, which includes data discovery, data cleaning, data integration, data labeling, data lineage. (1) Data discovery. Learning based data discovery enhances the ability of finding relevant data, which effectively finds out relevant data among a large number of data sources [34]. (2) Data cleaning. Dirty or inconsistent data can affect the training performance terribly. Data cleaning and integration techniques can detect and repair the dirty data, and integrate the data from multiple sources to generate high-quality data [142]. (3) Data labeling. With domain experts, crowdsourcing [75] and knowledge base [99], we can properly utilize manpower or existing knowledge to label a large number of training data for ML algorithms. (4) Data lineage. Data lineage depicts the relationship between input and output and is important to ensure ML models working properly. With database technologies like join and graph mapping, we can trace data relationships backwardly and forwardly [18], [20].

Model Training. Model training aims to train a good model that will be used for online inference. Model training is a time consuming and complicated process, and thus it requires optimization techniques, including feature selection, model selection, model management and hardware acceleration. (1) Feature selection. It aims to select appropriate features from a large number of possible features. It is time consuming to select and evaluate the possible features. Therefore, technologies like batching, materialization [152] are proposed to address this issue. (2) Model selection. It aims to select an appropriate model (and parameter values) from a large number of possible models. Thus, some parallelism techniques are proposed to accelerate this step, including task parallel [101], bulk synchronous parallel [66], parameter server [79] and model hop parallelism [104]. (3) Model management. Since model training is a trial-and-error process that needs to maintain many models and parameters that have been tried, it is necessary to design a model management system to track, store and search the ML models. We review GUI-based [9] and command-based model [137] management system. (4) Hardware acceleration. Hardwares, like GPU and FPGA, are also utilized to accelerate the model training. We introduce hardware acceleration techniques in row-store [92] and column-store [62] databases respectively.

Model Inference. Model inference aims to effectively infer the results using a trained model, and in-database optimization techniques include operator support, operator selection, and execution acceleration. (1) Operator support. An ML model may contain different types of operators (e.g., scalar, tensor), which have different optimization requirements.

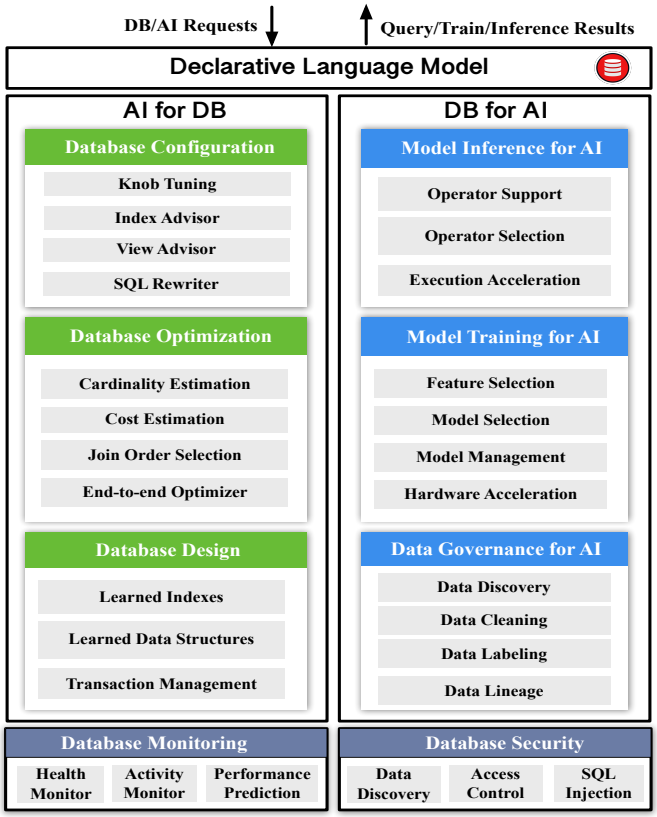

Fig. 1: The Overview of DB Meets AI

Thus some in-database techniques are proposed to support AI operators, including scalar operations [49], tensor operations [12], and tensor partitions [49]. (2) Operator selection. The same ML model can be converted to different physical operators, which may bring significant performance difference. In-database operator selection can estimate resource consumption and judiciously schedule the operators [12]. (3) Execution acceleration. Inference acceleration aims to enhance the execution efficiency. On one hand, in-memory databases compress sample/model data inside memory and conduct in-memory optimization [71]. On the other hand, distributed databases enhance execution efficiency by issuing tasks to different nodes [115] .

1.3 Contributions

We make the following contributions (see Figure 1).
(1) We review AI4DB techniques that utilize AI techniques to optimize database, including learning-based configuration tuning, optimizer, and index/view advisor (see Section 2).
(2) We review DB4AI techniques that utilize DB techniques to make AI easy-to-use, improve model performance and accelerate AI algorithm, including declarative language, data governance, model training and inference (see Section 3).
(3) We provide research challenges and future directions, including hybrid DB and AI data model, co-optimization, and hybrid DB and AI system (see Section 4).

2 AI For DB

AI techniques can be utilized to optimize databases from many aspects, including (1) learning-based database configuration, (2) learning-based database optimization, (3) learning-based database design, (4) learning-based database monitoring and protection, (5) learning-based security. This section reviews existing studies from these aspects.

2.1 Learning-based Database Configuration

Learning-based database configuration aims to utilize machine learning techniques to automate database configurations, e.g., knob tuning, index advisor, and view advisor.

2.1.1 Knob Tuning

Databases and big data analytics systems have hundreds of tunable system knobs (e.g., `Work_Mem`, `Max_Connections`, `Active_Statements`) [86], which control many important aspects of databases (e.g., memory allocation, I/O control, logging). Traditional manual methods leverage DBAs to manually tune these knobs based on their experiences but they always spend too much time to tune the knobs (several days to weeks) and cannot handle millions of database instances on cloud databases. To address this issue, self-tuning is proposed to automatically tune these knobs, which uses ML techniques to not only achieve higher tuning performance but less tuning time (thus saving the human efforts and enabling online tuning). We can categorize existing knob tuning techniques into four categories, including search-based tuning, traditional ML-based tuning, deep learning based tuning and reinforcement learning based tuning.

Search-based Tuning. To reduce the manpower, Zhu et al [157] propose a recursive bound-and-search tuning method *BestConfig*, which, given a query workload, finds the similar workloads from historical data and returns the corresponding knob values. Specifically, given n knobs, BestConfig divides the value range of each knob into k intervals and these knob intervals form a discrete space with k^n subspaces (bounded space). And then, in each iteration, *BestConfig* randomly selects k samples from the bounded space, and selects the sample with the best performance from the k selected samples, denoted as C_1. In the next iteration, it only gets samples from the bounded space closed to C_1. In this way, BestConfig iteratively reduces the bounded space and finally gets a good knob combination. However, this search-based method has several limitations. First, it is heuristic and may not find optimal knob values in limited time. Second, it cannot achieve high performance, because it needs to search the whole space.

Traditional ML-based Tuning. To solve the problems in the search-based tuning method, traditional ML models are proposed to automatically tune the knobs (e.g., Gaussian process [3], decision tree [36]). Aken et al propose an ML-based database tuning system *OtterTune* [3]. *OtterTune* uses Gaussian Process (GP) to recommend suitable knobs for different workloads. First, it selects some query templates, where each query template contains a query workload and its corresponding appropriate knob values. Second, it extracts internal state of the database (e.g., the number of pages read/written, the utilization of query cache) to reflect workload characteristics. From internal state features, *OtterTune* filters the irrelevant features using factor analysis, and then uses simple unsupervised learning methods (e.g., K-means) to select K features which are most related to

the tuning problem. And *OtterTune* uses these K features to profile the workload characters. Third, it uses these selected features to map the current workload to the most similar template. *OtterTune* directly recommends the knob configuration of this template as the optimal configuration. And it also inputs the query workload into the GP model to learn a new configuration for updating the model. Formally, it trains the model as follows. Given a training data (W, W', C', R), where W is a workload, W' is the similar workload template of W, C' is the recommended configuration of W', C'' is the recommend configuration by the GP model, and R is the performance difference between C' and C''. It trains the model by minimizing the difference between C' and C''. *OtterTune* uses these samples to train the model.

This ML-based method has good generalization ability and can perform well in different database environments. Moreover, it can make effective use of the experiences learned from the historical tasks and apply the experiences to future inference and training. However, it also has some limitations. Firstly, it adopts a pipeline architecture. The optimal solution obtained in the current stage is not guaranteed to be optimal in the next stage, and the models used in different stages may not be generalized well. Secondly, it requires a large number of high-quality samples for training, which are difficult to obtain. For example, database performance is affected by many factors (e.g., disk capacity, CPU status, workload features), and it is difficult to reproduce similar scenarios due to the huge search space. Thirdly, it cannot effectively support knob tuning with high-dimensional and continuous space. *OtterTune* still needs to filter out most knobs before utilizing the GP. For example, it only tunes 10 knobs on PostgreSQL.

Reinforcement Learning for Tuning. Reinforcement learning (RL) improves the generalization ability through continuous interactions with the environment (e.g., database state and workload). Zhang et al [154] propose a DRL-based database tuning system *CDBTune*. The main challenge of using DRL in knob tuning is to design the five modules in DRL. CDBTune maps the database tuning problem into the five modules in the reinforcement learning framework as follows. It takes the cloud database instance as the $Environment$, internal metrics of the instance as the $State$, the tuning model as the $Agent$, the knob tuning as the $Action$, and the performance change after tuning as the $Reward$. The $Agent$ adopts a neural network (the `Actor`) as the tuning strategy which takes as input the metrics from $State$ and outputs the knob values. Besides $Agent$ adopts another neural network (the `Critic`) for tuning the Actor, which takes as input the knob values and internal metrics and outputs the Reward. In a tuning procedure, the $Agent$ inputs the $state$ of the database instance. Based on the state, $Agent$ outputs a tuning action and applies it on the database instance. And then, it executes the workload on the database, gets the performance changes, and uses the change as the reward to update the `Critic` in the $Agent$. Next the `Critic` updates `Actor`, which captures the relationships between knob values and internal metrics.

Unlike traditional supervised learning, the training of reinforcement learning model does not require extensive high-quality data. Through the trial-and-error process, the learner in DRL model repeatedly generates (s_t, r_t, a_t, s_{t+1}), where s_t is the database state at time t, r_t is the reward at time t, and a_t is the action at time t, and uses the action to optimize the tuning strategy. With exploration and exploitation mechanisms, reinforcement learning can make a trade-off between exploring unexplored space and exploiting existing knowledge. Give an online workload, *CDBTune* runs the workload and obtains the metrics from the databases. Then *CDBTune* uses the `Actor` model to recommend the knob values.

However, there are still some limitations in *CDBTune*. First, *CDBTune* can only provide a coarse-grained tuning, e.g., tuning knobs for a workload, but cannot provide a fine-grained tuning, e.g., tuning knobs for some specific queries. Second, previous DRL-based tuning works directly use existing models (e.g., Q-learning [141], DDPG [82]); however Q-learning cannot support input/output with continuous values, and in DDPG, the agent tunes database only based on database state, without considering the workload features. Thus Li et al. [76] propose a query-aware tuning system *QTune* to solve the above problems. First, *QTune* uses a double-state deep reinforcement learning model (DS-DRL), which can embed workload characteristics and consider the effects of both the action (for inner state) and the workload (for outer metrics). Therefore, it can provide finer-grained tuning and can support online tuning even if workload changes. Second, *QTune* clusters the queries based on their "best" knob values and supports cluster-level tuning, which recommends the same knob values for the queries in the same cluster and different knob values for queries in different clusters.

In conclusion, DRL-based methods can greatly improve the tuning performance compared with traditional methods. Firstly, DRL does not need a lot of training data to train the model because it can generate training samples by iteratively running a workload under different database states. Secondly, it combines advanced learning methods (e.g., Markov Decision Process, Bellman function and gradient descent), and thus can adapt to database changes efficiently.

Deep Learning for Buffer Size Tuning. Above methods focus on tuning general knobs. There are also some methods that tune specific knobs to optimize database performance. Tan et al. [130] propose *iBTune*, which only tunes buffer pool size for individual database instances. It uses deep learning to decide the time to tune databases to ensure minimum negative effects on query latency and throughput. First, it collects samples of database state metrics (e.g., miss ratio), tuning action, and performance from the history records. Second, it uses these samples to train a pairwise neural network to predict the upper bounds of the latency. The input of the network includes two parts: current database instance and target database instance (after tuning buffer pool size), each of which includes the performance metrics (e.g., CPU usage, read I/O), and the encoding of current time (e.g., 21:59). And the output of the network is the predicted response time. If the predicted time is too high, *iBTune* will tune the memory size. *iBTune* achieves a balance between tuning performance and tuning frequency. Kunjir et al [72] propose a multi-level tuning method *RelM* for memory allocation. Different from *iBTune*, *RelM* first uses

TABLE 1: Methods of database tuning

Method	Tuning Time	Training Time	Quality	Adaptivity (workload)	Adaptivity (hardware)
Manual tuning	Days to Weeks	Months (train DBAs)	Medium -	×	×
Search-based [157]	Hours	Hours	Medium -	×	×
Traditional ML [3]	Seconds	Hours	High -	√	×
Reinforcement learning [154], [76], [72]	Milliseconds	Hours to Days	High	√	√
Deep learning [130]	Milliseconds	Hours to Days	Medium +	√	√

Guided Bayesian Optimization to compute the performance metric values (e.g. shuffle memory usage) based on the application types (e.g., SQL, shuffling). *RelM* uses these metrics to select knobs and uses DDPG to tune these selected knobs.

ML-based Tuning for Other Systems. Besides database tuning, there are many tuning methods for other data systems like Hadoop [50], [60] and Spark [105], [41], [38], [138]. Big data systems also have many knobs (e.g., >190 in Hadoop, $>$ 200 in Spark), which can have a significant impact on the performance [86]. However, tuning for big data systems is different from database tuning. (1) Big data systems like Spark have more diverse workloads (e.g., machine learning tasks on *MLlib*, real-time analytics on Spark streaming, graph processing tasks on *GraphX*), and they need to tune the systems for different tasks and environments. (2) Existing database tuning methods are mainly on single nodes, while most big data systems adopt a distributed architecture. (3) The performance metrics are different – big data systems focus on resource consumption and load balance, while database tuning optimizes throughput and latency.

Herodotou et al [50] propose a self-tuning system *Starfish* for Hadoop. For job-level tuning, *Starfish* captures online features while running the job (e.g., job, data, and cluster characters) and tunes the parameters based on estimated resource consumption (e.g., time, CPU, memory). For workflow-level tuning, a workflow can be distributed to different cluster nodes. So *Starfish* learns policies to schedule data across nodes based on both the local features and distributed system features. For workload-level tuning, *Starfish* mainly tunes system parameters like node numbers, node configurations according to the access patterns of those workflows and the network configurations. To tune these parameters, it simulates the actual working scenarios, and estimates the database performance after tuning based on the execution cost predicted by the database optimizer.

For Spark tuning, Gu et al [105] propose a machine learning based model to tune configurations of applications running on Spark. First, it uses N neural networks, each of which takes as input the default values of parameters and outputs a recommended parameter combination. Second, it uses a random forest model to estimate the performance of the N recommended combinations and chooses the optimal one to actually tune the configuration. This method is extremely useful for distributed cluster, where each network can be tuned based on the state of each node.

2.1.2 Index Selection

In DBMS, indexes are vital to speed up query execution, and it is very important to select appropriate indexes to achieve high performance. We first define index selection problem. Considering a set of tables, let C denote the set of columns in these tables and $size(c \in C)$ denote the index size of a column $c \in C$. Given a query workload Q, let $benefit(q \in Q, c \in C)$ denote the benefit of building an index on column c for query q, i.e., the benefit of executing query q with and without the index on column c. Given a space budget B, the index selection problem aims to find a subset C' of columns to build the index in order to maximize the benefit while keeping the total index size within B, i.e., $maximize \sum_{q \in Q, c \in C'} benefit(q, c)$, such that $\sum_{c \in C'} size(c) \leq B$.

There are several challenges. The first is how to estimate $benefit(q, c)$, and a well-known method is to use a Hypothetical Index [1], which adds the index information to the data dictionary of DBMS rather than creating the actual index, so that the query optimizer in DBMS can recognize the existence of the index and estimate the cost of query execution without building the real index. The estimated benefit of an index is the decrease of the estimated cost of query execution with and without the hypothetical index. The second is to solve the above optimization problem. There are two main solutions to address these challenges – offline index selection and online index selection. Offline index selection methods require DBAs to provide a representative workload and select an index scheme by analyzing this workload. Online index selection methods monitor the DBMS and select index scheme according to the change of workloads. The main difference between offline methods and online methods is that offline methods only select and materialize an index plan, while online methods dynamically create or delete some indexes according to the change of workloads.

Offline Index Selection. It relies on DBAs to choose some frequent queries from the query log as the representative workload, and uses the workload to select indexes. Chaudhuri et al propose an index selection tool *AutoAdmin* [15] for Microsoft SQL server. The main idea is to select well-performed index schemes for each query and then extend to multiple queries in Q. First, for each query $q_i \in Q$, *AutoAdmin* extracts indexable columns from the SQL queries. Second, *AutoAdmin* uses a naive enumeration algorithm to enumerate a collection of indexable columns as candidates, e.g., $\mathcal{I} = \{\{i_1, i_2, i_3, i_4\}, \{i_3, i_4, i_5, i_6\}, ...\}$. Then *AutoAdmin* selects the index scheme in $\mathcal{I}$ with the highest benefit for this query. Third, for each query, it has a corresponding optimal indexing strategy, and then for all queries in Q, *AutoAdmin* selects top-k schemes based on the benefit. Then for each top-k scheme, *AutoAdmin* uses a greedy algorithm to incrementally add indexable columns until the size is equal to a threshold B. Finally, the scheme with the highest benefit and within the storage budget will be selected. Zilio et al [136] model the index selection problem as a knapsack problem and propose DB2 Advisor. Similar to *AutoAdmin*, DB2 Advisor first enumerates index schemes with their benefit. Then, it models the index selection problem as a knapsack problem. More specifically, it regards each candi-

1. https://hypopg.readthedocs.io

date index scheme as an item, the size of the scheme as the item weight and the benefit of the scheme as the value. Then DB2 uses dynamic programming to solve this problem.

The downside of offline methods is that they are not flexible enough to handle the dynamic changes of workloads. Even worse, selecting a representative workload will increase the burden of the DBAs. To address these problems, some online methods are proposed as follows.

Online Index Selection. There are many online index selection methods, which can be divided into three categories: traditional online index selection methods, semi-automatic index selection method and ML-based index selection method. Traditional online index selection methods continuously analyze the workload and update the index scheme on-the-fly according to the change of workloads. Luhring et al. present a soft index autonomous management method based on the "observation-prediction-reaction" cycle [88]. First, it extracts indexable columns from queries and enumerates candidate index schemes. Then, it uses a greedy policy to select schemes that have the highest estimated benefit and adds them into the final result. Finally, the selected index scheme will be materialized when the workload of DBMS is not heavy. Schnaitte et al. propose COLT [120], which supports automatically online implementation of new indexes based on current index scheme. It models the index selection problem as a knapsack problem as we described in offline index selection (DB2) and applies dynamic programming to get an index scheme. Once the final index scheme is derived, it will be immediately materialized. However, traditional online methods do not take the experiences from DBAs into account. Also, continuous change of the index scheme may affect the stability of the DBMS and result in a high overhead.

Schnaitter proposed a semi-automatic index selection algorithm *WFIT* [121]. *WFIT* also works online and takes the feedback from DBAs into consideration. It monitors the DBMS in real time, dynamically analyzes the workload and enumerates some candidate schemes to tune the index structure. But before implementing the index schemes, *WFIT* needs DBAs to judge whether a column should be indexed or not. Then in the subsequent index selection process, *WFIT* will eliminate the column which should not be indexed from the index schemes according to DBAs experiences. Similarly, it can also add the column that should be indexed into the index schemes. Compared with traditional methods, *WFIT* does not need to select a representative workload and thus reduces the DBAs' burden. Though semi-automatic index selection methods take the experience from DBAs into consideration, these experience may not be useful. ML-based index selection methods can automatically learn some experience rather than DBAs' feedback and apply these experience to validate whether an index is useful.

ML-based index selection methods automatically learn experience from historical data rather than DBAs' feedback. Pedrozo et al. propose an index selection method *ITLCS* [110] based on a learning classifier system (LCS) as well as a genetic algorithm. First, *ITLCS* uses LCS to generate indexing rules in column level. Each rule consists of two parts: (i) index-related information from DBAs, e.g., "the percentage of null tuples in the column", "the data type in the column"; (ii) an action denotes whether to create or delete an index. Second, *ITLCS* uses a genetic algorithm to eliminate LCS rules and generates composited rules as the final indexing strategy. However, it is hard to generate the rules. Sadri et al [117] propose a reinforcement-learning-based index selection method. First, without expert rules, they denote workload features as the arrival rate of queries, column features as the access frequency and selectivity of each column. Second, they use the Markov Decision Process model (MDP) to learn from features of queries, columns, and outputs a set of actions, which denote creating/dropping an index.

TABLE 2: Methods of index advisor

Method	Scenario	Quality	Adaptability
Offline Index Selection [15], [136]	Static data	Low	Low
Online Index Selection [88], [120]	Dynamic data	Medium	Medium
Semi-Automatic Index Selection [121]	Dynamic data	High	Medium
ML-Based Index Selection [110], [117]	Dynamic data	High	High

2.1.3 View Advisor

View materialization is rather important in DBMS that utilizes views to improve the query performance based on the space-for-time trade-off principle. Judiciously selecting materialized views can significantly improve the query performance within an acceptable overhead. However, it is hard to automatically generate materialized views for ordinary users. Existing methods rely on DBAs to generate and maintain materialized views. Unfortunately, even DBAs cannot handle large-scale databases, especially cloud databases that have millions of database instances and support millions of users. Thus, it calls for the view advisor, which automatically identifies the appropriate views for a given query workload.

View advisor has been studied for many years and modern DBMSs such as Oracle DB and IBM DB2 provide tools to support materialized views [22], [158], [159]. There are two main tasks in view advisor: (1) Candidate view generation. View advisor should discover candidate views from the history workloads. (2) View selection. Since materializing all candidate views is impossible due to the system resource constraints, view advisor selects a subset of candidates as materialized views that can improve the performance most.

Candidate View Generation. Since the number of all possible views grows exponentially, the goal of view generation is to generate a set of high-quality candidate views that can be utilized for future queries. There are mainly two methods.

(1) Identify equivalent sub-queries that appear frequently in the workload. Dokeroglu et al. [25] propose a heuristic method which includes Branch-and-Bound, Genetic, HillClimbing, and Hybrid Genetic-Hill Climbing algorithms to find a near-optimal multiple queries execution plan in which queries share common computation tasks. They decompose queries into sub-queries which contain selection, projection, join, sorting, and data shipping. Next, to make these sub-queries have more chances to share common tasks, they generate alternative query plans for each query. Their plan generator interacts with a cost model

TABLE 3: Methods of view advisor

Method	Core Techniques	Quality	Scalability	Generalization Ability	Adaptability	On/Offline
Hybrid-GHCA [25]	Query Plan Graph, Heuristics	Medium	Medium	High	Medium	Online
IBM DB2 UDB [159]	Rule based rewriting, Greedy	Medium	High	High	Medium	Offline
BIGSUBS [58]	ILP, Probabilistic	Medium	High	High	High	Offline
CloudViews [59]	Query Plan Graph	High	High	High	Medium	Online
Wide-Deep [151]	RL	High	Low	Low	High	Online

that estimates the query cost by considering the potential reuse ability of its sub-queries. Then they detect common tasks among these alternative plans and search for a global plan with lower cost where queries share common tasks.

(2) Rewrite sub-queries to make a view answer more queries. Sub-queries directly extracted from the queries are still not general enough, because they cannot be generalized to other queries even with minor difference. To alleviate this problem, Zilio et al. [159] generalize extracted views by merging similar views, changing the select condition, adding "group by" clause. This reduces the resource consumption and more queries can utilize the views.

View Selection. Candidate views cannot be all materialized due to the system resource restriction. Therefore, the view advisor should choose a subset of views that lead to good performance to materialize and maintain them dynamically. The view selection problem aims to select a subset of candidates as materialized views. Assume that there is a workload and some candidate views, denoted as (Q, V), where $Q = \{q\}$ is the set of queries in the workload and $V = \{v\}$ is the set of candidate views. V' is the subset of views that are selected to be materialized. $C(q, V')$ is the cost of answering q in the presence of a set of materialized views V'. $M(v)$ is the maintenance cost of materialized view v. $|v|$ is the size that the view v will occupy in the disk space. τ is the disk space constraint such that the sum size of selected views cannot exceed. The goal of the view selection problem is to select a subset of views to be materialized to minimize the cost of answering the workload while not exceeding the disk space constraint.

$$\arg\min_{V' \subseteq V} \sum_{q \in Q} C(q, V') + \sum_{v \in V'} M(v), s.t., \sum_{v \in V'} |v| \leq \tau$$

We need to estimate the benefit and cost of a view and the view select problem is known to be NP-hard [29] due to the complex system environment constraints and the interactions between views. Since this is an NP-hard problem, there are many heuristics to obtain a near-optimal solution. Jindal et al. [58] solve the view selection problem as an integer linear programming (ILP) problem with an alternative formulation of Equation. 1. The goal is to minimize the cost of answering the workload by selecting a subset of views, V', to be materialized. Also, they study the problem that given each query q, which materialized views $V'_q \subseteq V'$ should be selected to answer the query. But solving this ILP problem is very time-consuming, so they split it into multiple smaller ILP problems and propose an approximate algorithm, *BIGSUBS*. The algorithm uses an iterative approach with two steps in each iteration. At each iteration, the first step is to decide whether a view should be materialized in current solution by a probabilistic method, which is similar to the mutation operation in the genetic algorithm. The probability is computed based on: (1) how much disk space the view occupies, and (2) the utility of the view, which is the estimated reduction of query cost when using this view. After temporally deciding the view selection state, the second step is to solve the remained smaller ILP problem by an ILP solver. In the second step, given each query, the algorithm will decide which view to be used to answer the query, and estimate the utility of each selected view. Then the utility will be fed back to the next iteration to select the candidate views until the selected views as well as their corresponding queries do not vary or the number of iterations is up to a threshold. The time complexity of solving the smaller ILP problems is more acceptable and the solution is near-optimal.

However, the above method is an offline method and takes hours to analyze a workload. For supporting online workloads, they propose an online computation reuse framework, *CLOUDVIEWS* [59]. The workflow is similar to [58]. It also first generates candidate views, and then given a query, studies how to select an appropriate view for the query. *CLOUDVIEWS* decomposes the query plan graph into multiple sub-graphs and chooses high-frequent sub-graphs as candidate views. Then it estimates the utility and cost of the candidate views online by collecting and analyzing the information from the previously executed workload, such as compile-time and run-time statistics. Given the candidate views and their estimated utility and cost, they convert the view selection problem to a packing problem, where the utility is regarded as the value of a view, the cost is regarded as the weight of a view, and the disk space constraint is regarded as the capacity. To make the view selection problem efficient, they also propose some heuristic algorithms. Suppose u_i is the utility of v_i and $|v_i|$ is the cost of v_i. They select k views with the highest score. The score can be either u_i or $\frac{u_i}{|v_i|}$. However, the downside is that they focus on recurring workloads, which means it can not be adapted rapidly to new workloads that have different query patterns and distributions.

To address these limitations, Yuan et al [151] propose to use reinforcement learning (RL) to address view selection and maintenance problems. First, they use a *wide-deep model* to estimate the benefit of using a materialized view to answer a query. Second, they model the view selection as an Integer Linear Programming (ILP) problem and use reinforcement learning to find the best policy.

Some materialized views may need to be evicted when a new one is ready to be created but there is not enough disk budget. For view eviction, a credit-based model is proposed [81] that evicts materialized views with the lowest credit when the storage limit is reached. The credit of a view is the sum of its future utility and recreation cost. Higher credit means that we will sacrifice more utility if we evict the view but cost more when we keep it. It is similar to the *Reward* function in *DQM* and can be calculated in a

similar way. However, because they train the RL model with real runtime of workloads, it has the assumption that the database environment will not change. This results in an expensive cost of model training in the cold-start step.

2.1.4 SQL Rewrite

Many database users, especially cloud users, may not write high-quality SQL queries, and SQL rewriter aims to transform SQL queries to the equivalent forms (e.g., pushing down the filters, transforming nested queries to join queries), which can be efficiently executed in databases [94]. Most of existing SQL rewrite methods adopt rule-based techniques [94], [6], [16], which, given a set of query rewrite rules, find the rules that can apply to a query and use the rule to rewrite the query. However, it is costly to evaluate various combinations of rewriting operations, and traditional methods often fail into sub-optimization. Besides, rewriting rules are highly related to applications, and it is hard to efficiently identify rules on new scenarios. Hence, machine learning ca be used to optimize SQL rewrite from two aspects. (1) Rule selection: since there are many rewriting rules, a reinforcement model can be used to make the rewrite decision. At each step, the agent estimates the execution cost of different rewrite methods and selects the method with the lowest cost. The model iteratively generates a rewrite solution and updates its decision strategy according to the results. (2) Rule generation: according to the sets of rewrite rules of different scenarios, a LSTM model is used to learn the correlations between queries, compilers, hardware features and the corresponding rules. Then for a new scenario, the LSTM model captures features inside gate cells and predicts proper rewrite rules.

2.2 Learning-based Database Optimization

Learning-based database optimization aims to utilize the machine learning techniques to address the hard problems in database optimization, e.g., cost/cardinality estimation, join order selection and end-to-end optimizer.

2.2.1 Cardinality and Cost Estimation

Cardinality estimation is one of the most challenging problems in databases, and it's often called 'Achilles Heel' of modern query optimizers [74] and has been studied for decades. Traditional cardinality estimation methods can be divided into three categories, data sketching [51], [51], histogram [87], [111] and sampling [84], [102]. However, they have the following drawbacks. Firstly, data sketching and histogram-based methods can only handle the data distribution of one column, while for multi-column tables or even multi-table joins, they will produce large errors due to column correlations. Secondly, although sampling-based methods can capture the correlations of multi-columns and multi-tables by using indexes, they cannot work well for sparse or complex queries due to the *0-tuple* problem.

Cost estimation predicts the resources usage of a physical execution plan for a query, including I/O usage and CPU usage. Traditional cost estimation methods utilize a cost model built upon the estimated cardinality to incorporate the physical operators. Compared with the estimated cardinality, the estimated cost provides more direct running overhead for guiding the plan selection.

Recently, database researchers propose to use deep learning techniques to estimate the cardinality and cost. The learning-based cardinality estimates can be classified into supervised methods and unsupervised methods.

Supervised Methods. We can further divide the supervised methods into the following categories according to the model they adopt.

(1) Mixture Model. Park et al [109] propose a query-based method with mixture models. The method uses a mixture model to fit observed predicates and selectivities of them by minimizing the difference between the model and observed distribution. It can avoid the overhead of multi-dimensional histograms. However, these methods do not support LIKE, EXISTS, and ANY keywords.

(2) Fully Connected Neural Network. Ortiz et al [108] build a cardinality estimation model with fully connected neural network and take an encoded query as input features. They conduct experiments to test trade-offs between model size and estimated errors. Dutt et al [28] propose a regression model on multi-dimensional numerical range predicates for selectivity estimation. The regression model can give an estimated selectivity with small space and time overhead, and it can learn the correlations of multiple columns, which outperform methods with attribute value independence assumption. However, this method is hard to capture the join correlations. Wu et al [143] propose a learning-based method for workloads in shared clouds named *CardLearner*, which extracts overlapped subqueries from workloads, and classifies them according to the structure. Each category of subqueries is a template, and *CardLearner* trains a cardinality estimation model for each template. In this way, *CardLearner* can replace traditional cardinality estimates for subqueries with shared templates. However, the effectiveness of this method is constrained by a workload.

(3) Convolutional Neural Network (CNN). Kipf et al [63] propose a multi-set CNN to learn the cardinality of joins. The model divides a query into three parts, selected tables, join conditions and filter predicates. Each part is represented by a convolutional network and concatenated after average pooling. This is the first method to represent whole queries and learn the cardinality in an end-to-end manner, and it can give an accurate cardinality with low evaluation overhead. However, this method is hard to be used in a plan-based query optimization scenario directly, because the representation of children nodes cannot be used by the parent in a query plan. Marcus et al [97] propose an end-to-end query optimizer named *Neo*. As an important ingredient of the reinforcement learning model, they propose a neural network which contains both query encoding and partial plan encoding to evaluate the best performance of the current query. For plan encoding, they use a CNN to aggregate joins layer by layer. For predicates encoding, they utilize the row vectors which are trained by word2vec models, and encode each predicate by leveraging the average value of selected rows. However, row vectors representation is time-consuming for online encoding.

(4) Recurrent Neural Network. Ortiz et al [108] propose a RNN-based model for estimating cardinality of left-deep plan. At each iteration, one node will be added into the plan tree, and the nodes sequence is the input of the RNN model.

TABLE 4: Methods on cardinality/cost estimation

Category	Model	References	Core Techniques	Estimates	Encodes	Multi-column	Multi-table
Supervised	Mixture	[109]	Mixture Model	Selectivity	Predicates	√	×
	NN	[28]	NN,XGBoost	Selectivity	Predicates	√	×
		[143]	LR,PR,NN	Cardinality	Job name, Input cardinality, Operators	√	√
		[108]	NN	Cardinality	Tables, Predicates	√	√
	CNN	[97]	Tree-CNN	Query latency	Query,Partial plan	√	√
		[63]	Multi-set CNN	Cardinality	Query	√	√
	RNN	[128]	Tree-structured LSTM	Cardinality,Cost	Predicates,Metadata,Operators	√	√
		[96]	Plan-structured RNN	Cost	Input Cardinality,Metadata,Operators	√	√
		[108]	RNN	Cardinality	Tables, Predicates	√	√
Unsupervised	KDE	[48]	KDE	Selectivity	Data samples	√	×
	Deep Likelihood	[45], [148]	Autoregression,DNN	Selectivity	Data,Predicates	√	×

Marcus et al [96] propose a RNN-based cost model built on estimated cardinality and other statistics parameters within traditional RDBMS. The model encodes the query plans into a tree-structured deep neural network. However, it doesn't encode the predicates and relies on cardinality estimated by RDBMS, and the inaccurate estimated cardinality can bring large errors to cost estimations. Sun et al [128] propose an end-to-end learning-based cost estimator with a tree-structured LSTM. It learns a representation for each sub-plan with physical operator and predicates, and outputs the estimated cardinality and cost simultaneously by using another estimation layer. Moreover, the method encodes keywords in the predicates by using word embedding.

Unsupervised Methods. There are also studies [48], [45], [148] on fitting underlying distributions of datasets by using unsupervised density models, but they are hard to support complex queries like multi-joins. Heimel et al [48] propose a Kernel Density Estimator (KDE) based selectivity estimators which is lightweight to construct and maintain with database changes, and it optimizes the Kernel Density Estimator model numerically for better estimation quality by selecting the optimal bandwidth parameters. KDE is fast to fit underline data distributions, is easy to construct and maintain, and is robust for data correlations. Hasan et al [45] and Yang et al [148] utilize autoregressive densities model to represent the joint data distribution among columns. The model returns a list of conditional densities present in the chain rule with an input tuple. To support range predicates, the model adopts progressive sampling to select the meaningful samples by leveraging the learned densities model, and works even with skewed data. However, it cannot support high-dimensional data or multi-table joins.

Above methods have gained great improvements, but they just support simple/fixed queries and queries with DISTINCT keyword may use different techniques in query planner (e.g., set-theoretic cardinality). Hence, Hayek et al [46] used two methods to handle general queries. First, they use a deep learning scheme to predict the unique rate R in the query results. with duplicate rows, where the query is denoted as a collection of (attributes, tables, join, predicates). Second, they extend existing cardinality methods by multiplying unique rate with duplicate results.

2.2.2 Join Order Selection

Join order selection plays a very important role in database systems which has been studied for many years [42]. Traditional methods typically search the solution space of all possible join orders with some pruning techniques, based on cardinality estimation and cost models. For example, the dynamic programming (DP) based algorithms [57] often select the best plan but have high computational overhead. Also, the generated plan by DP algorithms may have large costs due to the wrong cost estimation. Heuristic methods, *GEQO* [10], *QuickPick-1000* [139], and *GOO* [32], may generate plans more quickly, but may not produce good plans.

To address these problems, machine-learning-based methods are proposed to improve the performance of join order selection in recent years, which utilize machine learning to learn from previous examples and overcome the bias caused by the inaccurate estimation. Moreover, the learning-based methods can efficiently select a better plan in a shorter time. We can categorize existing methods into offline-learning methods and online learning methods.

Offline-Learning Methods. Some studies [127], [68], [95], [150] learn from the previous queries to improve the performance of future queries. Stillger et al [127] propose a learned optimizer LEO, which utilizes the feedback of query execution to improve the cost model of query optimizer. It uses a two-layer approach to guide the plan search. One layer represents statistic information from database and the other is the system catalog which is analyzed from past executions. When a query comes, LEO uses a combination of statistic and system catalog to estimate the cardinality and cost, and generates a plan. After a query is executed, LEO compares with the accurate cost and estimates cost of this query's plan and then updates the system catalog.

DQ [68] and *ReJoin* [95] are proposed to use neural network and reinforcement learning to optimize the join orders, inspired by *LEO*'s feedback-based methods. Both *DQ* and *ReJoin* use the cost of previous plans as training data to train the neural network so as to evaluate each join order. *DQ* uses a one-hot vector G to represent each join state. Each cell in a vector indicates the existence of each table in the join tree and the operation selection. Then it uses a multi-layer perceptron (MLP) with G as input to measure each join state and uses DQN to guide the join order selection. Once a plan is generated, the cost of the plan will be treated as a feedback to train the neural network. Different from *DQ*, the input of *ReJoin* is composed of a depth vector and a query vector. The depth vector represents the depth information of each table in the join tree and the query vector represents the query information. *ReJoin* uses the Proximal Policy Optimization(PPO) [122] to guide the join order selection. The results show that both *DQ* and *ReJoin* can generate good join order compared with PostgreSQL's optimizer with low cost while keeping high efficiency. However, the neural networks in *ReJoin* and *DQ* are simple, and can not represent the structure of join tree

sufficiently, and they cannot learn the latency of query plans.

Moreover, the above two methods cannot support updates on schemas. To address this problem, *RTOS*[150] uses a two stage training to generate better join order on latency with a well-designed neural network structure. In order to address the problem that the DNN design in *ReJoin* and *DQ* cannot catch the structure of join tree, *RTOS* proposes a model that utilizes the TreeLSTM to represent the join state. *RTOS* first designs the representation for column, table and join tree. Then the DQN is used to guide the join order generation after the representation is obtained. Next, *RTOS* first uses cost to pre-train the DNN and then utilizes latency information to train the neural network online. The results show that it can generate better join order with low latency.

Online-learning Methods. This class of methods [5], [135], [134] focus on learn a join order using adaptive query processing, which can change the join order even during the execution of the queries. Avnur et al [5] propose an adaptive query processing mechanism called *Eddy*, which combines the execution and optimization of a query. It learns and generates the join order during the online execution of a query. *Eddy* splits the query process as many operators, e.g., two join operators between three relations. *Eddy* uses two routing methods to control the order of these operator to handle the coming tuples: Naive *Eddy* and Lottery *Eddy*. Naive *Eddy* can route more tuples to the operator with less cost; while Lottery *Eddy* can route more tuples to the operator with smaller selectivity. However, these two routing methods are designed to specific scenarios and it calls for designing general routing algorithms to deal with more complex scenarios.

Tzoumas et al [135] model the query execution with *Eddy* as a reinforcement learning problem (*Eddy-RL*) and automatically learn the routing methods from the query execution without any human design. *Eddy* defines the eddy itself as an agent, the progress of tuples as state, the operators in the query as actions, the execution time as reward(cost). The routing methods that decide the order of operator are exactly what this RL problem will solve. *Eddy-RL* uses Q-learning [141] to solve this RL problem. It defines a Q function as a sum of all operators' cost. By minimizing the Q function, it guides which operator to choose each time. However, the above style optimizer does not analyze the relationship between expected execution time and the optimum will not discard intermediate results.

Trummer et al [134] propose a reinforcement learning model SkinnerDB based on *Eddy-RL* [135]. SkinnerDB uses the Upper Confidence Bounds to Trees (UCT) [64] instead of Q-learning in *Eddy-RL*, because *UCT* provides theoretical guarantees on accumulated regret over all choices. SkinnerDB divides the query execution into many small time slices and in each slice, it chooses one join order to execute. Using the real performance of a join order in the time slice, Skinner-DB trains the *UCT* to guide a better join order selection. Finally, Skinner-DB merges the tuples produced in each time slice to generate the final result.

2.2.3 Learning-based Optimizer

Although many researchers have tried to use machine learning methods to solve cost/cardinality estimation and join order selection problems, there are still many factors that need to be considered in physical plan optimization, e.g indexes and views selection. Join order selection methods [68], [95], [150], [134] provide the logical plan, rely on the database optimizer to select the physical operators and indexes, and utilize the cost model to generate the final physical plan. Recently, Marcus et al [97] propose an end-to-end optimizer *NEO* which does not use any cost model and cardinality estimation to generate the final physical plan. *NEO* is also an offline-learning method based on *ReJoin*[95]. Similar to *RTOS* [150], *NEO* also uses Tree-CNN to catch the structural information. It uses row vectors to represent the predicates. In order to produce the physical plan, *NEO* uses a one-hot vector to represent each physical operator selection and index selection in neural networks. Then *NEO* performs a DNN-guided search which keeps expanding the state with minimal value to find the physical plan. In addition, without any information from the cost model, *NEO* uses PostgreSQL's plan to pre-train the neural network and uses latency as feedback to train the neural network. This end-to-end method learns from the latency to generate the whole physical plan, which can be applied to any environment and robust to estimation errors.

2.3 Learning-based Design

Learning-based design aims to utilize machine learning techniques to design database components, e.g., indexes.

2.3.1 Learned Data Structure

The database community and machine learning community investigate learning-based structures, e.g., learned B-tree, using learning-based models to replace traditional indexes to reduce the index size and improve the performance.

Learned B+tree. Kraska et al. [65] propose that *indexes are models*, where the B^+tree index can be seen as a model that maps each query key to its page. For a sorted array, larger position id means larger key value, and the *range index* should effectively approximate the cumulative distribution function (CDF). Based on this observation, Kraska et al. [65] propose a recursive model index, which uses a learning model to estimate the page id of a key. This method outperforms B^+tree under the in-memory environment. However, it cannot support data updates, and it doesn't show effectiveness for secondary indexes.

Another learned index proposed in Galakatos [35], *Fitting-tree*, provides strict error bounds and predictable performance, and it supports two kinds of data insertion strategies. For in-place insertion strategy, *Fitting-tree* keeps ϵ extra insertion space at each end of the page to make in-place insertion not to violate the page error. However, the insertion cost may be high for large segments. Delta insertion strategy keeps a fixed-size buffer zone in each segment, and the keys will be inserted in the buffer and kept sorted. Once the buffer overflows, it has to splits and combines segments. Similar to *Fitting-tree*, the *Alex-index* [24] also reserves spaces for inserted keys. The difference is that reserved space in *Alex-index* is scattered and the inserted key would be put into the position predicted by the model directly. If the position is occupied, more gaps would be inserted (for gapped array) or expanding itself (for packed memory array). *Alex-index* can be more flexible for balancing the trade-offs between space and efficiency.

Tang et al. [131] propose a workload-aware learned index called *Doraemon*. *Doraemon* can incorporate read access pattern by making several copies of frequently visited queries in the training data, and frequent queries would make more contributions to the error and be optimized more than other queries. *Doraemon* reuses pre-trained models for the similar data distributions based on the observation that similar data distribution requires the same model structure.

Learned Secondary Index. Wu et al. [144] propose a succinct secondary indexing mechanism called *Hermit*, which leverages a Tiered Regression Search Tree (TRS-Tree) to capture the column correlations and outliers. The TRS-tree is a machine-learning enhanced tree index. Each leaf node contains a learned linear regression model which maps the target values to correlated values. Each internal node maintains a fixed number of pointers pointing to the children. *Hermit* uses three steps to answer a query, first searching TRS-Tree for mapping the target column to an existing index, leveraging the existing index to get candidate tuples, and finally validating the tuples. *Hermit* is effective in both in-memory and disk-based RDBMS.

Learned Hashmap. Kraska et al. [65] propose to approximate the CDF of keys as hash functions to distribute all the keys evenly in each hash bucket and make less conflicts.

Learned Bloom Filters. *Bloom filter* is a commonly used index to determine whether a value exists in a given set. But traditional bloom filter may occupy a large number of memory for the bit array and hash functions. To reduce the size of Bloom filters, Kraska et al. [65] propose a learning-based Bloom filter. They train a binary classifier model for recognizing whether a query exists in the dataset. New queries need to pass through the classifier firstly, and the negative ones should pass through a traditional Bloom filter further to guarantee no false negative exist. Mitzenmacher et al. [100] propose a formal mathematical method to guide how to improve the performance of learned Bloom filters. They propose a *Sandwich* structure which contains three layers. The first layer is a traditional Bloom filter aiming to remove most of the queries which are not in the dataset, the second layer is a neural network aiming to remove false positives, and the last layer is another traditional Bloom filter aiming to guarantee no false negatives. They provide both the mathematical and intuitive analysis to prove that the *Sandwich* structure is better than two-layer Bloom filter. Besides, they design Bloomier Filter which can not only determine key existence but can return the value associated with the key in the dataset. However, training a Bloom filter from scratch is not practical for ephemeral inputs stream with high throughput, and it motivates the work of Rae et al. [113] that proposes a learned Bloom filter with few-shot neural data structures to support data updates.

For multidimensional datasets, it's space and time consuming to search a set of single Bloom filters one by one. Macke et al. [91] propose an efficient learned Bloom filter for multidimensional data by using *Sandwich* structure. For attributes embedding, they represent values in the high-cardinality attributes by character-level RNNs to reduce the model size. Moreover, they select the best classifier cutoff threshold to maximize the KL divergence between the true positive and false positive rates. In order to reduce the influence of noisy in-index data, they introduce a shift parameter for each positive training sample.

Learned Index for Spatial Data. The conventional spatial indexes, e.g., $R\text{-}tree$, $kd\text{-}tree$, $G\text{-}tree$, cannot capture the distributions of underlying data, and the look-up time and space overhead could be optimized further with the learning-based techniques. For example, Wang et al. [140] propose a learned ZM index which first maps the multi-dimensional geospatial points into a 1-dimensional vector with Z-ordering, and then constructs a neural network index to fit the distributions and predicts the locations for queries.

Learned Index for High-dimensional Data. The nearest neighbor search (NNS) problem on high-dimensional data aims to find the k-nearest points of a query efficiently. Traditional methods for solving the approximate NNS problem can be divided into three categories, including hashing-based index, partition-based index and graph-based index. Some studies [26], [119], [116] improve the first two types of indexes by using machine learning techniques. Schlemper et al. [119] propose an end-to-end deep hashing method, which uses a supervised convolutional neural network. It combines two losses – similarity loss and bit rate loss, and thus it can discretize the data and minimize the collision probability at the same time. Sablayrolles et al. [116] propose a similar end-to-end deep learning architecture which learns a catalyzer function to increase the quality of subsequent coding phases. They introduce a loss derived from the Kozachenko-Leonenko differential entropy estimator to favor uniformity in the spherical output space. Dong et al. [26] reduce the high dimensional space partitions problem to balance graph partitioning and supervised classification problem. They firstly partition the KNN graph into balanced small partitions by using a graph partitioning algorithm *KaHIP*, and then learns a neural model to predict the probabilities that the KKNs of a query fall in a partition, and search the partitions with the large probabilities.

Learned KV-store Design. Idreos et al. [54], [53] show that data structures in key-value stores can be constructed from fundamental components, and the learned cost model can guide the construction directions. They define the *design space* as all the designs that can be described by fundamental design components such as fence pointers, links, and temporal partitioning, and *design continuums* is a subspace of *design space* which connects more than one designs. To design a data structure, they first identify the bottleneck of the total cost and which knob can be tweaked to alleviate it, and then tweak the knob in one direction until they reach its boundary or the total cost reaches the minimum. This process is similar to the gradient descent and can be conducted automatically.

2.3.2 Learned Transaction Management

As the number of CPU cores increases, concurrency control for heavy workloads becomes more challenging [124]. Effective workload scheduling can greatly improve the performance by avoiding the conflicts. We introduce learned transaction management techniques from two aspects: transaction prediction and transaction scheduling.

Transaction Prediction. Transaction prediction is important to database optimization (e.g., resource control, transaction

scheduling). Traditional workload prediction methods are rule-based. For example, a rule-based method [23] uses domain knowledge of database engines (e.g., internal latency, resource utilization) to identify signals t relevant to workload characteristics, such as memory utilization. And this method directly uses memory utilization to predict future workload trend. However, rule-based method wastes much time to rebuild a statistics model when workload changes, so Ma et al. [89] propose an ML-based system *QB5000* that predicts the future trend of different workloads. *QB5000* mainly contains three components, Pre-Processor, Cluster and Forecaster. First, `Pre-Processor` records incoming query features (e.g., syntax tree, arrival rate) and aggregates queries with the same template to approximate workload features. Second, `Cluster` clusters templates with similar arrival rates using modified *DBSCAN* algorithm. Third, `Forecaster` predicts the arrival rate patterns of queries in each cluster. `QB5000` tries six different models for forecasting, where the training data is the history workloads from the past observations.

Transaction Scheduling. Traditional database systems either schedule workload sequentially, which cannot consider potential conflicts, or schedule workloads based on the execution costs predicted by the database optimizer. But traditional database optimizer [73] estimates cost based on assumptions like uniformity and independence, which may be wrong when there are correlations between joined attributes. Hence Sheng et al. [124] propose a machine learning based transaction scheduling method, which can balance between concurrency and conflict rates. First, they estimate the conflict probabilities using supervised algorithm: they build a classifier M to identify whether any pairwise transactions (T_i, T_j) will be aborted or not, where T_i is the vector representation of the transaction query. The training data is collected by observing system logs that collect information when a transaction is committed or aborted and each sample is an abstract triples like (the feature of aborted transaction: $f(T_i)$, the feature of conflicting transaction: $f(T_j)$, label: *abort*), where $f(T)$ is the vector representation of the transaction T. Second, they assign transactions into the executing queue with the maximum throughput under an acceptable abort rate. Suppose (T_i^*, T_j^*) has the highest abort probability, they place T_i^* into the queue after T_j^* so they never execute concurrently.

2.4 Database Monitoring

Database monitoring records system running status and examines the workload to ensure database stability, which is very important to database optimization and diagnosis. For example, knob tuning relies on database monitoring metrics, e.g., system load, read/write blocks. We broadly divide database monitoring into three cases – database health monitor, activity monitor and performance prediction.

Database Health Monitor. Database health monitor (DHM) records database health related metrics, e.g., the number of queries per second, the query latency, to optimize database or diagnose failures. In [90], they assume that intermittent slow queries with similar key performance Indicators (e.g., $cpu.usage$, $mysql.tps$) have the same root causes. So they adopt two-stage diagnosis: (i) in offline stage, they extract slow SQLs from the failure records, cluster them with KPI states, and ask DBAs to assign root causes to each cluster; (ii) in online stage, for an incoming slow SQL, they match it to a cluster C based on similarity score of KPI states. If matched, they use the root cause of C to notify DBAs. Otherwise, they generate a new cluster and ask DBAs to assign the root causes. However, [90] cannot prevent potential database failure and it highly relies on DBA's experience. However, it is rather expensive to monitor many database metrics, because monitoring also consumes the resources. To tackle this issue, Taft et al. [129] propose an elastic database system *P-Store*, which combines database monitor with workload prediction. The basic idea is to proactively monitor database to adapt to workload changes.

Database Activity Monitor. Different from health monitor, database activity monitor (DAM) externally monitors and controls database activities (e.g., creating new accounts, viewing sensitive information), which is vital to protecting sensitive data. We divide DAM into two classes, activity selection and activity trace. For activity selection, there are different levels of database activities (e.g., DBA activities, user transactions including DML, DDL, and DCL). Traditional DAM methods are required to record all the activities on extra systems according to trigger rules [21]. For example, the company might create a rule that generates an alert every time a DBA performs a select query on a credit card column which returns more than 5 results. However, it is still a heavy burden to record all the activities, which brings frequent data exchanges between databases and monitoring systems. Hence, to automatically select and record risky activities, Hagit et al. [40] take database monitoring as a multi-armed bandits problem (MAB). *MAB model* is a decision-making algorithm that selects risky database activities by exploiting current policy and exploring new policies. The goal is to train an optimal policy with the maximal risk score. So for the *MAB model*, in each step, it samples some users with the highest risk for exploiting the policy and some users for exploring better policy. For activity trace, after deciding which activities to be monitored, we need to trace high risky activities and optimize database performance.

Performance Prediction. Query performance prediction is vital to meet the service level agreements (SLAs), especially for concurrent queries. Traditional prediction method [27] only captures logical I/O metrics (e.g., page access time), neglect many resource-related features, and cannot get accurate results. So Marcus et al [96] use deep learning to predict query latency under concurrency scenarios, including interactions between child/parent operators, and parallel plans. However, it adopts a pipeline structure (causing information loss) and fails to capture operator-to-operator relations like data sharing/conflict features. Hence, Zhou et al [156] propose a performance prediction method with graph embedding. They use a graph model to characterize concurrent queries, where the vertices are operators and edges capture operator correlations (e.g., data passing, access conflict, resource competition). They use a graph convolution network to embed the workload graph, extract performance-related features from the graph, and predict the performance based on the features.

2.5 Learning-based Security

Learning-based database security aims to use the machine learning techniques to ensure confidentiality, integrity and availability of databases. We review recent works on sensitive data discovery, access control, and SQL injection.

Learning-based Sensitive Data Discovery. Since sensitive data leakage will cause great financial and personal information loss, it is important to protect the sensitive data in a database. Sensitive data discovery aims to automatically detect and protect confidential data. Traditional methods use some user-defined rules (e.g., ID, credit card and password) to detect the sensitive data [44], [114]. For example, *DataSunrise*[2] is a search-based data discovery framework. It first defines some patterns (e.g., "$3[47][0-9]\{13\}\$$", "$\backslash d^*.\backslash d^*\$$") to define the sensitive data and then use the rules to detect the sensitive data by searching the patterns on the data. However, this method has several limitations. First, it is too expensive to search all the data, and it requires users to specify the candidate search columns to prune the search space. Second, it cannot automatically update rules for new data and thus may miss sensitive data if there is no user-defined rule on some unknown sensitive data. Bhaskar et al [11] propose an algorithm of discovering sensitive data patterns with machine learning. They adapt a Laplace model to learn the real access frequency of data records and then can take the frequently accessed records as candidate sensitive data. First, they formulate data discovery as a scoring problem: for each data record r in the dataset T, r is assigned a score $q(T, r)$, whose value is the access frequency in the database. Second, since the abstracted dataset may still be exponentially large, they recursively sample k patterns, which have not been selected so far from the dataset. Each time, for the k patterns, they compute the frequency with the Laplace model, which adjusts the noise rate to fit the true value according to the loss values. Third, to simplify the implementation, they directly add independent Laplace noise to $q(T, r)$ of each pattern r, and select k patterns with the highest perturbed frequencies, which represent the highest attach risk. In this way, they improve the accuracy of data discovery within resource limitations.

Access Control. It aims to prevent unauthorized users to access the data, including table-level and record-level access control. There have been several traditional methods for access control, such as protocol-based [39], role-based [126], query-based [145] and purpose-based [13]. However, these methods are mainly based on static rules, and advanced techniques (e.g., identity impersonation, metadata modification, illegal invasion) may fake access priority, and traditional methods cannot effectively prevent these attacks. Recently, machine learning algorithms are proposed to estimate the legality of access requests. Colombo et al [19] propose a purpose-based access control model, which customizes control polices to regulate data requests. As different actions and data content may lead to different private problems, this method aims to learn legal access purposes.

SQL Injection. SQL injection is a common and harmful vulnerability to database. Attackers may modify or view data that exceeds their priorities by bypassing additional information or interfering with the SQL statement, such as retrieving hidden data, subverting application logic, union attacks and etc [103]. For example, an application allows users to access the product information by filling the SQL statement like "SELECT price, abstract from product where pname='?' and released=1". But an attacker may retrieve the hidden information of unreleased products by adding extra information in the $pname$ value, like "SELECT price, abstract from product where pname='car'−− and is_released='yes';", where "−−" is a comment indicator in SQL and it removes the limitation of "released=1". However, traditional methods are rule-based (e.g., parameterized queries) and have two limitations. First, they take a long time to scan illegal parameters. Second, there are many variants of illegal parameters, which are not enumerable, and thus the traditional feature matching methods fail to recognize all attacks. Recently, there are mainly two types of SQL injection detection methods that utilize machine learning techniques, including classification tree [125], [85], fuzzy neural network [8]. Moises et al [85] propose a classification algorithm for detecting SQL injections. There are frequent SQL attacks caused by logical failures or bad filters in the query parameters. Hence they build a classifier tree based on tokens extracted from SQL queries to predict possible SQL injections. The training samples are queries with typical SQL injections and risk levels (dangerous/normal/none), which are collected from the database logs. However, this method requires much training data and cannot generalize knowledge to different detection targets. To address the problem of limited training samples, Batista et al [8] propose a fuzzy neural network (FNN) for SQL attacks. The basic idea is to identify attack patterns with fuzzy rules and memorize these rules inside neural network.

3 DB For AI

Existing machine learning platforms are hard to use, because users have to write codes (e.g., Python) to utilize the AI algorithms for data discovery/cleaning, modeling training and model inference. To lower the barrier for using AI, the database community extends the database techniques to encapsulate the complexity of AI algorithms and enables users to use declarative languages, e.g., SQL, to utilize the AI algorithms. In this section, we summarize the database techniques for reducing the complexity of using AI.

3.1 Declarative Language Model

Traditional machine learning algorithms are mostly implemented with programming languages (e.g., Python, R) and have several limitations. First, they require engineering skills to define the complete execution logic, e.g., the iterative patterns of model training, and tensor operations like matrix multiplication and flattening. Second, machine learning algorithms have to load data from database systems, and the data import/export costs may be high.

Hence, AI-oriented declarative language models are proposed to democratize machine learning with extended SQL syntax. We classify declarative language models into two categories, hybrid language model [33], unified language model [49], [98], [80], and drag-and-drop method [118], [33].

2. https://www.datasunrise.com

Hybrid language model. Hybrid language model, e.g., *BigQuery ML* [33], contains both AI and DB operations. Generally, for each query, it splits the statements into AI operations and DB operations. Then it executes AI operations on AI platforms (e.g., TensorFlow, Keras) and executes DB operators on databases. The advantage of the hybrid language models is that they are easy to implement, but the downside is that they have to frequently migrate data between DB and AI platforms, which results in low efficiency.

Unified language model. To fully utilize data management techniques, unified language models [49], [98], [80] are proposed to natively support AI queries in databases without data migration. Hellerstein et. al propose an in-database analytic method *MADlib* [49], which provides a suite of SQL-based algorithms for machine learning in three steps. (1) AI operators need frequent matrix computation including multiplication, transposition, etc, and MADlib implements a customized sparse matrix library within PostgreSQL. For example, linear algebra is coded with loops of basic arithmetic in C. (2) *MADlib* abstracts many AI operators inside databases, including data acquisition, access, sampling, and model definition, training, inference. (3) *MADlib* supports iterative training in databases. For a training process with n iterations, *MDAlib* first declares a virtual table. Then for each iteration it maintains the training results (e.g., gradients of neural units) of m samples as a view, and joins the virtual table with the view to update the model parameters.

Drag-and-Drop Interface. Some SQL statements are still complicated (e.g., nesting structure, multiple joins) and it is hard for inexperienced users to understand. Hence there are some studies that use drag-and-drop methods to use AI techniques [118], [33]. Spreadsheet is an indispensable tool for many data analysts. BigQuery ML [118] provides a virtual spreadsheet technology, *Connected Sheets*, which combines the simplicity of the spreadsheet interface with machine learning algorithms in database systems. Firstly, it presents billions of rows of data to users in a virtual spreadsheet and users can explore the data on that. Then, it automatically translates data operations performed by users into SQL statements and sends them to the database. In this way, data can be analyzed with conventional worksheet functions (e.g., formulas, pivot tables and graphs).

3.2 Data Governance

AI models rely on high-quality data, and data governance aims to discover, clean, integrate, label the data to get high-quality data, which is important for deploying AI models.

Data Discovery. Suppose an AI developer aims to build an AI model. Given a dataset corpus, the user requires to find relevant datasets to build an AI model. Data discovery aims to automatically find relevant datasets from data warehouse considering the applications and user needs. Many companies propose data discovery systems, like *Infogather* [147] in Microsoft and Goods [43] in Google. The former mainly focus on schema complement from an attribute level, which aims to enrich attributes in a table from a massive corpus of Web tables. The latter is a dataset-level method, which stores information like dataset scheme, similarity and provenance between datasets, and then users can search and manage datasets they want. However, the discovery process of such systems is built in and the information stored is predefined, and they cannot generalize to common use cases because limited relationships between datasets can be represented. To this end, Fernandez et al. propose *Aurum* [34], which is a data discovery system that provides flexible queries to search dataset based on users' demand. It leverages enterprise knowledge graph (EKG) to capture a variety of relationships to support a wide range of queries. The EKG is a hypergraph where each node denotes a table column, each edge represents the relationship between two nodes and hyperedges connect nodes that are hierarchically related such as columns in the same table.

Data Cleaning. Note that most of the data are dirty and inconsistent, and the dirty or inconsistent data will result in unreliable decisions and biased analysis. Therefore, it is necessary to clean the data. The pipeline of data cleaning consists of error detection and error repair (see [56] for a survey). Most data cleaning methods mainly focus on cleaning an entire dataset. However, data cleaning is task dependent and different tasks may use different cleaning techniques to repair different parts of the data. To address this problem, Wang et al. propose a cleaning framework for machine learning tasks *ActiveClean* [67]. Given a dataset and machine learning model with a convex loss, it selects records that can most improve the performance of the model to clean iteratively. *ActiveClean* consists of 4 modules, sampler, cleaner, updater and estimator. Sampler is used to select a batch of records to be cleaned. The selection criterion is measured by how much improvement can be made after cleaning a record, i.e., the variation of the gradient, which is estimated by the Estimator. Then the selected records will be checked and repaired by the Cleaner. Next, the Updater updates the gradient based on these verified dirty data. The above four steps are repeated until the budget is used up.

Data Labeling. Nowadays, some sophisticated machine learning algorithms like deep learning, always require a large number of training data to train a good model. Thus how to derive such massive amount of labeling data is a challenging problem. There are mainly three ways to obtain training data, domain experts, non-experts and distant supervision. Firstly, domain experts can provide high-quality labels, which will produce well-performed models. However, it is always too expensive to ask experts to label a large number of data, so active learning techniques [1], [146] are extended to leverage a smaller number of labeled data to train a model. Secondly, thanks to commercial public crowdsourcing platforms, such as Amazon Mechanical Turk (AMT) [3], crowdsourcing (see [75] for a survey) is an effective way to address such tasks by utilizing hundreds or thousands of workers to label the data. Thirdly, distant supervision [99] labels data automatically by making use of knowledge, such as Freebase or domain-specific knowledge. Traditionally, supervised learning always needs a large number of training data, which is expensive. Unsupervised methods do not need training data but they usually suffer from poor performance. Distant supervision combines their advantages. Suppose that we want to extract relations from a corpus without labeled data using distant supervision with the help of Freebase. In the training step, if a sentence

3. https://www.mturk.com

in a document contains two entities, which are an instance of one of Freebase relations, it extracts the features from the sentence, adds the feature vector for the relation and performs the training. Then in the testing step, given some pairs of entities as candidates, they use the features to predict the relation based on the trained model.

Data Lineage. Data lineage [112], [155] aims to depict the relationships between a machine learning pipeline. For example, suppose that we detect an erroneous result of a workflow and then want to debug it. It is very beneficial to retrospect the source data that generates the error. Also, if there are some dirty input data, it is helpful to identify the corresponding output data to prevent incorrect results. In summary, the relationships described by data lineage can be classified into two categories. (1) Backward relationships return the subset of input data that generate the given output records; (2) Forward relationships return the subset of output records that are generated from the given input records. There are several data lineage approaches building relationships between input and output data, including lazy approach [18], [20], eager linear capture approach [30], [55] and fine-grained data lineage [112], [155]. The lazy approaches [18], [20] regard the lineage queries as relational queries and execute them on the input relations directly. The advantage is that the base queries do not incur overhead, but the drawback is the computational cost of the lineage query execution. The eager linear capture approach [30], [55] builds a lineage graph to speed up the lineage query, and uses the paths on graphs to support backward and forward queries. The fine-grained data lineage system integrates the lineage capture logic and physical operators in databases tightly [112] and uses lightweight and write-efficient index to achieve a low-overhead lineage capture.

3.3 Model Training for AI

Model training is an indispensable step in applying AI algorithms, which includes feature selection, model selection, model management and model acceleration. In this section, we summarize the existing model training techniques.

Feature Selection. Feature selection (FS) selects features that may significantly affect the model performance. For ease of presentation, given a dataset of a relational table R, we define the entire feature set as $F = \{f_1, f_2 \dots f_n\}$. The goal of FS is to select an optimal subset $F* \subseteq F$, so as to train a model with the best performance. In addition, we use $R_{F'} \subseteq R$ to denote the sub-dataset corresponding to the feature subset $F' \subseteq F$. Generally speaking, the FS process consists of the following steps. (1) Generate a feature subset F' and the corresponding dataset $R_{F'}$. (2) Evaluate F' by building an ML model on $R_{F'}$. (3) Iteratively repeat the above two steps until a predefined model performance is achieved or all feature subsets have been evaluated.

Since the number of feature subsets is $O(2^n)$, a brute-force method to enumerate every subset is too expensive due to the large search space. Therefore, in the ML community, many approaches have been proposed to reduce the search space by generating some candidate feature subsets (see [47] for a survey). Recently, several works [69] have been proposed to leverage the DB optimization techniques to accelerate the FS process from feature subsets enumeration and feature subsets evaluation. Specifically, batching and materialization techniques [152] are utilized to reduce the feature subsets enumeration cost. Active learning based method [4] is utilized to accelerate the feature subsets evaluation process. Next, we will introduce them in detail.

Model Selection. Model selection aims to generate a model and set its hyper-parameters to maximize the quality given a specific measurement [70]. There are two classes of methods, traditional model selection and neural architecture search (NAS). The former focuses on selecting the best model from traditional ML models like SVM, Random Forest, KNN etc. The latter aims to build a well-performed neural architecture automatically, including model structure design and hyper-parameter settings, which is a current hot topic in both ML and DB community. In this survey, we will focus on DB-based techniques on NAS. In ML community, many automated machine learning techniques have been proposed to achieve well-performed models or reduce the latency of training one model at a time, like grid/random search, reinforcement learning method, Bayesian optimization etc (see [52] for a survey). However, a key bottleneck of this problem is model selection throughput, i.e., the number of training configurations is tested per unit time. High throughput allows the user to test more configurations during a fixed period, which makes the entire training process efficient. A straightforward way to enhance the throughput is parallelism, and the popular parallelism strategies include task parallel [101] , bulk synchronous parallel [66] , model hop parallelism [104] and parameter server [79].

Model Management. Data analysts usually build a machine learning model in an iterative approach. Given an ML task, they first start from some neat models, specify the training/testing data and loss function. Second, the model is evaluated on the data. Based on the results, the analysts modify the model and repeat the above steps until a well-performed model is derived. However, it is difficult for the data analysts to recap previous evaluation results to get some insights, because the models built before have not been recorded. Therefore, model management is proposed to track, store and search a large number of ML models, so that people can analyze, revise and share their models conveniently. In this part, we classify the model management system into two categories, i.e., GUI-based system [9], [14] and command-based system [137] .

Hardware Acceleration. The computer architecture community [93], [31] studies how to leverage hardware accelerator, like FPGA, to accelerate ML models. Given an ML task with long training time, where the data is stored in RDBMS, data scientists can use hardwares to accelerate the task. To make the hardware acceleration for ML easy-to-use in RDBMS, Mahajan [92] propose DAnA, a framework that takes ML queries as input, call the FPGA accelerator to fetch the data and conduct the ML computation automatically. More specifically, *DAnA* first designs a high-level programming language combining SQL and Python to define the ML algorithms and data required, where the SQL part specifies how to retrieve and manage the data and the Python part depicts the ML algorithms. Second, *DAnA* parses the query and utilizes $Striders$, which is a hardware mechanism that connects the FPGA and database. Specifically, $Striders$ retrieves the training data from the

buffer pool to the accelerator directly without accessing the CPU. Then it derives the feature vectors and labels. Finally, an execution model is designed to combine thread-level and data-level parallelism for accelerating the ML algorithms. For column-store database, Kara et.al [62] propose a framework, *ColumnML* that studies how to leverage hardware to accelerate the training of generalized linear models(GLMs). For data stored in column-wise, stochastic coordinate descent(SCD) [123] is applied to solve GLMs algorithms. *ColumnML* proposes a partition based SCD to improve the cache locality. Furthermore, column-store data is compressed and encrypted, which leads to a low efficiency for CPU to transform the data for training. Therefore, *ColumnML* utilizes FPGA to transform the data on-the-fly.

3.4 DB Techniques for Accelerating Model Inference

Model inference is to use a pre-trained model to predict the testing samples, including operator support, operator selection, and execution acceleration.

Operator Support. The database community studies enabling model inference inside databases and utilizes optimization techniques to accelerate model inference [49], [12], [133], [37], [149]. Different from traditional database operators (e.g., filter, join, sort), AI models involve more complex operator types, including scalar(0-dimension)/vector(1-dimension)/matrix (2-dimension)/tensor(N-dimension) operations. First, database natively supports scalar operations and can optimize the execution. Second, vector and tensor data can be converted into matrices: vector is a special matrix with one dimension while tensor can be partitioned into multiple matrices. Thus most of the existing studies focus on optimizing matrix data. Matrix operations in model inference include data pre-processing, forward propagation, and model training, which are usually time-consuming. Traditional AI systems rely on new hardware like GPU to enhance execution performance. Boehm et al. [12] propose an in-database machine learning system *SystemML*. *SystemML* supports matrix operations with user-defined aggregation functions, which provide parallel data processing in the column level. First, *SystemML* categorizes operations according to their access patterns, like cell-wise matrix additions and transpose. And then *SystemML* optimizes these operations with algebraic rewrites, adjusts the operator ordering of matrix multiplication chains, and compiles them into a DAG of low level aggregation operators like grouping, summing, and counting. Thus *SystemML* can efficiently execute matrix operators inside databases.

Operator Selection. The same ML model can be translated into different physical operators [7]. For example, linear regression can be interpreted into linear regularization operators or derivative operators. However, AI systems do not directly consider operator selection and leave this work to hardware like GPU, which may flatten sparse tensors and convert tensor decomposition into simpler matrix multiplications. But hardware-level selection often falls into local optimization, which cannot estimate the overall resource requirements and make errors. Database optimizer can efficiently estimate execution cost and optimize operator selection natively. Boehm et al. [12] propose an in-database method of operator selection. First, they select operations according to resource estimation functions of data sparsity, cluster size, or memory. For example, it estimates memory consumption of every operation on $\mathcal{M}(X)$, which denotes the memory estimate of a single-block matrix, and $\mathcal{M}(X_p)$, which denotes the memory estimate of a block-partitioned matrix. And the target is to select operation combinations with minimized total execution time under memory constraints. Second, in Spark, they further enhance execution efficiency by replacing selected operations into operators of Spark (e.g., Map, Reduce, Shuffle) as much as possible. Third, to avoid repeated read from HDFS, text parsing, and shuffle operators, which are time-consuming, they inject checkpoints after each persistent read. If any checkpoint reports one of the three operators, optimizer in Spark will remove/replace the operator in the next iteration.

Execution Acceleration. Different from model training, model inference needs to choose ML models and execute forward propagation to predict for different problems. Existing execution acceleration techniques include in-memory methods [71] and distributed methods[115], [2]. The former aims to compress data into memory and conduct in-memory computation as much as possible. The latter routes tasks to different nodes and reduces the burden of data processing and model computation using parallel computing.

4 Research Challenges and Future Work

4.1 AI4DB

There are still several challenges that utilize AI techniques to optimize databases.

Training Data. Most AI models require large-scale, high-quality, diversified training data to achieve high performance. However, it is rather hard to get training data in AI4DB, because the data either is security critical or relies on DBAs. For example, in database knob tuning, the training samples should be gotten based on DBAs' experiences. Thus it is hard to get a large number of training samples. Moreover, to build an effective model, the training data should cover different scenarios, different hardware environments, and different workloads. It calls for new methods that use a small training dataset to get a high-quality model.

Adaptability. The adaptability is a big challenge, e.g., adapting to dynamic data updates, other datasets, new hardware environments, and other database systems. We need to address the following challenges. How to adapt a trained model (e.g., optimizer, cost estimation) on a dataset to other datasets? How to adapt a trained model on a hardware environment to other hardware environments? How to adapt a trained model on a database to other databases? How to make a trained model support dynamic data updates?

Model Convergence. It is very important that whether a learned model can be converged. If the model cannot be converged, we need to provide alternative ways to avoid making delayed and inaccurate decisions. For example, in knob tuning, if the model is not converged, we cannot utilize the model for online knob suggestion.

Learning for OLAP. Traditional OLAP focuses on relational data analytics. However, in the big data era, many new data types have emerged, e.g., graph data, time-series data, spatial data, it calls for new data analytics techniques to analyze

these multi-model data. Moreover, besides traditional aggregation queries, many applications require to use machine learning algorithms to enhance data analytics, e.g., image analysis. Thus it is rather challenging to integrate AI and DB techniques to provide new data analytics functionality.

Learning for OLTP. Transaction modeling and scheduling are rather important to OLTP systems, because different transactions may have conflicts. It is promising to utilize learnign techniques to optimize OLTP queries, e.g., consistent snapshot [78], situ query processing [106]. However, it is not free to model and schedule the transactions, and it calls more efficient models that can instantly model and schedule the transactions in multiple cores and multiple machines.

4.2 DB4AI

In-database Training. It is challenging to support AI training inside databases, including model storage, model update and parallel training. First, it is challenging to store a model in databases, such that the model can be trained and used by multi-tenants, and we need to consider the security and privacy issues. Second, it is challenging to update a model, especially when the data is dynamically updated.

Accelerate AI Training using Database Techniques. Most of studies focus on the effectiveness of AI algorithms but do not pay much attention to the efficiency, which is also very important. It calls for utilizing database techniques to improve the performance of AI algorithms, e.g., indexes and views. For example, self-driving vehicles require a large number of examples for training, which is rather time consuming. Actually, it only requires some *important examples*, e.g., the training cases in the night or rainy day, but not many redundant examples. Thus we can index the samples and features for effective training. However, it is not easy to define which examples are important, e.g., manhole cover missing in a rainy day. Thus it calls for search by examples, which, given some examples, can find training samples based on the examples. In addition, it is also challenging to reuse the well-trained AI models by different users.

AI Optimizer. Existing studies use user-defined functions (UDF) to support AI models, which are not effectively optimized. It requires to implement the AI models as operators insider databases, and design physical operators for each operator. Most importantly, it requires to push down the AI operators and estimate the cost/cardinality of AI operators. It calls for an AI optimizer to optimize the AI training and inference. Furthermore, it is more important to efficiently support AI operators in a distributed environment.

Fault-tolerant Learning. Existing learning model training does not consider error tolerance. When a distributed training is conducted, a process crashes and the whole task will fail. We can combine the error tolerance techniques of database systems to improve the robustness of in-database learning. In order to ensure business continuity under predictable and unpredictable disasters, database systems must guarantee fault tolerance and disaster recovery capabilities.

4.3 Hybrid DB and AI

Hybrid Relational and Tensor Model. Databases adopt a relational model and many AI models use a tensor model. Traditional CPU is good at processing relational models and AI chips are good at processing tensor models. However, traditional CPU cannot efficiently process tensor models and AI chips cannot efficiently process relational models. It calls for effective methods that accelerate relational operations on AI chips, scheduling different operators across traditional CPU and AI chips, and supporting both relational and tensor model. Moreover, it is promising to study a unified data model to support scalar, vector, large-scale tensor and other different types of data.

Hybrid DB&AI Inference. Many applications require both DB and AI operations, e.g., finding all the patients of a hospital whose stay time will be longer than 3 days. A native way is to predict the hospital stay of each patient and then prune the patients whose stay time is larger than 3. Obviously this method is rather expensive, and it calls for a new optimization model to optimize both DB and AI [77], including new optimization model, AI operator push-down, AI cost estimation, and AI index/views.

Hybrid DB&AI System. Most of applications use both AI and DB operators and it calls for an end-to-end hybrid AI&DB system that can support the two operators. It requires to design a declarative language, e.g., AISQL that extends SQL to support AI operators, an AI&DB optimizer that co-optimize the two operations, an effective (distributed) execution engined that schedule the two types of tasks, and an appropriate storage engine.

5 Conclusion

In this paper, we summarize the recent techniques on AI4DB and DB4AI. The former focuses on utilizing AI techniques to address the data processing problems with high computation complexities, e.g., knob tuning, cost estimation, join order selection, index advisor and view advisor. The latter focuses on using DB techniques to reduce the complexity of using AI and accelerate AI models, e.g., declarative AI, and accelerating AI training and inferences. We also provide some research challenges and open problems in AI4DB, DB4AI, and hybrid DB and AI optimization.

Acknowledgement. This paper was supported by NSF of China (61925205, 61632016), Huawei, and TAL education.

References

[1] C. C. Aggarwal, X. Kong, Q. Gu, J. Han, and P. S. Yu. Active learning: A survey. In *Data Classification: Algorithms and Applications*, pages 571–606. 2014.

[2] D. Agrawal and et al. Rheem: Enabling multi-platform task execution. In *SIGMOD 2016*, pages 2069–2072, 2016.

[3] D. V. Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic database management system tuning through large-scale machine learning. In *SIGMOD 2017*, pages 1009–1024, 2017.

[4] M. R. Anderson and M. J. Cafarella. Input selection for fast feature engineering. In *ICDE 2016,*, pages 577–588, 2016.

[5] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD 2000*, pages 261–272, 2000.

[6] C. Baik, H. V. Jagadish, and Y. Li. Bridging the semantic gap with SQL query logs in natural language interfaces to databases. In *ICDE 2019*, pages 374–385, 2019.

[7] P. L. Bartlett, S. Boucheron, and G. Lugosi. Model selection and error estimation. *Machine Learning*, 48(1-3):85–113, 2002.

[8] L. O. Batista, G. A. de Silva, V. S. Araújo, V. J. S. Araújo, T. S. Rezende, A. J. Guimarães, and P. V. de Campos Souza. Fuzzy neural networks to create an expert system for detecting attacks by SQL injection. *CoRR*, abs/1901.02868, 2019.

[9] L. Bavoil, S. P. Callahan, C. E. Scheidegger, H. T. Vo, P. Crossno, C. T. Silva, and J. Freire. Vistrails: Enabling interactive multiple-view visualizations. In *VIS 2005*, pages 135–142, 2005.

[10] K. P. Bennett, M. C. Ferris, and Y. E. Ioannidis. A genetic algorithm for database query optimization. In *ICGA 1991*, pages 400–407, 1991.

[11] R. Bhaskar, S. Laxman, A. D. Smith, and A. Thakurta. Discovering frequent patterns in sensitive data. In *SIGKDD 2010*, pages 503–512, 2010.

[12] M. Boehm, M. Dusenberry, D. Eriksson, A. V. Evfimievski, F. M. Manshadi, N. Pansare, B. Reinwald, F. Reiss, P. Sen, A. Surve, and S. Tatikonda. Systemml: Declarative machine learning on spark. *PVLDB*, 9(13):1425–1436, 2016.

[13] J. Byun and N. Li. Purpose based access control for privacy protection in relational database systems. *VLDB J.*, 17(4):603–619, 2008.

[14] S. P. Callahan, J. Freire, E. Santos, C. E. Scheidegger, C. T. Silva, and H. T. Vo. Vistrails: visualization meets data management. In *SIGMOD 2006*, pages 745–747, 2006.

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

[16] M. Chavan, R. Guravannavar, K. Ramachandra, and S. Sudarshan. Dbridge: A program rewrite tool for set-oriented query execution. In *ICDE 2011*, pages 1284–1287, 2011.

[17] J. Chen, Y. Chen, and G. L. et al. Data management at huawei: Recent accomplishments and future challenges. In *ICDE*, pages 13–24, 2019.

[18] J. Cheney, L. Chiticariu, and W. C. Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.

[19] P. Colombo and E. Ferrari. Efficient enforcement of action-aware purpose-based access control within relational database management systems. In *ICDE 2016*, pages 1516–1517, 2016.

[20] Y. Cui, J. Widom, and J. L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.*, 25(2):179–227, 2000.

[21] C. Curino, E. P. C. Jones, S. Madden, and H. Balakrishnan. Workload-aware database monitoring and consolidation. In *SIGMOD 2011*, pages 313–324, 2011.

[22] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zaït, and M. Ziauddin. Automatic SQL tuning in oracle 10g. In *VLDB*, pages 1098–1109, 2004.

[23] S. Das, F. Li, V. R. Narasayya, and A. C. König. Automated demand-driven resource scaling in relational database-as-a-service. In *SIGMOD 2016*, pages 1923–1934, 2016.

[24] J. Ding, U. F. Minhas, H. Zhang, Y. Li, C. Wang, B. Chandramouli, J. Gehrke, D. Kossmann, and D. B. Lomet. ALEX: an updatable adaptive learned index. *CoRR*, abs/1905.08898, 2019.

[25] T. Dökeroglu, M. A. Bayir, and A. Cosar. Robust heuristic algorithms for exploiting the common tasks of relational cloud database queries. *Appl. Soft Comput.*, 30:72–82, 2015.

[26] Y. Dong, P. Indyk, I. P. Razenshteyn, and T. Wagner. Learning sublinear-time indexing for nearest neighbor search. *CoRR*, abs/1901.08544, 2019.

[27] J. Duggan, U. Çetintemel, O. Papaemmanouil, and E. Upfal. Performance prediction for concurrent database workloads. In *SIGMOD 2011*, pages 337–348, 2011.

[28] A. Dutt, C. Wang, A. Nazi, S. Kandula, V. R. Narasayya, and S. Chaudhuri. Selectivity estimation for range predicates using lightweight models. *PVLDB*, 12(9):1044–1057, 2019.

[29] H. G. et al. Selection of views to materialize under a maintenance cost constraint. In *ICDT*, pages 453–470, 1999.

[30] P. A. et al. Trio: A system for data, uncertainty, and lineage. In *VLDB 2006*, pages 1151–1154, 2006.

[31] A. P. et.al. A reconfigurable fabric for accelerating large-scale datacenter services. In *ISCA 2014*, pages 13–24, 2014.

[32] L. Fegaras. A new heuristic for optimizing large queries. In *DEXA 1998*.

[33] S. Fernandes and J. Bernardino. What is bigquery? In *ICDAS*, pages 202–203, 2015.

[34] R. C. Fernandez, Z. Abedjan, F. Koko, G. Yuan, S. Madden, and M. Stonebraker. Aurum: A data discovery system. In *ICDE 2018*, pages 1001–1012, 2018.

[35] A. Galakatos, M. Markovitch, C. Binnig, R. Fonseca, and T. Kraska. Fiting-tree: A data-aware index structure. In *SIGMOD*, pages 1189–1206, 2019.

[36] E. Gallinucci and M. Golfarelli. Sparktune: tuning spark SQL through query cost modeling. In *EDBT 2019*, pages 546–549, 2019.

[37] Z. Gharibshah, X. Zhu, A. Hainline, and M. Conway. Deep learning for user interest and response prediction in online display advertising. *Data Science and Engineering*, 5(1):12–26, 2020.

[38] A. Gounaris, G. Kougka, R. Tous, C. T. Montes, and J. Torres. Dynamic configuration of partitioning in spark applications. *IEEE Trans. Parallel Distrib. Syst.*, 28(7):1891–1904, 2017.

[39] M. L. Goyal and G. V. Singh. Access control in distributed heterogeneous database management systems. *Computers & Security*, 10(7):661–669, 1991.

[40] H. Grushka-Cohen, O. Biller, O. Sofer, L. Rokach, and B. Shapira. Diversifying database activity monitoring with bandits. *CoRR*, abs/1910.10777, 2019.

[41] J. Gu, Y. Li, H. Tang, and Z. Wu. Auto-tuning spark configurations based on neural network. In *ICC 2018*, pages 1–6, 2018.

[42] L. M. Haas. Review - access path selection in a relational database management system. *ACM SIGMOD Digital Review*, 1, 1999.

[43] A. Y. Halevy, F. Korn, N. F. Noy, C. Olston, N. Polyzotis, S. Roy, and S. E. Whang. Goods: Organizing google's datasets. In *SIGMOD 2016*, pages 795–806, 2016.

[44] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD 2000*, pages 1–12, 2000.

[45] S. Hasan, S. Thirumuruganathan, J. Augustine, N. Koudas, and G. Das. Multi-attribute selectivity estimation using deep learning. *CoRR*, abs/1903.09999, 2019.

[46] R. Hayek and O. Shmueli. Nn-based transformation of any SQL cardinality estimator for handling distinct, and, OR and NOT. *CoRR*, abs/2004.07009, 2020.

[47] X. He, K. Zhao, and X. Chu. Automl: A survey of the state-of-the-art. *CoRR*, abs/1908.00709, 2019.

[48] M. Heimel, M. Kiefer, and V. Markl. Self-tuning, gpu-accelerated kernel density models for multidimensional selectivity estimation. In *SIGMOD*, pages 1477–1492, 2015.

[49] J. M. Hellerstein, C. Ré, and F. S. et al. The madlib analytics library or MAD skills, the SQL. *PVLDB*, 5(12):1700–1711, 2012.

[50] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A self-tuning system for big data analytics. In *CIDR 2011*, pages 261–272, 2011.

[51] S. Heule, M. Nunkesser, and A. Hall. Hyperloglog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm. In *EDBT*, pages 683–692, 2013.

[52] F. Hutter, L. Kotthoff, and J. Vanschoren, editors. *Automated Machine Learning: Methods, Systems, Challenges*. Springer, 2018.

[53] S. Idreos, N. Dayan, W. Qin, M. Akmanalp, S. Hilgard, A. Ross, J. Lennon, V. Jain, H. Gupta, D. Li, and Z. Zhu. Design continuums and the path toward self-designing key-value stores that know and learn. In *CIDR*, 2019.

[54] S. Idreos, N. Dayan, W. Qin, M. Akmanalp, S. Hilgard, A. Ross, J. Lennon, V. Jain, H. Gupta, D. Li, and Z. Zhu. Learning key-value store design. *CoRR*, abs/1907.05443, 2019.

[55] R. Ikeda, H. Park, and J. Widom. Provenance for generalized map and reduce workflows. In *CIDR 2011*, pages 273–283, 2011.

[56] I. F. Ilyas and X. Chu. *Data Cleaning*. ACM, 2019.

[57] Y. E. Ioannidis and Y. C. Kang. Left-deep vs. bushy trees: An analysis of strategy spaces and its implications for query optimization. In *SIGMOD 1991.*, pages 168–177, 1991.

[58] A. Jindal, K. Karanasos, and S. R. et al. Selecting subexpressions to materialize at datacenter scale. *PVLDB*, 11(7):800–812, 2018.

[59] A. Jindal, S. Qiao, H. Patel, Z. Yin, J. Di, M. Bag, M. Friedman, Y. Lin, K. Karanasos, and S. Rao. Computation reuse in analytics job service at microsoft. In *SIGMOD*, pages 191–203, 2018.

[60] S. Kadirvel and J. A. B. Fortes. Grey-box approach for performance prediction in map-reduce based platforms. In *ICCCN 2012*, pages 1–9, 2012.

[61] H. Kaneko and K. Funatsu. Automatic database monitoring for process control systems. In *IEA/AIE 2014*, pages 410–419, 2014.

[62] 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.

[63] A. Kipf, T. Kipf, B. Radke, V. Leis, P. A. Boncz, and A. Kemper. Learned cardinalities: Estimating correlated joins with deep learning. In *CIDR 2019*, 2019.

[64] L. Kocsis and C. Szepesvári. Bandit based monte-carlo planning. In *ECML 2006*, pages 282–293, 2006.

[65] T. Kraska, A. Beutel, and E. H. C. et al. The case for learned index structures. In *SIGMOD*, pages 489–504, 2018.

[66] T. Kraska, A. Talwalkar, J. C. Duchi, R. Griffith, M. J. Franklin, and M. I. Jordan. Mlbase: A distributed machine-learning system. In *CIDR 2013*, 2013.

[67] S. Krishnan, J. Wang, E. Wu, M. J. Franklin, and K. Goldberg. Activeclean: Interactive data cleaning for statistical modeling. *PVLDB*, 9(12):948–959, 2016.

[68] S. Krishnan, Z. Yang, K. Goldberg, J. M. Hellerstein, and I. Stoica. Learning to optimize join queries with deep reinforcement learning. *CoRR*, abs/1808.03196, 2018.

[69] A. Kumar, M. Boehm, and J. Yang. Data management in machine learning: Challenges, techniques, and systems. In *SIGMOD 2017*, pages 1717–1722, 2017.

[70] A. Kumar, R. McCann, J. F. Naughton, and J. M. Patel. Model selection management systems: The next frontier of advanced analytics. *SIGMOD Record*, 44(4):17–22, 2015.

[71] M. Kunjir and S. Babu. Thoth in action: Memory management in modern data analytics. *PVLDB*, 10(12):1917–1920, 2017.

[72] M. Kunjir and S. Babu. Black or white? how to develop an autotuner for memory-based analytics [extended version]. *CoRR*, abs/2002.11780, 2020.

[73] V. Leis, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *PVLDB*, 9(3):204–215, 2015.

[74] V. Leis, B. Radke, A. Gubichev, A. Kemper, and T. Neumann. Cardinality estimation done right: Index-based join sampling. In *CIDR*, 2017.

[75] G. Li, J. Wang, Y. Zheng, and M. J. Franklin. Crowdsourced data management: A survey. *IEEE Trans. Knowl. Data Eng.*, 28(9):2296–2319, 2016.

[76] G. Li, X. Zhou, and S. L. et al. Qtune: A query-aware database tuning system with deep reinforcement learning. *VLDB*, 2019.

[77] G. Li, X. Zhou, and S. Li. Xuanyuan: An ai-native database. *IEEE Data Eng. Bull.*, 42(2):70–81, 2019.

[78] L. Li, G. Wang, and G. W. et al. A comparative study of consistent snapshot algorithms for main-memory database systems. *CoRR*, abs/1810.04915, 2018.

[79] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B. Su. Scaling distributed machine learning with the parameter server. In *OSDI 2014*, pages 583–598, 2014.

[80] X. Li, B. Cui, Y. Chen, W. Wu, and C. Zhang. Mlog: Towards declarative in-database machine learning. *PVLDB*, 10(12):1933–1936, 2017.

[81] X. Liang, A. J. Elmore, and S. Krishnan. Opportunistic view materialization with deep reinforcement learning. *CoRR*, abs/1903.01363, 2019.

[82] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. In *ICLR 2016*, 2016.

[83] Z. Lin, X. Li, and X. Kuang. Machine learning in vulnerability databases. In *ISCID 2017*, pages 108–113, 2017.

[84] R. J. Lipton, J. F. Naughton, and D. A. Schneider. Practical selectivity estimation through adaptive sampling. In *SIGMOD*, pages 1–11, 1990.

[85] M. Lodeiro-Santiago, C. Caballero-Gil, and P. Caballero-Gil. Collaborative sql-injections detection system with machine learning. In *IML 2017*, pages 45:1–45:5, 2017.

[86] J. Lu, Y. Chen, H. Herodotou, and S. Babu. Speedup your analytics: Automatic parameter tuning for databases and big data systems. *PVLDB*, 12(12):1970–1973, 2019.

[87] X. Lu and J. Guan. A new approach to building histogram for selectivity estimation in query processing optimization. *Computers & Mathematics with Applications*, 57(6):1037–1047, 2009.

[88] M. Lühring, K. Sattler, K. Schmidt, and E. Schallehn. Autonomous management of soft indexes. In *ICDE 2007*, pages 450–458, 2007.

[89] L. Ma, D. V. Aken, and A. H. et al. Query-based workload forecasting for self-driving database management systems. In *SIGMOD 2018*, pages 631–645, 2018.

[90] M. Ma, Z. Yin, and S. Z. et al. Diagnosing root causes of intermittent slow queries in cloud databases. In *PVLDB*, 2020.

[91] S. Macke, A. Beutel, T. Kraska, M. Sathiamoorthy, D. Z. Cheng, and E. H. Chi. Lifting the curse of multidimensional data with learned existence indexes. *NIPS*, 2018.

[92] D. Mahajan, J. K. Kim, J. Sacks, A. Ardalan, A. Kumar, and H. Esmaeilzadeh. In-rdbms hardware acceleration of advanced analytics. *PVLDB*, 11(11):1317–1331, 2018.

[93] D. Mahajan, J. Park, E. Amaro, H. Sharma, A. Yazdanbakhsh, J. K. Kim, and H. Esmaeilzadeh. TABLA: A unified template-based framework for accelerating statistical machine learning. In *HPCA 2016*, pages 14–26, 2016.

[94] N. Makrynioti and V. Vassalos. Declarative data analytics: a survey. *CoRR*, abs/1902.01304, 2019.

[95] R. Marcus and O. Papaemmanouil. Deep reinforcement learning for join order enumeration. In *SIGMOD 2018*, pages 3:1–3:4, 2018.

[96] R. Marcus and O. Papaemmanouil. Plan-structured deep neural network models for query performance prediction. *Proc. VLDB Endow.*, 12(11):1733–1746, 2019.

[97] R. C. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul. Neo: A learned query optimizer. *PVLDB*, 12(11):1705–1718, 2019.

[98] X. Meng, J. K. Bradley, and et al. Mllib: Machine learning in apache spark. *J. Mach. Learn. Res.*, 17:34:1–34:7, 2016.

[99] M. Mintz, S. Bills, R. Snow, and D. Jurafsky. Distant supervision for relation extraction without labeled data. In *ACL 2009*, pages 1003–1011, 2009.

[100] M. Mitzenmacher. A model for learned bloom filters and optimizing by sandwiching. In *NeurIPS*, pages 462–471, 2018.

[101] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica. Ray: A distributed framework for emerging AI applications. In *OSDI 2018*, pages 561–577, 2018.

[102] M. Müller, G. Moerkotte, and O. Kolb. Improved selectivity estimation by combining knowledge from sampling and synopses. *PVLDB*, 11(9):1016–1028, 2018.

[103] B. Nagpal, N. Chauhan, and N. Singh. A survey on the detection of SQL injection attacks and their countermeasures. *JIPS*, 13(4):689–702, 2017.

[104] S. Nakandala, Y. Zhang, and A. Kumar. Cerebro: Efficient and reproducible model selection on deep learning systems. In *DEEM@SIGMOD 2019*, pages 6:1–6:4, 2019.

[105] N. Nguyen, M. M. H. Khan, and K. Wang. Towards automatic tuning of apache spark configuration. In *CLOUD 2018*, pages 417–425, 2018.

[106] M. Olma, M. Karpathiotakis, and I. A. et al. Adaptive partitioning and indexing for in situ query processing. *VLDB J.*, 29(1):569–591, 2020.

[107] J. Ortiz, M. Balazinska, J. Gehrke, and S. S. Keerthi. Learning state representations for query optimization with deep reinforcement learning. In *DEEM@SIGMOD*, pages 4:1–4:4, 2018.

[108] J. Ortiz, M. Balazinska, J. Gehrke, and S. S. Keerthi. An empirical analysis of deep learning for cardinality estimation. *CoRR*, abs/1905.06425, 2019.

[109] Y. Park, S. Zhong, and B. Mozafari. Quicksel: Quick selectivity learning with mixture models. *CoRR*, abs/1812.10568, 2018.

[110] W. G. Pedrozo, J. C. Nievola, and D. C. Ribeiro. An adaptive approach for index tuning with learning classifier systems on hybrid storage environments. In *HAIS 2018*, pages 716–729, 2018.

[111] V. Poosala, Y. E. Ioannidis, P. J. Haas, and E. J. Shekita. Improved histograms for selectivity estimation of range predicates. In *SIGMOD*, pages 294–305, 1996.

[112] F. Psallidas and E. Wu. Smoke: Fine-grained lineage at interactive speed. *PVLDB*, 11(6):719–732, 2018.

[113] J. W. Rae, S. Bartunov, and T. P. Lillicrap. Meta-learning neural bloom filters. In *ICML*, pages 5271–5280, 2019.

[114] S. Ruggieri, D. Pedreschi, and F. Turini. DCUBE: discrimination discovery in databases. In *SIGMOD 2010*, pages 1127–1130, 2010.

[115] J. M. Rzeszotarski and A. Kittur. Kinetica: naturalistic multi-touch data visualization. In *CHI 2014*, pages 897–906, 2014.

[116] A. Sablayrolles, M. Douze, C. Schmid, and H. Jégou. Spreading vectors for similarity search. In *ICLR*, 2019.

[117] Z. Sadri, L. Gruenwald, and E. L. et al. Online index selection using deep reinforcement learning for a cluster database. In *ICDEW*, 2020.

[118] T. Schindler and C. Skornia. Secure parallel processing of big data using order-preserving encryption on google bigquery. *CoRR*, abs/1608.07981, 2016.

[119] J. Schlemper, J. Caballero, A. Aitken, and J. R. van Amersfoort. Deep hashing using entropy regularised product quantisation network. *CoRR*, abs/1902.03876, 2019.

[120] K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis. On-line index selection for shifting workloads. In *ICDE 2007*, pages 459–468, 2007.

[121] K. Schnaitter and N. Polyzotis. Semi-automatic index tuning: Keeping dbas in the loop. *PVLDB*, 5(5):478–489, 2012.

[122] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[123] S. Shalev-Shwartz and A. Tewari. Stochastic methods for l_1-regularized loss minimization. *J. Mach. Learn. Res.*, 12:1865–1892, 2011.

[124] Y. Sheng, A. Tomasic, T. Sheng, and A. Pavlo. Scheduling OLTP transactions via machine learning. *CoRR*, abs/1903.02990, 2019.

[125] N. M. Sheykhkanloo. A learning-based neural network model for the detection and classification of SQL injection attacks. *IJCWT*, 7(2):16–41, 2017.

[126] K. Sohr, M. Drouineaud, G. Ahn, and M. Gogolla. Analyzing and managing role-based access control policies. *IEEE Trans. Knowl. Data Eng.*, 20(7):924–939, 2008.

[127] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. LEO - db2's learning optimizer. In *VLDB 2001*, pages 19–28, 2001.

[128] J. Sun and G. Li. An end-to-end learning-based cost estimator. *PVLDB*, 13(3):307–319, 2019.

[129] R. Taft, N. El-Sayed, M. Serafini, Y. Lu, A. Aboulnaga, M. Stonebraker, R. Mayerhofer, and F. J. Andrade. P-store: An elastic database system with predictive provisioning. In *SIGMOD 2018*, pages 205–219, 2018.

[130] J. Tan, T. Zhang, F. Li, J. Chen, Q. Zheng, P. Zhang, H. Qiao, Y. Shi, W. Cao, and R. Zhang. ibtune: Individualized buffer tuning for large-scale cloud databases. *PVLDB*, 12(10):1221–1234, 2019.

[131] C. Tang, Z. Dong, M. Wang, Z. Wang, and H. Chen. Learned indexes for dynamic workloads. *CoRR*, abs/1902.00655, 2019.

[132] P. Tang, W. Qiu, Z. Huang, H. Lian, and G. Liu. SQL injection behavior mining based deep learning. In *ADMA 2018*, pages 445–454, 2018.

[133] S. Tian, S. Mo, L. Wang, and Z. Peng. Deep reinforcement learning-based approach to tackle topic-aware influence maximization. *Data Science and Engineering*, 5(1):1–11, 2020.

[134] I. Trummer, J. Wang, D. Maram, S. Moseley, S. Jo, and J. Antonakakis. Skinnerdb: Regret-bounded query evaluation via reinforcement learning. In *SIGMOD 2019*, pages 1153–1170, 2019.

[135] K. Tzoumas, T. Sellis, and C. S. Jensen. A reinforcement learning approach for adaptive query processing. *History*, 2008.

[136] G. Valentin, M. Zuliani, D. C. Zilio, G. M. Lohman, and A. Skelley. DB2 advisor: An optimizer smart enough to recommend its own indexes. In *ICDE 2000*, pages 101–110, 2000.

[137] M. Vartak, H. Subramanyam, W. Lee, S. Viswanathan, S. Husnoo, S. Madden, and M. Zaharia. Modeldb: a system for machine learning model management. In *SIGMOD 2016*, page 14, 2016.

[138] S. Venkataraman, Z. Yang, M. J. Franklin, B. Recht, and I. Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *NSDI 2016*, pages 363–378, 2016.

[139] F. Waas and A. Pellenkoft. Join order selection - good enough is easy. In *BNCOD 2017*.

[140] H. Wang, X. Fu, J. Xu, and H. Lu. Learned index for spatial queries. In *MDM*, pages 569–574, 2019.

[141] C. Watkins and P. Dayan. Technical note q-learning. *Machine Learning*, 8:279–292, 1992.

[142] G. M. Weiss and H. Hirsh. Learning to predict rare events in event sequences. In *KDD*, pages 359–363, 1998.

[143] C. Wu, A. Jindal, S. Amizadeh, H. Patel, W. Le, S. Qiao, and S. Rao. Towards a learning optimizer for shared clouds. *PVLDB*, 12(3):210–222, 2018.

[144] Y. Wu, J. Yu, Y. Tian, R. Sidle, and R. Barber. Designing succinct secondary indexing mechanism by exploiting column correlations. In *SIGMOD*, pages 1223–1240, 2019.

[145] C. Xu, J. Xu, H. Hu, and M. H. Au. When query authentication meets fine-grained access control: A zero-knowledge approach. In *SIGMOD 2018*, pages 147–162, 2018.

[146] N. T. G. L. Xuedi Qin, Yuyu Luo. Deepeye: An automatic big data visualization framework. *Big Data Mining and Analytics*, 1(1):75, 2018.

[147] M. Yakout, K. Ganjam, K. Chakrabarti, and S. Chaudhuri. Infogather: entity augmentation and attribute discovery by holistic matching with web tables. In *SIGMOD 2012*, pages 97–108, 2012.

[148] Z. Yang, E. Liang, A. Kamsetty, C. Wu, Y. Duan, X. Chen, P. Abbeel, J. M. Hellerstein, S. Krishnan, and I. Stoica. Selectivity estimation with deep likelihood models. *CoRR*, abs/1905.04278, 2019.

[149] H. T. F. X. J. L. Yaojing Wang, Yuan Yao. A brief review of network embedding. *Big Data Mining and Analytics*, 2(1):35, 2019.

[150] X. Yu, G. Li, and C. C. et al. Reinforcement learning with tree-lstm for join order selection. In *ICDE 2020*, pages 196–207, 2019.

[151] H. Yuan, G. Li, L. Feng, J. Sun, and Y. Han. Automatic view generation with deep learning and reinforcement learning. In *ICDE*, 2020.

[152] C. Zhang, A. Kumar, and C. Ré. Materialization optimizations for feature selection workloads. In *SIGMOD 2014*, pages 265–276, 2014.

[153] H. Zhang, B. Zhao, H. Yuan, J. Zhao, X. Yan, and F. Li. SQL injection detection based on deep belief network. In *CSAE 2019*, pages 20:1–20:6, 2019.

[154] J. Zhang, Y. Liu, K. Zhou, G. Li, Z. Xiao, B. Cheng, J. Xing, Y. Wang, T. Cheng, L. Liu, M. Ran, and Z. Li. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *SIGMOD 2019*, pages 415–432, 2019.

[155] Z. Zhang, E. R. Sparks, and M. J. Franklin. Diagnosing machine learning pipelines with fine-grained lineage. In *HPDC 2017*, pages 143–153, 2017.

[156] X. Zhou, J. Sun, G. Li, and J. Feng. Query performance prediction for concurrent queries using graph embedding. In *VLDB*, 2020.

[157] Y. Zhu, J. Liu, M. Guo, Y. Bao, W. Ma, Z. Liu, K. Song, and Y. Yang. Bestconfig: Tapping the performance potential of systems via automatic configuration tuning. *CoRR*, abs/1710.03439, 2017.

[158] D. C. Zilio, J. Rao, S. Lightstone, G. M. Lohman, A. J. Storm, C. Garcia-Arellano, and S. Fadden. DB2 design advisor: Integrated automatic physical database design. In *VLDB*, pages 1087–1097, 2004.

[159] D. C. Zilio, C. Zuzarte, S. Lightstone, W. Ma, G. M. Lohman, R. Cochrane, H. Pirahesh, L. S. Colby, J. Gryz, E. Alton, D. Liang, and G. Valentin. Recommending materialized views and indexes with IBM DB2 design advisor. In *ICAC*, pages 180–188, 2004.

Xuanhe Zhou received his bachelor's degree in Computer Science and Technology from the Beijing University of Posts and Telecommunications in 2019. He is currently a PhD student in the Department of Computer Science, Tsinghua University, Beijing, China. His research interests lie in the interdisciplinary technologies of database and machine learning.

Chengliang Chai received his bachelor's degree in Computer Science and Technology from the Harbin Institute of Technology in 2015. He is currently a PhD student in the Department of Computer Science, Tsinghua University, Beijing, China. His research interests lie in crowdsourcing data management and data mining.

Guoliang Li is currently working as a professor in the Department of Computer Science, Tsinghua University, Beijing, China. He received his PhD degree in Computer Science from Tsinghua University, Beijing, China in 2009. His research interests mainly include data cleaning and integration, spatial databases, crowdsourcing, and AI &DB co-optimization.

Ji Sun received his bachelor's degree in Computer Science from the Beijing University of Posts and Telecommunications in 2016. He is currently a PhD student in the Department of Computer Science, Tsinghua University, Beijing, China. His research interests include query processing and machine learning for database.