Db2une: Tuning Under Pressure via Deep Learning

Alexander Bianchi
York University, CA
IBM CAS, CA
bianchia@yorku.ca

Andrew Chai
York University, CA
IBM CAS, CA
abfchai@yorku.ca

Vincent Corvinelli
IBM Canada Ltd.
vcorvine@ca.ibm.com

Parke Godfrey
York University, CA
IBM CAS, CA
godfrey@yorku.ca

Jarek Szlichta
York University, CA
IBM CAS, CA
szlichta@yorku.ca

Calisto Zuzarte
IBM Canada Ltd.
calisto@ca.ibm.com

ABSTRACT

Modern database systems including IBM Db2 have numerous parameters, "knobs," that require precise configuration to achieve optimal workload performance. Even for experts, manually "tuning" these knobs is a challenging process. We present Db2une, an automatic query-aware tuning system that leverages *deep learning* to maximize performance while minimizing resource usage. Via a specialized transformer-based query-embedding pipeline we name QBERT, Db2une generates context-aware representations of query workloads to feed as input to a stability-oriented, on-policy deep reinforcement learning model. In Db2une, we introduce a multi-phased, database meta-data driven training approach—which incorporates cost estimates, interpolation of these costs, and database statistics—to efficiently discover optimal tuning configurations without the need to execute queries. Thus, our model can scale to very large workloads, for which executing queries would be prohibitively expensive. Through experimental evaluation, we demonstrate Db2une's efficiency and effectiveness over a variety of workloads. We compare it against the state-of-the-art query-aware tuning systems and show that the system provides recommendations that surpass those of IBM experts.

PVLDB Reference Format:
Alexander Bianchi, Andrew Chai, Vincent Corvinelli, Parke Godfrey, Jarek Szlichta, and Calisto Zuzarte. Db2une: Tuning Under Pressure via Deep Learning. PVLDB, 17(12): 3855 - 3868, 2024.
doi:10.14778/3685800.3685811

1 INTRODUCTION

1.1 Motivation

Database systems such as IBM Db2 contain many configuration parameters, often referred to as "knobs," each with the potential to influence query performance and resource usage [4, 45]. These knobs govern various aspects of the system, such as query optimization, resource allocation, and compiler algorithms. Setting these parameters correctly, "tuning the knobs," is essential to generate *query execution plans* (QEPs) that both optimize system performance and resource usage. For IBM Db2, tuning knobs include *configuration parameters* [11] and *registry variables* [12]. Tuning is often done by experts, such as by *database administrators* (DBAs). Such tuning by hand, however, is both time-consuming and challenging, due to the sheer number of performance-impacting knobs, and the interdependence amongst the knobs, where adjusting one affects the others [11]. As well, an optimal tuning configuration for one workload is likely far from optimal for another. The increasing complexity of modern systems and workloads challenges experts significantly, despite their expertise [7]. Additionally, manual tuning does not scale, say, to thousands of database instances in the cloud.

Given these challenges, automated tuning systems have been proposed [2, 45]. These systems employ a range of machine-learning algorithms to find optimal tuning configurations for specific workloads. Still, these have significant drawbacks. First, the systems require a ***costly training process***, since evaluating database-system performance would seem to necessitate workload execution. For OLAP workloads especially, queries can have long runtimes, even with a good tuning configuration. This heavy training load can lead to extended database downtime. To circumvent this, some approaches [2, 5] clone the database, which itself is costly.

Second, automatic tuning systems can perform poorly due to ***inaccurate workload characterization***. The effectiveness of a tuning system's configuration hinges on an accurate workload characterization that can inform the tuning requirements. *Query-aware* systems attempt to address this by bringing a deeper understanding of the queries, and the query plans, that are involved [19, 24].

Lastly, these systems often ***unnecessarily waste resources***. In cloud-based database-system deployments, users must pay for CPU, memory resources, and I/O bandwidth. In such a setting, efficient resource usage may be as important as overall performance. The less expensive resource allocation option would be preferred if similar latency could be achieved for a workload, say, with 64GB rather than 128GB of RAM. Neglecting resource conservation and focusing solely on performance enhancement may be reasonable when provisioning fixed resources for a dedicated server. However, it becomes unreasonable when resources are shared or paid for per deployment, such as in cloud computing environments.

1.2 Knob Effects Example

Knobs affect the performance of query evaluation by modifying the behavior of the optimizer, and by provisioning resources for

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 17, No. 12 ISSN 2150-8097.
doi:10.14778/3685800.3685811

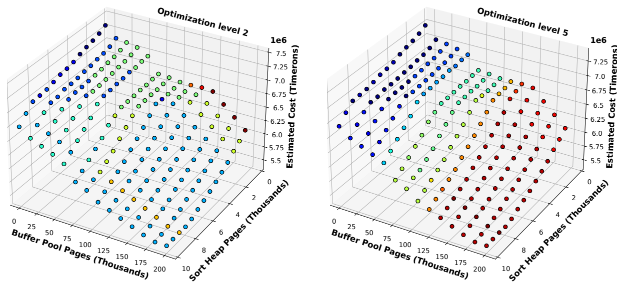


Figure 1: Cost estimates of Q_{23} at varied settings.

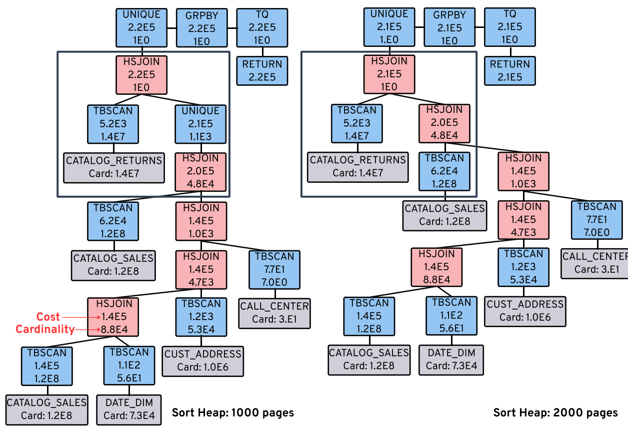


Figure 2: QEPs for Q_{16} for different values of knobs.

database operations. In IBM Db2, *optimization level* specifies which query-optimization techniques will be employed when composing QEPs. The *buffer pool* and *sort heap* knobs set the memory allocations for those, respectively. Their sizes affect the choices of operators during query planning. An insufficient allocation to either leads to less efficient operations and spillage to disk. These settings compete for main memory, a limited resource, however.

Figure 1 plots how varying *buffer pool* and *sort heap* at two different *optimization levels* changes cost estimates for TPC-DS's *query* #23 ($\mathbf{Q}_{23}$) [31]. Note how a change on one knob affects the responsiveness to changes on the others. While we illustrate this here with three knobs, this behavior extends to a multi-dimensional parameter space of knobs to be tuned. Thus, this is a complex search space that is prohibitively expensive to search exhaustively, and difficult to tune manually within. In addition to affecting the estimated cost, knob settings inform the optimizer's plan selection, which may change the structure of a plan. Figure 1 illustrates this, with the same structured plans assigned the same color. Figure 2 illustrates how changing the *sort heap* knob from 1,000 pages to 2,000 pages changes the QEP for $\mathbf{Q}_{16}$ in TPC-DS. Each box is an operator; the three lines in the box report the operator's *type*, and its *estimated cost* and *estimated cardinality* (in exponential notation), respectively. At the lower knob settings, shown on the left, the optimizer chooses to perform duplicate removal, `UNIQUE`, which is not present in the other plan. This reduces the cardinality from the result of the prior `HSJOIN` from `4.8E+4` to `1.1E+3`, done to avoid spilling when executing the subsequent `HSJOIN`. With *sort heap* set at 2,000 pages, the `HSJOIN` likely fits in memory, so removing duplicates is judged not significant enough to offset the cost of the `UNIQUE` operation. Thus, changes in plan structure and operations can result in changes to query performance. For tuning, therefore, we need to encode the structure and statistics of the QEPs.

1.3 Contributions

We present Db2une, an automatic knob-tuning system for IBM Db2, which provides query-aware tuning recommendations for analytical workloads and a transformer-based, query-embedding pipeline via *deep learning*. We provide an architectural overview in Section 2. This work makes the following contributions towards addressing the challenges outlined in Section 1.1.

1. **QEP Embeddings with Context (Section 3).** We propose QBERT, a transformer-based pipeline for generating fixed-length vector embeddings of SQL queries using QEPs, capturing both statistics and structure of the query plan. Thus, QBERT advances beyond prior representation learning techniques such as the *featurization* of QTune [24] and the *bag-of-words* approach of BLUTune [19, 20] by encoding both of these aspects. QBERT embeddings are context-aware, with structurally different QEPs being represented differently, even when they share common operators and statistics. This is necessary for recommending optimal knob configurations for unobserved queries.
2. **A Knob-Tuning System (Section 4).** Db2une leverages *deep reinforcement learning* (DRL) to provide effective and efficient tuning recommendations for diverse query workloads.
   (a) **Stable Deep Learning Approaches for Tuning.** We employ *proximal policy optimization* (PPO), a DRL algorithm identified as state-of-the-art by experts such as OpenAI [33], to tuning data systems. Previous DRL-based work use *off-policy* algorithms [25] that are sensitive to the quality of training data, and have been shown to have convergence issues [2]. PPO belongs to a class of *on-policy* algorithms that are more stable.
   (b) **Multi-phased & Database Meta-data Driven Training.** *Performance Metrics for Multi-phased Training.* We select three interchangeable performance training metrics for use in DRL knob tuning: IBM Db2 *cost estimates*; *interpolated cost estimates* from samples; and actual *query runtimes*. We leverage cost estimates from the IBM Db2 optimizer as our primary training metric, allowing for effective tuning without running expensive analytical queries for training data. We employ estimated costs as a fast and scalable training metric for large workloads, where repetitive query executions are prohibitively expensive. We enhance this further by using sampled IBM Db2 cost estimates and interpolation, for rapid model pre-training over the most relevant knobs. This allows for *fine-tuning* over additional knobs in a multi-phased fashion, using cost estimates and runtimes. *Database Meta-data for Large-Scale Training.* We also propose an approach to training based on *database meta-data*, rather than relying on the full physical database. This method replicates database objects and statistics from the

production system to a test environment with limited resources. This facilitates effective training for very large databases with terabytes of data.
    (c) **Performance and Resource Reward.** We design a *reward function* that guides the DRL agent to learn optimal knob settings, consisting of two components assessing *performance* and *resource* usage. Performance improvement is evaluated using the metrics of cost estimates and runtimes, while an adjustable metric for reducing resource usage—termed *back pressure*—incorporated into the reward encourages resource-effective tuning, as would be needed for cloud-based applications.
(3) **Experimental Validation (Section 5.)**
    (a) **Tuning System Comparisons (Section 5.1).** We demonstrate the efficiency of Db2une across various workloads, showcasing significant improvements when compared against state-of-the-art *query-aware* tuning systems of up to 60%. These systems include BLUTune, which employs QEP2Vec embeddings [19], and QTune, which uses a query featurization approach [24] (which we call Featurization herein). We measure the execution times of unseen test queries under the recommended knob settings from each model. We run two experiments: first, in which a tuning configuration is set for each test query; and second, in which a single tuning configuration is set for the workload of the test queries.
    (b) **Training Approach Evaluation (Section 5.2)** Via an evaluation suite, we determine the trade-offs of overhead and effectiveness for training regimes using our different performance metrics. We show how extremely effective training is just over estimated and interpolated costs, paired with PPO, for very little training overhead. This offers a scalability that other systems simply cannot. We evaluate Db2une trained over database statistics for a very large 3TB TPC-DS database simulating a production environment. This demonstrates its effectiveness in achieving high performance while optimizing resource usage, and reducing human effort, delivering recommendations that surpass those of IBM experts.
    (c) **Back Pressure Evaluation (Section 5.3).** We evaluate the effectiveness of back pressure to minimize resource usage while not degrading performance. We run three experiments: to study the best way to incorporate back pressure into the training; to compare Db2une against itself without back pressure, both setting per query and per workload; and to demonstrate back pressure's ability to conserve resource allocation while preserving performance.

We discuss related works in Section 6, and conclude in Section 7.

2 SYSTEM OVERVIEW

Db2une is a query-aware knob tuning system for IBM Db2, designed to optimize both for performance and resource efficiency for analytical workloads. It is a proprietary prototype for internal use by IBM experts. The system is used in two ways: to *train* a model for tuning recommendations; and to *make* tuning recommendations on request (using the trained model).

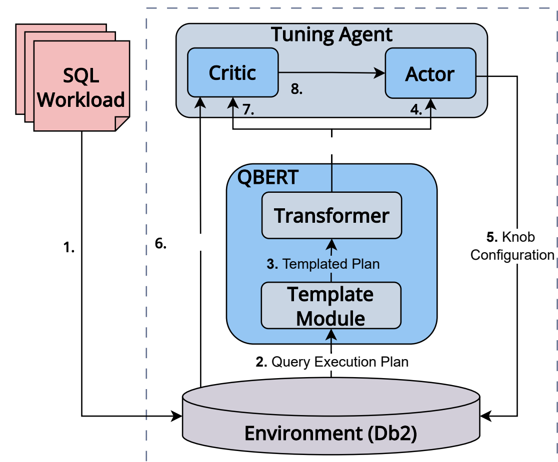


Figure 3: Architectural overview of Db2une.

Figure 3 overviews the architecture of Db2une. A *query workload* is defined as a collection of SQL queries executed over a specific database (its schema and its corresponding data). A *query execution plan* (QEP) details the steps the database system undertakes to retrieve the queried data, including the chosen join order, join type, associated costs, cardinalities of intermediate results, and underlying statistics (as seen in Figure 2).

To train a model, a tuning request is made with a target workload (*Step* 1). The IBM Db2 environment processes the workload with respect to the current tuning configuration (initially, this is its default configuration), generating a QEP per query (*Step* 2). Db2une's QBERT pipeline extracts critical tuning information from the QEPs, mapping features to templates (*Step* 3). The transformer generates a low-dimensional *embedding* for each plan (Section 3). QBERT then aggregates these into a workload embedding vector, which is passed to the *actor model* within the *deep reinforcement learning* (DRL) *tuning agent* (*Step* 4). (Note that a QBERT embedding can represent an individual query or a workload.) The agent, based on *proximal-policy optimization*, recommends a knob configuration aimed at optimizing performance while efficiently allocating resources (Section 4). The recommended configuration is then applied to IBM Db2, which generates new QEPs for the workload based on the new tuning (*Step* 5). A *reward* value is determined, based on the performance cost and the amount of resources allocated (*Step* 6). The workload embedding from *Step* 4 is also passed to the *critic model* in the tuning agent (*Step* 7). Via the reward and the workload embedding, the critic learns how beneficial the actor's actions were, and determines an *advantage* value to guide the actor's future tuning recommendations (*Step* 8). This process is repeated, enabling Db2une to explore the space of knob settings through trial-and-error, until convergence. To use Db2une to make a tuning recommendation, only *Steps* 1 through 5 are executed. The DBA screens final knob recommendations.

3 QUERY PLAN REPRESENTATIONS

To determine an optimal tuning configuration, a tuning system must identify and represent *features* of the target workload that align with the tuning requirements. These could encompass database

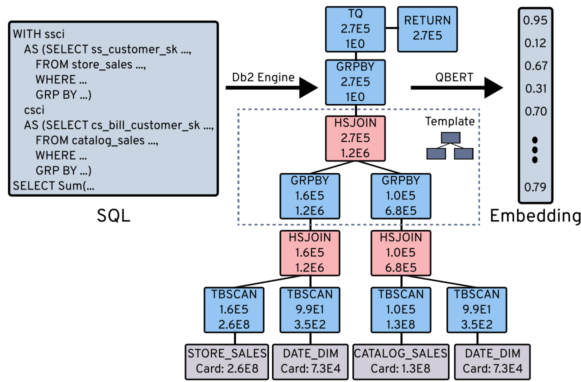


Figure 4: Query plan for Q_{97} representation via QBERT.

metrics and query-workload characteristics. A *query-aware* system informs tuning decisions by focusing on specific aspects of the query workload, including the query operators, operator structures, and costs associated with CPU and input/output (I/O). Being able to identify, for example, that a particular query is memory-intensive would inform the system to increase the memory-related knobs such as sort heap and buffer pool sizes.

Previous query-aware systems have demonstrated the ability to determine knob settings, such as QTune [24] and BLUTune [19, 20], and to identify problematic components [21]. These methods suffer from several limitations, however. QTune relies on a simplistic featurization approach, aggregating operator costs across an entire query, and hard-coding schema table and attribute names, which limits its ability to generalize to other query workloads. BLUTune's QEP2Vec model, inspired by Doc2Vec neural document embeddings [22], treats query plans as documents and operators as a bag of words. This fails to capture the inherent structure of the query plans, however. It also produces non-deterministic query embeddings, which hinders the tuning learning process.

To address these limitations, we propose the QBERT *pipeline* for templating query plans and generating embeddings from them. Leveraging a transformer architecture, QBERT represents query plans in a low-dimensional vector space. This facilitates clustering based on structure and costs while anonymizing schema names. This is more adaptable and effective for query-aware knob tuning. Figure 4 provides an overview of how QBERT transforms a query plan into a vector embedding, via an example with Query #97 ($\mathbf{Q}_{97}$) from the TPC-DS benchmark.[1] A templating process is subsequently employed to extract relevant tuning information from these plans, as detailed next in Section 3.1. This extracted information serves as input to the QBERT transformer model, responsible for generating the embeddings, discussed in Section 3.2.

3.1 The Templating Process

Learning tuning configurations involves observing how the performance changes while processing the workload from tuning to tuning. Unlike the SQL queries themselves which remain unchanged across tuning configurations, the query plans do change. As such, we opt to learn representations of the QEPs as generated by IBM Db2's cost-based optimizer. The effectiveness of a new tuning, in turn, is assessed by observing changes in the query plans [6]. To extract relevant information from QEPs, we employ a *templating process*, which represents a QEP as a sequence of templates. A template encodes the operators' information, and costs, akin to the "boxes" in a QEP plot, as seen in Figure 4.

To transform a QEP into templates, we traverse its graph structure by a depth-first walk to extract the physical operators and their attributes. During this traversal, we use a hash function to map a relevant window of operators, a *sub-plan*, to a template. These templates encapsulate operator types and comparable cost and cardinality ranges (within upper and lower bounds). The costs are a weighted sum of I/O, CPU, and communication costs, measured in *timerons* [13] within IBM Db2. The hashing function assigns the same template for sub-plans when they share the same operator types and exhibit similar costs and cardinalities. If no template exists for a sub-plan with operators within a specific cost and cardinality range, a new template is created. Templates are represented as *<operator window>–<attribute range>*, allowing for different ranges of costs and cardinality to be associated with the same window of operators. For example, as shown in Figure 4, a template might represent a combination of a Hash Join (HSJOIN) connecting two grouping by (GRPBY) operators. The template is labeled accordingly based on the range of costs and cardinality. If costs and cardinality increase, the template label will reflect a greater range. Similarly, changes in operator combinations, such as replacing a Hash Join with a Merge-Sort Join (MSJOIN), result in new template labels.

Our architecture supports templating for any operators. We presently template for join type (e.g, Nested-Loop Join (NLJOIN), Hash Join (HSJOIN), and Merge-Sort Join (MSJOIN)), and neighboring operators, such as table scan (TBSCAN), index scan (IXSCAN), unique (UNIQUE), sorting (SORT), and grouping by (GRPBY). We abstract table and attribute names into canonical symbols, allowing us to characterize QEPs based on their most significant features. This facilitates the identification of (sub)plans with similar characteristics and structures across queries and query workloads, even when underlying tables and attributes differ. By decomposing the QEP into an ordered sequence of templates, the templating process enables us to learn query representations that are context-aware.

3.2 QBERT Embeddings

A sequence of templates representing a QEP can be arbitrarily long, which does not mesh well as input for a machine learning model. As such, we seek to map this into a low-dimensional embedding space, in which a QEP is represented by a fixed-length vector, while preserving the semantic richness of the QEP. We conceptualize transforming a QEP template sequence into an embedding as a *natural language processing* (NLP) problem, where a QEP template sequence *is to* a document *as* its templates *are to* words.

Towards this end, QBERT employs a *transformer* architecture, a state-of-the-art neural network in the field of NLP, to create contextually aware embeddings [9, 40]. Specifically, it leverages the Bidirectional Encoder Representations from Transformers (BERT) architecture [9], known for its effectiveness in generating contextually rich embeddings via self-supervised training tasks. BERT processes a sequence of tokens (e.g., words in text) that are divided

[1] IBM Db2 offers a command-line tool db2exfmt to explain its generated query execution plan. We use this to pull graphical representations of query plans as shown herein.

into segments (e.g., sentences). It undergoes training on two main tasks: *masked-language modeling* and *next-segment prediction*. Following observations made in RoBERTa [26] for the task of question answering where the former is more performant, we have observed the same for us. Thus, we restrict our focus to that herein. For the masked-language task, BERT is given a sequence with some tokens randomly replaced by a placeholder mask. Its goal is to predict the original tokens at these masked-out positions accurately.

The *masked-template prediction* task is our version of the masked-language task: the model learns to predict the masked portions of the query-plan sequence. For example, the model can identify that a filter operation usually occurs after a table scan (TBSCAN), but before a nested-loop join (NLJOIN). It may determine the reason for the high costs associated with certain operations is due to their placement relative to preceding operators.

The *attention mechanism* in the transformer's neural networks mimics human attention, allowing the model to focus on the parts of the input data most relevant for performing a given task. This dynamically weighs different input elements for more accurate, context-aware outputs. BERT's bidirectional nature complements this well by providing contextual information from both directions within the sequence. This enables the model to focus dynamically on portions of the input QEP sequence during the prediction of masked-out tokens, thereby uncovering and understanding complex relationships amongst the operators.

Figure 5 illustrates our QBERT pipeline. The templated QEP undergoes *tokenization*, to represent it as a sequence of vocabulary elements (tokens). QBERT employs a word-piece tokenizer [9], decomposing template segments into sub-word units where operators serve as root words, and their associated costs and cardinalities become suffixes. This helps the model to generalize to unseen templates by splitting unseen tokens into smaller seen tokens. The tokenized templates, $\mathtt{Tok}_1, \ldots, \mathtt{Tok}_n$, are input to the QBERT transformer where they then are mapped to *learned template embeddings*, $\mathtt{Emb}_1, \ldots, \mathtt{Emb}_n$. These template embeddings are then *augmented* by the transformer neural encoder into *contextual embeddings*, $\mathtt{Emb}'_1, \ldots, \mathtt{Emb}'_n$, to encode context information. The inclusion of contextual information allows the transformer to interpret the ordering of templates, preventing it from perceiving the sequence as a bag of words. The template embeddings traverse multiple encoders, leveraging multiple attention mechanisms to encode complex relationships and contextual information.

The components of the pipeline boxed with a dashed black line are present just during the training phase of the transformer, specifically for the masked template prediction task. The transformer training based on this task is sketched in Algorithm 1. Given a set of queries, the QEPs are generated (*Line* 3), which then undergo the templating process (*Line* 4). These templates are tokenized and randomly masked with a special token, MASK_TOKEN, with some probability; e.g., 15% (*Line* 10). (In the figure, the masked elements are encircled with a dashed black line and thatched.) The transformer proceeds with a forward pass (*Line* 13), computing contextual embeddings for the masked-token embeddings via the use of the encoders. The masked-template contextualized embeddings are used as input to a prediction network that aims to predict the masked templates based on the encoded contextual information (*Line* 14). The *softmax* activation function is applied to calculate

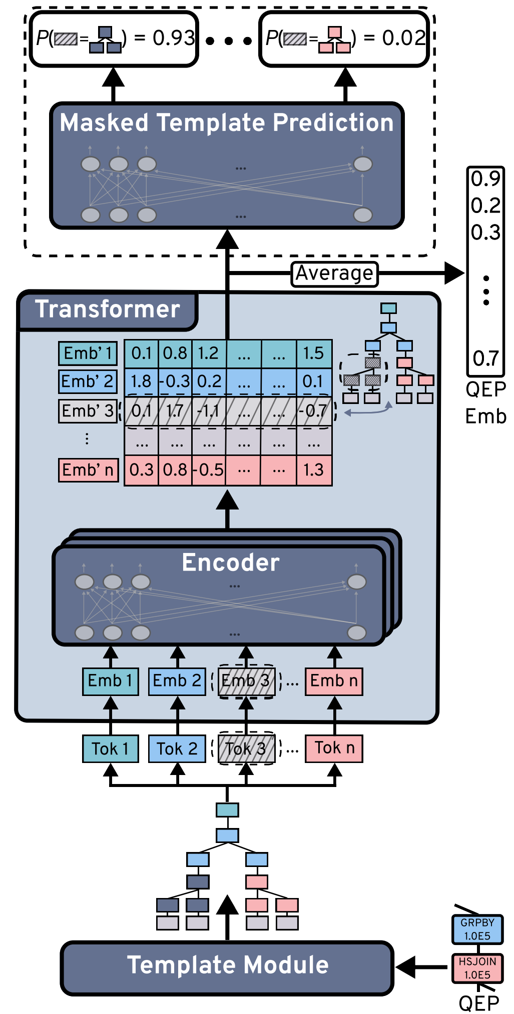


Figure 5: The QBERT pipeline.

"probabilities" for each template (over the entire vocabulary) to determine its likelihood of being the masked template (*Line* 15).

For instance, the predicted probability for the masked dark-blue colored template is 93%. The model's performance is assessed using cross-entropy loss, which compares the distribution of predicted templates (93% for the dark-blue colored and 2% for the pink colored template) against the actual template distribution (100% for the dark-blue colored and 0% for the pink colored template). This indicates that the prediction was accurate, as only the dark-blue colored template was masked. Iterative refinement of the transformer's weights based on multi-class cross-entropy loss function (*Lines* 16–17) allows the model to identify contextual template embeddings that predict the masked templates with high accuracy.

Ultimately, the transformer outputs a contextual embedding for each template. These are averaged to obtain an embedding to represent the entire query plan. This *QEP embedding*, indicated as $\langle 0.9, \ldots, 0.7 \rangle$ in Figure 5, captures the operators' properties and their relationships. Similarly, we obtain a workload embedding by averaging the workload's QEP embeddings. There are a variety of tasks, such as clustering and knob tuning, that can use QBERT embeddings. Our experiments in Section 5 show that our embeddings

Algorithm 1 QBERT-Train

```
1:  Input: dataset of queries {q_n}_{n=1}^{N_data}, initial parameters θ
2:  for n ∈ [N_data] do
3:      q_n ← GET-QEP(q_n)
4:      q_n ← TEMPLATE-QEP(q_n)
5:      q_n ← TOKENIZE-TEMPLATES(q_n)
6:  end for
7:  for i ∈ [N_epochs] do
8:      for n ∈ [N_data] do
9:          for t ∈ [len(q_n)] do
10:             q̃_n[t] ← MASK_TOKEN with probability p_mask
11:         end for
12:         T̃ ← {t ∈ [len(q_n)] : q̃_n[t] = MASK_TOKEN}
13:         X ← TRANSFORMER-FORWARD(q̃_n|θ)
14:         X ← LINEAR(X)
15:         P ← SOFTMAX(X)
16:         loss(θ) ← − Σ_{t∈T̃} q_n[t] log P(t)
17:         θ ← θ − α · ∇loss(θ)
18:     end for
19: end for
```

allow Db2une to achieve significantly better tuning performance than competing methods.

4 AUTOMATIC TUNING SYSTEM

To navigate the complexity of the tuning configuration space, we employ deep reinforcement learning (DRL), using *actor-critic* networks, as illustrated in Figure 3. This approach trains two neural networks in parallel: an *actor*, which learns a *policy* π to map an input *state* to a set of recommended *actions*; and a *critic*, which learns a *value function* to estimate the effectiveness of the actions. The actor updates its policy as guided by the critic's values to learn which actions are beneficial, and which are detrimental, in a given state. The actor and critic together constitute an *agent*, tasked with learning how to maximize *rewards* within its *environment*. This is achieved by combining *exploration*, where new strategies are tried, and *exploitation*, where known strategies are employed.

In the context of Db2une, the agent interacts with the target IBM Db2 database environment. Knob configurations are recommended via the actor's actions, which align with the current *state* of the model. This state is represented by the QBERT workload embedding at the current knob settings. Applying a new tuning configuration changes the state. A *reward* (Section 4.3) is subsequently calculated based on the changes in performance metrics (Section 4.2), and resource allocation. The critic uses this reward to learn how effective the current tuning is, computing an *advantage* score to provide feedback to guide the actor's future recommendations.

4.1 Stability-Oriented, On Policy Tuning

Prior work in DRL for database knob tuning [2, 5, 16, 17, 24, 43] has employed *off-policy* algorithms such as *deep deterministic policy gradient* (DDPG) [25]. On the one hand, these methods favor sample efficiency, which allows for the reuse of expensive sampled data. On the other hand, they have difficulty converging on tuning tasks [2] and are sensitive to the quality of training data [42]. DDPG specifically exhibits instability and is brittle to hyperparameter selection [14, 18]. As off-policy algorithms update the current policy using experiences collected under older policies, misalignment between past and target behaviors can hinder learning. Additionally, fluctuations in reward signals lead to disruptive policy updates, undermining the stability of DRL algorithms [3].

We employ an *on-policy* algorithm, in contrast to this prior work. On-policy algorithms update the policy based on the data collected while following that policy. This is more stable and exhibits better convergence [32, 33]. A potential drawback of on-policy algorithms is that they discard past experiences after each update. This requires fresh data for each iteration. This can be limiting in environments in which sampling is expensive, as is sampling latency and throughput from the database environment.

Our system mitigates these drawbacks in two ways. First, we pretrain the policy using quickly-obtainable cost estimates instead of using runtimes (Section 4.2). Second, we employ the *proximal policy optimization* (PPO) on-policy algorithm [33], recognized as state of the art by OpenAI [30]. PPO is recognized for its stability and adaptability, and can handle discrete and continuous action spaces. This makes it particularly well suited for knob tuning. PPO enhances training stability by limiting the magnitude of policy updates in each training iteration. This is accomplished by evaluating the deviation of the current policy π_θ from the previous policy $\pi_{\theta_{old}}$ by using their ratio, denoted as $f_t(\theta)$, as in Equation 1:

$$f_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \tag{1}$$

The ratio $f_t(\theta)$ calculates the *relative likelihood* of the current policy π_θ choosing action a_t at state s_t compared to the old policy $\pi_{\theta_{old}}$ at time step t. For the task of tuning, a_t represents the knob configuration, and s_t represents the query workload embedding. The ratio indicates the extent of the divergence between this and the old policy. If $f_t(\theta) > 1$, the current policy is more likely to select action a_t at state s_t, than under the old policy. Conversely, if $f_t(\theta) < 1$, the current policy is less likely to choose a_t at s_t,

Db2une employs the advantage actor-critic framework, an improvement over the original actor-critic algorithm used in QTune [24] and CDBTune [43]. In the original framework, the critic learns the Q-value function $Q(s_t, a_t)$, which estimates the overall expected reward for an actor-agent in state s_t that performs an action a_t, following a specific policy until the end of the episode. The Q-value alone, however, does not capture the relative quality of an action compared against other possible actions. To address this, the *V-value* function $V(s_t)$ is introduced, which indicates the overall expected reward of the agent by performing an "average" action in state s_t, then following the policy until the end of the episode. The *advantage*, denoted as $\widehat{A}(a_t, s_t)$, of a given state-action pair is defined in Equation 2 as the difference between the Q-value and V-value. Thus, the advantage captures how good the action taken is compared against other possible actions in that state.

$$\widehat{A}(s_t, a_t) = Q(s_t, a_t) - V(s_t) \tag{2}$$

For conciseness, we refer to the reward from taking action a_t at state s_t simply as r_t. Since the Q-value can be expressed as the sum of the intermediate rewards r_t discounted by the parameter γ, minus the V-value at the end of the episode at timestamp T, we

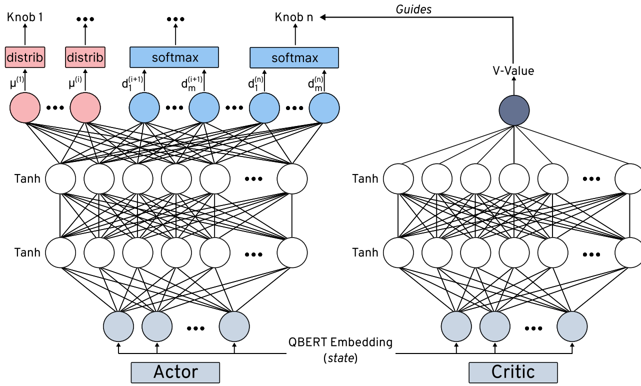


Figure 6: Tuning agent based on advantage actor-critic.

can rewrite the advantage as in Equation 3.

$$\widehat{A}(a_t, s_t) = \sum_{i=t}^{T-1} \gamma^{i-t} r_i + \gamma^{T-t} V(s_T) - V(s_t) \tag{3}$$

PPO's clipped surrogate objective function, denoted by L^{clip} in Equation 4, restricts updates by constraining the ratio $f_t(\theta)$. If the ratio exceeds a predefined threshold, the update is *clipped*, ensuring it does not deviate too far from the previous policy.

$$L^{clip} = \widehat{\mathbb{E}}[\min(f_t(\theta)\widehat{A}_t, \text{clip}(f_t(\theta), 1-\epsilon, 1+\epsilon)\widehat{A}_t)] \tag{4}$$

The objective function in Equation 4 aims to maximize the product between $f_t(\theta)$ and $\widehat{A}(a_t, s_t)$, indicating the current policy is more likely to favor advantageous actions than the old policy. To prevent excessively large updates, $f_t(\theta)$ is clipped within $[1-\epsilon, 1+\epsilon]$, with the threshold ϵ typically set to 0.2 [33]. The final objective function is determined by selecting the minimum value between the clipped and unclipped objectives. The policy is updated multiple times each episode to promote efficiency.

Figure 6 illustrates our actor-critic framework. The actor models multiple probability distributions and can sample discrete and continuous actions. A discrete knob may lack a linear mapping to a continuous variable. Being able to model simultaneously continuous and discrete variables is an advantage over algorithms such as DDPG, used by QTune and CDBtune, which are limited exclusively to modeling continuous variables. The actor processes the QBERT state embedding through two fully connected layers with *tanh* activations. The outputs represents the means of the normal distributions for sampling continuous actions (pink nodes), and the *logits* of categorical distributions for discrete actions (blue nodes). The variance of each of the normal distributions serves as a model parameter. The critic also processes the QBERT state embedding to compute the state's V-value, $V(s)$. The V-values and the rewards are used to calculate the advantage (Equation 2), which, in turn, guides the actor-network to minimize the objective function (Equation 4).

4.2 Multi-phased & Meta-Data Driven Training

Performance Metrics: Multi-phased Training. We support three *training metrics* to generate our reward: query runtimes; IBM Db2 "timeron" cost estimates; and an interpolated estimate of those costs. These are interchangeable, enabling training on one and *fine-tuning* on another via transfer learning in a *multi-phased* approach. This allows us to select initial metrics that facilitate faster model training, to be followed by fine-tuning on more precise metrics.

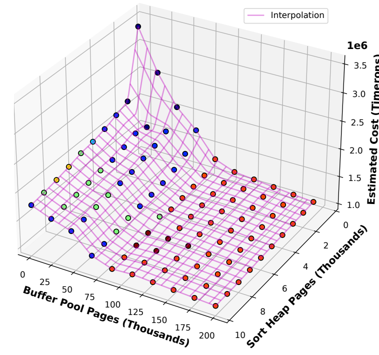


Figure 7: Example of interpolation for Q_{64}.

Prior tuning work relies predominantly on query workload runtimes, but this *does not* scale for workloads with expensive analytical queries and very large databases. Poor tuning configurations (including the default) sampled during training can lead to unreasonably long runtimes. Therefore, we opted to use IBM Db2 cost estimates as our primary training metric. These estimates, generated by the query optimizer for a given query and knob configuration during QEP compilation, provide valuable insights into query performance. While traditional optimizer cost estimates have faced challenges with accuracy [10, 23], recent advancements in machine learning-based cardinality estimation have significantly improved these estimates, offering promising enhancements for tuning efforts [29, 36]. Cost estimates do not capture the impact of every knob, such as the degree of parallelism, however, this approach allows many of the most important knobs to settle into optimal settings.

This *does* scale; for TPC-DS, we empirically verified that the overhead in time for compiling the queries during training increased by 9% when scaling from 1GB to 100GB (and by only 7% when scaling from 10GB to 100GB). In contrast, training based on execution times can vary by orders of magnitude—ranging, from minutes to hours for 1GB versus 100GB. Furthermore, for a specific dataset size (e.g., 100GB), compilation times remain relatively stable across various knob configurations. Meanwhile, execution times for different knob configurations can vary significantly, often leading to query plans for which the execution time can be hours longer, or may not complete at all, having a significant impact on training. In Section 5, we demonstrate that effective tuning can be provided via cost estimates, with the ability to fine-tune models further using query runtimes, controlled by the number of training steps.

Despite using cost estimates, the process of modifying database settings, even virtually, and re-compiling the QEP for each sample remains the main time bottleneck during training. To address this issue further, we implement a process in our model to estimate the IBM Db2 cost estimates themselves. We initially sample QEPs with their corresponding costs and knob settings. Then using the sampled costs at given settings, we create a function to interpolate linearly between the sampled costs. Figure 7 illustrates this interpolated function overlaid over sampled costs for TPC-DS's $\mathbf{Q}_{64}$. During training, we use this function to obtain an interpolated cost estimate. The embedding, used as the input state for Db2une, is the nearest-neighbor sampled QEP. As QEPs with small changes

in knob settings can have similar structures, and only vary slightly in cost metrics, the resulting embedding is sufficiently similar for tuning purposes. In Section 5.2, we show that by removing the need to change database system settings and re-compile QEPs, we improve the time to train significantly, while obtaining reasonable results which can be further fine-tuned. Our interpolation is limited in the number of knobs we can simultaneously tune, of course, since sampling involves combinations of knob values. Thus, we prioritize sampling the most impactful knobs for a given workload. IBM experts have observed that typically it is a subset of knobs that are most influential.

Meta-Data Driven Training. A significant challenge for tuning database systems in prior work is *downtime*, as the database is typically taken offline during training. While cloning the database is a workaround for this, this introduces potential concerns regarding space, resources, data duplication time and synchronization, and network transfer costs. We propose a novel method to train over database meta-data instead of the full physical database. We leverage the `db2look` tool, which extracts data definition language (DDL) statements for replicating database objects and generates `UPDATE STATISTICS` for replicating catalog statistics on a test system. This facilitates the synchronization of query optimizer configuration settings with a production database.

Moving the training of the tuning agent onto the test system allows for `Db2une` to experiment with different knob configurations without the risk of affecting customer workloads on the production system [8]. The `db2fopt` command is used during training to let us adjust memory parameters used by the query optimizer virtually. This setup enables a test system with limited resources (such as memory and CPU) to generate query plans mirroring those of a higher-capacity production system. The query plans cannot be executed on the test system since no copy of the data exists. However, this is not an issue since our training focuses on reducing QEP estimated costs. Consequently, we can train the model solely on the database statistics, even when our test system lacks the resources of the production system. This method performs and scales well for very large databases with terabytes of data (Section 5.2)

4.3 Performance and Back Pressure Reward

The reward signal plays a vital role in guiding the agent to learn which knobs work best for a given workload. Our reward function, r_t, consists of two components, each assessing a key tuning metric. First, we evaluate how much our knob recommendation enhances the *performance* of queries, denoted by r_{perf}. Second, we measure the reduction in system *resources*, compared against other beneficial knob settings, denoted by r_{res}. Then r_t is the sum of these, with r_{res} weighted by a binary parameter $\delta \in \{0, 1\}$.

$$r_t = r_{perf} + \delta r_{res} \tag{5}$$

Performance. We design r_{perf} to enhance performance, employing metrics such as the cost estimates generated by `IBM Db2`'s optimizer and actual query runtimes (detailed in Section 4.2). This is calculated as the difference in workload performance attributed to the agent's current knob recommendation and the performance of its previous and initial knob recommendations [24, 43]. This provides the agent with immediate feedback, indicating whether its tuning actions improve or worsen performance throughout an episode. Thus, r_{perf} is defined as in Equation 6.

$$r_{perf} = \begin{cases} ((1 + r_{prev})^2 - 1)|1 + r_{init}| & \text{if } r_{init} > 0 \\ -((1 - r_{prev})^2 - 1)|1 - r_{init}| & \text{if } r_{init} \leq 0 \end{cases} \tag{6}$$

Here, r_{init} is the performance difference between the current performance and the initial sampled performance, normalized by the initial performance, and r_{prev} is the performance change between the current cost and the previous sampled performance, normalized by the previous performance. After calculating r_{perf}, we normalize its value to the range $[-5, 5]$ to maintain a controlled scaling.

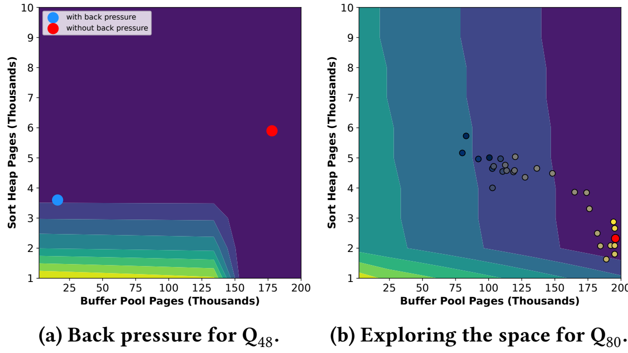
Back Pressure. Our design of r_{res} is motivated by the application of automated tuning systems in multi-tenant settings, including cloud environments. In such settings, database tuning should only use as many resources as needed to achieve a high performance level to prevent the customer from paying for unnecessary resources. Other query-aware tuning systems lack the utilization of resource rewards [24] or are constrained by upper-bound resource limits [19, 20]. In contrast, our system implements *back pressure*—we refer to the idea of pushing back on a high resource allocation as "back pressure"—into the reward to accommodate this need for intelligent resource management. For two knob configurations that result in the same performance, the one that used fewer resources is favored. During training, we track the best-performing configuration seen for a given query. If a newly sampled configuration's performance falls within α of the best, we calculate an additional incentive or penalty for each knob that has back pressure applied. Thus, this parameter α controls the tolerance for change in performance before back pressure is applied.

For each knob, we have its current recommended resource setting, r_{rec}, its resource setting in the best-seen performance settings, r_{best}, and its maximum setting, r_{max}. A parameter β is used to weigh the magnitude of the resource reward. The resource reward is then calculated for each knob i independently, as in Equation 7.

$$r_{res}^{(i)} = \beta^{(i)} \frac{r_{rec}^{(i)} - r_{best}^{(i)}}{r_{max}^{(i)}} \tag{7}$$

For knobs that allocate shared resources—for instance, such as buffer pool and sort heap do for memory—the resource reward is calculated by subtracting the current number of recommended pages of memory and the pages allocated to achieve the best-seen performance, normalized as per Equation 7. Back pressure can also be applied to discrete knobs with ordinal values. For such a knob, we map its n ordered discrete values onto $[1, \ldots, n]$, and set its maximum value to be n. Given this, we can then treat discrete and continuous knobs the same for the purpose of computing r_{res}'s as per Equation 7. Given tuning knobs $1, \ldots, k$, the overall resource award, r_{res}, is then simply the sum over the individual knob resource awards: $r_{res} = \sum_{i=1}^{k} r_{res}^{(i)}$.

In `IBM Db2`, for example, the *optimization level* knob has seven settings: 0–3, 5, 7, and 9. It dictates which optimizer algorithms will be used and the number of plans that will be considered for a SQL statement. A higher setting indicates a higher degree of optimization. We found that on large, real-world workloads, setting this too high may worsen runtime performance. The system often selects the same plan as it would have with a lower optimization level, but at a greatly increased compile time, sometimes even exceeding the



(a) Back pressure for Q_{48}. **(b) Exploring the space for Q_{80}.**

Figure 8: Db2une setting knobs.

actual execution time. Applying back pressure to this knob encourages the model to select the lowest optimization level to achieve the highest-performing plan, potentially reducing compilation times. However, we do not explicitly enforce resource constraints when they are not captured in cost estimates (i.e., compile time).

Figure 8 shows a projection onto buffer pool and sort heap allocations for Db2une training runs with respect to two TPC-DS queries, $\mathbf{Q}_{48}$ and $\mathbf{Q}_{80}$. In Figure 8a, we illustrate a use case of our reward function. The contour plot shows the estimated costs of a given query over buffer pool and sort heap settings. The higher cost regions are shown in the bottom-left in yellow-green, and the lowest cost "plateau" in the upper-right in dark blue. (Figure 8b is discussed shortly below.) The buffer pool and sort heap settings are measured in pages of memory. The red dot indicates Db2une's recommended tuning for this query when trained without back pressure. The blue dot represents the recommendation from the same model after undergoing additional training with back pressure implemented. Notably, this recommendation remains within the lowest cost plateau, but uses significantly fewer memory pages as per the buffer pool setting, thus achieving optimal cost efficiency with reduced resource allocation.

Back Pressure Variations. We consider three approaches for applying the back pressure in the reward function. First, we consider Db2une *without* any back pressure. Second, we apply *split* back pressure by conducting one training pass without back pressure, followed by another pass with it. Lastly, we implement *full* back pressure by incorporating it into the reward for the entire training.

With split back pressure, due to the large, complex search space, we initially guide the agent's learning using only the performance reward. This allows it to converge on the best knob settings, regardless of resources. After this, we enable the resource reward, r_{res}, weighted by the parameter δ, alongside the performance reward, r_{perf}, as per Equation 5, to encourage the actor-agent to prefer knob configurations that result in high performance, but while using as few resources as possible. Based on our experimental evaluation (Section 5.3), we observed that split back pressure offers the best trade-off between efficiency and resource allocation.

4.4 The Training Process

Algorithm 2 pseudocode details the training process for Db2une's PPO model. Multiple passes of N *episodes* are conducted over each query in the target workload (or until convergence, if sooner). Each training episode executes with t steps in the IBM Db2 environment

Algorithm 2 Train-PPO

1: **Input:** policy parameters θ and value function parameters ϕ
2: **while** $i \leq N$ **and not** *converged* **do**
3:     **for** $t \in \{1, \ldots, T\}$ **do**
4:         $D_t \leftarrow (s_t, a_t, r_t, \log \pi_{\theta_{old}}(a_t|s_t), V(s_t))$
5:     **end for**
6:     **for** $j \in \{1, \ldots, J\}$ **do**
7:         $loss(\theta) \leftarrow \widehat{\mathbb{E}}[\min(f_t(\theta)\widehat{A}_t, \text{clip}(f_t(\theta), 1-\epsilon, 1+\epsilon)\widehat{A}_t)]$
8:         $loss(\phi) \leftarrow \sum_{t \in T}(V_\phi(s_t) - R_t)^2$
9:         $\theta \leftarrow \theta - \alpha \cdot \nabla loss(\theta)$
10:         $\phi \leftarrow \phi - \beta \cdot \nabla loss(\phi)$
11:     **end for**
12:     $\theta_{old} \leftarrow \theta, i \leftarrow i + 1$
13: **end while**

(*Lines* 3–5). An episode samples multiple knob settings based on the current policy, and evaluates their effect on workload queries. IBM Db2's optimizer processes a query from the target workload, generating a QEP under the default knob settings. QBERT embeds this QEP, and the PPO agent predicts knob settings. These settings are then applied to the database system, a new QEP is generated under them, and this is used to compute the performance and back pressure rewards for the predicted knob settings.

At each step t, we store in a buffer D_t the state, action, reward, probability of the policy taking the action, and V-value (*Line* 4). After calculating the advantage at the episode's end, we update the actor and critic networks for J steps (*Lines* 6–11) using stochastic gradient descent over their objective functions (*Lines* 7–8), where R_t is the sum of discounted intermediate rewards.

In Figure 8b, we show an example use case of tuning two knobs, buffer pool and sort heap. The contour plot depicts the estimated cost of a query at various settings, with the lowest cost represented in dark blue. The dots, ranging from black to yellow to finally red, highlight recommended settings during training. This illustrates the system's exploration process, ultimately recommending a setting in the lowest cost region.

5 EXPERIMENTAL VALIDATION

We conduct an experimental validation of Db2une, showcasing its efficiency, and effectiveness. The experiments were run on a machine with an Intel i7-8565U CPU and 32GB of RAM and a RTX3070 GPU, running IBM Db2 Enterprise Edition. We evaluate Db2une over the standard TPC-DS [31] and TPC-H [34] benchmarks, and over an IBM client-inspired analytical workload, referred to as IBM Client, that includes common OLAP operations like window functions, union operators, and complex nested joins and subqueries. Each database is column-organized and sized at 100GB (column compressed to 36GB), unless stated otherwise. We cap memory usage of DB2 to 2GB. For each workload, we measure the execution times of test queries that are unseen during training. Query embedding models (i.e., QBERT, QEP2Vec) are trained once on GPU and used across all experiments, while the tuning agents are re-trained on CPU across different workloads.

5.1 Query and Workload Level Tuning

We evaluate the Db2une system against BLUTune [19, 20], which uses QEP2Vec embeddings, and against QTune [24], which uses

`Featurization` (denoted herein as `QTune-F`) over the TPC-DS, TPC-H, and IBM Client workloads. To demonstrate `Db2une`'s transfer-learning effectiveness, we evaluate `Db2une` against `BLUTune` and `QTune` where each system has been trained over either the TPC-DS or TPC-H workload, then tested over the TPC-H or IBM Client workload, respectively.

Exp-1: Query Level. We first evaluate `Db2une`'s efficiency tuning on each query. As illustrated in Figure 9, `Db2une`'s knob configurations result in significant performance gains over the TPC-DS, TPC-H, and IBM Client workloads, achieving a 23.3% to 50.1% reduction in execution time over `BLUTune`, and a 30.9% to 60.4% reduction over `QTune-F`. To demonstrate `Db2une`'s effectiveness to re-use learned tuning models, we train the systems on TPC-DS and then test them on TPC-H (TPC-DS → TPC-H), and train on TPC-H and test on IBM Client (TPC-H → IBM Client). `Db2une` shows a 32.3% and 48.2% execution-time reduction compared against `BLUTune`, and a 44.1% and 51.0% reduction compared against `QTune-F`, over TPC-DS → TPC-H and TPC-H → IBM Client, respectively.

Exp-2: Workload Level. We next evaluate how the `Db2une` system tunes once per workload. As shown in Figure 10, `Db2une` again outperforms substantially `BLUTune` and `QTune-F` over the TPC-DS, TPC-H, and IBM Client workloads, achieving a 25.0% to 48.5% reduction in execution time over `BLUTune`, and a 31.2% to 56.3% reduction over `QTune-F`. `Db2une` shows a 33.2% and 47.2% execution-time reduction compared against `BLUTune`, and a 43.1% and 50.1% reduction compared against `QTune-F`, over TPC-DS → TPC-H and TPC-H → IBM Client, respectively.

5.2 Multi-phased & Meta-Data Training

Exp-3: Performance Metrics. We next evaluate the effectiveness of multi-phased training for knob tuning, considering different combinations of our training metrics (Section 4.2). A model is either fully trained using a single performance metric (e.g., interpolated cost or estimated cost), *or pre-trained* with one performance metric during a first training pass, and then *fine-tuned* by a more precise metric during a second training pass. For the latter, we pre-train on interpolation and fine-tune on estimated cost, or pre-train on estimated cost and fine-tune on runtime. When fine-tuning on runtime, we implement timeouts to prevent excessively long runs caused by poor initial knob settings. For pre-training with interpolation, we train for buffer pool, sort heap, and optimization level (with the remaining knobs set at their default values), as these have the most significant impact on cost estimates and runtime.

Table 1 reports the cumulative times for training and testing `Db2une` over the TPC-DS, TPC-H and IBM Client workloads.[2] For instance, for TPC-DS, compared against training on interpolation (*interpolation*/−), pre-training on interpolation then fine-tuning on cost (*interpolation*/*cost*), yields a 3% improvement in testing time, at the cost of an 81% increase in training time. Compared against the *interpolation*/*cost* model, *cost*/− yields a further 6% improvement in testing time, at an additional cost of a 44% increase in training time. Lastly, compared against *cost*/−, *cost*/*runtime* yields yet a further 4% improvement in testing time, but at an additional cost of a 94% increase in training time. Note that the reported training times for `Db2une` are on average 40% faster than those for `BLUTune` and `QTune`, due primarily to faster convergence during training.

[2]To construct our interpolation function via sampling took on average 30 minutes. This is an *off-line* cost, as it is done once, when the data is sampled. This is reused over any number of training runs. Thus, this off-line cost is not included in the table.

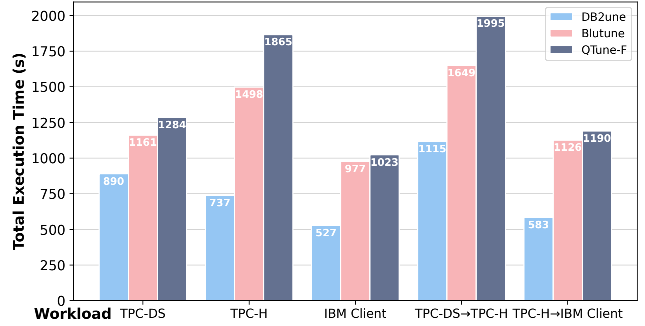


Figure 9: Query level tuning.

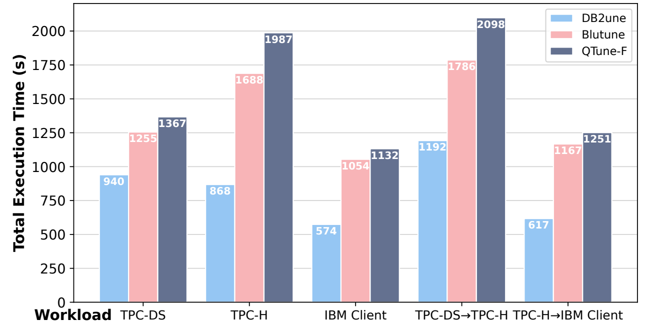


Figure 10: Workload level tuning.

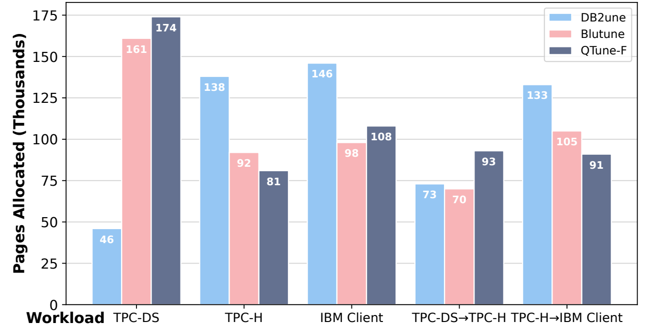


Figure 11: Memory allocation of tuning methods.

We do not report the results of training fully on runtime as this is *extremely* time-consuming. Many of our training runs had to be aborted because the initially untrained model would recommend poor knob parameters, which resulted in queries that would run for hours. We conclude that training solely on runtime is simply not practical for large workloads. We observed similar trends over the TPC-H and IBM Client workloads as reported in Table 1. Our experiments confirm that we can achieve significant time savings during training while not significantly sacrificing model performance. While cost estimates offer a favorable trade-off between training time and testing time, it is ultimately up to the administrator to decide which method is most suitable for their specific needs, considering whether investing the additional training time needed by the more accurate performance metrics is worthwhile for the corresponding performance gains.

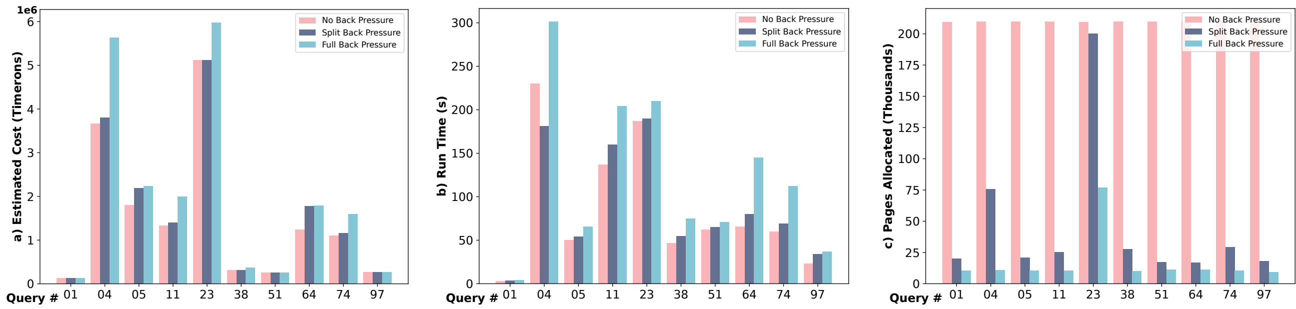


Figure 12: Comparison of Db2une with and without back pressure over test queries.

Table 1: Times with training metrics and fine tuning.

pre-training: fine tuning:	interp. –	interp. cost	cost –	cost runtime
training time (TPC-DS)	19m	1h 41m	3h	48h
testing time (TPC-DS)	978s	946s	890s	856s
training time (TPC-H)	8m	54m	1h 46m	30h
testing time (TPC-H)	887s	833s	737s	692s
training time (IBM Client)	13m	1h 12m	1h 57m	42h
testing time (IBM Client)	636s	542s	527s	495s

Exp-4: Tuning with Database's Meta-data. As we have shown that we can effectively train via cost estimations, this effectively means we do not need the database itself but just the database's meta-data—schema, constraints, and data statistics—in order to train! To demonstrate, we used a 3TB TPC-DS database, partitioned by the IBM team across eight nodes, to simulate a production environment. Using `db2look`, we extracted the DDL and `update statistics` statements to replicate the database objects and statistics, but not the data, in our test system. We used the `db2fopt` command to adjust memory-related knobs, such as buffer pools, to beyond the actual resource limits available in our test environment. Thus, the test system mimics the resources of the production system.

Training `Db2une` in our test environment for the 3TB TPC-DS workload took 4 hours and 47 minutes. This training time is quite reasonable in light of the performance improvements the model conferred. Applying the model's recommended configurations, we observed up to 18.5% improvement over the individual queries, with an average improvement of 5%, with *none* more expensive than via the tuning configurations recommended by IBM experts. In addition, `Db2une`'s tuning allocates 130 thousand *fewer* pages in total over the buffer pool and sort heap. Interestingly, it allocated 376 thousand more pages for sort heap but 506 thousand fewer for buffer pool than the IBM experts had. This surprised the IBM team, as this ratio between sort heap and buffer pool differed from their heuristic rules based upon past tuning experiences with other workloads. Furthermore, this tuning task was achieved automatically, alleviating the human burden to tune manually, which IBM experts report can take hours, even days.

5.3 Back-Pressure Evaluation

Exp-5: Ways To Apply Back Pressure. To study the inclusion of back pressure into the reward function, we evaluate three approaches over the TPC-DS workload: with *no*, *split*, and *full* back pressure. We measure the changes in the estimated cost, in query execution time, and in pages of memory allocated. Figure 12a plots the estimated cost of each test query as provided by the `IBM Db2` optimizer at `Db2une`'s recommended settings. Figure 12b shows the actual runtime for each. The no back pressure model achieves the lowest estimated cost for each query. Split achieves a close second-best performance, with nearly identical results on five of the ten test queries, and only slight increases for the rest. Full is not as good, showing either similar or worse results than the split model on each query. These results are reflected in the runtimes. (Note that $\mathbf{Q}_4$ is an outlier in that the split model outperforms the full.) Figure 12c shows the pages of memory allocated by each model, revealing a significant memory-resource reduction for both the split- and full-back pressure models, just as anticipated.

Table 2: Summary of back-pressure models.

	no BP	split BP	full BP
estisimated cost	15.2M	16.4M	20.3M
avg. total runtime	864s	891s	1, 226s
change in est. cost & runtime	–	7% & 3%	25% & 42%
avg. change in memory	–	78%	91.8%

Table 2 summarizes the models' performance. Split shows a slight 3% increase in total runtime (and 7% in estimated cost), in exchange for a significant 78% average reduction in memory allocation per query! Full demonstrates a worse 42% increase in runtime (and 25% in estimated cost), but with a 92% reduction in memory allocation. We surmise that full performs worse than split as an early application of back pressure may discourage exploration of higher-performing settings. As our goal is to avoid significant increases in runtime while economizing on resource allocation, we advocate for the split back-pressure approach.

Exp-6: Performance with and without Back Pressure. We now compare `Db2une` (thus, with split back pressure) against `Db2une` but with no back pressure, tuning per query and tuning per workload. We aggregate over the workloads for each of the following. For query tuning, the increase in execution times is 0.7%, 3%, and 6% for the TPC-H, TPC-DS and IBM Client workloads, respectively. For workload tuning, it is 5.9%, 7.7%, and 9% for the TPC-DS, TPC-H and IBM Client workloads, respectively. Thus, loss in performance is minimal. Meanwhile, back pressure dramatically decreases the allocated memory. For query tuning, this decrease is 78%, 15.8%, and 22% and for workload tuning, this decrease is 78%, 34%, and 30% for the TPC-DS, TPC-H and IBM Client workloads, respectively.

Exp-7: Reduction in Resource Allocation. In Figure 11, we compare the resource allocation, specifically for the number of

pages allocated by Db2une's recommended tuning, against those of BLUTune's and QTune-F's for tuning by workload. Over TPC-DS, Db2une allocates 71.4% fewer resources than BLUTune does, and 73.5% fewer than QTune-F. Db2une finds that a large buffer pool is unnecessary, as long as there is sufficient sort heap. Thus it allocates less memory for the buffer pool than do BLUTune and QTune-F. Over IBM Client, Db2une finds the opposite: allocating more memory for a larger buffer pool is better. It uses 48.9% more resources than BLUTune does, and 35.2% more than QTune-F. Over TPC-H, Db2une uses 50% more resources than BLUTune, and 70% more than QTune-F. One reason Db2une may identify that these workloads require a larger buffer pool is due to their large fact tables. For example, the LINE_ITEM fact table in TPC-H at 100GB contains about 600 million rows, which is about twice as large as the STORE_SALES table, the largest table in TPC-DS. With the smaller size of tables on TPC-DS, a smaller buffer pool was sufficient, as long as the sort heap was large enough. Over TPC-H, however, since the queries were, on average, accessing a larger fact table, the buffer pool is more critical. Over TPC-DS → TPC-H, Db2une uses 4.1% more memory resources than BLUTune does, but 21.5% *less* than QTune-F. Finally, over TPC-H → IBM Client, Db2une uses 26.7% more resources than BLUTune does, and 46.2% more than QTune-F. Thus, Db2une's resource usage in transfer-learning scenarios varies, based on workload-specific needs, demonstrating its adaptability.

6 RELATED WORK

Many systems for automatic database system knob tuning using machine learning have been developed over the last several years; e.g., [1, 5, 16, 24, 38, 41, 43, 44]. In this section, we focus on the ones most relevant to the Db2une system.

OtterTune [1, 2] identifies and ranks the knobs with the strongest impact on performance by applying Lasso, maps the given workload to a repository of previously collected performance measurements, and employs Bayesian optimization using a Gaussian process to choose knob settings. This approach produces effective recommendations, however, requires a large number of costly training samples. Bayesian optimization, while shown as effective for knob tuning, is generally less effective in large search spaces [15, 28], which may limit its use for tuning complex analytical workloads.

CDBTune [43] introduces DRL as a method for automatic knob tuning. It uses a deep deterministic policy gradient model (DDPG), an off-policy-based learning method. This approach relies on a trial-and-error method for learning knob settings, which improves upon existing work by alleviating the need for a large number of expensive high-quality training samples. However, CDBTune is limited in that it can provide only coarse-grained tunings (i.e., for specific workload types, such as read-only). It is also not query-aware. It only reacts to the anticipated change to data system health metrics, such as counters for data reads and lock timeouts.

Gur et al. [17] propose a multi-model tuning solution that employs DDPG to address deployments with varying workloads. The authors use IBM Db2 monitoring to track the memory allocated. Their approach is particularly suited for handling a variety of OLTP workloads with varying patterns. However, in scenarios with numerous distinct workload patterns, the training and maintenance of these models can be quite costly.

QTune [24] extends on CDBTune to be query-informed by using a double-state DDPG (DS-DDPG) method. By performing a featurization of queries—accounting for factors such as table and attribute involvement and aggregated costs—QTune converts queries into vectors. These vectors serve as the initial state and input to a predictor model, which is trained to forecast changes in the internal database system state. This predicted state then serves as the input to the second model, which in turn predicts knob settings. This approach falls short in capturing the structure of queries and their cardinalities. Thus, it does not generalize effectively to new queries and workloads that involve different table names and attributes.

BLUTune [19] is also a query-aware approach, leveraging learned representations of query plans as input states for its DRL model. This method employs a QEP2Vec component to capture a more comprehensive range of query information within the learned embeddings than does QTune's featurization technique. However, QEP2Vec's use of a bag-of-operators approach, similar to the bag-of-words technique in Doc2Vec, fails to capture the complex structural details of query plans. Moreover, QEP2Vec embeddings exhibit non-determinism, which introduces challenges for the DRL agent due to inconsistent state representations across episodes. This complicates the learning process, affecting the tuning system's effectiveness.

Query representation methods have been proposed for various optimization tasks beyond database tuning. SQLBERT combines graph neural networks and transformers to generate rich SQL embeddings for tasks like cardinality estimation [37]. Unlike SQLBERT, which focuses on encoding SQL queries for different optimization tasks, QBERT encodes query plans to capture stats related to physical operators for effective database tuning. QPSeeker uses an LSTM architecture to encode query plans, capturing the interactions of physical operators over the tables and the data types and distributions to select effective query plans [39]. Tree-LSTM architectures have also been used to encode query plans for cost estimation [35], encoding information such as physical query operations, predicates, columns, tables, indexes, and tuples. In contrast to these methods, QBERT anonymizes table attributes, metadata, and tuples for schema-independent tuning.

7 CONCLUSIONS

Db2une has been well received within IBM, and is proving to be a valuable tool both for company support and in database development, as our novel system effectively addresses the problem of automatic knob tuning for IBM Db2. Our system is query-aware via QBERT embeddings, converges using stable deep learning approaches and scales well for complex and large workloads using multi-phased and database meta-data driven training, while encouraging resource-conscious tuning. Our experimental evaluations have demonstrated significant improvements by Db2une over previous state-of-the-art query-aware tuning systems and IBM experts. In future work, we plan to incorporate distributed computing, as in our prior work on query problem determination [27], to knob tuning to improve further the efficiency.

ACKNOWLEDGMENTS

This research was undertaken thanks in part to funding from the Canada First Research Excellence Fund and IBM CAS.

REFERENCES

[1] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. ACM, 1009–1024. https://doi.org/10.1145/3035918.3064029

[2] Dana Van Aken, Dongsheng Yang, Sebastien Brillard, Ari Fiorino, Bohan Zhang, Christian Billian, and Andrew Pavlo. 2021. An Inquiry into Machine Learning-based Automatic Configuration Tuning Services on Real-World Database Management Systems. *Proc. VLDB Endow.* 14, 7 (2021), 1241–1253. https://doi.org/10.14778/3450980.3450992

[3] Oron Anschel, Nir Baram, and Nahum Shimkin. 2017. Averaged-DQN: Variance Reduction and Stabilization for Deep Reinforcement Learning. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017*, Vol. 70. PMLR, 176–185. http://proceedings.mlr.press/v70/anschel17a.html

[4] Peter Belknap, Benoît Dageville, Karl Dias, and Khaled Yagoub. 2009. Self-Tuning for SQL Performance in Oracle Database 11g. In *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009*. IEEE Computer Society, 1694–1700. https://doi.org/10.1109/ICDE.2009.165

[5] Baoqing Cai, Yu Liu, Ce Zhang, Guangyu Zhang, Ke Zhou, Li Liu, Chunhua Li, Bin Cheng, Jie Yang, and Jiashu Xing. 2022. HUNTER: An Online Cloud Database Hybrid Tuning System for Personalized Requirements. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. ACM, 646–659. https://doi.org/10.1145/3514221.3517882

[6] Surajit Chaudhuri and Vivek R. Narasayya. 1998. AutoAdmin 'What-if' Index Analysis Utility. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data (SIGMOD '98)*. ACM Press, 367–378. https://doi.org/10.1145/276304.276337

[7] Guilherme Damasio, Vincent Corvinelli, Parke Godfrey, Piotr Mierzejewski, Alexandar Mihaylov, Jaroslaw Szlichta, and Calisto Zuzarte. 2019. Guided automated learning for query workload re-optimization. *Proc. VLDB Endow.* 12, 12 (2019), 2010–2021. https://doi.org/10.14778/3352063.3352120

[8] Sudipto Das, Miroslav Grbic, Igor Ilic, Isidora Jovandic, Andrija Jovanovic, Vivek R. Narasayya, Miodrag Radulovic, Maja Stikic, Gaoxiang Xu, and Surajit Chaudhuri. 2019. Automatically Indexing Millions of Databases in Microsoft Azure SQL Database. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. ACM, 666–679. https://doi.org/10.1145/3299869.3314035

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019*. Association for Computational Linguistics, 4171–4186. https://doi.org/10.18653/V1/N19-1423

[10] Bailu Ding, Sudipto Das, Wentao Wu, Surajit Chaudhuri, and Vivek R. Narasayya. 2018. Plan Stitch: Harnessing the Best of Many Plans. *Proc. VLDB Endow.* 11, 10 (2018), 1123–1136. https://doi.org/10.14778/3231751.3231761

[11] IBM Db2 Documentation. 2022. Configuration parameters summary. https://www.ibm.com/docs/en/db2/11.5?topic=parameters-configuration-summary Accessed: 2023-12-07.

[12] IBM Db2 Documentation. 2022. Db2 registry and environment variables. https://www.ibm.com/docs/en/db2/11.5?topic=variables-registry-environment Accessed: 2023-12-07.

[13] IBM Db2 Documentation. 2024. Explain information for data operators. https://www.ibm.com/docs/en/db2/11.5?topic=information-explain-data-operators Accessed: 2024-07-10.

[14] Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. 2016. Benchmarking Deep Reinforcement Learning for Continuous Control. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016*, Vol. 48. JMLR.org, 1329–1338. http://proceedings.mlr.press/v48/duan16.html

[15] David Eriksson and Martin Jankowiak. 2021. High-dimensional Bayesian optimization with sparse axis-aligned subspaces. In *Proceedings of the Thirty-Seventh Conference on Uncertainty in Artificial Intelligence, UAI 2021 (Proceedings of Machine Learning Research)*, Vol. 161. AUAI Press, 493–503. https://proceedings.mlr.press/v161/eriksson21a.html

[16] Jia-Ke Ge, Yanfeng Chai, and Yunpeng Chai. 2021. WATuning: A Workload-Aware Tuning System with Attention-Based Deep Reinforcement Learning. *J. Comput. Sci. Technol.* 36, 4 (2021), 741–761. https://doi.org/10.1007/S11390-021-1350-8

[17] Yaniv Gur, Dongsheng Yang, Frederik Stalschus, and Berthold Reinwald. 2021. Adaptive Multi-Model Reinforcement Learning for Online Database Tuning. In *Proceedings of the 24th International Conference on Extending Database Technology, EDBT 2021*. OpenProceedings.org, 439–444. https://doi.org/10.5441/002/EDBT.2021.48

[18] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. 2018. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018*, Vol. 80. PMLR, 1856–1865. http://proceedings.mlr.press/v80/haarnoja18b.html

[19] Connor Henderson, Spencer Bryson, Vincent Corvinelli, Parke Godfrey, Piotr Mierzejewski, Jaroslaw Szlichta, and Calisto Zuzarte. 2022. BLUTune: Query-informed Multi-stage IBM Db2 Tuning via ML. In *Proceedings of the 31st ACM International Conference on Information and Knowledge Management (CIKM '22)*. 3162—-3171. https://doi.org/10.1145/3511808.3557117

[20] Connor Henderson, Vincent Corvinelli, Parke Godfrey, Piotr Mierzejewski, Jaroslaw Szlichta, and Calisto Zuzarte. 2023. BLUTune: Tuning Up IBM Db2 with ML. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 3615–3618. https://doi.org/10.1109/ICDE55515.2023.00281

[21] Shrainik Jain and Bill Howe. 2018. Query2Vec: NLP Meets Databases for Generalized Workload Analytics. *CoRR* abs/1801.05613 (2018). arXiv:1801.05613 http://arxiv.org/abs/1801.05613

[22] Quoc V. Le and Tomás Mikolov. 2014. Distributed Representations of Sentences and Documents. In *Proceedings of the 31th International Conference on Machine Learning, ICML 2014*, Vol. 32. 1188–1196. http://proceedings.mlr.press/v32/le14.html

[23] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (2015), 204–215. https://doi.org/10.14778/2850583.2850594

[24] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. QTune: A Query-Aware Database Tuning System with Deep Reinforcement Learning. *Proc. VLDB Endow.* 12, 12 (2019), 2118–2130. https://doi.org/10.14778/3352063.3352129

[25] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2016. Continuous control with deep reinforcement learning. In *International Conference on Learning Representations, ICLR 2016*. http://arxiv.org/abs/1509.02971

[26] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *CoRR* abs/1907.11692 (2019). arXiv:1907.11692 http://arxiv.org/abs/1907.11692

[27] Alexandar Mihaylov, Vincent Corvinelli, Parke Godfrey, Piotr Mierzejewski, Jaroslaw Szlichta, and Calisto Zuzarte. 2021. Scalable Learning to Troubleshoot Query Performance Problems. In *Proceedings of the 30th ACM International Conference on Information and Knowledge Management (CIKM '21)*. ACM, 4016–4025. https://doi.org/10.1145/3459637.3481947

[28] Riccardo Moriconi, Marc Peter Deisenroth, and K. S. Sesh Kumar. 2020. High-dimensional Bayesian optimization using low-dimensional feature spaces. *Mach. Learn.* 109, 9-10 (2020), 1925–1943. https://doi.org/10.1007/S10994-020-05899-Z

[29] Parimarjan Negi, Ziniu Wu, Andreas Kipf, Nesime Tatbul, Ryan Marcus, Sam Madden, Tim Kraska, and Mohammad Alizadeh. 2023. Robust Query Driven Cardinality Estimation under Changing Workloads. *Proc. VLDB Endow.* 16, 6 (2023), 1520–1533. https://doi.org/10.14778/3583140.3583164

[30] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F. Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems, NeurIPS 2022*. http://papers.nips.cc/paper_files/paper/2022/hash/b1efde53be364a73914f58805a001731-Abstract-Conference.html

[31] Meikel Pöss, Raghunath Othayoth Nambiar, and David Walrath. 2007. Why You Should Run TPC-DS: A Workload Analysis. In *Proceedings of the 33rd International Conference on Very Large Data Bases, VLDB 2007*. ACM, 1138–1149. http://www.vldb.org/conf/2007/papers/industrial/p1138-poess.pdf

[32] John Schulman, Sergey Levine, Pieter Abbeel, Michael I. Jordan, and Philipp Moritz. 2015. Trust Region Policy Optimization. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015*, Vol. 37. 1889–1897. http://proceedings.mlr.press/v37/schulman15.html

[33] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. *CoRR* abs/1707.06347 (2017). arXiv:1707.06347 http://arxiv.org/abs/1707.06347

[34] Alkis Simitsis, Panos Vassiliadis, Umeshwar Dayal, Anastasios Karagiannis, and Vasiliki Tziovara. 2009. Benchmarking ETL Workflows. In *Performance Evaluation and Benchmarking, First TPC Technology Conference, TPCTC 2009*, Vol. 5895. Springer, 199–220. https://doi.org/10.1007/978-3-642-10424-4_15

[35] Ji Sun and Guoliang Li. 2019. An End-to-End Learning-based Cost Estimator. *Proc. VLDB Endow.* 13, 3 (2019), 307–319. https://doi.org/10.14778/3368289.3368296

[36] Waikeat Tan, Mohammed Alhamid, Mohamad Kalil, Ronghao Yang, Vincent Corvinelli, Calisto Zuzarte, and Liam Finnie. 2021. Query predicate selectivity using machine learning in Db2®. In *Proceedings of the 31st Annual International Conference on Computer Science and Software Engineering (CASCON '21)*. ACM, 143–152. https://doi.org/10.5555/3507788.3507808

[37] Xiu Tang, Sai Wu, Mingli Song, Shanshan Ying, Feifei Li, and Gang Chen. 2022. PreQR: Pre-training Representation for SQL Understanding. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. ACM, 204–216. https://doi.org/10.1145/3514221.3517878

[38] Immanuel Trummer. 2022. DB-BERT: A Database Tuning Tool that "Reads the Manual". In *Proceedings of the 2022 International Conference on Management of*

Data (SIGMOD '22). ACM, 190–203. https://doi.org/10.1145/3514221.3517843

[39] Christos Tsapelas and Georgia Koutrika. 2024. QPSeeker: An Efficient Neural Planner combining both data and queries through Variational Inference. In *Proceedings 27th International Conference on Extending Database Technology, EDBT 2024*. OpenProceedings.org, 307–319. https://doi.org/10.48786/EDBT.2024.27

[40] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems, NeurIPS 2017*. 5998–6008. https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[41] Junxiong Wang, Immanuel Trummer, and Debabrota Basu. 2021. UDO: Universal Database Optimization using Reinforcement Learning. *Proc. VLDB Endow.* 14, 13 (2021), 3402–3414. https://doi.org/10.14778/3484224.3484236

[42] Ziyu Wang, Victor Bapst, Nicolas Heess, Volodymyr Mnih, Rémi Munos, Koray Kavukcuoglu, and Nando de Freitas. 2017. Sample Efficient Actor-Critic with Experience Replay. In *International Conference on Learning Representations, ICLR 2017*. OpenReview.net. https://openreview.net/forum?id=HyM25Mqel

[43] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. 2019. An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. ACM, 415–432. https://doi.org/10.1145/3299869.3300085

[44] Xinyi Zhang, Hong Wu, Zhuo Chang, Shuowei Jin, Jian Tan, Feifei Li, Tieying Zhang, and Bin Cui. 2021. ResTune: Resource Oriented Tuning Boosted by Meta-Learning for Cloud Databases. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21)*. ACM, 2102–2114. https://doi.org/10.1145/3448016.3457291

[45] Xinyang Zhao, Xuanhe Zhou, and Guoliang Li. 2023. Automatic Database Knob Tuning: A Survey. *IEEE Trans. Knowl. Data Eng.* 35, 12 (2023), 12470–12490. https://doi.org/10.1109/TKDE.2023.3266893