openGauss: An Autonomous Database System

Guoliang Li[§], Xuanhe Zhou[§], Ji Sun[§], Xiang Yu[§], Yue Han[§], Lianyuan Jin[§], Wenbo Li[§], Tianqing Wang [♣], Shifu Li [♣]

[§] Department of Computer Science,Tsinghua University, Beijing, China

[♣] Gauss Department, Huawei Company, Beijing, China

liguoliang@tsinghua.edu.cn;{zhouxuan19,sun-j16,x-yu17,han-y19,jinly20,li-wb17}@mails.tsinghua.edu.cn;{wangtianqing2,lishifu}@huawei.com

ABSTRACT

Although learning-based database optimization techniques have been studied from academia in recent years, they have not been widely deployed in commercial database systems. In this work, we build an autonomous database framework and integrate our proposed learning-based database techniques into an open-source database system openGauss. We propose effective learning-based models to build learned optimizers (including learned query rewrite, learned cost/cardinality estimation, learned join order selection and physical operator selection) and learned database advisors (including self-monitoring, self-diagnosis, self-configuration, and self-optimization). We devise an effective validation model to validate the effectiveness of learned models. We build effective training data management and model management platforms to easily deploy learned models. We have evaluated our techniques on real-world datasets and the experimental results validated the effectiveness of our techniques. We also provide our learnings of deploying learning-based techniques.

PVLDB Reference Format:
Guoliang Li, Xuanhe Zhou, Ji Sun, Xiang Yu, Yue Han, Lianyuan Jin, Wenbo Li, Tianqing Wang, Shifu Li. openGauss: An Autonomous Database System. PVLDB, 14(12): 3028 - 3041, 2021.
doi:10.14778/3476311.3476380

PVLDB Artifact Availability:
https://gitee.com/opengauss/openGauss-AI.

1 INTRODUCTION

Learning-based database optimization techniques have been widely studied [4, 19, 20, 33–36, 38, 48, 55, 59, 69, 73, 75]. Recently, commercial database systems try to deploy learning-based techniques to optimize database systems. For example, Oracle utilizes learning-based methods to automatically recommend the materialized views [3, 11]. DB2 is integrating learned cost estimation into optimizers [30, 63]. However, most of learning-based techniques have not been widely deployed in commercial database systems. In this work, we build an autonomous database framework and integrate our proposed learning-based techniques [20, 36, 37, 55, 69, 70, 74, 75] into an open-source database system, openGauss [1].

Proceedings of the VLDB Endowment, Vol. 14, No. 12 ISSN 2150-8097.
doi:10.14778/3476311.3476380

Challenges to Integrate Learning-based Techniques into openGauss. It is challenging to build an end-to-end learning-based database system. (1) *Model Selection.* If we want to replace a database component (e.g., cost estimation) with a learning-based model, we should select an appropriate machine learning (ML) model. For example, deep learning models are effective for cost estimation as they can capture the correlation between columns; deep reinforcement learning models are effective for knob tuning as they can learn knob configurations in high-dimensional continuous space. Thus we need to design effective models for learned database components. (2) *Model Validation.* It is hard to evaluate whether a learned model is effective and outperforms non-learning methods. For example, whether a knob tuning strategy really works for a workload? It requires to design a validation model to evaluate a learned model. (3) *Model Management.* Different database components may use different ML models and it is important to provide a unified ML platform to achieve a unified resource scheduling and a unified model management. (4) *Training Data Management.* Effective models require high-quality training data and it requires to effectively collect training data and manage the data to train models. To address these challenges, we build an autonomous database framework and integrate it into openGauss, with five main components.

(1) Learned Optimizers in openGauss. We propose learned query rewrite, learned cost/cardinality estimation [55], learned plan generator [69] (including join order selection and physical operator selection). Learned query rewrite uses a Monte Carlo tree search based method to judiciously rewrite a SQL query to an equivalent yet more efficient query by considering rewrite benefits and rewrite orders. Learned cost/cardinality estimation uses tree-LSTM to simultaneously estimate the cost and cardinality. Learned plan generator utilizes deep reinforcement learning to select a good join order and appropriate physical operators.

(2) Learned Database Advisors in openGauss. We develop learning-based self-monitoring, self-diagnosis, self-configuration, and self-optimization techniques to monitor, diagnose, configure, and optimize databases [37, 74]. Self-monitoring monitors databases metrics and uses the metrics to facilitate other components. Self-diagnosis uses tree-LSTM to detect the anomalies and identify the root causes of the anomalies. Self-configuration uses a deep reinforcement learning method to tune the knobs [36]. Self-optimization uses an encoder-reducer model to recommend views and uses a deep reinforcement model to recommend indexes [20, 70].

(3) `Model Validation`. To validate whether a model is effective for a workload, we propose a graph embedding based performance prediction model in openGauss [75] to validate the effectiveness of a learned model. For a model in learned optimizers or advisors, we predict the performance before deploying the model. If the performance gets better, we deploy the model; drop it otherwise.

(4) `Model Management in openGauss`. As most modes in learned optimizers and advisors are realized by combining multiple reinforcement learning or deep learning algorithms, we provide an machine-learning (ML) platform to achieve a unified resource scheduling, model management, and facilitate the provision of a unified computing platform. It encapsulates the complexity of ML and provides the capabilities of training, prediction, model management.

(5) `Training Data Management in openGauss`. We collect the runtime database metrics (e.g., resource consumption, lock/latch information), historical SQL queries (e.g., query latency) and system logs (e.g., anomalies) as training data. Since different training modules may require different training data, we judiciously organize the data, including (1) judiciously organizing the correlated columns into the same table to reduce the join overhead; (2) judiciously selecting the training data to train a learned model.

Contributions. We make the following contributions.
(1) We develop an autonomous database framework and integrate our learning-based techniques into an open-source database system.
(2) We propose effective learning-based models for learned optimizers and learned database advisors.
(3) We devise effective training data management and model management platforms to deploy learning-based modules.
(4) We propose an effective validation model to validate the effectiveness of a learned model.
(5) We evaluated our techniques and the experimental results validated the effectiveness of our techniques.
(6) All of the codes have been open-sourced[1]. Our proposed techniques have been used by real customers, e.g., the sixth biggest bank of China, *Postal Savings Bank of China*, and the biggest mobile company of China, *China Mobile*.

2 SYSTEM OVERVIEW

In this section, we present the overview of our autonomous framework. Besides the core components of traditional databases (e.g., SQL parser, optimizer, execution engine, storage engine), `openGauss` has five learning-based components, `learned optimizer`, `learned advisor`, `model validation`, `training data management`, and `model data management`.

(1) `Learned Optimizer`. Most of problems in optimizer (e.g., query rewrite, cost estimation, and plan generation) are NP-hard, and existing optimization techniques adopt heuristics, which may fall in local optimum. To address these problems, our `learned optimizer` uses learning-based techniques to improve the performance.

(*i*) `Learned Rewriter` uses *a Monte Carlo tree search based method* to rewrite a SQL query to an equivalent yet more efficient query. The basic idea is that we first build a policy tree, where the root is the original query, and a tree node is a rewritten query from its parent by applying a rewrite rule. A brute-force method enumerates all possible rewrite strategies to get a full policy tree and chooses the optimal one with the lowest cost. However it is prohibitively expensive to enumerate every possible rewritten queries.

[1]https://gitee.com/opengauss/openGauss-AI

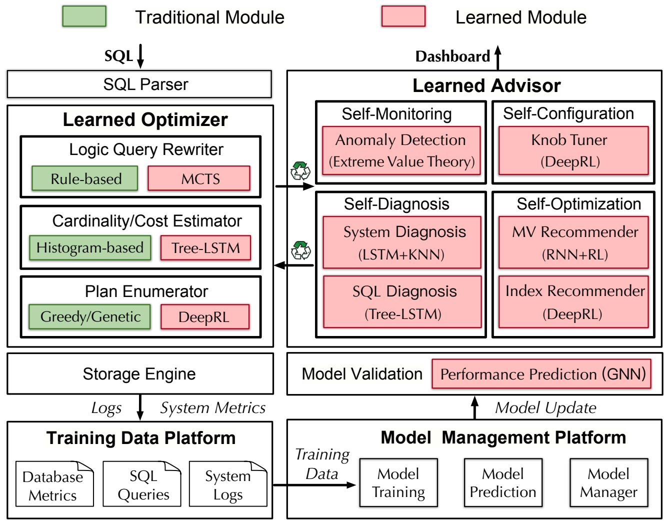


Figure 1: The `openGauss` Architecture

Thus we propose a Monte Carlo tree method, which judiciously selects a search strategy to get the best rewritten strategy.

(*ii*) `Learned Cost Estimator` [55] uses *a deep learning based method* to estimate the cost and cardinality of a query, which can capture the correlation between different columns. Note that the query plan is a tree structure, and the plan is executed in a bottom-up manner. Intuitively, the cost/cardinality of a plan should be estimated based on its sub-plans. To this end, we design a tree-structured model that matches the plan naturally, where each model can be composed of several sub-models in the same way as a plan is made up of sub-plans. We use the tree-structured model to estimate the cost/cardinality of a plan.

(*iii*) `Learned Plan Generator` [69] uses *a deep reinforcement learning based method* to generate an optimized query plan. The basic idea is to model the plan enumerator process as a Markov decision process in order to select an optimized join order and physical operator (e.g, nested-loop join, hash join or index join). We present a novel learned optimizer that uses reinforcement learning with tree-structured long short-term memory (LSTM) for join order selection. We adopt graph neural networks to capture the structures of join trees which support the updates of database schema and multi-alias of table names. The model can automatically select appropriate physical operators.

(2) `Learned Advisor`. Existing database monitor, configuration, diagnosis, optimization methods (e.g., knob tuning, slow SQL diagnosis, index/view advisor) rely on database administrators (DBAs), which is expensive and cannot adapt to large-scale instances (e.g., cloud database). Thus we propose learning-based methods for self-monitoring, self-configuration, self-diagnosis, self-optimization, in order to automatically and judiciously optimize the database.

(*i*) `Self-Monitoring [37]`. It monitors the database status and provides database runtime metrics (e.g., CPU usage, response time, running log). For `anomaly detection`, we propose a LSTM-based auto-encoder model to automatically detect the anomalies based on data distributions and metrics correlations. We use the encoder to turn database metrics into a low-dimensional representation

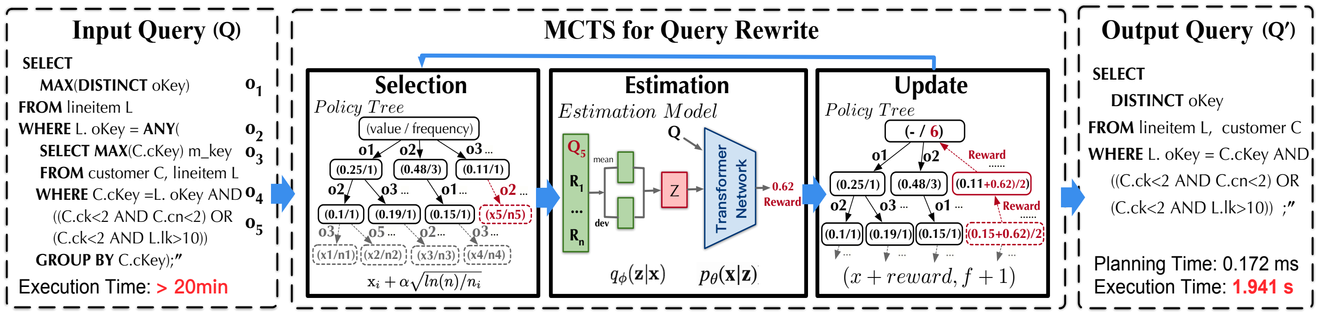

Input Query (Q)
SELECT MAX(DISTINCT oKey) FROM lineitem L WHERE L. oKey = ANY(SELECT MAX(C.cKey) m_key FROM customer C, lineitem L WHERE C.cKey =L. oKey AND ((C.ck<2 AND C.cn<2) OR (C.ck<2 AND L.lk>10)) GROUP BY C.cKey);"
Execution Time: > 20min
MCTS for Query Rewrite
Selection
Estimation
Update
Output Query (Q')
SELECT DISTINCT oKey FROM lineitem L, customer C WHERE L. oKey = C.cKey AND ((C.ck<2 AND C.cn<2) OR (C.ck<2 AND L.lk>10)) ;"
Planning Time: 0.172 ms
Execution Time: 1.941 s

Figure 2: Monte Carlo Tree Search Based Query Rewrite

and uses the decoder to restore the original data. The data that the model cannot reconstruct well is taken as an anomaly.

(*ii*) `Self-Diagnosis`. It aims to automatically diagnose the anomalies. `System Diagnosis` diagnoses the root cause of system anomalies (e.g., lock conflicts) and `SQL Diagnosis` diagnoses the root cause of SQL anomalies (e.g., slow SQL). We adopt a tree-LSTM model for self-diagnosis, which identifies the root causes by using a softmax function.

(*iii*) `Self-Configuration [36]`. It aims to automatically configure the database systems, e.g., knob tuning. `Learned Knob Tuner` adopts a deep reinforcement learning technique to tune the knobs. We use an actor-critic model to automatically select the appropriate knob values. Our model can support SQL-level, session-level and system-level knob tuning.

(*iv*) `Self-Optimization [20, 70]`. It automatically optimizes the databases, e.g., index/view advisor, for a query workload. `Learned Index Advisor` uses deep reinforcement learning to automatically recommends indexes. `Learned View Advisor` proposes an encoder-decoder model to automatically recommends views.

(3) `Model Validation`. To validate whether a model is effective for a workload, we propose a graph embedding based performance prediction model in openGauss [75], which can predict the performance if we employ a learned model. For a model in learned optimizers or advisor, we can predict the performance if we deploy the learned model. If the performance becomes better, we deploy the model; drop it otherwise. For example, if we use a deep reinforcement learning model to recommend the knob values, we predict the performance if we set these recommended values. If the performance becomes better, we deploy the recommended values.

(4) `Training Data Platform`. It collects training data from databases from several aspects. (*i*) Database metrics: database running status, e.g., queries per second (QPS), CPU usage rate, cache hit rate. These are usually represented by time-series data. (*ii*) SQL Query. It collects SQL queries and their statistics like physical plan, response time, and duration time. (*iii*) Database Log. It collects running logs. As different training modules require different training data, we judiciously organize the data, including (1) organizing the correlated columns into the same table to reduce the join overhead; (2) selecting the training data for a model.

(5) `Model Data Management`. It integrates the frequently-used ML capabilities, provides a unified application access interface, and supports the management and scheduling of ML tasks.

3 LEARNED DATABASE OPTIMIZER

We present the learned database optimizer, including learned query rewrite, cost estimation, and plan generation.

3.1 Query Rewriter

Query rewrite aims to transform a slow SQL query into an equivalent one with higher performance, which is a fundamental problem in query optimization [33, 73]. The performance of a slow SQL query (due to redundant or inefficient operations) can be improved by orders of magnitude if it is rewritten in an appropriate way (e.g., removing redundant operators, swapping two operators). For example, in Figure 2, if the query Q is rewritten into an equivalent query Q' by eliminating the redundant aggregation `MAX(c_custkey)` and pulling up the subquery, it achieves over 600x speedup.

To address this problem, in openGauss, we adopt Monte Carlo Tree Search (MCTS) in the query rewrite module. The module is composed of three parts as shown in Figure 12. Firstly, we build a *policy tree* where the root node is the input SQL query and a non-root node is a rewritten query from its parent by applying some rewrite rules. And then we utilize Monte Carlo Tree Search (MCTS) [9, 26] to efficiently explore and find rewrite orders that gain the most time reduction. We design the Upper Confidence bounds (UCB) to select promising or uncovered tree branches to expand the *policy tree* in order to find optimized rewrite queries. Secondly, we need to estimate the *potential benefit* of a tree node (an intermediate rewritten query) that may be rewritten further by other rewrite rules, and we design a deep estimation model that estimates the potential benefit of an intermediate rewritten query based on the query features, rewrite rules and table schema (e.g., index, column cardinality). Thirdly, to enhance the search efficiency, especially when the query has numerous logic operators, we propose a multi-agent MCTS algorithm that explores different rewrite orders on the *policy tree* in parallel.

Workflow. Figure 12 shows an example of finding the rewrite orders of query Q. We first initiate the *policy tree* with a root node, which denotes the input query Q. And then, with MCTS, we iteratively explore new tree nodes (new rewrite orders) on the *policy tree*. For the *policy tree* in Figure 12, we first compute UCB values for all the candidate branches, i.e., $UCB_i = x_i + \alpha\sqrt{\frac{ln(f)}{f_i}}$, where x_i is the estimated cost reduction when rewriting from node i, f_i is the frequency (the number of iterations) that the node i is selected, f is the total iteration number, and α is a hyper-parameter that adjusts the exploration times. As the branch (o_3, o_2) has the highest UCB score, we expand the branch (o_3) into (o_3, o_2), i.e., the new node denotes the query rewritten by o_3 and o_2 in order (selection). Next, we input the rewritten query Q' into the deep estimation model, which computes the reward of Q', denoting the potential cost reduction when further rewriting Q' with a transformer network (estimation). Finally, we update the existing nodes in (o_3, o_2), where the cost reduction is added by the reward and the selection

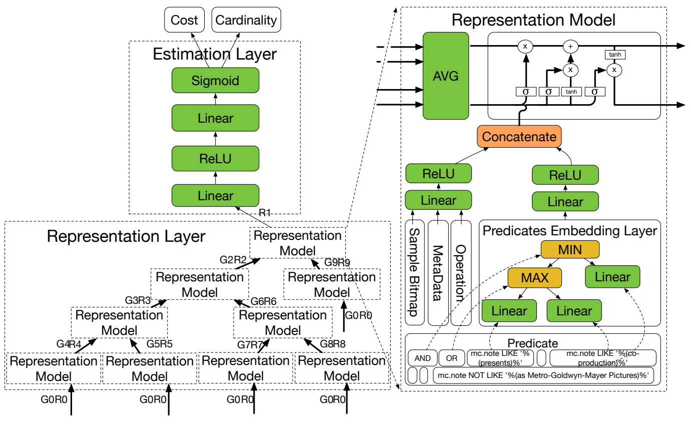


Figure 3: Tree-LSTM Based Cost Estimator

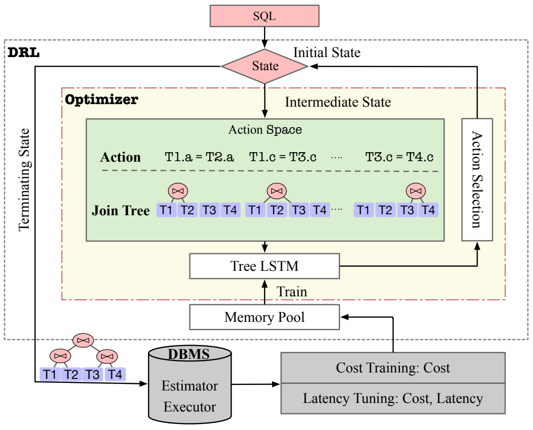


Figure 4: DRL Based Plan Generator

frequency is added by one. After reaching the maximal iteration number or no new leaf nodes, we output the rewritten query with the maximal cost reduction in the leaf nodes of the *policy tree*.

3.2 Learned Cost Estimator

Traditional cardinality estimation in modern DBMS (e.g., PostgreSQL) still brings large errors, especially when the number of involved columns are large, because they can hardly capture the correlations of different columns. For example, histogram-based methods only capture distribution on a single column; sampling-based methods suffer from 0-tuple problem for high dimensional data. Recently, neural network is utilized for solving cost estimation problems [21, 25, 46, 49, 66, 67]. In openGauss, we adopt a tree-structured LSTM model to estimate both cardinality and cost for arbitrary execution plan and outperforms traditional methods [55]. Figure 3 shows the workflow and design of tree-LSTM based cost estimator.

3.2.1 Workflow. For offline training, the training data are collected from historical queries, which are encoded into tensors by *Feature Extractor*. Then the training data is fed into the *Training Model* and the model updates weights by back-propagating based on current training loss. For online cost estimation, when the query optimizer asks the cost of a plan, *Feature Extractor* encodes it in a up-down manner recursively. If the sub-plan rooted at the current node has been evaluated before, it extracts representation from *Representation Memory Pool*, which stores a mapping from a query plan to its estimated cost. If the current sub-plan is new, *Feature Extractor* encodes the root and goes to its children nodes. We input the encoded plan vector into *Tree-structured Model*, and then the model evaluates the cost and cardinality of the plan and returns them to the query optimizer. Finally, the estimator puts all the representations of 'new' sub-plans into *Representation Memory Pool.*

3.2.2 Model Design. Cost estimation model has three components, *embedding layer*, *representation layer* and *estimation layer*.

Embedding layer. The embedding layer embeds a sparse vector to a dense vector. For operations, meta data, predicates, openGauss uses a one-hot vector. We also use a fix-sized 0-1 vector to capture some samples, where each bit denotes whether the corresponding tuple satisfies the predicate of the query node. If the data tuple matches the predicate, the corresponding bit is 1; 0 otherwise. openGauss uses a one-layer fully connected neural network with ReLU activator to embed these vectors. However, the structure of the *Predicate* vector is complicated, because it contains multiple AND/OR semantics. openGauss uses a MIN pooling layer to represent OR operator and a MAX pooling layer to represent AND operator. For string data, openGauss uses a Skip-gram model to learn representations of substrings to appear in queries.

Representation Layer. The representation layer aims to capture the global cost from leaf nodes to the root and avoid information loss (e.g., correlation between columns). Each representation model has three inputs, the *embedding vector E*, the representation vector $[G^l_{t-1}, R^l_{t-1}]$ of its left child, and the representation vector $[G^r_{t-1}, R^r_{t-1}]$ of its right child.

Estimation Layer. The estimation layer is the final layer which outputs the cost/cardinality results according to the representation of the plan structure. openGauss takes two-layer fully connected neural network as an estimation layer.

3.3 Learned Plan Enumerator

The plan enumeration usually consists of join order selection and physical operation selection. Traditional methods search the solution space to find a good plan based on the cost estimation model. The dynamic programming based algorithms [22] enumerate the solution space to find the best plan but takes exponential time complexity. Heuristic methods [62] prune the solution space to speed up the search operation, but often generate poor results. Recently, some studies [29, 43] apply deep reinforcement learning (DRL) [5] on plan enumeration. Similar to dynamic programming, these works model the plan generation process as a Markov decision process (MDP) and reduce the search space from exponential level to polynomial level. However they adopt a fixed-length representation of the join tree, which makes them hard to handle the schema updates and the multi-aliases in a query. To address these problems, we propose a learned plan enumerator [69] to make the optimizer generate the plan with low execution time and easy to handle the changes in the schema. We apply the Deep Q-Network (DQN) [47] with Tree-LSTM to find the plan of a SQL query.

Workflow. Figure 4 shows the framework and workflow of the openGauss's plan enumerator. Given a query q, the plan enumerator first initializes an empty state s_0 which contains only the basic

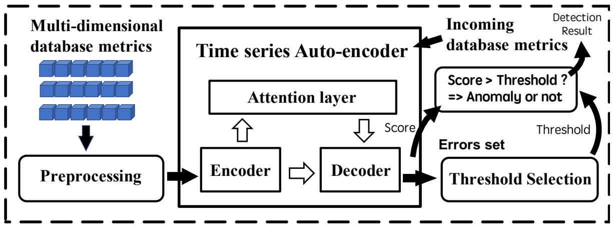


Figure 5: LSTM-Based Anomaly Detection

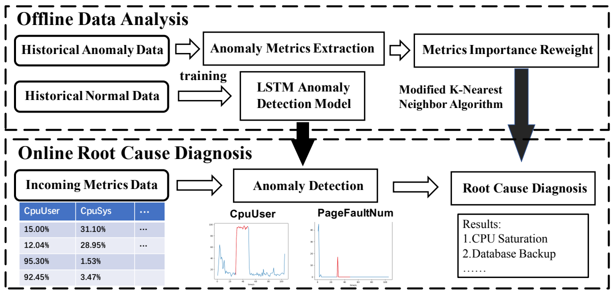
information of the query. The intermediate state s_i is a plan forest that contains many partial plan tree. Each time, RL policy will choose the next two tables t_{i1}, t_{i2} to join and which operation o_i(hashjoin, nestloop, etc) to be used. An action a_i is the combination of the join and the operation $a_i = (t_{i1}, t_{i2}, o_i)$. The action is retrieved according to the q-value from our tree-LSTM neural network $Q(s_i, t_{i1}, t_{i2}, o_i)$. $a_i = \arg\min_{t_1,t_2} Q(s_i, t_1, t_2, o_i)$. RL continuously chooses the next action until the whole plan tree is built. We divide the training process into two steps, cost training and latency tuning. In cost training, the cost of the plans will be used to train the TreeLSTM, which can be efficiently trained to get initial results. In latency tuning, only a few plans' latency will be used as training data to fine-tune the neural network.

Tree-LSTM Based Q-network. DQN uses the Q-network to estimate and find the next action. In the plan enumerator, Q-network will choose which plan tree is better. To capture the tree structure of the plan tree, we design a Tree-LSTM based Q-network Q to estimate each plan tree. We have three kinds of leaf nodes in a plan tree: column, table, operation. First, we use the predicate (e.g., selectivity) to represent each column c as $R(c)$. Then a pooling layer is to collect information of all columns in each table to give the table's representation $R(t) = Pool(R(c_{t1}), R(c_{t2}), \ldots)$. The representation of the operator c is a one-hot vector. For example, if we have three join operations (Hash Join, Nested Loop, Merge Join) and o = Hash Join, we can set $R(o) = (1, 0, 0)$. The column representations $R(c)$, table representations $R(t)$ and operation representations $R(o)$ are the leaves of the plan tree. The Depth-First-Search is used to traverse the plan trees to give the final representations. The tree-LSTM layer T will be applied to each node to get the representation of the plan tree. For an intermediate state s_i that not all tables are joined together, the plan state is a plan forest F which contains several plan trees $F = \{tree_1, tree_2, \ldots\}$. A child-sum (CS) layer is used as a root layer to capture the forest information $R(s_i) = R(F) = CS(R(tree_1), R(tree_2))$. After we get the representation of the plan state $R(s_i)$ of query q, a dense layer is used to construct the final Q-network $Q = \text{Dense}(R(s_i), q)$

4 LEARNED DATABASE ADVISOR

4.1 Self-Monitoring

Detecting database anomalies at runtime is very important. However this is rather hard to manually handle in time. Hence, we develop a self-monitoring module to proactively discover the anomalies. We monitor 500+ database metrics (e.g., response time, CPU usage, memory usage, disk used space, cache hit rate) and then detect the anomalies based on the metrics. For example, when the metric is far from the prediction result, there may be an anomaly with high probability. Then the self-diagnosis module diagnoses the root cause, and calls the corresponding optimization functions to repair this anomaly. Thus, the database monitor can drive the



CpuUser	CpuSys	...
15.00%	31.10%	...
12.04%	28.95%	...
95.30%	1.53%	
92.45%	3.47%	

Results:
1.CPU Saturation
2.Database Backup
……

Figure 6: System-level Diagnosis

subsequent operations (e.g., driving the database diagnosis module for root cause analysis, driving the optimizations for knob tuning).

Anomaly Detection. When an anomaly happens inside or outside the database, it can be reflected by corresponding metrics and system logs [14]. Thus, openGauss diagnoses in real-time by analyzing database and operating system metrics. A unified training data platform continuously collects metrics and logs from the database and operating system such as QPS, running log. Putting these data together forms time-series data. Traditional statistics-based anomaly detection algorithms are designed for one-dimensional data. These methods neglect correlation between time series data thus cannot achieve high accuracy. Deep learning based algorithms take correlation into account but require labeled data. To address these problems, we propose to employ a reconstruction-based algorithm to discover anomalies. The normal time-series data always have a regular pattern and abnormal patterns will have large possibilities to be anomalies. We employ a LSTM-based auto-encoder with an attention layer. The original time-series data are encoded into a low-dimensional representation. The decoder parses the representation and tries to restore the original data. The training loss is reconstruction quality. The model learns the distribution of these multi-dimensional data and obtains reconstruction ability. The data that cannot be reconstructed (error exceeds the threshold) are reported as an anomaly. Our system applies a statistical method "Extreme Value Theory" to determine the dynamic threshold. Hence, users need to set system sensitivity like 1% or 5%, and it will calculate the corresponding threshold by historical data. In Figure 5, openGauss first conducts standardization on training data. Then the processed data are fed into time series auto-encoder for updating model's parameters. After model has the ability to reconstruct normal database metrics. openGauss collects reconstruction errors and calculate threshold.

4.2 Self-Diagnosis

Root-cause diagnosis is vital to autonomous databases. We consider system diagnosis and SQL diagnosis. The former analyzes the root cause of system anomalies, e.g., IO contention, network congestion, insufficient disk space; the latter analyzes the root cause of slow SQLs, e.g., lock conflicts and no indexes.

4.2.1 System-Level Diagnosis. The database performance may degrade due to some inside or outside factors like adding index on frequently updating tables or network congestion. Existing databases usually employ experienced DBA to diagnose the root causes manually, which is a difficult task since the state metrics for a database are complex and lack of domain knowledge. Previous automatic diagnosis methods rely on expert knowledge heavily [7] or introduce non-negligible extra cost [13].

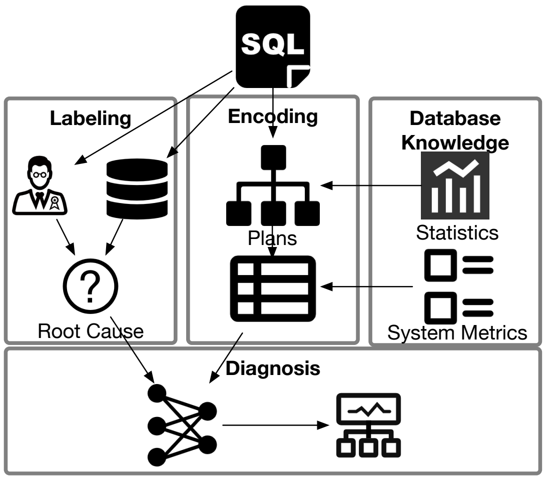


Figure 7: SQL-level Diagnosis

To address this problem, we propose a system-level diagnosis framework. It regards diagnosis as a classification problem and finds root cause by making use of features in metrics data. In other words, it is lightweight and will not affect the performance. In addition, it optimizes the diagnosis model by analyzing historical data so experts just need to label a few anomaly cases. The framework consists of the offline and online stages as shown in Figure 6. In the offline stage, it collects two types of metrics: normal data and abnormal data. It utilizes normal data to train an LSTM-based auto-encoder model. Abnormal data consist of anomaly cases and each case contains database metrics data and the root cause label. For abnormal data, it extracts representative metrics and stores them as a knowledge base. We apply Kolmogorov-Smirnov test to measure the state of every dimension metrics data quantitatively to unify these information in one model. Thus it encodes each anomaly case into an vector. Each dimension represents the state of a database metrics. In the online stage, once openGauss detects an anomaly, it calls the root cause module, which first adopts Kolmogorov-Smirnov test on each database metrics to generate the anomaly vector, and then finds similar anomaly cases and locates the root cause.

4.2.2 SQL-level Diagnosis. SQL diagnosis module aims to find the time-consuming operator of a SQL without actually running it.

Challenges (1) *The execution plan does not exist.* It's time-consuming to record execution plans in log files, and thus the execution plan of a SQL query is not kept in our metrics. (2) *A one-model-fits-all approach is necessary.* As the estimated cost by statistic-based optimizer may not be accurate and not all databases have learning-based cost estimators, we need another model to decide the root cause of slow queries. Moreover, the SQL diagnosis module in openGauss should support different datasets and workloads with complex grammar features. (3) SQL-level root cause diagnosis should also consider system metrics. System status have effects on operator performance. For example, IO contention makes sequence scan operator slower.

Workflow. Figure 7 shows the architecture of SQL-level diagnosis module. For offline model training, openGauss first collects training queries labeled by running logs or experts, a root cause label of each query is the slowest operator. Then the SQL query is transformed into an execution plan by *Plan Generator* using database statistics (e.g., index columns histogram, table cardinality, number of distinct values). Next, each plan is encoded into a vector by depth-first searching the nodes of the plan tree and concatenate with real-time system metrics (e.g., CPU/IO load). Next the vectors are fed into training models, and a gradient-based optimizer compares output with label and conducts parameters optimization. For online diagnosing, the input query is also transformed into an execution plan by *Plan Generator*, and the plan is encoded into plan-structured vectors and concatenated with real-time system metrics in *Feature Encoder*. At last, the model outputs root cause for the query.

Schema-free Encoding. Data statistics have been used for generating execution plans, and our query encoding is based on plan trees, we depth-first search the whole plan structure and encode the plan into a preorder vector. To generalize model to different databases, we encode each column or table with statistics (instead of one-hot). Specifically, for a scan operator, we encode the scan types (e.g., sequential scan, index scan) as a one-hot vector, and encode normalized table cardinality and estimated cost/cardinality as float values. For join operator, we encode the join types as a one-hot vector, and encode estimated cost/cardinality as float values. For aggregation and sort operators, we encode the number of rows and distinct values on group-by keys as float values, and encode estimated cost/cardinality as float values. For nested sub-plan, we encode it as a new execution plan and insert into an array. System metrics for each query is also encoded as a vector.

Model Design. We adopt a light-weight model for SQL diagnosis. As system metric vector is shorter than plan vector, we use two fully-connected modules to learn representations of plan and system metrics separately and transform them into two vectors with the same length, and then concatenate them as a vector and input it into output layer. When sub-plans exist, representations of different sub-plans are combined into a single vector with an average pooling layer. In the output layer, the model outputs probabilities for all possible root causes by using a softmax function, and select the one with the largest possibility as the root cause.

4.3 Self-Configuration

Databases have hundreds of tunable knobs (e.g., over 400 knobs in openGauss) and it is important to set appropriate values for the knobs, such as memory management, logging, concurrency control. However, manually tuning the knobs is time-consuming, and other automatic tuning methods (e.g., heuristic sampling or machine learning methods [4, 36, 71, 76]) cannot efficiently tune the knobs. For example, ML methods [4, 36, 71] first train models on small datasets and require retraining before migrating to large datasets. Besides, in a practical scenario, it is important to initialize the DRL model with a near-optimal state (i.e., pre-trained model) for quick convergence.

To address those problems, we propose a hybrid tuning module, which has four parts: (1) `DB Side`: the DB_Agent extracts the characteristics of the database instance, including the internal status and current knob settings. (2) `Algorithm Side` contains the tuning algorithms, including search-based algorithms (e.g., Bayesian optimization, particle swarm algorithm) and deep reinforcement learning (e.g., Q-learning, DDPG [40]). (3) `Tuning Side` iteratively conducts knob tuning. At each step, it inputs the database status and query features (the state vector) and outputs recommended knob values using the tuning algorithm in `Algorithm Side`. (4)

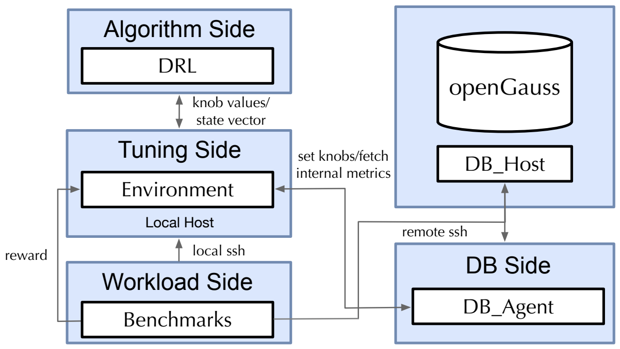


Figure 8: DRL-Based Knob Tuning

`Workload Side` conducts stress tests by the clients, which runs benchmark tasks and feeds back the execution performance (e.g., average throughput). To achieve practical online tuning, we provide three tuning modes:
1) `Rule Mode` logs in the database, obtains the state of the currently running workload, and generates a knob recommendation based on expert experience. The report includes the warnings of unreasonable knob values (e.g., "too high *max_connections* with small number of CPU cores"), potential risks, and recommended knob value ranges. Since there are hundreds of knobs, the `Rule Mode` can first reduce the tuning space, based on which the `Learned Mode` further tunes the database for higher performance.
2) `Training Mode`: with the workload type provided by the clients, we iteratively modify knob values and execute test workloads. And we collect the execution information as training samples and use the samples to train the RL model so that the user can load the model through the `Tune Mode` for finer-granularity tuning later;
3) `DRL Mode` uses optimization algorithms for knob tuning. Currently, two types of algorithms are supported: one is global search (e.g., particle swarm algorithm), and the other is deep reinforcement learning. DRL can achieve the highest performance, but needs to generate a well-trained model; on the other hand, the global search algorithm does not need to be trained in advance and can directly search knob configurations and tune the database.

4.4 Self-Optimization

4.4.1 MV Recommender. Materialized views (MV) are rather important in DBMS that can significantly improve the query performance based on the space-for-time trade-off principle. However, it is hard to automatically recommend and maintain MVs without DBAs. Furthermore, even DBAs cannot handle large-scale databases, especially cloud databases that have millions of instances and millions of users. Recently, there are some studies on MV recommendation [20, 23, 70]. There are two challenges: (1) There are many potential MVs and finding an optional set of MVs is time-consuming; (2) Estimating the cost of MV-aware optimized query is essential while the traditional estimation method is not accurate enough. To address these challenges, we adopt an learning-based MV recommendation method as shown in Figure 9.

MV Candidate Generation. We analyze the workload to find common subqueries for MV candidate generation, where a subquery is a subtree of the syntax tree for relational algebra. Common subqueries are the equivalent or similar rewritten subqueries among different queries. Common subqueries with a high frequency and computation cost will be selected as MV candidates.

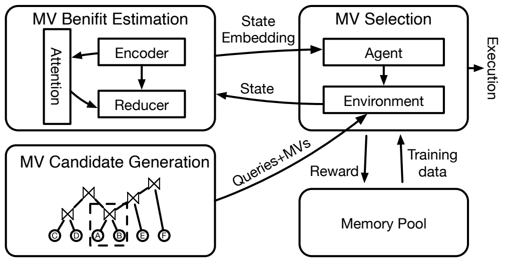


Figure 9: Encoder-Reducer Based View Recommender

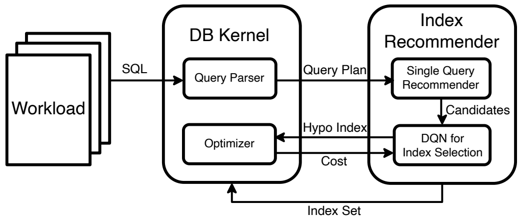


Figure 10: DRL-Based Index Recommender

MV Estimation. We regard `benefit` as the saved execution time from executing a query by making use of a set of views. We propose an `Encoder-Reducer` model to predict the benefit of using a view to answer a query by using features of queries and MVs. Meanwhile, we estimate the cost of a view including the space cost and the view generation time cost using `Encoder-Reducer` model.

MV Selection. Given a space budget, we select a subset of MV candidates to maximize the total benefit of answering queries within the space budget. We model this selection problem as an integer programming problem and propose a reinforcement learning (RL) model to address it.

4.4.2 Index Recommender. Indexes are essential for the database system to achieve high performance. A proper set of indexes can significantly speed up query execution. Nevertheless, index recommendation is a complex and challenging problem. First, indexes mutually affect performance, making it difficult to quantify the gain of a new index accurately. Second, the number of index combinations is enormous. For example, the developers may just build indexes on any accessed columns and cause great space waste. And it is a laborious work to filter out useless indexes. To address these problems, we propose a learning-based method with three steps. Firstly, to reduce the computation cost, we extract representative queries to represent the entire workload, and recommend indexes for each query independently. We parse the query and extract the potentially useful columns from different clauses. According to the table and column statistics, these columns are selected and scored. The indexes of each query together constitute the set of candidate indexes for the given workload. Secondly, to estimate the benefit of candidate indexes, we use the hypothetical index to simulate the creation of actual indexes, which avoids the time and space overhead required for creating a physical index. Based on the hypothetical index, we can evaluate the index's impact on the specified query statement by invoking the database's optimizer. We estimate the accurate cost (e.g., disk space usage, creation time, and total cost of using indexes) for hypothetical indexes. Thirdly, a subset of index candidates is selected to maximize the benefit. We formulate the problem as a deep reinforcement learning (DRL) problem and apply Deep Q-Network (DQN) to pick indexes.

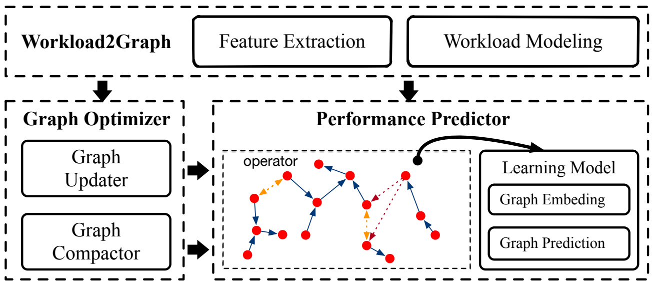


Figure 11: Embedding Based Model Validation

5 MODEL VALIDATION

Model validation is a vital task in learning-based database systems. We utilize it to validate learning-based models, which can serve many database applications (e.g., knob tuning and monitoring) and support them to meet the service level agreements (SLAs).

To validate a model, we predict the performance if we deploy the learned model. If the performance becomes better, we deploy the model; drop it otherwise. Traditional prediction methods are designed for single queries [65, 72]. They cannot effectively predict the performance for concurrent queries, because they cannot capture the correlations between concurrent queries, e.g., resource competition and mutual effect, which can significantly affect concurrent queries' performance. To address this issue, we adopt a graph embedding based performance prediction method , `Workload2Graph`, in `openGauss` [75], which provides real-time query performance prediction for concurrent and dynamic workloads. Firstly, we utilize a graph model to capture the workload characteristics, in which the vertices represent operator features extracted from query plans, and edges between two operators denote the query correlations and resource competitions between them. Secondly, we feed the workload graph into the prediction model, in which we propose a graph embedding algorithm to embed graph features in operator level (e.g., the operator features and $\mathcal{K}$-hop neighbors) [18, 51, 57, 58], and utilize a deep learning model to predict query performance. Moreover, if a graph is too large, it may affect the prediction efficiency. Hence, we propose a graph compaction algorithm, which drops redundant vertices and combines similar vertices.

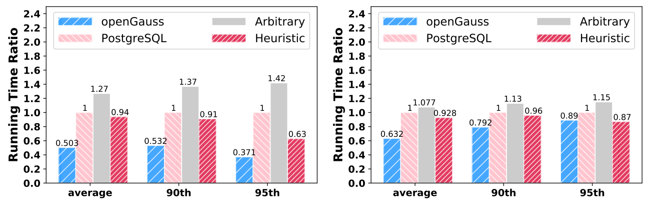
Workflow. As shown in Figure 11, given a query workload with multiple concurrent queries, we aim to predict execution time for each query. First, `Workload2Graph` extracts feature from the query workload that may affect the performance of concurrent queries. For instance, it extracts operators from the query plans and obtains the correlations between different operators, e.g., data sharing between operators and lock conflicts between operators. Moreover, it also obtains the statistics, configurations, and states from database system views. Then `Workload2Graph` utilizes these features to characterize the behaviors of concurrent queries in the form of a graph. Lastly, `PerformancePredictor` adopts a graph-based learning model to embed the graph and predicts the query performance using a deep learning model. For a large graph with high concurrency queries, we conduct semi-supervised training (i.e., with some labeled vertices for prediction accuracy and other unlabeled vertices for prediction stability) and utilize batch gradient descent to enhance training efficiency.

6 EXPERIMENT

We compared with state-of-the-art learning-based techniques and an open-source database `PostgreSQL` [2].

Table 1: Datasets.

Name	Mode	Table	Size(GB)	#Query
`JOB`	RO	21	3.7	113
`TPC-C`	RW	9	1.30	912,176
`TPC-H`	RO	8	1.47	22
`TPC-DS`	RO	25	1.35	99



(a) TPC-H (b) JOB

Figure 12: Query Rewrite Comparison (90th denotes the queries that run slower than 90% queries).

System Setting. We used TensorFlow and Pytorch as backend engines to train learned models and the trained model was used inside the database kernel. We implemented a data collection module inside the database kernel, which collected statistical and system metrics and called TensorFlow to train models via the RPC protocol.

Datasets. To facilitate re-producing the results, we used four widely-used datasets as shown in Table 1.

6.1 Learned Optimizer

6.1.1 Learned Query Rewrite. We conducted experiments on `TPC-H` and `JOB` to compare the query rewrite in `openGauss` with three baselines, i.e., top-down query rewrite in `PostgreSQL`, arbitrary query rewrite that selected a random rewrite order (`Arbitrary`), and heuristic query rewrite that greedily used rewrite rules (`Heuristic`). `openGauss` and `PostgreSQL` were implemented in database engines. For `Arbitrary` and `Heuristic`, we extracted 82 rewrite rules in Calcite [6] and rewrote queries with corresponding strategies. We used a tool SQL-smith (https://github.com/anse1/sqlsmith) to generate 15,750 and 10,673 slow queries (>1s) for `TPC-H` and `JOB` respectively.

As shown in Figures 12(a)-12(b), `openGauss` outperformed the other methods in all the cases, i.e., over 49.7% execution time reduction for `TPC-H` and over 36.8% for `JOB`. The reasons were two-fold. Firstly, `openGauss` explored rewrite orders with lower execution cost than the default top-down order in `PostgreSQL`. For example, with an outer join, `PostgreSQL` cannot push down predicates to the input table, while `openGauss` solved the problem by first converting the outer-join into an inner-join and then pushing down the predicate. Secondly, the estimation model in `openGauss` predicted the potential cost reduction, with which `openGauss` selected promising rewrite orders. Besides, `openGauss` worked better on `TPC-H` than `JOB`, because `TPC-H` queries contained many subqueries that were removed by *query rewrite*, while the multi-joins in `JOB` queries would be further optimized by plan enumerator.

6.1.2 Learned Cost Estimation. We conducted experiments on `JOB` to compare cost estimator in `openGauss` with baselines, including cost estimator in popular databases (`PostgreSQL`, `MySQL` and `Oracle`). Our methods included tree-LSTM based estimator with different string embeddings (hash `TLSTM-Hash` and embedding `TLSTM-Emb`), and tree-LSTM model with MIN-MAX pooling for complex predicate (`TPool`). Table 2 showed the results, and we made the following observations.

Table 2: Cost Estimation – Test errors on the JOB workload

Cardinality	median	90th	95th	99th	max	mean	Cost	median	90th	95th	99th	max	mean
PostgreSQL	184	8303	34204	106000	670000	10416	PostgreSQL	4.90	80.8	104	3577	4920	105
MySQL	104	28157	213471	1630689	2487611	60229	MySQL	7.94	691	1014	1568	1943	173
Oracle	119	55446	179106	697790	927648	34493	Oracle	6.63	149	246	630	1274	55.3
openGauss (TLSTM-Hash)	11.1	207	359	824	1371	83.3	TLSTM-Hash	4.47	53.6	149	239	478	24.1
openGauss (TLSTM-Emb)	11.6	181	339	777	1142	70.2	TLSTM-Emb	4.12	18.1	44.1	105	166	10.3
openGauss (TPool)	**10.1**	**74.7**	**193**	**679**	**798**	**47.5**	TPool	**4.07**	**11.6**	**17.5**	**63.1**	**67.3**	**7.06**

Table 3: Plan Generation – GMRL to DP

	JOB	TPC-H
openGauss	**0.67**	**0.92**
ReJoin	1.14	0.96
QP100	NA	1.03
QP1000	1.90	1.00
Skinner-C	0.89	1.03
DQ	1.23	0.99

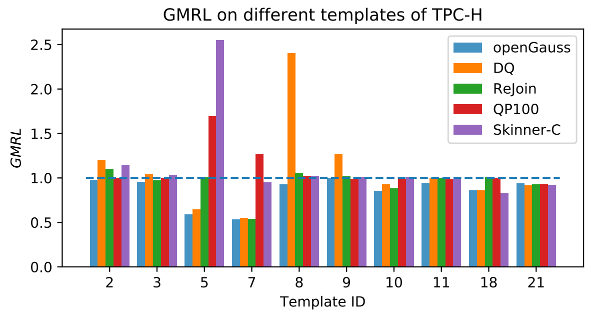


Figure 13: Plan Generation–GMRL on TPC-H (DP is 1).

Summary. Traditional methods (PostgreSQL,MySQL,Oracle) had large errors on cardinality estimation for harder queries in the JOB workload. The reason was that the distribution passed to the root of the plan was not accurate using statistical or sampling methods, and traditional methods estimated cardinality of most of queries as 1 (the true value is from 0 to millions). The cost estimation error was less than cardinality estimation, and the learned methods still outperformed traditional methods by 1-2 orders of magnitude. The difference between TLSTM-Emb and TPool was the structure of the predicate embedding model, and TPool outperformed TLSTM-Emb on all the cardinality errors and cost errors, because the tree model with Min-Max Pooling represented the compound predicate better and trained a more robust model for cardinality and cost estimation.

6.1.3 Learned Plan Enumerator. We conducted experiments on JOB and TPC-H to compare the latency of the plans generated by the learned plan enumerator in openGauss with the baselines. We chose dynamic programming(DP) in PostgreSQL, two DQL methods DQ [29] and ReJoin [43], a heuristic method QP-1000(quick pick 1000 plans) [62], and the Monte-Carlo tree search-based methods skinner-C [60] as our baselines. We used a geometric average based on Geometric Mean Relevant Latency (GMRL) as the metric. The DP was the baseline in GMRL.

$$GMRL = (\Pi_{i=1}^{n} \frac{Latency(q_i)}{Latency_{DP}(q_i)})^{\frac{1}{n}}$$

Table 3 showed the overall GMRL of different methods on JOB and TPC-H datasets. openGauss outperformed all other methods on two benchmarks. The GMRL lower than 1 indicated that openGauss achieved a better plan than DP on latency. Specifically, the GMRL of openGauss was 0.67 on JOB means that the plan of openGauss took 67% time of DP on average. The queries in TPC-H were typically short (< 8 relations) which limited the search space. It made all methods get similar GMRL on TPC-H. The 0.92 of openGauss showed that it saved the 8% time compared with DP. We grouped queries by their templates to analyze which queries openGauss can optimize well. Figure 13 showed the GMRL on TPC-H. All methods generated similar plans on most queries. For T5 and T7, DRL based methods found better plans, compared with other methods.

Table 4: Knob Tuning Comparison – openGauss (R) denotes rule based tuning, openGauss (D) denotes DRL based tuning.

	TPC-H (s)	JOB (s)	TPC-C (tpmC)
PostgresQL	121.3	220.19	5552
DBA	95.1	193.37	7023
openGauss (R)	94.3	192.81	7574
openGauss (D)	**82.7**	**163.88**	**12118.4**

6.2 Learned Advisor

6.2.1 Learned Knob Tuning. We compared the performance of knob tuning with DBA and PostgreSQL. Table 4 showed the results. openGauss outperformed PostgreSQL and DBA in all the cases. The reasons were two fold. First, the Rule Mode in openGauss reduced the value ranges and recommends knob values based on rules of thumb, and achieved better tuning performance than DBA. Second, the DRL Mode in openGauss further explored the experience of Rule Mode. As it utilized an exploitation-and-exploration policy to efficiently search better knobs settings, openGauss achieved better performance than DBA, e.g., over 15% latency reduction for JOB and TPC-H. The Rule Mode had lower tuning performance than the DRL Mode, but gained higher tuning efficiency (e.g., 4s for the Rule Mode and 8min for the DRL Mode in TPC-H). The reasons were two fold. Firstly, the DRL Mode adjusted the knob values based on the reward values, the execution results and causes overhead. Secondly, for a new scenario, the DRL Mode took time to fine-tune the reinforcement learning model, which learned new mapping from the workload and database state to obtain ideal knob settings.

6.2.2 Learned View Advisor. To evaluate view advisor, we evaluated the module on different budgets and compared our DQN model with BIGSUB [23] and traditional algorithms on JOB.

(1) TopValue: A greedy algorithm, using the metric of sum benefit for each MV. The sum benefit of an MV was the sum of the benefit of using this MV answering each query. MVs with top sum benefit was selected within the budget. (2) TopUValue: A greedy algorithm, using the metric of unit benefit, sum benefit/size, for each MV. (3) TopFreq: A greedy algorithm, using the metric of frequency for each MV. (4) BIGSUB [23]: An iterative method. It optimized views and queries separately in each iteration. It first flipped selection state of views by a specified probability, and then selected views to optimize queries using an integer linear programming solver. (5) openGaussNS: Our DQN model without the semantic vector in state representation from Encoder-Reducer model. (6) openGauss: Our

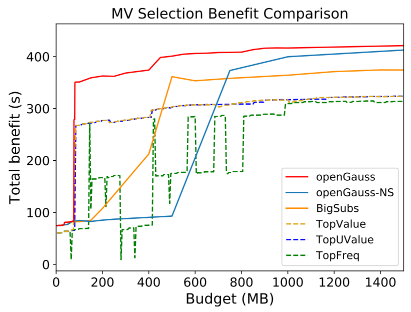


Figure 14: View Advisor – MV Selection (JOB).

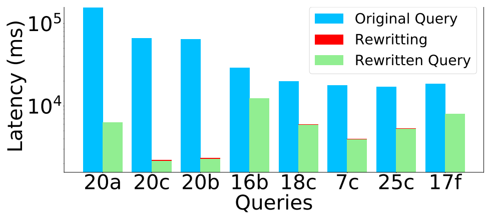


Figure 15: View Advisor (JOB).

DQN model with the semantic vector from the Encoder-Reducer model. All methods were based on our benefit/cost estimation. We compared the effectiveness of these methods on optimizing the JOB workload under different budget size from 5MB to 2000MB. The result was shown in Figure 14.

openGauss vs Heuristics. openGauss outperformed TopValue, TopUValue and TopFreq. The reasons were two-fold. Firstly, openGauss estimated the benefit of utilizing multiple MVs while the greedy methods cannot, because they were expensive to enumerate all combinations. Thus, greedy methods approximated the benefit by summing up the individual benefit which were not accurate. Secondly, the performances of greedy methods were not stable during the increase of budget while openGauss grew stable. The reason was that greedy methods were more likely to fall in local optimum, and they selected MVs with higher benefit or unit benefit, but higher benefit leaded to larger size that wasted the budget. While openGauss adjusted the earlier selection in the subsequent iterations. When an MV fell in local optimum, DQN model preferred not to select this MV.

openGauss vs BIGSUB. openGauss outperformed BIGSUB by 12.5%. The reasons were two-fold. Firstly, BIGSUB flipped a view by the probability that relied on the benefit and cost of this view. Views were considered independent with each other. openGauss captured the correlation between views. Secondly, BIGSUB might fall in local optimum. openGauss learned to select views and avoided local optimum as low rewards made the model to change the action.

Efficiency on MV-aware query rewriting. We evaluated the latency of our MV-aware query rewriting. We rewrote the queries in the JOB workload and compared the latency of original queries with the total latency of rewritten queries and rewriting. Figure 15 showed the result. The average query rewriting latency in JOB was 64.75ms which was small compared to the slow queries.

Summary. Our DQN model and semantic vector improved the quality of MV selection.

6.2.3 Learned Index Advisor. To evaluate learned index advisor in openGauss, we conducted the experiments on TPC-H and TPC-C and compared our methods with default indexes and DBA designed indexes. Table 5 showed the results. We observed that openGauss outperformed DBA and default indexes on both workloads. This indicated that openGauss identified important indexes for both OLAP and OLTP systems, because we encoded system statistics into state representation so that our index advisor could update index configuration dynamically.

Table 5: Index Advisor

	TPC-H (s)	TPC-C (tpmC)
openGauss	**122.9**	**10202**
DBA	130.1	10001
Default	140.8	9700

Table 6: Anomaly Detection(TPC-C)

	Precision	Recall	F1-score
openGauss	**0.795**	0.776	**0.785**
VAE	0.302	**0.821**	0.441
GAN	0.554	0.745	0.635

6.2.4 Anomaly Detection. We compared anomaly detection with two state-of-the-art methods, VAE-based method(VAE)[41] and GAN-based methods(GAN)[32]. The results were shown in Table 6. We observed that openGauss outperformed VAE and GAN, because database monitoring data were noisy and influenced by workload and deep generative methods cannot model them properly. Our method achieved higher performance, because our auto-decoder model could effectively capture the data distributions and patterns.

6.2.5 SQL Diagnosis. We compared with *query-embedding based method*, *cost-based method* and *plan-embedding based method*. *Query-embedding based method* encoded SQL into vectors with query features and real-time system metrics. *Cost-based method* used cost estimated by using statistics-based cost estimator in PostgreSQL to determine the root cause. *Plan-embedding based method* was adopted in openGauss, and it considered both execution operators and system metrics. We reported the accuracy and latency of these methods. Table 7 showed the results. For accuracy, openGauss outperformed PostgreSQL estimator by around 10%, and outperformed query-embedding method by around 20%. That's because feature and model designed by openGauss considered both SQL execution path and real-time system metrics, and it supported different schemas. For latency, *query-embedding based method* outperformed openGauss due to ignoring execution plan when encoding, but this caused worse accuracy with limited training queries. openGauss was reasonable because it could achieve near optimal accuracy with 4 orders of magnitude less time comparing with diagnosis by actually running the query.

6.2.6 System-Level Diagnosis. We compared system-level diagnosis with two baselines and state-of-the-art DBSherlock [68]. Two baselines adopted decision tree and normal *k*-nearest neighbor as classifiers. DBSherlock detected metrics predicates between normal and abnormal data and built corresponding causal model for each type of anomalies. The results were shown in Table 8. openGauss outperformed the other three methods, because our method emphasized metrics with more importance and achieved higher accuracy. DBSherlock cannot adapt to complex workload since the causal models were sensitive to training data.

Our learning-based techniques have been used by the real customers, e.g., the sixth biggest bank of China, *Postal Savings Bank of China*. For example, learning-based knob tuning in openGauss achieved 27% latency reduction for analytical queries and over

Table 7: SQL-Level Diagnosis(4 Datasets)

	Precision	Recall	Latency(ms)
openGauss	**0.913**	**0.922**	0.528
query-embedding	0.739	0.794	**0.035**
PostgreSQL	0.826	0.831	0.372
Actual running	1.0	1.0	3561

Table 8: System-Level Diagnosis(TPC-C)

	Precision	Recall	F1-score
openGauss	**0.885**	**0.871**	**0.869**
kNN	0.815	0.771	0.765
Decision Tree	0.836	0.824	0.826
DBSherlock	0.826	0.553	0.549

120% throughput increase for high-concurrency transactions. The index recommender in openGauss automatically selected beneficial indexes and can improve the throughput by 100%.

6.3 Model Validation

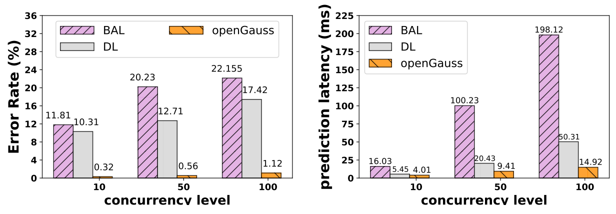
We compared performance prediction in openGauss with two state-of-the-art methods, BAL-based method (BAL) [15], DL-based method [46] (DL). BAL estimated the average buffer access latency and uses linear regression to predict query latency for concurrent queries. DL adopted a plan-structured neural network to predict performance for a single query. We compared prediction accuracy and prediction time on JOB. And the results were shown in Figure 16.

Prediction Accuracy. As shown in Figure 16, openGauss had the lowest error rate, around 29.9x lower than BAL, 22.5x lower than DL. The reasons were two-fold. First, the workload graph in openGauss encoded concurrency factors like resource contentions, which increased query latency of JOB by over 20%, in comparison with serial execution. Instead, BAL collected buffer access latency (BAL) and DL relied on single query features. Second, openGauss utilized graph embedding network to directly map the structural information into performance factors and could improve the generality when the workload changed. Instead, BAL utilized a linear regression method that required many statistical samples for a single workload.

Prediction Latency. As shown in Figure 16, openGauss took the least prediction latency than BAL and DL; and when the concurrency level increased, the prediction latency of openGauss was relatively stable. For openGauss, the prediction model concurrently predicted the execution time of all the vertices. It embedded the localized graphs for all the vertices in the workload graph, and so the total prediction time of a workload was close to predicting for the vertex with the largest localized graph. For BAL, it required the longest prediction time because it predicted the performance while executing the workload. For DL, it propagated intermediate data features across the query plan tree in a bottom-up fashion, which took relatively long time than openGauss.

Summary. openGauss outperformed state-of-the-arts methods in prediction accuracy and latency. For prediction accuracy, openGauss outperformed existing methods by 11–30 times. For latency, openGauss outperformed existing methods by 20%–1,227%.

We summarize our learnings from the practices of deploying openGauss. (1) *Feature selection is very important.* We need to select effective features based on different application scenarios. (2) *Model selection is vital.* We need to design effective learning models by considering multiple factors (e.g., optimization targets, input features).



(a) Mean Square Error (b) Prediction Latency

Figure 16: Model Validation (JOB).

(3) *The validation model is vital for deploying learning algorithms.* We need to build a validation model to decide whether a learned model can be deployed. (4) *The model management platform is vital to reduce redundant training time and adapt to workload changes.* We need to provide a unified model data management platform. (5) *Training data management is important.* We need to build a training data management platform to judiciously collect training data.

7 RELATED WORK

7.1 Autonomous Databases Systems

There were some autonomous database systems studied by academia [28, 37, 50]. Peloton [50] predicted the workload arrival rate with clustering and RNN, and deployed optimization actions like knob tuning, index/mv selection. SageDB [28] provided a vision to specialize the database implementation by learning the data distribution (CDF models) and designing database components based on the knowledge, e.g., learned index, learned query scheduling. However, they did not implement real systems. Moreover, there were some autonomous practices in commercial databases [3, 10–12, 30, 33, 63]. Oracle's autonomous data warehouse [3, 11] optimized the OLAP services with tuning choices like predicate indexing and materialized views. Alibaba's SDDP [33] detected database status based on history statistics and provided automatic optimization like hot/cold separation and buffer tuning. DB2 [30, 63] and Microsoft [10, 12] supported statistics-based self-configuration (e.g., knob tuning) and self-maintenance (e.g., resource allocation, data partitioning). Amazon Redshift supported automatic distribution-key selection with hybrid algorithms[48]. However, these autonomous techniques were mainly cost-based heuristics; while openGauss is an autonomous database system that validates, trains, and implements state-of-the-art learning-based components [20, 20, 36, 55, 69, 75] on an open-source database [1].

7.2 Learned Database Optimizer

Query Rewrite. Related works [17, 52] defined heuristic rules to automatically translate a set of linear algebra operators. They first transformed SQL queries into a standardized form, and then applied rewriting rules to these translated queries. However, they [17, 52] heuristically matched rules from top down, but cannot estimate benefits of different rewrite orders, and may give sub-optimal queries.

Cost/Cardinality Estimation. Existing studies proposed either supervised learning [25, 49] or unsupervised learning [21, 66, 67] to learn the joint dataset distributions and estimate cardinalities. Marcus [46] proposed a tree-structured model to represent plan structure with operator embeddings and estimated information(e.g., cost, cardinality) from databases. We [55] proposed a tree-structured LSTM model to represent plan structure which did not rely on estimated information provided by databases. Comparing to existing studies, our system has several superiorities. Our tree-LSTM learns the plan structure, and supports both cardinality and cost estimations without external estimation. Our predicate embedding techniques support non-trivial predicates (e.g., 'Like' predicates).

Plan Generation. Existing learning based plan enumerator [29, 44, 45, 61, 69] usually tried to apply reinforcement learning on the decision making(join order selection, operator selection). SkinnerDB [61] used a Monte-Carlo tree search based methods to change the join order on the fly. SkinnerDB required to measure the plan's performance and switched to another plan when query execution which relied on its customized in-memory database. DQ and ReJoin [29, 45] introduced the idea of DRL into the plan enumerator. It proved that the DRL can handle query enumerator process. DRL based methods can give the plans with low cost after enough training. The most similar work to ours was NEO [45]. Both NEO and our work tried to catch the plans' structure information during plan generation. NEO used a value neural network to search the plan with low latency. NEO first encoded each node in a join tree into a feature vector and then uses Tree Convolution Network to get the representation of join tree. Different from NEO, our work used the Tree-LSTM to catch the plans' structure. Our work supports database updates efficiently that estimates both latency and cost.

7.3 Learned Database Advisor

Knob Tuning. ML-based knob tuning [4, 56, 71] was proposed to automate knob tuning and achieved close or even better results than DBAs. However, existing techniques were unaware of the incoming workload characteristics and ML-based methods cannot efficiently adapt to new scenarios. We [36] proposed a query-aware tuning system. We first encoded query features into changes of state metrics, and then recommended knob values based on the predicated state metrics. Besides, our method supported query/workload/cluster-level tuning for various performance requirements.

Index Selection. Most of index selection methods [27] used what-if calls or hypothetical index to estimate index benefits and design different heuristic algorithms to choose final index sets. Some studies [27, 31, 53, 54] applied reinforcement learning based methods to index selection problem. They regarded the problem as a DRL problem and used RL algorithms to select indexes. Existing methods usually took long time to recommend indexes, especially for large workloads. To improve the efficiency of index advisor, we extracted representative queries from workload so that our methods can identify important indexes more quickly. Besides, we encoded system statistics into state representation of our deep reinforcement learning model so that it can update index configuration dynamically.

Materialized View Selection. Learning-based MV selection was proposed recently [23, 39, 70]. Jindal et al. [23] proposed an iteration based heuristic method to select MVs. Liang et al. [39] proposed an RL method to learn a policy of view creation and eviction from the historical execution statistics. Yuan et al. [70] proposed an RL method to select and create MVs for workloads. We [20] proposed a DL+RL method to estimate the benefit of MVs and select MVs for dynamic workloads. We used a deep learning model to estimate the benefit of MVs instead of collecting latency directly from runtimes so that we did not retrain model for different workloads. We split the MVs selection state into small states instead of a fixed-length (#MVs) state so that our RL model can fit dynamic workloads.

Model Validation. Duggan et al. [15] proposed Buffer Access Latency (BAL) to capture the joint effects of disk and memory contention on query performance. Wu et al. [64] proposed an analytic model to predict dynamic workloads. Marcus et al. proposed deep learning [46]. For a single query, it builded tree-structured network to predict query latency, which encoded interactions between child/parent operators. However, existing methods cannot predict the performance of concurrent queries. We [75] proposed a graph embedding based performance prediction model that embedded the query and data features with a workload graph and predicted execution time with the embeddings.

Learned Database Diagnosis. Benoit et al [7] proposed a framework for diagnosing and tuning OLTP DBMS performance, which relied on expert knowledge heavily. Yoon et al [68] presented a practical tool for assisting DBAs diagnosing performance. The system can highlight anomaly metrics by predicates but its diagnosis performance degraded sharply when workload changes. openGauss adopted distribution difference rather than predicate to describe anomaly metrics and was more robust. Borisov et al [8] focused on databases on network-attached server-storage infrastructure. Kalmegh et al [24] focused on analyzing inter-query contention for data analytics like Spark. Oracle[13] improved diagnosis accuracy by introducing a common currency called "Database Time" by collecting more trace data. Microsoft[16] built an automatic database troubleshooting system on Azure SQL Databases. Alibaba[42] proposed a framework to find root cause for intermittent slow queries. These studies were devised on the cloud environment and consider particular database metrics. Our method was more general and could be used in either cloud databases or on-premise databases.

8 CONCLUSION

We proposed an autonomous database system openGauss that integrated machine learning techniques into database systems including learned optimizers and learned advisors. The former designed learned query rewriter, learned cost estimator and learned plan generator to optimize database optimizer. The latter designed learned knob tuning, learned database diagnosis, learned view/index advisor to optimize database engine. We also proposed a model validation model to validate new learned model. Experimental results showed that the learned methods outperformed traditional techniques. There are still some future works. First, learning-based models rely on offline learning and it requires to address the cold-start problem (e.g., adapting to new databases). Second, learning-based models need to support database updates (e.g., schema update). Third, it is expensive to use a machine learning platform, e.g., TensorFlow, which cannot provide instant feedback. Thus it requires to design lightweight in-database machine learning algorithms. Fourth, based on the "one size does not fit all" paradigm, it requires to automatically select appropriate methods for different scenarios.

ACKNOWLEDGMENTS

This paper was supported by NSF of China (61925205, 61632016), Huawei, Beijing National Research Center for Information Science and Technology(BNRist), and TAL education. Guoliang Li and Xuanhe Zhou are corresponding authors.

REFERENCES

[1] [n.d.]. https://github.com/opengauss-mirror.

[2] [n.d.]. https://www.postgresql.org/.

[3] Rafi Ahmed, Randall G. Bello, Andrew Witkowski, and Praveen Kumar. 2020. Automated Generation of Materialized Views in Oracle. *VLDB* 13, 12 (2020), 3046–3058. http://www.vldb.org/pvldb/vol13/p3046-ahmed.pdf

[4] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *SIGMOD*. 1009–1024. https://doi.org/10.1145/3035918.3064029

[5] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. 2017. A Brief Survey of Deep Reinforcement Learning. *CoRR* abs/1708.05866 (2017). arXiv:1708.05866 http://arxiv.org/abs/1708.05866

[6] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, and et al. 2018. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *SIGMOD*. 221–230. https://doi.org/10.1145/3183713.3190662

[7] Darcy G. Benoit. 2005. Automatic Diagnosis of Performance Problems in Database Management Systems. In *ICAC*. IEEE Computer Society, 326–327. https://doi.org/10.1109/ICAC.2005.12

[8] Nedyalko Borisov, Sandeep Uttamchandani, Ramani Routray, and Aameek Singh. 2009. Why Did My Query Slow Down. In *CIDR*. www.cidrdb.org. http://www-db.cs.wisc.edu/cidr/cidr2009/Paper_72.pdf

[9] Cameron Browne, Edward Jack Powley, and Daniel Whitehouse and. 2012. A Survey of Monte Carlo Tree Search Methods. *IEEE Trans. Comput. Intell. AI Games* (2012). https://doi.org/10.1109/TCIAIG.2012.2186810

[10] Surajit Chaudhuri and Vivek R. Narasayya. 1998. AutoAdmin 'What-if' Index Analysis Utility. In *SIGMOD*. https://doi.org/10.1145/276304.276337

[11] Dinesh Das, Jiaqi Yan, Mohamed Zaït, and et al. 2015. Query Optimization in Oracle 12c Database In-Memory. *VLDB* (2015). https://doi.org/10.14778/2824032.2824074

[12] Sudipto Das, Feng Li, Vivek R. Narasayya, and et al. 2016. Automated Demand-driven Resource Scaling in Relational Database-as-a-Service. In *SIGMOD*. https://doi.org/10.1145/2882903.2903733

[13] Karl Dias, Mark Ramacher, Uri Shaft, Venkateshwaran Venkataramani, and Graham Wood. 2005. Automatic Performance Diagnosis and Tuning in Oracle. In *CIDR*. www.cidrdb.org, 84–94. http://cidrdb.org/cidr2005/papers/P07.pdf

[14] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. 2017. DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning. In *SIGSAC*. ACM. https://doi.org/10.1145/3133956.3134015

[15] Jennie Duggan, Ugur Çetintemel, Olga Papaemmanouil, and Eli Upfal. 2011. Performance prediction for concurrent database workloads. In *SIGMOD 2011*. 337–348. https://doi.org/10.1145/1989323.1989359

[16] Dejan Dundjerski and Milo Tomasevic. 2020. Automatic database troubleshooting of Azure SQL Databases. *IEEE Transactions on Cloud Computing* (2020).

[17] Béatrice Finance and Georges Gardarin. 1991. A Rule-Based Query Rewriter in an Extensible DBMS. In *ICDE*. IEEE Computer Society, 248–256. https://doi.org/10.1109/ICDE.1991.131472

[18] Ming Gao, Leihui Chen, Xiangnan He, and Aoying Zhou. 2018. BiNE: Bipartite Network Embedding. In *SIGIR*. 715–724. https://doi.org/10.1145/3209978.3209987

[19] Zhabiz Gharibshah, Xingquan Zhu, Arthur Hainline, and Michael Conway. 2020. Deep Learning for User Interest and Response Prediction in Online Display Advertising. *Data Science and Engineering* 5, 1 (2020), 12–26. https://doi.org/10.1007/s41019-019-00115-y

[20] Yue Han, Guoliang Li, Haitao Yuan, and Ji Sun. 2021. An Autonomous Materialized View Management System with Deep Reinforcement Learning. In *ICDE*.

[21] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2020. DeepDB: Learn from Data, not from Queries! *VLDB* 13, 7 (2020), 992–1005. http://www.vldb.org/pvldb/vol13/p992-hilprecht.pdf

[22] Yannis E. Ioannidis and Younkyung Cha Kang. 1991. Left-Deep vs. Bushy Trees: An Analysis of Strategy Spaces and its Implications for Query Optimization. In *SIGMOD*. 168–177. https://doi.org/10.1145/115790.115813

[23] Alekh Jindal, Konstantinos Karanasos, Sriram Rao, and Hiren Patel. 2018. Selecting Subexpressions to Materialize at Datacenter Scale. *VLDB* 11, 7 (2018), 800–812. https://doi.org/10.14778/3192965.3192971

[24] Prajakta Kalmegh, Shivnath Babu, and Sudeepa Roy. 2019. iQCAR: inter-Query Contention Analyzer for Data Analytics Frameworks. In *SIGMOD*. ACM, 918–935. https://doi.org/10.1145/3299869.3319904

[25] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. 2019. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *CIDR*. www.cidrdb.org. http://cidrdb.org/cidr2019/papers/p101-kipf-cidr19.pdf

[26] Levente Kocsis and Csaba Szepesvári. 2006. Bandit Based Monte-Carlo Planning. In *ECML*. https://doi.org/10.1007/11871842_29

[27] Jan Kossmann, Stefan Halfpap, Marcel Jankrift, and Rainer Schlosser. 2020. Magic mirror in my hand, which is the best in the land? An Experimental Evaluation of Index Selection Algorithms. *VLDB* 13, 11 (2020), 2382–2395. http://www.vldb.org/pvldb/vol13/p2382-kossmann.pdf

[28] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H. Chi, and et al. 2019. SageDB: A Learned Database System. In *CIDR*. http://cidrdb.org/cidr2019/papers/p117-kraska-cidr19.pdf

[29] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph M. Hellerstein, and Ion Stoica. 2018. Learning to Optimize Join Queries With Deep Reinforcement Learning. *CoRR* abs/1808.03196 (2018). arXiv:1808.03196 http://arxiv.org/abs/1808.03196

[30] Eva Kwan, Sam Lightstone, Adam Storm, and Leanne Wu. 2002. Automatic configuration for IBM DB2 universal database. *Proc. of IBM Perf Technical Report* (2002).

[31] Hai Lan, Zhifeng Bao, and Yuwei Peng. 2020. An Index Advisor Using Deep Reinforcement Learning. In *CIKM*. 2105–2108. https://doi.org/10.1145/3340531.3412106

[32] Dan Li, Dacheng Chen, and et al. 2019. MAD-GAN: Multivariate Anomaly Detection for Time Series Data with Generative Adversarial Networks. In *ICANN*. 703–716. https://doi.org/10.1007/978-3-030-30490-4_56

[33] Feifei Li. 2019. Cloud native database systems at Alibaba: Opportunities and Challenges. *VLDB* (2019). https://doi.org/10.14778/3352063.3352141

[34] Guoliang Li, Xuanhe Zhou, and Lei Cao. 2021. AI Meets Database: AI4DB and DB4AI. In *SIGMOD*. 2859–2866. https://doi.org/10.1145/3448016.3457542

[35] Guoliang Li, Xuanhe Zhou, and Lei Cao. 2021. Machine Learning for Databases. *VLDB* (2021).

[36] Guoliang Li, Xuanhe Zhou, Bo Gao, and Shifu Li. 2019. QTune: A Query-Aware Database Tuning System with Deep Reinforcement Learning. In *VLDB*.

[37] Guoliang Li, Xuanhe Zhou, and Sihao Li. 2019. XuanYuan: An AI-Native Database. *IEEE Data Eng. Bull.* 42, 2 (2019), 70–81. http://sites.computer.org/debull/A19june/p70.pdf

[38] Mingda Li, Hongzhi Wang, and Jianzhong Li. 2020. Mining Conditional Functional Dependency Rules on Big Data. *Big Data Mining and Analytics* 03, 01, Article 68 (2020), 16 pages.

[39] Xi Liang, Aaron J. Elmore, and Sanjay Krishnan. 2019. Opportunistic View Materialization with Deep Reinforcement Learning. *CoRR* abs/1903.01363 (2019). arXiv:1903.01363 http://arxiv.org/abs/1903.01363

[40] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, and et al. 2016. Continuous control with deep reinforcement learning. In *ICLR*. http://arxiv.org/abs/1509.02971

[41] Shuyu Lin, Ronald Clark, Robert Birke, Sandro Schönborn, Niki Trigoni, and Stephen J. Roberts. 2020. Anomaly Detection for Time Series Using VAE-LSTM Hybrid Model. In *ICASSP*. IEEE, 4322–4326. https://doi.org/10.1109/ICASSP40776.2020.9053558

[42] Minghua Ma, Zheng Yin, Shenglin Zhang, and et al. 2020. Diagnosing Root Causes of Intermittent Slow Queries in Large-Scale Cloud Databases. *VLDB* 13, 8 (2020), 1176–1189. http://www.vldb.org/pvldb/vol13/p1176-ma.pdf

[43] Ryan Marcus and Olga Papaemmanouil. 2018. Deep reinforcement learning for join order enumeration. In *SIGMOD Workshop*. 3.

[44] Ryan Marcus and Olga Papaemmanouil. 2018. Deep Reinforcement Learning for Join Order Enumeration. In *SIGMOD*. ACM, 3:1–3:4. https://doi.org/10.1145/3211954.3211957

[45] Ryan C. Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *VLDB* 12, 11 (2019), 1705–1718. https://doi.org/10.14778/3342263.3342644

[46] Ryan C. Marcus and Olga Papaemmanouil. 2019. Plan-Structured Deep Neural Network Models for Query Performance Prediction. *VLDB* 12, 11 (2019), 1733–1746. https://doi.org/10.14778/3342263.3342646

[47] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. 2013. Playing Atari with Deep Reinforcement Learning. *CoRR* abs/1312.5602 (2013). arXiv:1312.5602 http://arxiv.org/abs/1312.5602

[48] Panos Parchas, Yonatan Naamad, and Peter Van Bouwel and. 2020. Fast and Effective Distribution-Key Recommendation for Amazon Redshift. *VLDB* (2020). http://www.vldb.org/pvldb/vol13/p2411-parchas.pdf

[49] Yongjoo Park, Shucheng Zhong, and Barzan Mozafari. 2020. QuickSel: Quick Selectivity Learning with Mixture Models. In *SIGMOD*. ACM, 1017–1033. https://doi.org/10.1145/3318464.3389727

[50] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, and et al. 2017. Self-Driving Database Management Systems. In *CIDR*. http://cidrdb.org/cidr2017/papers/p42-pavlo-cidr17.pdf

[51] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. DeepWalk: online learning of social representations. In *KDD*. 701–710. https://doi.org/10.1145/2623330.2623732

[52] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. 1992. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *SIGMOD*. 39–48. https://doi.org/10.1145/130283.130294

[53] Zahra Sadri, Le Gruenwald, and Eleazar Leal. 2020. Online Index Selection Using Deep Reinforcement Learning for a Cluster Database. In *ICDE*. IEEE, 158–161. https://doi.org/10.1109/ICDEW49219.2020.00035

[54] Ankur Sharma, Felix Martin Schuhknecht, and Jens Dittrich. 2018. The Case for Automatic Database Administration using Deep Reinforcement Learning. *CoRR* abs/1801.05643 (2018). arXiv:1801.05643 http://arxiv.org/abs/1801.05643
[55] Ji Sun and Guoliang Li. 2019. An End-to-End Learning-based Cost Estimator. *VLDB* 13, 3 (2019), 307–319. https://doi.org/10.14778/3368289.3368296
[56] Jian Tan, Tieying Zhang, and et al. 2019. iBTune: Individualized Buffer Tuning for Large-scale Cloud Databases. *PVLDB* 12, 10 (2019), 1221–1234. http://www.vldb.org/pvldb/vol12/p1221-tan.pdf
[57] Jian Tang, Meng Qu, and Qiaozhu Mei. [n.d.]. PTE: Predictive Text Embedding through Large-scale Heterogeneous Text Networks. In *SIGKDD*.
[58] Jian Tang, Meng Qu, Mingzhe Wang, and et al. 2015. LINE: Large-scale Information Network Embedding. In *WWW*. 1067–1077. https://doi.org/10.1145/2736277.2741093
[59] Shan Tian, Songsong Mo, Liwei Wang, and Zhiyong Peng. 2020. Deep Reinforcement Learning-Based Approach to Tackle Topic-Aware Influence Maximization. *Data Science and Engineering* 5, 1 (2020), 1–11. https://doi.org/10.1007/s41019-020-00117-1
[60] Immanuel Trummer, Junxiong Wang, Deepak Maram, Samuel Moseley, Saehan Jo, and Joseph Antonakakis. 2019. SkinnerDB: Regret-Bounded Query Evaluation via Reinforcement Learning. In *SIGMOD*. ACM, 1153–1170. https://doi.org/10.1145/3299869.3300088
[61] Immanuel Trummer, Junxiong Wang, Deepak Maram, Samuel Moseley, Saehan Jo, and Joseph Antonakakis. 2019. SkinnerDB: Regret-Bounded Query Evaluation via Reinforcement Learning. In *SIGMOD*.
[62] Florian Waas and Arjan Pellenkoft. 2000. Join order selection (good enough is easy). In *British National Conference on Databases*.
[63] David Wiese, Gennadi Rabinovitch, Michael Reichert, and Stephan Arenswald. 2008. Autonomic tuning expert: a framework for best-practice oriented autonomic database tuning. In *CASCON*.
[64] Wentao Wu, Yun Chi, Hakan Hacigümüs, and Jeffrey F. Naughton. 2013. Towards Predicting Query Execution Time for Concurrent and Dynamic Database Workloads. *PVLDB* 6, 10 (2013), 925–936. https://doi.org/10.14778/2536206.2536219
[65] Wentao Wu, Yun Chi, Shenghuo Zhu, Jun'ichi Tatemura, Hakan Hacigümüs, and Jeffrey F. Naughton. 2013. Predicting query execution time: Are optimizer cost models really unusable?. In *ICDE*. 1081–1092. https://doi.org/10.1109/ICDE.2013.6544899
[66] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. 2020. NeuroCard: One Cardinality Estimator for All Tables. *CoRR* abs/2006.08109 (2020). arXiv:2006.08109 https://arxiv.org/abs/2006.08109
[67] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Peter Chen, Pieter Abbeel, Joseph M. Hellerstein, Sanjay Krishnan, and Ion Stoica. 2019. Deep Unsupervised Cardinality Estimation. *VLDB* 13, 3 (2019), 279–292. https://doi.org/10.14778/3368289.3368294
[68] Dong Young Yoon, Ning Niu, and Barzan Mozafari. 2016. DBSherlock: A Performance Diagnostic Tool for Transactional Databases. In *SIGMOD*. ACM, 1599–1614. https://doi.org/10.1145/2882903.2915218
[69] Xiang Yu, Guoliang Li, Chengliang Chai, and Nan Tang. 2020. Reinforcement Learning with Tree-LSTM for Join Order Selection. In *ICDE*. IEEE, 1297–1308. https://doi.org/10.1109/ICDE48307.2020.00116
[70] Haitao Yuan, Guoliang Li, Ling Feng, Ji Sun, and Yue Han. 2020. Automatic View Generation with Deep Learning and Reinforcement Learning. In *ICDE*. IEEE, 1501–1512. https://doi.org/10.1109/ICDE48307.2020.00133
[71] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, and et al. 2019. An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning. In *SIGMOD*. 415–432. https://doi.org/10.1145/3299869.3300085
[72] Ning Zhang, Peter J. Haas, Vanja Josifovski, Guy M. Lohman, and Chun Zhang. 2005. Statistical Learning Techniques for Costing XML Queries. In *VLDB*. 289–300. http://www.vldb.org/archives/website/2005/program/paper/wed/p289-zhang.pdf
[73] Xuanhe Zhou, Chengliang Chai, Guoliang Li, and Ji Sun. 2020. Database Meets Artificial Intelligence: A Survey. *TKDE* (2020).
[74] Xuanhe Zhou, Lianyuan Jin, Sun Ji, Xinyang Zhao, Xiang Yu, Shifu Li, Tianqing Wang, Kun Li, and Luyang Liu. 2021. DBMind: A Self-Driving Platform in openGauss. *VLDB* (2021).
[75] Xuanhe Zhou, Ji Sun, Guoliang Li, and Jianhua Feng. 2020. Query Performance Prediction for Concurrent Queries using Graph Embedding. *VLDB* (2020). http://www.vldb.org/pvldb/vol13/p1416-zhou.pdf
[76] Yuqing Zhu, Jianxun Liu, and et al. 2017. BestConfig: tapping the performance potential of systems via automatic configuration tuning. In *SoCC*. 338–350. https://doi.org/10.1145/3127479.3128605