AutoIndex: An Incremental Index Management System for Dynamic Workloads

Xuanhe Zhou*, Luyang Liu†, Wenbo Li*, Lianyuan Jin*, Shifu Li†, Tianqing Wang †, Jianhua Feng*
*Department of Computer Science, Tsinghua University, China. zhouxuan19@mails.tsinghua.edu.cn
†Huawei Company, China. liuluyang2@huawei.com

***Abstract*—Indexes are vital to enhance the lookup on single or multiple columns, and building proper indexes can significantly improve the database performance. Existing works focus on adding new indexes that can benefit the read queries, but they have several limitations. First, real-world workloads may have numerous queries and it is tricky to analyze their index requirements and find the most beneficial indexes within resource limit. Second, they fail to consider the update of existing indexes, which may be redundant or even have negative effects to current workload. Third, they cannot estimate the index maintenance costs, which are affected by multiple index utilization factors and can significantly affect the index benefits, especially for high-write-ratio workloads. To address those challenges, we propose an incremental index management system AUTOINDEX for dynamic workloads. First, to support incremental index management, we map the incoming queries into query templates and efficiently generate promising candidate indexes from matched templates. And then we propose to utilize *Monte Carlo Tree Search* to incrementally add indexes from the candidate indexes or remove indexes from existing indexes, so as to ensure high workload performance. Besides, we propose a deep index estimation model, which integrates the practical experience to extract critical cost features and applies deep regression to estimate index benefits from historical index management data. We have implemented the modules like *candidate index generation* and *index estimator* in an open-sourced database system *openGauss*. Experimental results showed that our method outperformed existing approaches on both testing and real-world workloads.**

***Index Terms*—index management, database, machine learning**

I. Introduction

Indexes in data management systems can speed up data retrieval at the cost of index maintenance and storage overhead, which can significantly affect the workload performance. And *index management* aims to judiciously create or remove indexes so as to ensure (1) the index size is within storage constraint and (2) the workload performance is optimized. Next we use two cases to demonstrate the importance and challenges in index management.

Example 1: Index Management in Real Scenarios. In the real transactional scenarios, there are numerous workload queries, which have various index requirements (e.g., accessing different columns). And it is laborious for DBAs to analyze the index requirements of those queries and find the optimal index combinations. For example, as shown in Figure 1, for the withdraw business (around 2.2M queries) in banking scenario, DBAs have carefully crafted 263 indexes, but there are still many redundant indexes. By judiciously removing the redundant indexes and adding the performance-critical indexes (e.g., high lookup frequency), we can save over

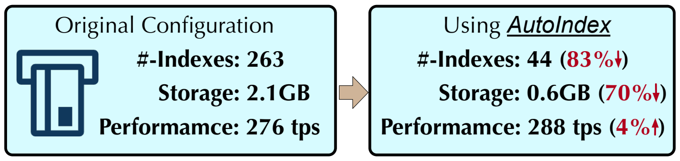
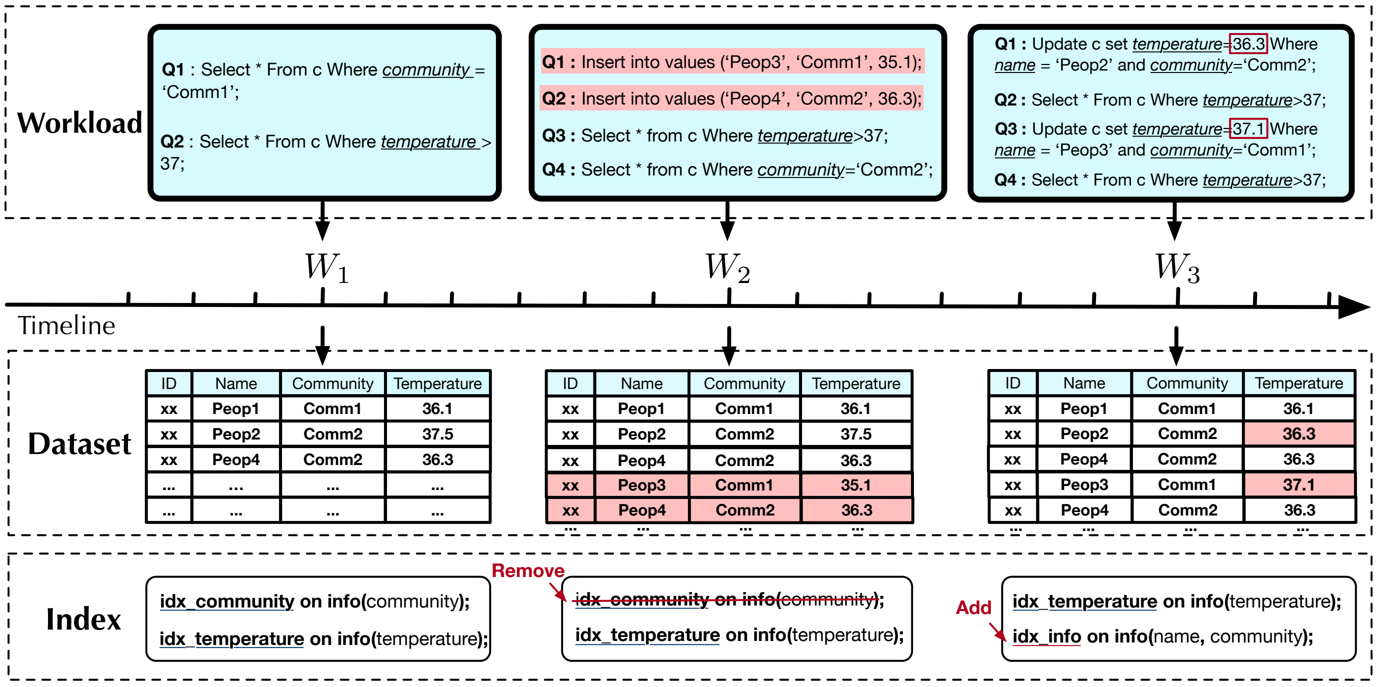


Fig. 1. An Example Index Management for the Withdraw business in Banking Scenario. AUTOINDEX removes 83% indexes, saves 70% storage space, and still achieves 4% throughput improvement.

70% storage space and gain even better performance (e.g., over 4% throughput increase). Hence, it is vital to conduct effective index management so as to optimize the workload performance within storage constraints.

Example 2: Index Management for Dynamic Workloads. Moreover, the workloads may continuously change in real scenarios. Figure 2 illustrates an example of index management for dynamic workloads. For an epidemic database, the table records the information of potentially infected people. There are three historical workloads, which have their own index requirements: (1) At the beginning of the epidemic, the table contained rare data and *the workloads were some random queries accessing the table* (W_1). And we can directly recommend indexes on the frequently-used columns, i.e., *idx_temperature* and *idx_community*; (2) Afterwards, the epidemic quickly spread to other communities, and there were insert queries that recorded new potentially-infected people (W_2). Here we found the maintenance cost of index *idx_community* was higher than the reduced query costs, and thus we removed the index *idx_community*; (3) Finally, after the epidemic got controlled, there were rare inserts but many update queries that refreshed the people's temperatures (W_3), and so we can build multi-column index on the columns *name* and *community* so as to speed up the *temperature* updates. Meanwhile, although tuples in *temperature* were frequently updated, we reserved the index *idx_temperature* because the reduced query cost (for Q_2, Q_4) by *idx_temperature* is higher than the maintenance cost of *idx_temperature*.

Key Observations. From above two examples we have three observations. First, workloads may have different index preference, and so we need to update indexes based on workload characters so as to ensure high performance. Second, the index benefits depend on both the cost reduction for read queries and index update overhead for write queries. Third, data columns have correlations (e.g., columns in the same predicate), and



Workload
Q1 : Select * From c Where community = 'Comm1';
Q2 : Select * From c Where temperature > 37;
Q1 : Insert into values ('Peop3', 'Comm1', 35.1);
Q2 : Insert into values ('Peop4', 'Comm2', 36.3);
Q3 : Select * from c Where temperature>37;
Q4 : Select * from c Where community='Comm2';
Q1 : Update c set temperature=36.3 Where name = 'Peop2' and community='Comm2';
Q2 : Select * From c Where temperature>37;
Q3 : Update c set temperature=37.1 Where name = 'Peop3' and community='Comm1';
Q4 : Select * From c Where temperature>37;
W_1
W_2
W_3
Timeline
Dataset
ID Name Community Temperature
xx Peop1 Comm1 36.1
xx Peop2 Comm2 37.5
xx Peop4 Comm2 36.3
xx Peop3 Comm1 35.1
xx Peop4 Comm2 36.3
xx Peop2 Comm2 36.3
xx Peop3 Comm1 37.1
Index
idx_community on info(community);
idx_temperature on info(temperature);
Remove
~~idx_community on info(community)~~;
Add
idx_info on info(name, community);

Fig. 2. An Example of Incremental Index Management.

building multi-column index may gain higher benefit than multiple single-column indexes.

Limitations of Existing Methods. Most existing index management methods [5], [31], [2], [3], [26] utilize cost estimator in the database to estimate the index benefit and greedily select indexes with maximum estimated cost reduction. And they have three limitations. First, most methods work in query level, i.e., adding indexes as long as they have positive effects to at least one query, and they ignore the correlations between indexes (e.g., index on column A can be replaced by multi-column index on (A, B)) and queries (e.g., read and write queries on the same columns), which are vital to index management. Second, the candidate index space can be very large, but they adopt greedy algorithms (e.g., top-k, hill-climbing) and may cause sub-optimal solutions. Third, they estimate index benefits based on the database cost estimator, which ignores the index maintenance cost and may make significant errors.

Moreover, there are some machine learning based methods in index management [8], [21], [25]. They either utilize deep reinforcement learning to select indexes [21], [25], or design deep learning model to estimate index benefits [8]. However, we argue that those methods have their own limitations and cannot be directly used in our problem.

(1) *DRL-based index selection*. First, DRL requires extremely long training time (e.g., days to weeks) for a specific workload, and so cannot efficiently work for dynamic workloads. Second, it is tricky for DRL to support index removal, since it only reserves current index selection state and cannot go back to previous steps by the learned policy.

(2) *DL-based estimation method*. They recommend for single queries (e.g., comparing query plans before/after using the indexes), and they fail to estimate in workload level, which may cause performance regression because some queries may only appear once and never reoccur, or the recommended index has high update frequency. Moreover, they mainly work for analytical queries, and cannot estimate the index update costs for write queries. In many scenarios, there are a large number of redundant or even negative benefit indexes that cannot be directly reflected by query plans, and existing plan-based ML method fails to identify those indexes.

Challenges. As observed above, index management for dynamic workloads is challenging and existing methods suffer from three main challenges. First, *how to efficiently capture the index requirements* (**C1**). The real-world workloads may contain numerous queries and they are continuously changing. Second, *how to effectively update indexes so as to ensure high performance* (**C2**). There can be a large number of candidate indexes and we need to conduct incremental index update based on existing indexes. Third, *how to estimate the index benefits for both read and write queries* (**C3**). It is hard to estimate the overall index benefits, which involve multiple factors around data lookup and index maintenance.

Our Proposed Methods. To address above challenges, we propose an incremental index management system AUTOINDEX for dynamic workloads in the real scenarios, which inputs the coming queries and historical index statistics, and updates the existing index set for performance improvement (e.g., higher throughput) within resource limit. First, AUTOINDEX monitors the workload performance and issues index update requests if there are performance regression caused by index problems. Second, AUTOINDEX proposes an incremental index management method, which matches coming workloads with templates, extracts promising candidate indexes from the templates (for **C1**), and utilizes *Monte Carlo Tree Search* to select high-benefit indexes based on both the existing and candidate indexes (for **C2**). In MCTS, to efficiently estimate the benefits of different indexes, AUTOINDEX proposes an index benefit estimation model which trains a deep regression model to estimate the overall index benefit (according to both read and write queries) (for **C3**).

Contributions. We make the following contributions:
(1) We propose a workload-level index management system, which incrementally updates indexes based on both historical and incoming queries under storage constraints.
(2) We propose a *MCTS*-based index update method, which maintains a policy tree to represent existing indexes and updates indexes by judiciously exploring the policy tree.
(3) We propose an index benefit estimation method, which computes critical cost features by practical experience and utilizes deep regression to estimate the benefits based on the computed features.
(4) We have implemented AUTOINDEX in an open-sourced database openGauss [1] (e.g., in-kernel estimator). Experimental results showed that our method outperformed existing approaches on both testing and real-world workloads.

II. PRELIMINARIES

A. Indexes and Benefits

Building indexes can significantly reduce the lookup time, because it can directly locate the required data within a relatively small region. Hence, we first define the benefits of single or multiple indexes.

Benefit of single indexes. For a workload W and any index I, we define the benefit of I as the difference of execution costs with/without the index, i.e., $B(I) = \sum(cost(W) - cost(W, I))$, where $cost(W)$ denotes executing W without index and $cost(W, I)$ denotes the execution cost of W using index I.

Example 1: In Figure 2, for workload W_1, queries Q_1, Q_2 access the columns $community$ and $temperature$, whose selectivity is no less than than 1/3. Hence, we can build single indexes on the two columns to enhance Q_1, Q_2.

Remark. We interchangeably use execution cost and performance in the following sections if there is no ambiguity. And we will formally define and explain how to compute the execution cost with different indexes in Section V.

Benefit of multiple indexes. Similarly, given a set of indexes $\mathcal{I}$, the benefit of $\mathcal{I}$ can be written as $B(\mathcal{I}) = \sum(cost(W) - cost(W, \mathcal{I}))$. However, different from single indexes, multiple indexes have correlations with each other when speeding up any workload.

Example 2: In Figure 2, for workload W_2, similar to the index $idx\_temperature$, multi-column index on ($temperature$, $community$) can also speed up Q_3. However, it brings extra maintenance cost to update $community$, and so we only reserve $idx\_temperature$ for W_2. Besides, for workload W_3, we cannot enhance Q_1 only with the index on $community$. Instead, we need to separately build indexes on both $name$ and $community$ (or a multi-column index on ($name$, $community$)).

In summarization, to improve the overall index benefits, it is vital to select indexes that can benefit lookup queries, cause little maintenance overhead, and have minor function overlaps with other indexes (more details in Section IV).

B. Index Management

Index Management for Static Workloads. Given a workload W and a storage budge B, if a query needs to filter out many tuples (high selectivity) which takes much time, it is vital to build indexes (e.g., B+Tree) on columns involved in the query, such that reducing the lookup time. Hence, when any query contains the columns in the index, it is possible to utilize the index to optimize the query.

Definition 1 (Index Management): Given a workload W and storage budget B, there exist a large number of indexes which may optimize at least one workload query in W, i.e., a set $\mathcal{I}$ of candidate indexes. Index management aims to judiciously select a subset of candidate indexes $\mathcal{I}^c \subseteq \mathcal{I}$ such that (i) the total size of indexes in $\mathcal{I}^c$ is within B and (ii) the workload performance is optimized.

Example 3: Figure 2 shows an example of index management. For W_1, it has two representative queries whose predicates separately access columns $community$ and $temperature$. We can separately create two indexes on the two columns so as to speed up the lookup time of Q_1, Q_2.

Index Management for Dynamic Workloads. However, in reality, the workload may continuously change (e.g., read/write ratio, access patterns). Hence, we need to conduct *incremental index management* (IIM) for dynamic workloads. That is, given existing indexes built based on historical workloads, for a newly coming workload, the *IIM problem* aims to incrementally update the indexes (e.g., adding/removing indexes) so as to ensure high performance on the new workload.

Definition 2 (IIM problem): Given existing indexes $\mathcal{I}$ built based on historical workloads and the storage budget B, for a newly coming workload W, incremental index management is to update the index set $\mathcal{I}$ into $\mathcal{I}'$ by removing redundant indexes or adding new indexes, such that (1) the size of $\mathcal{I}'$ does not exceed B and (2) $\mathcal{I}'$ can optimize the performance of the future workload, i.e., $\sum(cost(W, \mathcal{I}) - cost(W, \mathcal{I}')) > 0$, where $cost(W, \mathcal{I})$ denotes the execution cost of W using the index set $\mathcal{I}$. And the index set is continuously updated based on the incoming workloads.

Example 4: Figure 2 shows an example of incremental index management. Before executing the workload W_3, the index set is $\{idx\_temperature\}$. Within the workload, Q_2, Q_4 can utilize the existing index $idx\_temperature$, but Q_1, Q_3 require an additional index on $(name, community)$. Hence, we update the index set as $\{idx\_temperature, idx\_info\}$, which does not exceed the resource limit and can further optimize the performance of W_3.

[1] github.com/zhouxh19/autoindex

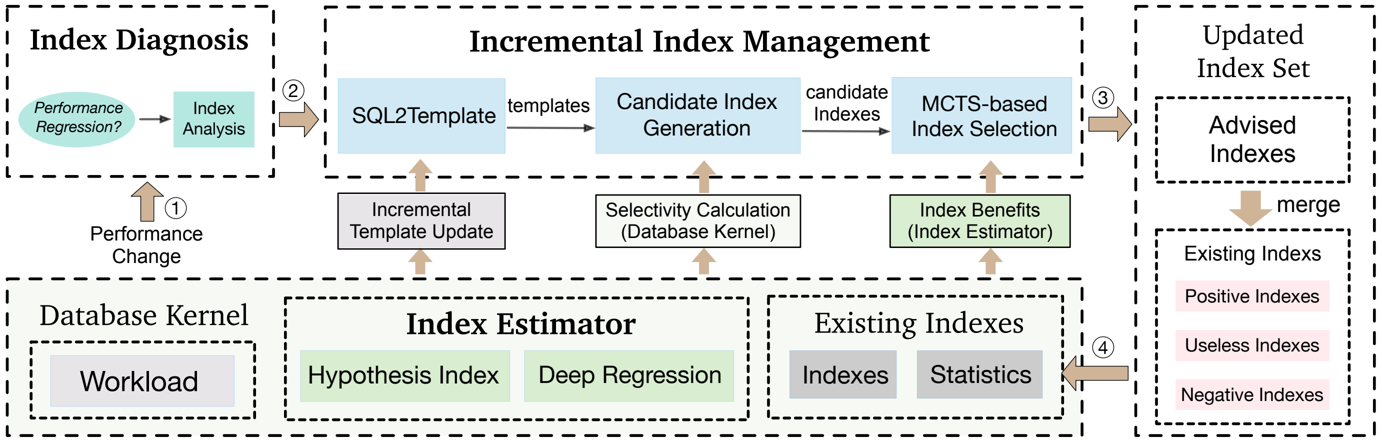


Fig. 3. Framework of AUTOINDEX.

III. SYSTEM OVERVIEW

In this section, we first explain the motivation of utilizing *Monte Carlo Tree Search* in our problem, and then we present the overall framework of AUTOINDEX, with which we can solve the challenges in basic MCTS and efficiently update the existing indexes based on the workload characters.

Motivation of using MCTS. Recap that *index update* is one of the core parts in incremental index management. Specifically, index update is important because it needs to (1) remove existing low-efficiency indexes and (2) add promising indexes. Moreover, there are correlations between indexes and workloads, and greedily selecting high-benefit indexes may fall into sub-optimum. For example, in TPC-DS, separately creating indexes on $i\_manufact\_id$ and $date\_dim$ may have minor benefits (directly dropped in hill-climbing algorithm) . Instead, if we create the two indexes together, the execution time of query $Q32$ is greatly reduced because the subquery in $Q32$ is enhanced only when the two indexes are available.

To address this issue, we utilize the guided searching policy in MCTS to incrementally remove/add indexes based on the existing policy tree (intermeidate states) and efficiently find promising solutions by evaluating the long-term index benefits (Section IV). However, basic MCTS still encounters several challenges in our problem. First, it takes long exploration time if the index space is too large (candidate index generation). Second, dynamic workloads may cause the estimated values of the tree nodes out-of-the-date (incremental update). Third, it requires efficient cost estimator to estimate the benefits of selected index combinations (index benefit estimation). Thus, next we introduce the framework of AUTOINDEX that helps to solve those challenges.

Workflow. For any new workload being executed in the database, we first diagnose the index problems when performance regression occurs. If any index problem is identified, we generate candidate indexes from the workload queries (logged in the server that runs the index management process) and utilize *Monte Carlo Tree Search* (MCTS) to explore for the optimal combination of both candidate and existing indexes under the resource constraints. In MCTS, we propose an index benefit estimation method that estimates index benefits based on both the read and write queries. Finally, we update the existing index set with the recommended indexes, where we also figure out redundant or negative indexes based on the index benefit estimation results.

Index Diagnosis. `Index Diagnosis` module monitors the system metrics during workload execution. Any time this module detects abnormal status (e.g., performance regression), it will call the *index analysis* component to decide whether there are any needs to update the existing indexes (e.g., removing redundant indexes, creating beneficial indexes). For example, we compute the ratio of three classes of indexes, i.e., (i) beneficial indexes that have not been created, (ii) rarely-used indexes, and (iii) indexes that have negative effects to the workload performance. If the ratio of those indexes is higher than a threshold, we will issue an index tuning request to the `Index Recommendation` module.

Index Management. For any index tuning request, the `Index Recommendation` module inputs the workload and index statistics and outputs the recommended indexes. This part contains three components: *SQL2Template*, *Candidate Index Generation*, *Index Selection*. At first, since the workload can be large (e.g., with millions of queries) and it is costly to recommend indexes in workload level, we use *SQL2Template* to map the workload queries into a fixed number of query templates, which denote the set of most-frequently-used access patterns. For each query template, we use *Candidate Index Generation* to parse predicates from the query clauses (e.g., *FROM*, *WHERE*, *GROUP*, and *ORDER*), and generate candidate indexes based on the columns involved in each predicate. For example, for predicate "a=$ and b>$", we will generate a candidate index on (a, b). Finally, with the candidate indexes, we utilize *Index Selection* to recommend optimal indexes based on current workload. In particular, we maintain a policy tree built on existing indexes, based on which we explore new optimal indexes by taking actions like adding candidate indexes or removing existing indexes.

Moreover, we can support index type selection for the data partitioning scenarios, which is vital to improve the workload

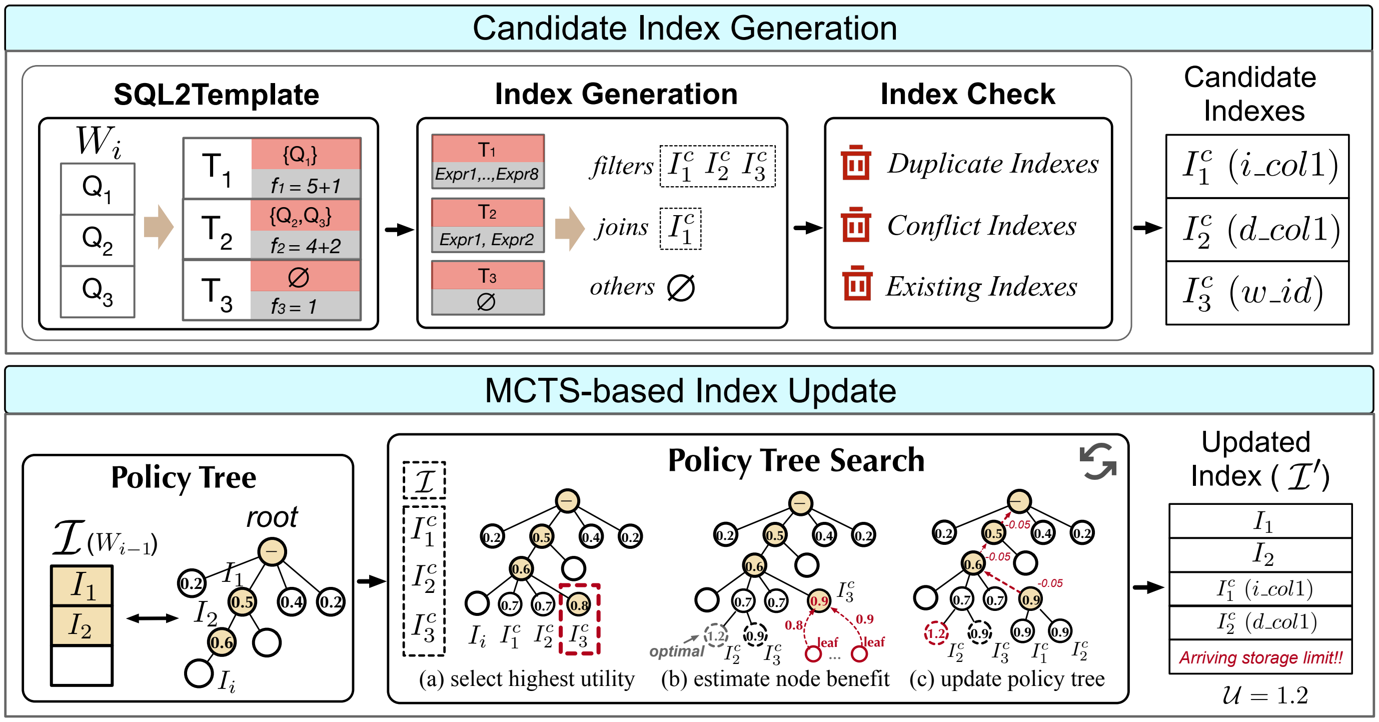


Fig. 4. Incremental Index Management.

performance (e.g., "global" index has high lookup speed, but takes much storage space; and "local" index is less efficient but takes much less space).

Index Estimator. Generally current database cannot estimate the index maintenance costs, which involve complex factors (e.g., the page splitting strategy) and are vital to estimate the overall performance of workloads with different indexes. Hence, in `Index Estimator`, we design a deep regression model, which inputs the workload features and indexes (together with the index statistics), and outputs the execution cost of the workload. That model is trained with numerous historical index management data and can efficiently diversify the costs of queries under different indexes.

IV. Incremental Index Management

In this section, we introduce how to conduct incremental index management. Existing index management methods only support creating indexes for single queries or static workloads [5], [31], [8], and there are several remaining challenges. First, *how to efficiently extract candidate indexes from real-world workloads*. In real scenarios, there can be millions of queries and it is costly to analyze the required indexes for each query (**C1.1**). Second, *how to represent such a large index space* and efficiently search for promising index combinations. Existing methods heuristically select high-benefit indexes, where they ignore the combined benefits of multiple indexes and often find suboptimal solutions (**C1.2**). Third, *how to effectively adapt to dynamic workloads*. For a coming workload, the queries and index requirements may significantly change, which requires to update the estimated benefits of existing indexes, synchronize the index requirements of current workload, and design efficient algorithms so as to merge new beneficial indexes into existing indexes (**C1.3**). Before illustrating the details, we first define the policy tree.

Policy Tree. In order to represent the index space and conduct incremental index management, we build a *policy tree*, where the root denotes the initial index set (e.g., primary columns, some distinct columns) and other nodes denote all the possible index combinations. The advantage of *policy tree* is that it can represent both the previous selected indexes (explored tree nodes) and new index combinations (unexplored tree nodes).

Example 5: In Figure 4, before *index update*, the policy tree has three selected nodes and we can see there are two existing indexes $\{I_1, I_2\}$. For the coming workload, we add a candidate index I_3^c and there is a new explored node in the policy tree, which denotes the index set $\{I_1, I_2, I_3^c\}$.

With the *policy tree*, we accordingly solve the above three challenges so as to efficiently search promising index combinations on the *policy tree* for the coming workloads.

A. Template-based Candidate Index Generation.

For any coming workload W, we first generate candidate indexes from the queries in W, which bring new index combinations, i.e., more unexplored nodes in the existing *policy tree* (for **C1.1**). Specifically, this procedure includes three main steps.

Step 1: Reduce workload size by mapping queries into templates. Since workloads may involve numerous queries and it is costly to analyze the index requirements of each query, first we propose to maintain a set of query templates, each of which represent a class of queries with similar index requirements. For example, in Figure 4, queries Q_2, Q_3 have similar access patterns (e.g., used columns, sql format) and so they are both mapped to T_2. More concretely, for any new query, we replace the predicate values in the query with placeholders and match that query with the most similar template. If none of the templates is matched, we record that query as a new template. Note that we only reserve limited template (e.g., 5000 for TPC-C), and so we will update the templates when arriving the maximum number (Section IV-B). Remark. This trick is useful, because, in many scenarios (e.g., online balance inquiry), many queries come from the same templates and only some predicate values are different.

Step 2: Exploit candidate indexes based on the matched templates. Next we generate candidate indexes based on the matched query templates. There are two problems. First, indexes can be derived from various query clauses (e.g., the filter predicate in *WHERE* clause, the expression in *ORDER* clause) and we need to extract useful indexes from those clauses based on their effectiveness. Second, for boolean predicates, there can be complex predicate expressions (e.g., expressions linked by multiple *ANDs* or *ORs*), which have various equivalent forms. We may derive different candidate indexes from those forms.

Example 6: For predicates *"(a AND b) OR (a AND c)"* and *"a AND (b OR c)"*, they are two forms of the same expression, but the former requires two indexes on (a, b) and (a, c), while the latter requires three indexes on a, b, c, which may perform worse if the results of *"(b OR c)"* is large.

To solve those problems, for any query template, we first extract expressions from different clauses (expression extraction), and then separately generate indexes from those expressions (index generation).

Expression Extraction. Based on the index effectiveness, we extract expressions from the query template, which can be divided into three types, including the filter predicates, join predicates, and other expressions: (1) For *filter predicates*, we can enhance the lookup of single tables, i.e., single-column indexes for atomic predicates and multi-column indexes for composite predicates linked by *AND*s; (2) For *join predicates*, we can enhance table joins by building indexes for the driven tables of the join; (3) For other expression that occur in the *GROUP* or *ORDER* clauses, we can speed up their operations by creating indexes on the involved columns.

Index Generation. We separately generate indexes from the three types of expressions:

(1) *Filter predicates*: Firstly, we rewrite the boolean predicate expressions in *Disjunctive Normal Form* (DNF) [6], [4], which provides a unified form and simplifies predicate factorization. For any atomic predicate or composite predicate linked by *AND*s (derived from predicate factorization), if its selectivity is higher than a threshold (e.g., 1/3), we generate candidate index on the column(s) of the predicate; otherwise, we assume it has low selectivity and give up the index.

(2) *Join predicates*: We extract atomic joins between any two tables. For any atomic join, we generate a candidate index on the join column of the driven table, which is generally the smaller table and looked up during join.

(3) *Other expressions*: For expressions in clauses like *GROUP* and *ORDER*, if the expression actually takes effects (e.g., the columns in the *GROUP* clause are not distinct), we generate candidate indexes on the involved columns.

Step 3: Check and remove redundant indexes. With the generated indexes, we filter out duplicate indexes, merge indexes based on the leftmost matching principle. For example, for indexes on (a, b) and a, only (a, b) is reserved because (a, b) can also enhance the access of column a. Then we remove the indexes that already exist in the database. Finally, the remaining indexes form the candidate index set.

B. MCTS-based Index Update.

After generating the candidate index set, whose size is greatly reduced (compared with picking any used columns as indexes), we utilize MCTS to update the existing index set with the candidate indexes in the *policy tree*. Recap that, in the *policy tree*, each node corresponds to a combination of existing or candidate indexes. And so *index update* is to expand the existing policy tree based on the candidate indexes so as to find new index combinations (expended tree nodes) with higher performance improvement (for **C1.2**).

However, even if we have filtered most useless indexes in last step, the index space is still very large (e.g., the factorial of dozens of indexes). To efficiently find the optimal node in the policy tree, we propose a *Monte Carlo Tree Search* based strategy that judiciously explores the nodes to obtain the optimal node. The core idea in MCTS [23] is to balance between exploitation (high benefit) and exploration (low frequency) when searching on the policy tree, which helps to avoid sub-optimum. To apply MCTS, the first step is to define the benefits of any node in the policy tree.

Node Utility. A node has higher node utility if it is on the *path* from the root to the optimal node. However, it is hard to predict whether the node is on the optimal path since each node may have numerous descendant nodes. To address this issue, given a node, we compute its *node utility* by considering two main factors:

(1) *Node Benefit* $\mathcal{B}(v_i)$. Given a policy tree, we define the global node benefit of a node v_i as the highest cost reduction of v_i or v_i's descendant nodes,

$$\mathcal{B}(v_i) = \begin{cases} cost(W) - cost(W, \mathcal{I}_i) & leaf\ \ node \\ \max(cost(W) - cost(W, \mathcal{I}_i)), v_j \in V_i & otherwise \end{cases}$$

where V_i denotes the descendant nodes of v_i. If v_i is a leaf node, $\mathcal{B}(v_i)$ equals to the difference of workload performance with/without indexes; otherwise, $\mathcal{B}(v_i)$ is the highest performance improvement of its descendant nodes. Since v_i may have numerous descendant nodes, we randomly explore several leaf nodes rooted at v_i or descendant nodes that arrive the storage constraint (e.g., 5 leaf nodes for dozens of indexes) to enhance the exploration procedure.

Obviously, we want to access the node with high benefit. However, the estimated benefit may not be accurate, and if we always access such nodes, we may miss the real optimal node. To address this issue, we also consider the access frequency of a node and balance the benefit and frequency in order to avoid falling into a local optimum or wrong directions.

(2) *Access frequency* $\mathcal{F}(v_i)$. The access frequency represents the number of visits of the node v_i when selecting new child nodes. Besides node benefits, we tend to select the node that is rarely accessed in order to try more possible indexes and avoid falling in local-optimum. For example, in Figure 4, when selecting from the candidate indexes, if we greedily select I_3^c, we can only further select I_1^c or I_2^c for storage limit. Instead, the combination of I_1^c and I_2^c can achieve higher benefits, and so we need to try out less frequently selected nodes.

This way, we define the node utility as the upper confidence bound (UCB) of the probability that v_i is on the path from the root to the optimal node, by considering node benefit $\mathcal{B}(v_i)$ and access frequency $\mathcal{F}(v_i)$,

$$\mathcal{U}(v_i) = \mathcal{B}(v_i) + \gamma\sqrt{\frac{ln(\mathcal{F}(v_0))}{\mathcal{F}(v_i)}}$$

where $\mathcal{F}(v_0) = \sum_{i \geq 1} \mathcal{F}(v_i)$ is the number of total accesses, γ is the exploration parameter that adjusts the amount of explorations of uncovered index combinations.

MCTS-based Index Update. As shown in Figure 4, to efficiently update the selected indexes (explored nodes) and explore new indexes (unexplored nodes), the complete index update procedure includes three steps.

Step 1: Node Selection and Expansion. As the node with the maximum utility has the largest possibility of leading to the optimal node, we select the node v with the maximum utility to explore. (i) If the selected node has not been expanded, we enumerate candidate indexes $I_i \in \mathcal{I}^c$. For each candidate index I_i, we generate a new node representing the indexes in v plus I_i within storage limit. (ii) If the node has been expanded, we select a node with the maximum utility from the descendants of this node.

Step 2: Node Utility Computation. For the selected node v, we estimate the node benefit of v. Specifically, we randomly explore $\mathcal{K}$ descendants of v and take the maximum estimated cost reduction as the node benefit of v. Note that, to provide more accurate estimation, the query cost is estimated by our index benefit model which takes both the index effects to read and write queries into consideration (see Section V).

Step 3: Utility Update. As the node utility of v is updated from 0 to $\mathcal{B}(v)$, some ancestors may have smaller benefit than v, and we need to update their subsequent cost reductions based on that of v, i.e., redirecting to a descendant node with higher cost reduction.

Remark. We repeat above steps until arriving the maximum iteration number or meeting the performance expectation, i.e., the estimated cost reduction is approaching $\max(\mathcal{B}(v_i) - \mathcal{B}(v_0))$, where v_i is a tree node.

C. *Incremental Template Update.*

Since we rely on templates (which denote the index requirements of distinct queries) to extract candidate indexes, it is vital to update the templates when the workload has changed. However, it is tricky to forecast workloads, which are affected by many factors outside databases. Hence, we have summarized some practical experience to handle workload changes (for **C1.3**).

First, for most workloads, there are always some query templates that frequently occur. And so we actually can foresee the main trend of future queries based on historical queries, i.e., familiar historical templates have high possibility to recur. Hence, similar to the *LRU* strategies, we only reserve templates that are most frequently matched.

Second, there is some common knowledge that hints when the workload may have significant changes, e.g., the update frequencies of most historical templates are lower than a threshold. In those cases, we need to multiple a decay factor with the frequency values of all the templates, remove the templates with low frequency values, and use the templates of most recent workloads (e.g., within 2 days).

V. Index Benefit Estimation

In index management, accurate index benefit estimation is vital to the final performance [8]. However, existing methods have two limitations. First, most methods [5], [31], [2], [3], [26] rely on the empirical cost estimator in databases, which can make significant errors [34], [14]. Second, deep learning based method [8], [32], [28], [41] has much higher accuracy, but it mainly works for anlytical queries. Moreover, those methods do not consider the costs of updating indexes for write queries, which is extremely important for index management under high-write-ratio scenarios.

To efficiently estimate the index benefit, we need to address three challenges. First, actually building indexes wastes much time and storage space, we utilize the *hypothesis index technique* to save the index creation overhead [2] (**C2.1**). Second, although the cost estimator in databases like PostgreSQL is relatively powerful, they cannot estimate index update costs, which depend on the execution mechanism and are hard to estimate (**C2.2**). Third, traditional cost model is based on the weighted sum of cost features, where the weights are static and may cause great errors. We can utilize historical index data to train a deep regression model, which dynamically learns the weights of cost features and achieves higher accuracy (**C2.3**).

We solve this problem by computing cost features based on practical experience (e.g., how to compute the CPU overhead). Foremost, we define the execution costs.

Execution Cost. Generally, for any query q, the *execution cost* is computed based on the IO and CPU consumption. If q is a write query, there can be additional index update cost. Hence, the execution cost of query q can be written as,

$$cost(q) = Model(\mathcal{C}^{data},\ \mathcal{C}^{io},\ \mathcal{C}^{cpu})$$

[2]github.com/opengauss-mirror/openGauss-server/search?q=hypopg_index

, where $\mathcal{C}^{data}$ is the data processing cost, $\mathcal{C}^{io}$ and $\mathcal{C}^{cpu}$ are separately the IO and CPU costs for index update.

There have been many cost estimation methods that focus on accurately estimating the data lookup cardinality/cost [27], [29]. Instead, we aim to estimate the index update costs, which heavily depend on the mechanism of query execution. For example, for a high-level deep estimation model, it is hard to capture the effects of splitting index pages unless we encode that knowledge within the cost features.

Remark. First, among the write queries, *updates* and *inserts* instantly update the index, which can cause overhead to the queries; while *deletes* update the index after finishing the query, whose index update cost is 0. Second, there are some special mechanisms for index update that are hard to estimate. For example, there can be in-placement update, i.e., the new and old index tuples are recorded in the same heap page, where the *index update cost* is greatly reduced. Here we mainly focus on more general index update cases (e.g., involving disk IO).

A. Computing Cost Features

There are many index statistics that are available within the database kernel (e.g., the number of accessed pages, the tree height of the indexes). Based on those statistics, we can efficiently compute the IO and CPU costs.

IO Cost. IO costs are mainly spent in accessing pages from disk. And so we can use the accessed page number to reflect the io number, and the io cost can be written as $\mathcal{C}^{io} = |pages| * seq\_page\_cost$, where seq_page_cost is a hyper parameter.

CPU Cost. We mainly use the number of accessed tuples to reflect the CPU cost, which is mainly composed of the *start up cost* and *running cost*. *Start up cost* (t_{start}) denotes the cost to find desired index tuples, and *running cost* ($t_{running}$) is for inserting the new or updated tuples.

$$\mathcal{C}^{cpu} = t_{start} + t_{running}$$

$$t_{start} = \{ceil(log(N)) + (H+1) * 50\} * cpu\_operator\_cost$$

$$t_{running} = N_{insert} * cpu\_index\_tuple\_cost$$

, where N denotes the number of index tuples, H denotes the tree height of the index, N_{insert} is the inserted tuple number, and $cpu\_index\_tuple\_cost$ is the hyper parameter.

B. Deep Regression for Cost Model

After computation, the IO and CPU costs have already provided the critical cost information and taking them as the execution cost features can significantly reduce the estimation complexity. However, traditional methods simply sum up those costs based on static weights (e.g., $\mathcal{C}^{io} + 0.01\mathcal{C}^{cpu}$), which can be inaccurate in many cases. To solve this problem, we propose to design a one-layer deep regression model, i.e., $cost(q) = Sigmoid(W_{cost} \cdot \mathcal{C} + b_{cost})$, where W_{cost} and b_{cost} are dynamically learned from historical data, so as to achieve higher estimation accuracy.

VI. EXPERIMENTS

In this section, we conducted extensive experiments to evaluate the proposed techniques and studied the following aspects of our approaches.

RQ1: *Did* AUTOINDEX *gain higher performance (e.g., throughput, total latency) than state-of-the-art methods?* We focus on the workload performance in both testing and real banking scenarios.

RQ2: *How well did important components in* AUTOINDEX *perform?* For space constraint, we mainly report the experimental results on *template-based candidate index generation*.

RQ3: *How well did* AUTOINDEX *performance for dynamic scenarios?* It is vital to support various index requirements in AUTOINDEX. In particular, we ask two sub-questions. **RQ3.1:** Can AUTOINDEX ensure high performance for dynamic workloads? **RQ3.2:** Can AUTOINDEX generalize to various storage constraints?

A. Experiment Setting

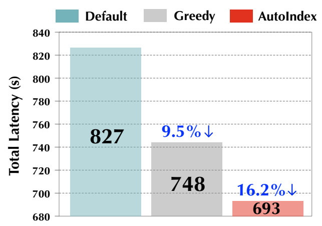
Implementation. We deployed AUTOINDEX in the open-sourced database system *openGauss* [1], [18], [39], where *index selection* was written in Python scripts, and other modules were implemented inside the database kernel.

Environment. In standard testing, we provisioned on a server with 16GB RAM, 256GB disk, and 4Gh CPU. In banking scenario, we used a three-node cluster with one primary node and two replicates. We trained the one-layer estimation model on the same server, for which we sampled 0.01% workload queries and conducted 9-fold cross validation.

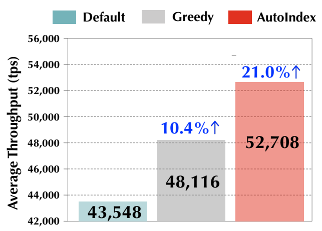
Datasets. In standard testing, we used four standard datasets, including TPC-C1x, TPC-C10x, TPC-C100x, and TPC-DS. For TPC-C, it is a standard OLTP benchmark, which contains 10 tables. For TPC-DS, it is a standard OLAP benchmark, which contains 25 tables and around 1G data. In the banking scenario, we tested AUTOINDEX on the hybrid workload of summarization and withdrawal services (over 2.2M queries). The dataset contains 144 tables and 1G data.

Performance Metrics. We evaluated AUTOINDEX with two aspects of metrics. (1) `Workload Performance`: for a workload, the performance includes the total latency, average throughput, and the number of optimized queries; (2) `Management Overhead`: we used the latency of updating indexes to estimate the overhead caused by index management.

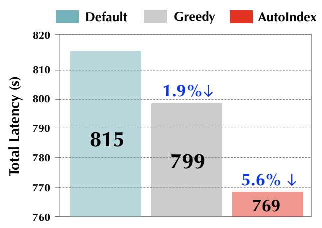
Baseline Methods. We mainly compared with two baselines. First, we compared with default index configurations (`Default`), e.g., indexes on the primary columns for the testing datasets and manually-crafted indexes for the real datasets. In many cases `Default` gained good performance, and `AutoIndex` aimed to find better indexes. Second, we implemented a heuristic method (`Greedy`), commonly adopted in existing works [2], [3], [26]. `Greedy` greedily selected indexes with the highest benefits until arriving resource limit. To ensure the fairness, `Greedy` and `AutoIndex` utilized the same cost estimation method (see Section V).



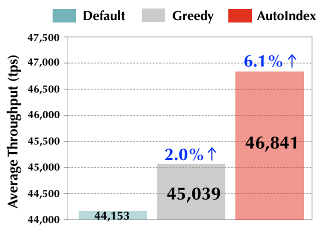
(a) Latency (TPC-C1x)



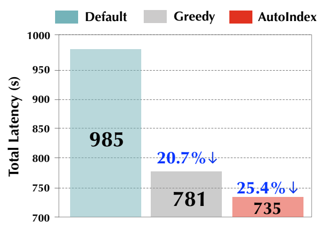
(b) Throughput (TPC-C1x)



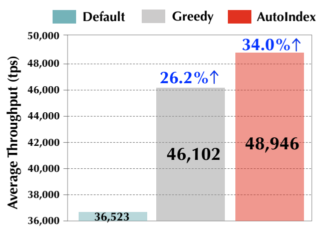
(c) Latency (TPC-C10x)



(d) Throughput (TPC-C10x)



(e) Latency (TPC-C100x)



(f) Throughput (TPC-C100x)

Fig. 5. Performance Comparison on three TPC-C datasets.

TABLE I
ADDED INDEXES COMPARED WITH DEFAULT (TPC-C1X). *cost* ↓ DENOTES THE PERCENTAGE OF REDUCED COST WITHIN ALL THE SELECTED INDEXES.

Greedy	AutoIndex	Cost ↓
(o_c_id, o_w_id, o_d_id)	(o_c_id, o_w_id, o_d_id)	99.4%
	s_quality	21.4%
	(o_c_id, o_d_id)	3.6%

B. Performance Comparison

To answer **RQ1**, we evaluated the index management results on two standard benchmarks and the real-world banking scenarios. We separately demonstrated the performance on TPC-C (Figures 5(a)- 5(f)) and TPC-DS (Figure 6- 7).

Performance Comparison in Testing Datasets. We compared the workload performance (e.g., total latency, throughput) with two state-of-the-art methods, i.e., default configuration (Default), and heuristic method (Greedy).

TPC-C. As shown in Figures 5(a)- 5(f), AutoIndex outperformed Default and Greedy in all the cases. For example, for TPC-C100x, AutoIndex gained over 25.4% latency reduction and 34% throughput improvement than Default; and gained over 4.7% latency reduction and 7.8% throughput improvement than Greedy.

The reasons are three-fold. First, there can be numerous high-concurrency queries in the TPC-C workloads. We have two main observations: (i) they are simple SPJ queries and (ii) many queries come from similar templates. For those queries, AutoIndex can efficiently convert those queries into a few templates, which helped filter most queries and simplified index selection. Instead, Greedy enumerated each query and parsed the candidate indexes from those queries. However, some queries did not frequently occur. So even if the indexes created for those queries can reduce the query cost, if the index cannot be utilized by other queries, it had minor impact on the overall performance. Second, compared with Greedy which only selected atomic indexes (extracted from predicates) with high benefits, AutoIndex considered the combined benefits of multiple indexes. For example, table I demonstrated the indexes separately selected by AutoIndex and Greedy. We can find that, AutoIndex selected two more indexes, i.e., $s\_quality$ and (o_c_id, o_w_id), which had relatively low benefits if we separately compute them, based on which Greedy has dropped them. Instead, AUTOINDEX explored different index combinations in the policy tree and selected the two indexes because them gained extremely high benefits when used together. Third, Default could also gain relatively good performance, because the indexes in Default were most frequently accessed columns or the primary keys. However, some indexes in Default were also frequently updated for the write operations, which may cause negative effects instead. Hence, AutoIndex and Default relied on our index benefit estimator that took the update effects into consideration, and can efficiently avoid selecting columns with high maintenance costs. Moreover, by combining different indexes as multi-column indexes, AutoIndex can gain extremely high cost reduction. For example, by building the index (o_c_id, o_w_id, o_d_id), the query cost was reduced from 406.45 to 20.45 and gained over 99% cost reduction.

TPC-DS. Similarly, in the TPC-DS dataset, AutoIndex also achieved best performance. Instead of directly analyzing the overall performance, we have also demonstrated the detailed optimization levels of the TPC-DS queries. First, in Figure 6, we have three observations: (i) Most queries can be optimized by AutoIndex, since it could judiciously select beneficial indexes that enhanced most queries; (ii) AutoIndex worked better than Greedy because the TPC-DS queries were more complex than the TPC-C queries, and so there were more correlations between the indexes and queries. This way, AutoIndex could utilize the UCT function in MCTS method to estimate the long term benefits of different queries and indexes. Instead, Greedy could only evaluate the benefits of single queries, and led to suboptimal solutions; (iii) For TPC-DS, AutoIndex selected a few more indexes than Greedy (i.e., 9 indexes by AutoIndex and 3 indexes by Greedy). Since the total size of the indexes was still within the resource limit and it was acceptable to gain higher performance at the cost of storage space. Note that, we will discuss the performance changes under different storage limits in the following sections. Second, as shown in Figure 7, another interesting observation is that the optimization levels of some queries in AutoIndex were much higher than queries

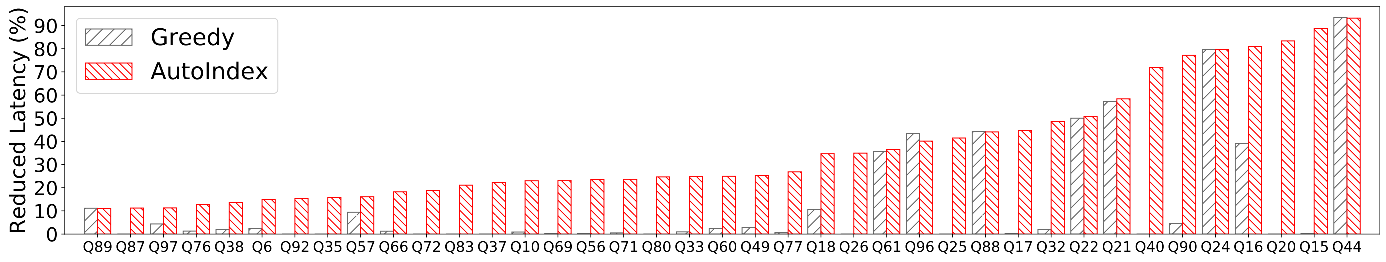


Fig. 6. Execution time reduction of TPC-DS queries.

in `Greedy`. For example, in `AutoIndex`, the execution time of over 44 queries was reduced by over 10%, while there were only 15 queries in `Greedy`. The reason was that `AutoIndex` built more beneficial indexes, which can serve for more queries. Moreover, the co-effects of indexes selected by `AutoIndex` may bring in higher cost reduction. For example, for a query "select * from t1, (select * from t2 where a' = 2) where a = 1 and t1.b = t2.b" (we replace the table and column names for easy description), `Greedy` separately tested indexes on a and a', which can only enhance their own table scan and had relatively low cost benefit. Instead, `AutoIndex` built indexes on both a and a', based on which the two table scans can utilize indexes and enhanced the join operation.

Performance Comparison in Real Datasets. Besides the standard testing datasets, we also verified the effectiveness of `AutoIndex` in the real services of a banking scenario. In this part we actually have conducted two experiments, including removing redundant indexes (Figure 1) and adding beneficial indexes (Table II- III).

(1) For index removal, as shown in Figure 1, recap that we could remove over 80% indexes and still achieve performance improvement. There are three main observations. First, with the lightweight modules like SQL2Template and candidate index generation, `AutoIndex` can finish index management for the 2.2M SQL queries within 11 minutes. Note that the SQL collection process has minor impact on the workload performance, i.e., $< 1\%$. Second, `AutoIndex` has efficiently reduced 83% indexes and 70% disk space, because there are many redundant or even negative indexes that can be identified by our benefit estimation model. Third, to answer why we can achieve throughput improvement after removing so many indexes, one reason is that removing indexes can enhance available memory searching and optimize the utilization of memory space.

(2) For index creation, as shown in Table II and Table III, `AutoIndex` could gain obvious performance improvement. It took minor storage space for storing the new indexes. Specifically, `AutoIndex` only created 33 more indexes, which could serve for different workloads from the hybrid services. That was because it identified the index requirements of different typical queries, and so the most common queries in the two services were identified and optimized. While methods that separately considered each workload (like manual methods), it could not achieve that performance of `AutoIndex`. Second, we can find the summarization service gained a bit higher throughput improvement, because it was of OLAP type and

TABLE II
PERFORMANCE IMPROVEMENT IN BANKING SCENARIO

	Default	AutoIndex
# Non Primary Index	601	+33
The Size of Disk Space (G)	24.4	+1.27G
Summarization Service (tps)	235	+10%
Withdrawal Flow Service (tps)	488	+6%

TABLE III
EXAMPLE RECOMMENDED INDEXES IN THE BANKING SCENARIOS.

Index	Query Cost (no index)	Query Cost (with index)
ind15	12.33	8.32
ind20	59495	7655
ind32	0.101	0.051

contained more complex queries that owned higher optimization potential. Third, as shown in Table III, we showcased some typical indexes selected by `AutoIndex`. We can see, at most, *ind20* can reduce over 98.7% execution time and significantly improved the service quality. In the future we will conduct more experiments to verify the effectiveness of our system in different online businesses.

C. Evaluating Template-based Index Management

To answer **RQ2**, the results are shown in Figure 8. We can see that template-level index management can significantly enhance indexing without obvious performance regression. And there are several observations. First, template-based can reduce over 98.5% overhead, including candidate index generation and existing index update. Because there were numerous TPC-C queries, but those queries may come from a few query templates based on the service types. And thus we can directly capture the query patterns based on the matched templates, rather than asking the index preferences of each query. That was because they could only occur in one of the templates. Second, there were no significant performance regression, i.e., query-based method only outperformed `AutoIndex` by 0.1%, which was not so necessary in comparison with the significant time wasted in candidate index generation by the query-level method. And the reason why `AutoIndex` did not work much worse was two-fold. First, as talked above, the templates were relatively few for TPC-C, i.e., simple queries were more easily captured. Second, we can infer the queries that were not matched with any templates in `AutoIndex`, with the index benefit estimation model, which

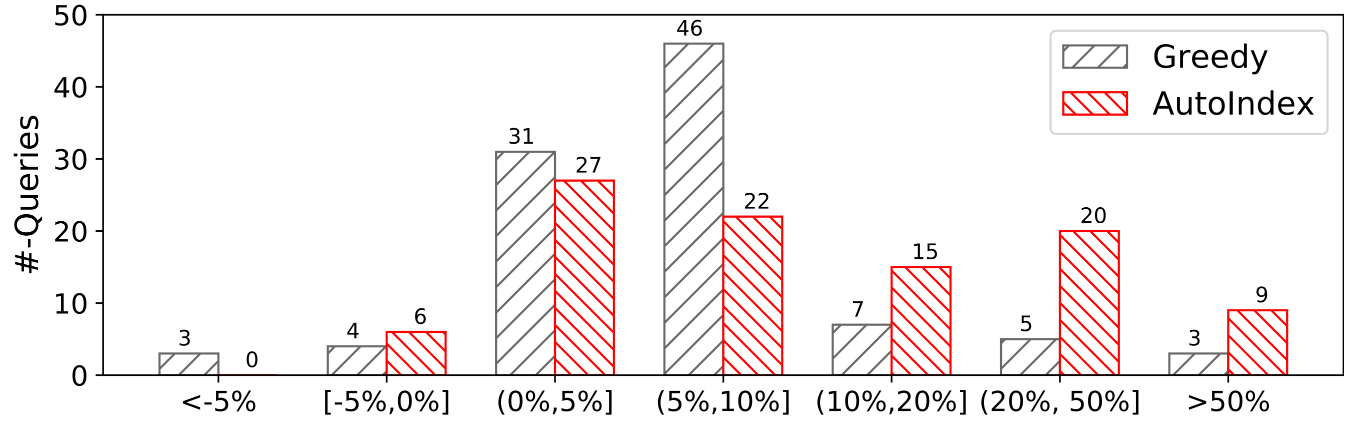
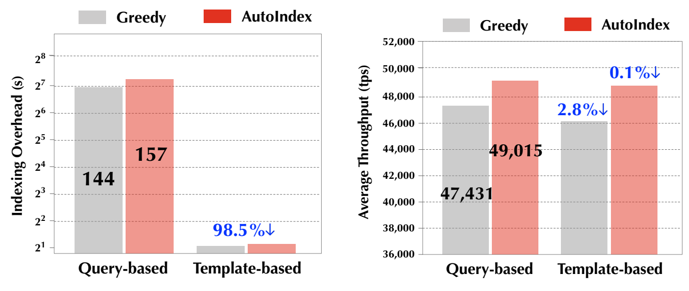


Fig. 7. Performance Comparison on TPC-DS dataset. AUTOINDEX optimized 29 more queries than `Greedy` whose execution time is reduced by over 10%.



(a) Overhead (b) Performance

Fig. 8. Performance Evaluation for Template-based Index Generation.

provided important hints of the index preference of the overall workload, which helped to avoid select suboptimal indexes.

D. *Evaluating Adaptivity for Dynamic Workloads*

To answer **RQ3.1**, we tested the performance of `AutoIndex` and the other two methods on dynamic TPC-C workloads. We continuously issued TPC-C tasks (with different concurrency), and every five minutes we conducted once index management. Note that, there were many insert operations in TPC-C workloads and they caused the table data to grow, and so the performance slightly decreased in the `Default` method.

As shown in Figure 9, `AutoIndex` can adaptively recommend high-performance indexes during workload execution and worked better than the other two methods. There are several observations. First, compared with `Greedy`, `AutoIndex` outperformed `Default` and `Greedy`, because (i) `AutoIndex` efficiently characterized the workload features (e.g., access patterns, frequencies) and the candidate generation model can select a small part of candidate indexes for updating the index set; (ii) `Greedy` can capture changes with the index estimator and prepared index (e.g., very bad estimation results of the recommended indexes, or most templates were not matched for a period of time) and it helped AUTOINDEX to update the templates based on current workloads and find high benefit single indexes. Second, AUTOINDEX still worked best under different storage constraints (i.e., from no limit to 50M). Because the tree search policy in `AutoIndex` can efficiently select indexes that took small space and had high benefits, because it selected optimal indexes by the exploration-and-exploitation strategy. If the storage has arrived limit, it will try out other branches and found better solutions. Instead, `Greedy` only considered high benefits and cannot select any more indexes after picking a few indexes and arriving the resource limit. Third, `AutoIndex` also met performance regression when the storage constraint got smaller, since effective indexes usually took more space to record distinct tuples. Fourth, in some cases, `AutoIndex` got better performance under even smaller storage constraint, i.e., from no limit to 150M. In this case, we found that some indexes may have both relatively small storage space and high performance. Hence, for `AutoIndex`, it is vital to explore untouched part in the policy tree so as to find such index combinations. Fifth, `AutoIndex` achieved lower index latency than `Greedy`, which needed to enumerate each coming query. Hence `AutoIndex` is more suitable to provide online index management for dynamic workloads.

E. *Evaluating Adaptivity for Various Storage Constraints*

To answer **RQ3.2**, as shown in Figure 10, AUTOINDEX still worked best under different storage constraints (i.e., $\{nolimit, 150M, 100M, 50M\}$). We have three main observations. First, `AutoIndex` can achieve high performance for various storage constraints. The tree search policy in `AutoIndex` can efficiently select indexes that took small space but high benefits, because it selected optimal indexes based on the exploration-and-exploitation strategy (the utility function). If arriving the storage limit, `AutoIndex` will try out other branches and found better solutions. Instead, `Greedy` only considered high benefits and cannot select any more indexes after picking a few indexes and arriving the resource limit. Second, `AutoIndex` also met performance regression when the storage constraint got smaller, since effective indexes usually took more space to record distinct tuples. Third, in some cases, `AutoIndex` got better solution

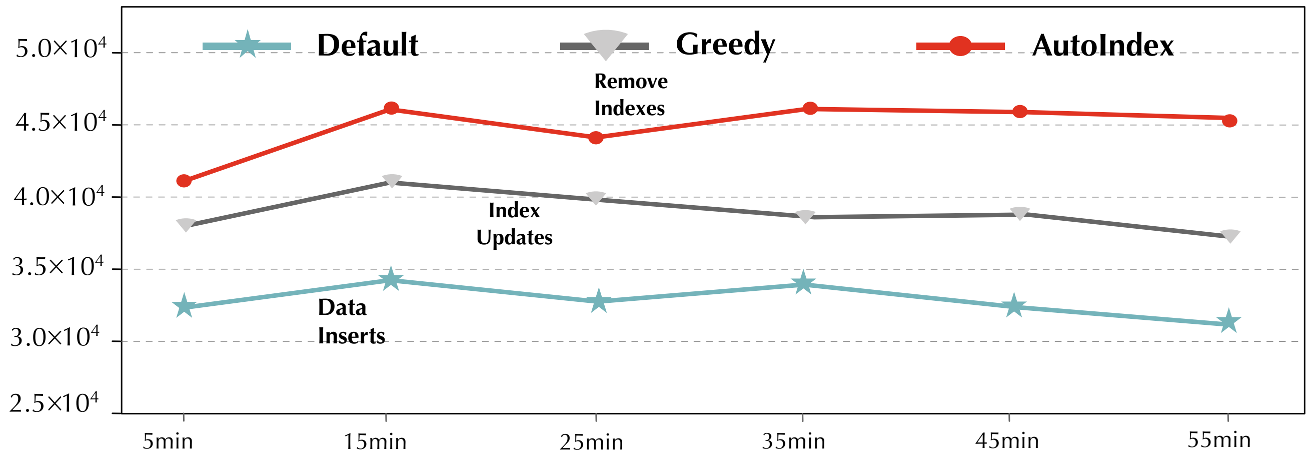


Fig. 9. Throughput Changes for Dynamic Workloads on TPC-C100x dataset.

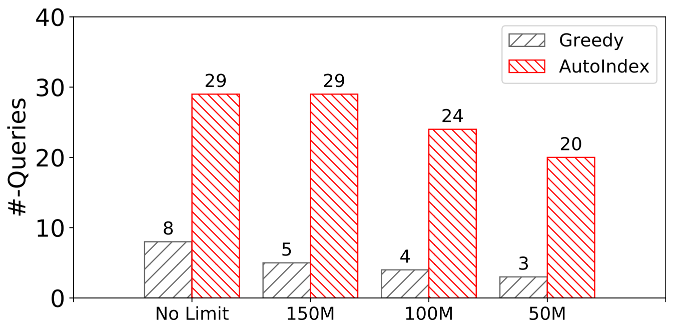


Fig. 10. Performance Under Different Storage Limits (TPC-C100x).

with even smaller storage constraint, i.e., from no limit to 150M, `AutoIndex` recommended indexes with even higher benefit. Hence, we found some indexes may have both cheaper price but high performance. It is vital to explore untouched part in the policy tree so as to find such index combinations.

VII. RELATED WORK

Index Management. Existing index management methods can be broadly divided into two categories, including the optimizer-based methods and ML-based methods. For *optimizer-based methods*, they rely on traditional database estimator to estimate the costs of queries with different indexes, and utilize some heuristic algorithms (e.g., top-$\mathcal{K}$, hill-climbing) to select indexes that can cause execution cost reduction [5], [31], [2], [3], [26], [19]. However, the estimated costs can be inaccurate and mislead to suboptimal solutions. For *machine learning based method*, to solve the problem of inaccurate cost estimation, Ding et al [8] proposed to design a neural network that estimates index benefits based on the plans before/after creating indexes, based on which they recommend any indexes with positive benefits. However, it does not consider (i) the correlations between indexes and queries (e.g., the maintenance cost may be higher than reduced lookup cost) and (ii) the removal of redundant/negative indexes, which can significantly affect the performance.

Machine Learning for Databases. There are increasingly more works that adopt machine learning in database optimization [38], [16], [33], [15], [22] (e.g., deep reinforcement learning for join enumeration [26], [12], [24], [35], [13], [7], view management [36], [9], data partition [10]). However, the advanced DRL methods (e.g., DQN [36], [11], DDPG [37], [20]) cannot be directly applied to our problem for two reasons. First, RL is based on static environment-actor interactions and takes long time (e.g., several days or weeks) to adapt to dynamic workloads [30], [17], which is intolerable in reality. Second, advanced RL methods like DDPG only support creating several indexes at each iteration, but it is hard to support other actions like index removal, because it cannot go back to previous steps. Hence, we propose to utilize a lightweight exploration method (MCTS) [23], [30], [40], which can not only balance between exploration-and-exploitation (finding promising index solutions), but maintains a policy tree (representing all the index solutions), which helps to update historical indexes (parent nodes) and add new indexes (child nodes) for dynamic workloads.

VIII. CONCLUSION

In this paper we proposed an incremental index management system AUTOINDEX for real-world dynamic workloads. We proposed a template-based candidate index generation method to efficiently capture the index requirements of coming workloads. We proposed a MCTS-based index selection method which incrementally updated the indexes to ensure high performance. We proposed a deep index estimation model to estimate the index benefits based on read and write queries. We have implemented the cost estimator inside the database kernel of openGauss. Experimental results showed that our method outperformed existing approaches on both testing and real-world workloads.

Acknowledgement Jianhua Feng is the corresponding author. This paper was supported by NSF of China (61925205, 62072261), Huawei, TAL education, and Beijing National Research Center for Information Science and Technology (BNRist).

References

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

[2] S. Agrawal, S. Chaudhuri, L. Kollár, A. P. Marathe, V. R. Narasayya, and M. Syamala. Database tuning advisor for microsoft SQL server 2005. In *VLDB*, pages 1110–1121, 2004.

[3] S. Agrawal, S. Chaudhuri, L. Kollár, A. P. Marathe, V. R. Narasayya, and M. Syamala. Database tuning advisor for microsoft SQL server 2005: demo. In F. Özcan, editor, *SIGMOD*, pages 930–932. ACM, 2005.

[4] S. Chaudhuri, P. Ganesan, and S. Sarawagi. Factorizing complex predicates in queries to exploit indexes. In *SIGMOD*, pages 361–372, 2003.

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

[6] S. Cluet and C. Delobel. A general framework for the optimization of object-oriented queries. In *SIGMOD*, pages 383–392, 1992.

[7] L. Cui, J. Chen, W. He, H. Li, W. Guo, and Z. Su. Achieving approximate global optimization of truth inference for crowdsourcing microtasks. *Data Science and Engineering*, 6(3):294–309, 2021.

[8] B. Ding, S. Das, R. Marcus, W. Wu, S. Chaudhuri, and V. R. Narasayya. AI meets AI: leveraging query executions to improve index recommendations. In *SIGMOD*, pages 1241–1258, 2019.

[9] Y. Han, G. Li, H. Yuan, and J. Sun. An autonomous materialized view management system with deep reinforcement learning. In *ICDE*, 2021.

[10] B. Hilprecht, C. Binnig, and U. Röhm. Learning a partitioning advisor for cloud databases. In *SIGMOD*, 2020.

[11] S. Huang, Y. Wang, T. Zhao, and G. Li. A learning-based method for computing shortest path distances on road networks. In *ICDE*, pages 360–371, 2021.

[12] 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.

[13] H. Lan, Z. Bao, and Y. Peng. A survey on advancing the dbms query optimizer: Cardinality estimation, cost model, and plan enumeration. *Data Science and Engineering*, 6(1):86–101, 2021.

[14] V. Leis, A. Gubichev, A. Mirchev, and P. A. B. et al. How good are query optimizers, really? *VLDB*, 2015.

[15] G. Li and X. Zhou. Machine learning for data management: A system view. In *ICDE*, 2022.

[16] G. Li, X. Zhou, and L. Cao. AI meets database: AI4DB and DB4AI. In *SIGMOD*, pages 2859–2866, 2021.

[17] G. Li, X. Zhou, and L. Cao. Machine learning for databases. *Proc. VLDB Endow.*, 14(12):3190–3193, 2021.

[18] G. Li, X. Zhou, S. Ji, X. Yu, Y. Han, L. Jin, W. Li, T. Wang, and S. Li. opengauss: An autonomous database system. *VLDB*, 2021.

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

[20] G. Li, X. Zhou, S. Li, and B. Gao. Qtune: A query-aware database tuning system with deep reinforcement learning. *PVLDB*, 2019.

[21] G. P. Licks, J. M. C. Couto, P. de Fátima Miehe, R. D. Paris, D. D. A. Ruiz, and F. Meneguzzi. Smartix: A database indexing agent based on reinforcement learning. *Appl. Intell.*, 50(8):2575–2588, 2020.

[22] C. Lin, J. Zhuang, J. Feng, H. Li, X. Zhou, and G. Li. Adaptive code learning for spark configuration tuning. In *ICDE*, 2022.

[23] A. Liu, J. Chen, M. Yu, Y. Zhai, X. Zhou, and J. Liu. Watch the unobserved: A simple approach to parallelizing monte carlo tree search. In *ICLR*, 2020.

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

[25] R. M. Perera, B. Oetomo, B. I. P. Rubinstein, and R. Borovica-Gajic. DBA bandits: Self-driving index tuning under ad-hoc, analytical workloads with safety guarantees. In *ICDE*, pages 600–611. IEEE, 2021.

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

[27] J. Sun and G. Li. An end-to-end learning-based cost estimator. *VLDB*, 2019.

[28] J. Sun, J. Zhang, Z. Sun, G. Li, and N. Tang. Learned cardinality estimation: A design space exploration and A comparative evaluation. *Proc. VLDB Endow.*, 15(1):85–97, 2021.

[29] I. Trummer. Exact cardinality query optimization with bounded execution cost. In P. A. Boncz, S. Manegold, A. Ailamaki, and et al, editors, *SIGMOD*, 2019.

[30] 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.

[31] 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 D. B. Lomet and G. Weikum, editors, *ICDE*, pages 101–110. IEEE Computer Society, 2000.

[32] J. Wang, C. Chai, J. Liu, and G. Li. FACE: A normalizing flow based cardinality estimator. *Proc. VLDB Endow.*, 15(1):72–84, 2021.

[33] W. Wang, M. Zhang, G. Chen, H. V. Jagadish, and et al. Database meets deep learning: Challenges and opportunities. *SIGMOD Rec.*, 2016.

[34] W. Wu, Y. Chi, S. Zhu, J. Tatemura, H. Hacigümüs, and J. F. Naughton. Predicting query execution time: Are optimizer cost models really unusable? In *ICDE*, pages 1081–1092, 2013.

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

[36] H. Yuan, G. Li, L. Feng, and et al. Automatic view generation with deep learning and reinforcement learning. In *ICDE*, 2020.

[37] L. Zhang, C. Chai, X. Zhou, and G. Li. Learnedsqlgen: Constraint-aware sql generation using reinforcement learning. In *SIGMOD*, 2022.

[38] X. Zhou, C. Chai, G. Li, and J. Sun. Database meets artificial intelligence: A survey. *TKDE*, 2020.

[39] X. Zhou, L. Jin, S. Ji, and et al. Dbmind: A self-driving platform in opengauss. *Proc. VLDB Endow.*, 14(12):2743–2746, 2021.

[40] X. Zhou, G. Li, C. Chai, and J. Feng. A learned query rewrite system using monte carlo tree search. *PVLDB*, 2022.

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