Tuning Big Data Systems via Deep Learning

Alexander Bianchi

A thesis submitted to the
Faculty of Graduate Studies in partial
fulfillment of the requirements for the degree of

Master of Science

in

Graduate Program in Computer Science

York University
Toronto, Ontario
April 2025

© Alexander Bianchi, 2025

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. Db2une uses a specialized transformer-based query-embedding pipeline and graph neural networks to feed as input to a stability-oriented 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 scales to large workloads where executing queries repeatedly would be prohibitively expensive. Through experimental evaluation, we demonstrate Db2une’s efficiency and effectiveness over a variety of workloads. We show that Db2une provides recommendations surpassing those of other state-of-the-art systems and IBM experts.

Dedication

I dedicate this thesis to my family, whose constant support and encouragement have been the foundation of my academic journey.

I would like to express my appreciation to my supervisors, Dr. Jarek Szlichta and Dr. Parke Godfrey. Dr. Szlichta's guidance and support have shaped a lot of my academic and personal growth, and his mentorship, along with Dr. Godfrey's, has been instrumental to my success. I thank them both for passing on their love of learning to me.

I am thankful to my IBM colleagues, Vincent Corvinelli and Calisto Zuzarte, for generously sharing their expertise and deepening my understanding of data systems.

Finally, I thank my friends and lab mates Andrew Chai, Justin So, and Rafael Dolores. I'll fondly remember the time we spent as graduate students, it has been a great pleasure to work and grow alongside you.

Acknowledgements

I would like to acknowledge Andrew Chai and Rafael Dolores of York University for their contribution towards the `Db2une` system. Andrew contributed the design and implementation of the interpolated cost reward function described in Section 2.3.2 and the corresponding experiments in Section 4.2. He also helped with the back pressure evaluation in Section 4.3. Rafael assisted in the design of the `Db2une` interface as described in Section 3.1.

Contents

Abstract ii

Dedication iii

Acknowledgements iv

Table of Contents ix

List of Figures ix

List of Abbreviations and Symbols x

1 Introduction 1

1.1 Motivation . . . 1

1.2 Knob Effects Example . . . 3

1.3 Contributions . . . 5

2 System Overview 10

2.1 Transformer-Based Embedding Strategy . . . 12

2.1.1 The Templating Process . . . 14

2.1.2 QBERT Embeddings . . . 16

2.2 Graph Contrastive Learning Embeddings 22
2.3 Automatic Tuning System 27
2.3.1 Stability-Oriented, On Policy Tuning 28
2.3.2 Multi-phased & Meta-Data Driven Training 32
2.3.3 Performance and Back Pressure Reward 35
2.3.4 The Training Process 40

3 System Demonstration 42
3.1 Demonstration of Knob Tuning Interface 43
3.2 Query Aware Tuning and Transfer Learning 45

4 Experimental Evaluation 47
4.1 Query and Workload Level Tuning 47
4.2 Multi-phased & Meta-Data Training 49
4.3 Back-Pressure Evaluation 53
4.4 Embedding Quality 56

5 Related Work 61

6 Conclusion 64
6.1 Conclusion 64
6.2 Future Work 65

Copyright 67

List of Figures

1.1 Cost estimates of $\mathbf{Q}_{23}$ at varied settings. 4

1.2 QEPs for $\mathbf{Q}_{16}$ for different values of knobs. 5

2.1 Architectural overview of `Db2une`. 11

2.2 A query plan for $\mathbf{Q}_{97}$ representation via `QBERT`. 13

2.3 The `QBERT` pipeline. 18

2.4 A Generalization of The GCL Model for Embedding Query Plans . . 26

2.5 Tuning agent based on advantage actor-critic. 31

2.6 Example of interpolation for $\mathbf{Q}_{64}$. 34

2.7 `Db2une` setting knobs. 39

3.1 Architecture Overview of The `Db2une` Web System © IEEE 2025 . . 43

3.2 `Db2une` Training Screen © IEEE 2025 44

3.3 `Db2une` Tuning Screen © IEEE 2025 45

4.1 Query level tuning. 50

4.2 Workload level tuning. 50

4.3 Memory allocation of tuning methods. 51

4.4 Comparison of `Db2une` with and without back pressure over test queries. 54

4.5 The QEPs for $\mathbf{Q}_7$, $\mathbf{Q}_{79}$, $\mathbf{Q}_6$, and $\mathbf{Q}_{25}$ in the IBM expert case study. . 60

List of Tables

4.1 Times with training metrics and fine tuning. 52
4.2 Summary of back-pressure models. 53
4.3 GNN and QBERT Comparison 58

List of Abbreviations and Symbols

SQL: .... Structured Query Language

DDL: .... Data Definition Language

QEP: .... Query Execution Plan

IBM: .... International Business Machines Corporation

DBA: .... Database Administrator

GCL: .... Graph Contrastive Learning

GNN: .... Graph Neural Network

DRL: .... Deep Reinforcement Learning

PPO: .... Proximal Policy Optimization

DDPG: .... Deep Deterministic Policy Gradient

HSJOIN: .... Hash Join

MSJOIN: .... Merge Join

NLJOIN: .... Nested Loop Join

TBSCAN: .... Table Scan

Chapter 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 [1], [2]. 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 are optimal for system performance and resource usage. For `IBM Db2`, tuning knobs include *configuration parameters* [3] and *registry variables* [4]. Tuning is often performed by experts, such as *database administrators* (DBAs). However, manual tuning is 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 [3]. In addition, 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 [5]. This becomes an even greater problem

when considering the growing trend of cloud database services, where it becomes infeasible to tune tens of thousands of database instances manually.

Given these challenges, automated tuning systems have been proposed [1], [6]. These systems employ a range of machine-learning algorithms to find optimal tuning configurations for specific workloads. While these systems can equal or surpass the expertise of DBAs, they have their drawbacks. First, the systems require a ***costly training process***, since evaluating database-system performance would seem to necessitate workload execution. For analytical workloads especially, queries can have long runtimes even with a good tuning configuration, and exceedingly longer runtimes, or even failures, without. Such long run times can be expected while training the tuning system and may lead to extended database downtime. Workload execution is a problem for deep-learning-based methods that must collect a high-quality training set. To circumvent this, some approaches [6], [7] 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. A poor choice of query representation can limit the adaptability of the tuning system to new queries or schemas. At the simplest, such systems may just rely on the database statistics to inform their tuning decisions. Better, they may rely on representations of the query workload. This can be based on *query-featurization* methods or learned representations. However, these better approaches still lack adaptability for new queries or datasets. The best would be a model that could learn query representations adaptable to unseen queries in new workloads.

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 becomes critical as the customer wants the best performance but at the lowest cost. 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. Most automatic tuning solutions neglect conserving resources, seeking only to increase performance. This may be reasonable when the goal is to provision the fixed resources of a server one owns, *and* which is not shared with other services. This ceases to be reasonable, however, when the resources are shared, or must be paid for per deployment, such as for cloud computing.

1.2 Knob Effects Example

Knobs affect the performance of query evaluation by modifying the behavior of the optimizer, and by provisioning resources for database operations. In `IBM Db2`, *optimization level* specifies which query-optimization techniques will be used by the database system when composing QEPs. A QEP details the sequence of steps the database system will take to execute a query. The *buffer pool* and *sort heap* knobs set the memory allocations for those, respectively. Their sizes control resource allocation and may allow or disallow specific operator selections due to the availability of resources. Insufficient resource allocation may lead to less efficient operations or disk spilling, These settings compete, however, in that each is an allocation of main memory.

Figure 1.1 plots how varying *buffer pool* and *sort heap* at two different *optimization levels* changes cost estimates for *query* #23 ($\mathbf{Q}_{23}$) of the TPC-DS benchmark [8]. Note

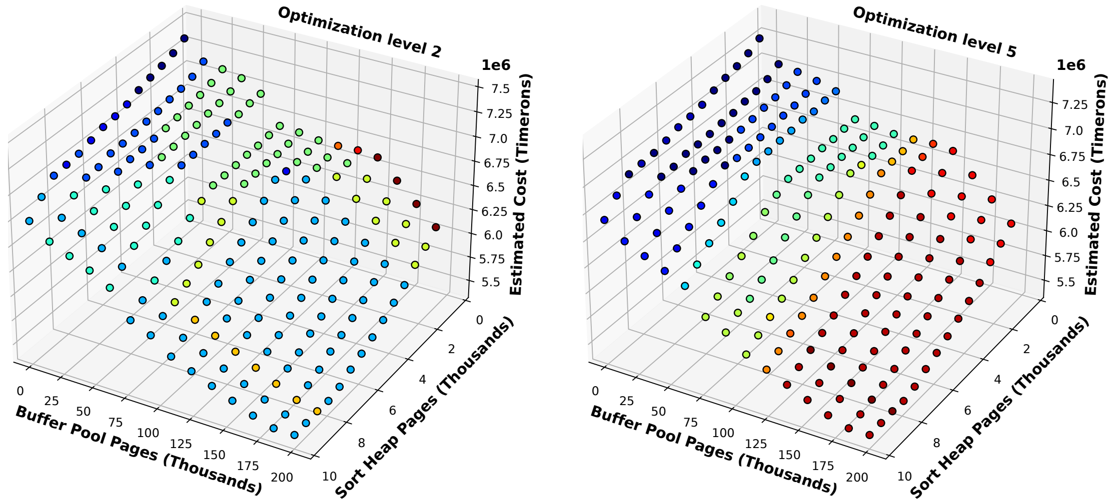


Figure 1.1: Cost estimates of $\mathbf{Q}_{23}$ at varied settings.

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 1.1 illustrates this, with the same structured plans assigned the same color. Figure 1.2 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 *Hash join* (`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,

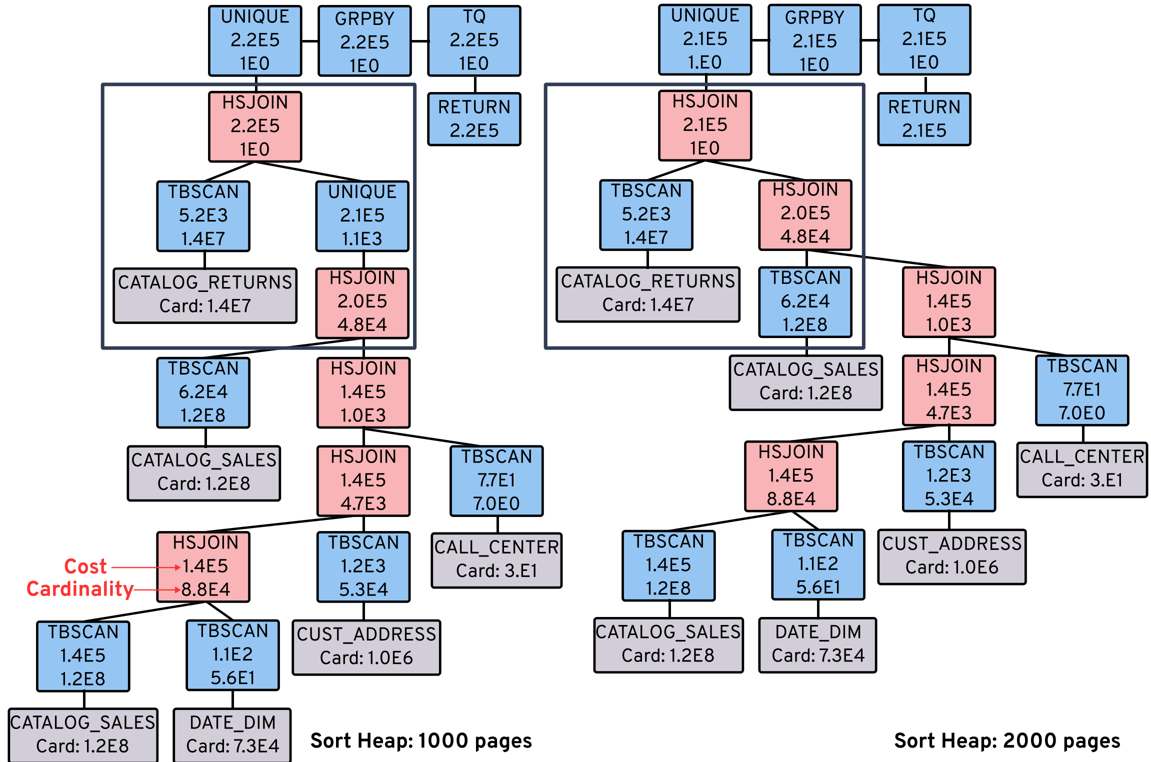


Figure 1.2: QEPs for $\mathbf{Q}_{16}$ for different values of knobs.

the `HSJOIN` likely fits in memory, and the benefit of 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. Therefore, for effective tuning 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 query-embedding pipeline via *deep learning*. The system has resulted in two publications; one for the 50th International Conference on Very Large Databases (VLDB) [9] and one to

appear in the 41st IEEE International Conference on Data Engineering (ICDE) [10]. 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 2.1 & 2.2).** We propose two novel methods for learning SQL query representations from QEPs. Our transformer-based pipeline, `QBERT`, generates fixed-length vector embeddings that go beyond prior query-aware techniques [11], [12] by encoding both structural and statistical aspects of query plans. `QBERT`'s context-aware embeddings distinguish structurally different QEPs, even when they share similar operators or statistics. This enables effective knob configuration recommendations for unseen queries. Additionally, we introduce a graph contrastive learning method that perturbs QEPs through sub-plan sampling and feature masking. A graph neural network is trained to maximize similarity between perturbed versions of the same plan while distinguishing unrelated plans, capturing structural relationships and cost-based attributes for embeddings transferable across various data system optimization tasks.

2. **A Knob-Tuning System (Section 2.3).** `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) [13], to tuning data systems. Previous DRL-based work use *off-policy* algorithms [14] that are sensitive to the quality of training data, and have been shown to have convergence issues [6]. PPO belongs to a class of *on-policy* algorithms that are more stable.

    (b) **Multi-phased & Database Meta-data Driven 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. 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 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 cost estimates and runtime metrics, while an adjustable metric for reducing resource usage—termed *back pressure*—incorporated into the reward encourages resource-effective tuning.

3. **System Demonstration (Section 3).** We create an interactive application for `Db2une` showing how we can tune data systems to boost performance, and how its query representation models aid users in understanding the query plans generated

from complex analytical workloads.

4. **Experimental Validation (Section 4).**

   (a) **Tuning System Comparisons (Section 4.1).** 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 [12], and `QTune`, which uses a query featurization approach [11]. 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 4.2)** Via an evaluation suite, we determine the trade-offs of overhead and effectiveness for training regimes using our different performance metrics. We show how effective training is with estimated and interpolated costs, paired with PPO, for very little training overhead. 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 compete with those of IBM experts.

   (c) **Back Pressure Evaluation (Section 4.3).** 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.

(d) **Embedding Evaluation (Section 4.4).** We quantitatively compare our two proposed embedding methods and discuss their tradeoffs. We also gauge the quality and superiority of our embeddings through a case study with IBM experts to compare similar query plans produced by `Db2une`'s embeddings, `BLUTune`'s `QEP2Vec`, and `QTune`'s `Featurization`.

We discuss related works in Section 5, and conclude in Section 6.

Chapter 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 query workloads. 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 2.1 overviews the architecture of Db2une. A *query workload* is defined as a collection of SQL queries executed over a specific database (its schema and 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 1.2).

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). The QEP is passed either to the QBERT pipeline or the GNN model to generate an embedding. Db2une's QBERT pipeline extracts critical tuning information

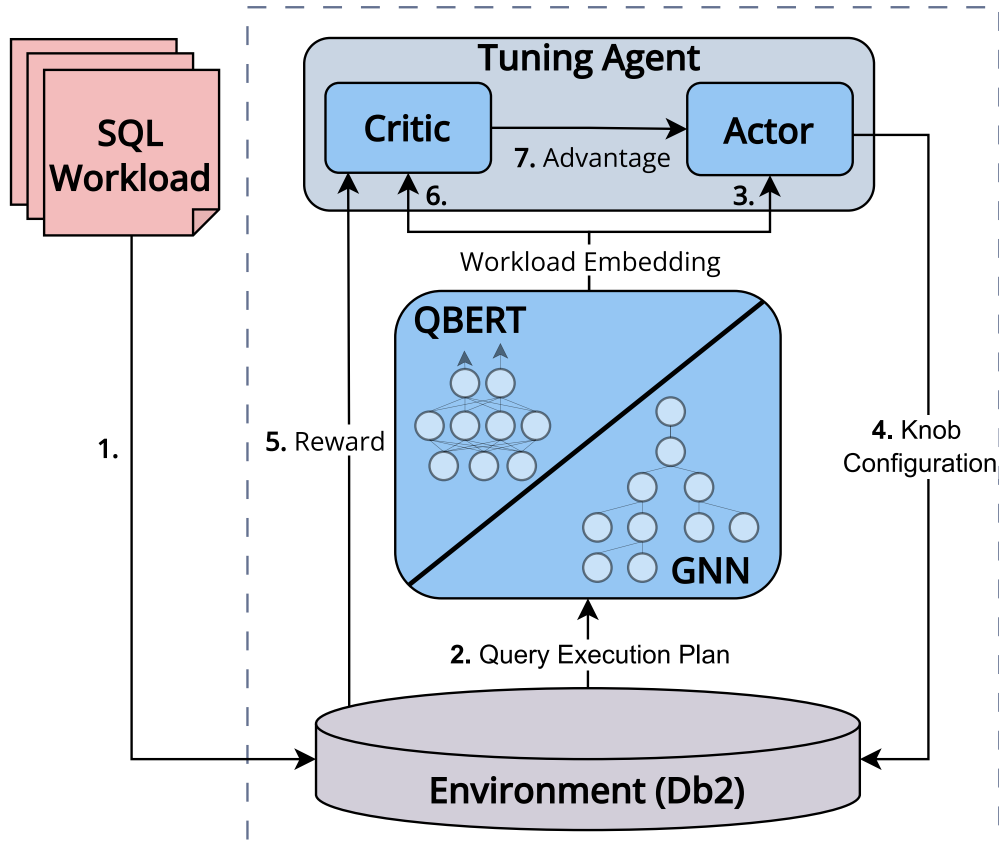


Figure 2.1: Architectural overview of `Db2une`.

from the QEPs, mapping features to templates. The transformer generates a low-dimensional *embedding* for each plan (Section 2.1). `QBERT` then aggregates these into a workload embedding vector. Alternatively, the `GNN` model maps the QEP to a graph object before applying a series of graph convolutions to generate an embedding from the raw workload features (Section 2.2). Said embedding is passed to the *actor model* within the *deep reinforcement learning* (DRL) *tuning agent* (*Step* 3). The agent, based on *proximal-policy optimization*, recommends a knob configuration aimed

at optimizing performance while efficiently allocating resources (Section 2.3). The recommended configuration is then applied to `IBM Db2`, which generates new QEPs for the workload based on the new tuning adjustment (*Step* 4). A *reward* value is determined, based on the performance cost and the amount of resources allocated (*Step* 5). The workload embedding from *Step* 3 is also passed to the *critic model* in the tuning agent (*Step* 6). Via the reward and the workload embedding, the critic learns how beneficial the actor's actions were for the workload and determines an *advantage* value to guide the actor's future tuning recommendations (*Step* 7). 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 4 are executed. The DBA validates final knob recommendations.

2.1 Transformer-Based Embedding Strategy

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 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` [11] and `BLUTune` [12], [15], and to identify problematic com-

ponents [16]. 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 [17], 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.

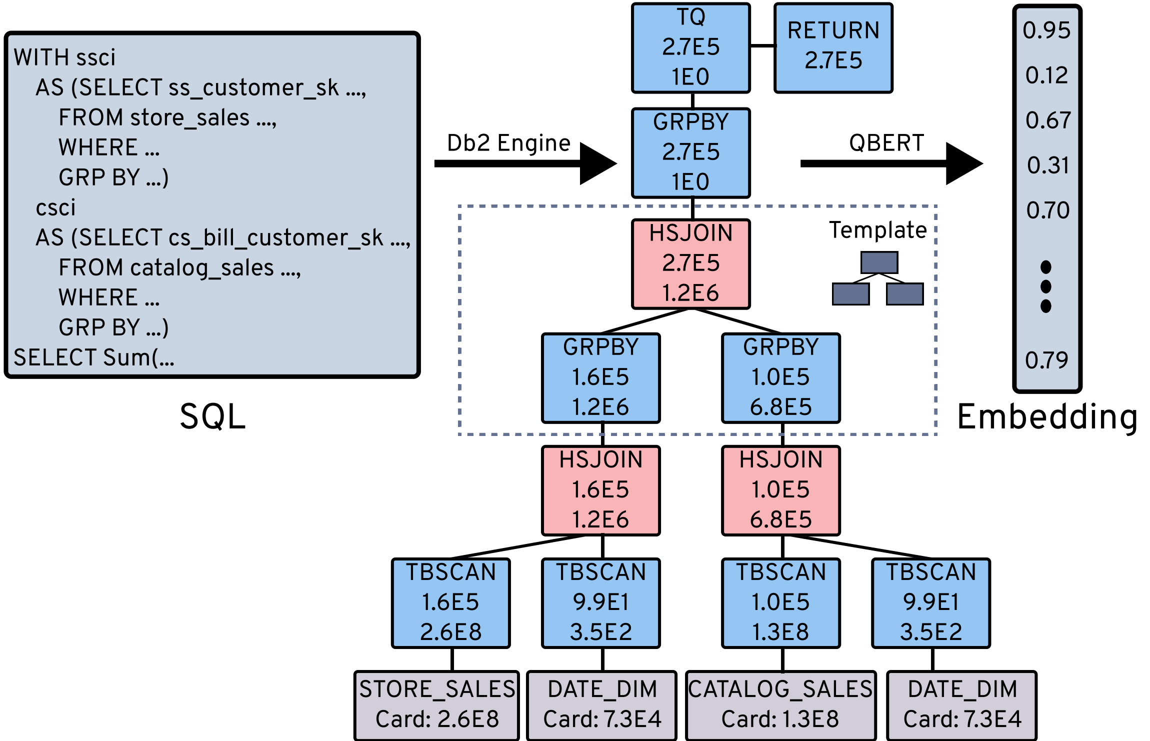


Figure 2.2: A query plan for $\mathbf{Q}_{97}$ representation via `QBERT`.

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. We show this is more adaptable and effective for query-aware knob tuning. Figure 2.2 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 2.1.1. This extracted information serves as input to the `QBERT` transformer model, responsible for generating the embeddings, discussed in Section 2.1.2.

2.1.1 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. Since the QEP generated for a query is dependent on the tuning, 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 [18]. Optimal knob settings result in plans that exhibit lower costs. QEPs additionally provide graph structures that capture relationships amongst operators and the underlying statistics. 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 2.2.

To transform a QEP into templates, we traverse its graph structure by a depth-

[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.

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* [19] 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 2.2, 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, allowing us to characterize QEPs based on their most important features. This facilitates the identification of (sub)plans with similar characteristics and structures across queries and query workloads, even when their underlying tables and attributes differ.

2.1.2 QBERT Embeddings

As a sequence of templates representing a QEP is still an arbitrary structure, we seek to map them into a fixed-length representation suitable for input to machine learning models. An accurate representation of query plans is crucial for effective database tuning. This representation must include details such as the costs and cardinalities of physical operators. The ability to distinguish between query plans, especially those that share similar operator costs and cardinalities, often relies on understanding the structural differences. This underscores the importance of a query representation that captures these contextual linkages. To address this challenge, we introduce `QBERT`, an embedding pipeline that maps high-dimensional QEPs into a low-dimensional embedding space. `QBERT` excels in encapsulating the cost and cardinality estimates, as well as the structural intricacies of these plans. 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 [20], [21]. Specifically, it leverages the Bidirectional Encoder Representations from Transformers (`BERT`) architecture [21], 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 into segments (e.g., sentences). It undergoes training on two main tasks: *masked-language modeling* and *next-segment prediction*.

In masked language modeling, BERT is given a sequence with some tokens randomly replaced by a placeholder mask, and its model's goal is to predict the original

tokens at these masked positions accurately. During the segment prediction task, BERT evaluates pairs of segments to determine whether the second segment follows the first, treating it as a binary classification problem.

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. For our next-segment prediction task, we decompose query plans into segments representing 2-, 3-, and 4-way joins to predict if one sub-plan follows another. However, we have empirically verified that its impact is less significant than masked-language modeling, aligning with observations made in RoBERTa [22] for the task of question answering. Thus, in the remaining discussion, we focus on the masked-template prediction task.

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.

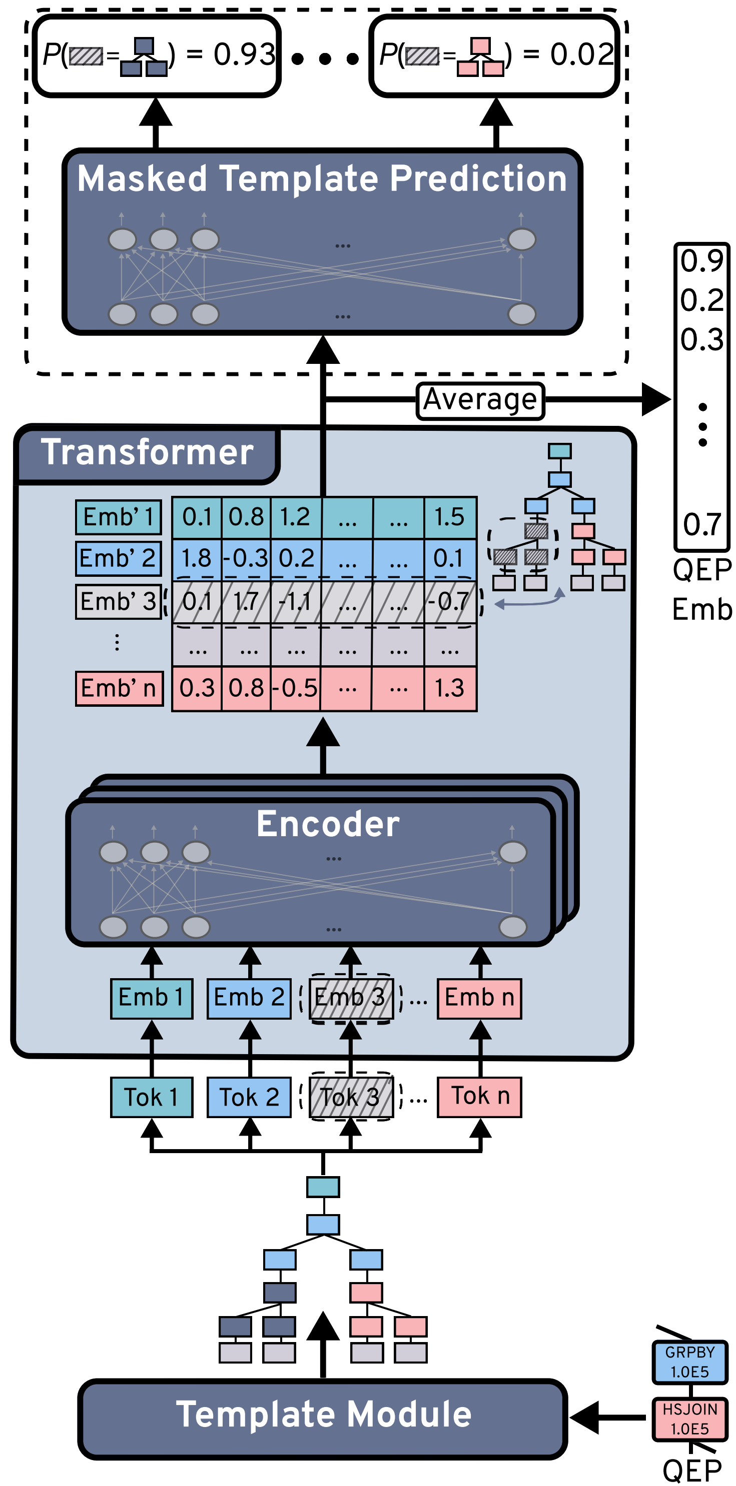


Figure 2.3: The QBERT pipeline.

Figure 2.3 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 [21], 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, $\texttt{Tok}_1, \ldots, \texttt{Tok}_n$, are input to the QBERT transformer where they then are mapped to *learned template embeddings*, $\texttt{Emb}_1, \ldots, \texttt{Emb}_n$. These template embeddings are then *augmented* by the transformer neural encoder into *contextual embeddings*, $\texttt{Emb}'_1, \ldots, \texttt{Emb}'_n$, 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 into the embeddings.

In Figure 2.3, 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 "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.

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) ← −∑_{t∈T̃} q_n[t]logP(t)
17:        θ ← θ − α · ∇loss(θ)
18:    end for
19: end for
```

After training the model, we can obtain an embedding v for any query q using the process in Algorithm 2. Based on the query, the `IBM Db2` cost-based optimizer generates a QEP (Line 2). We template the QEP to retrieve the tuning information (Line 3). The forward pass starts with the tokenization of the templated QEP, where each token is assigned an initial embedding based on its token ID and position in the sequence (Line 5). These initial embeddings are stored in the matrix Q (Line 6). The next step involves passing Q through several encoder blocks (Lines 7-12). Each token in the input sequence receives a contextual embedding through this process. These embeddings, which now carry contextual information, are averaged to obtain a final QEP embedding v (Line 13). This *QEP embedding*, indicated as $\langle 0.9, \ldots, 0.7 \rangle$ in Figure 2.3, 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 4 show that our embeddings allow `Db2une` to achieve notably better tuning performance than competing methods.

Algorithm 2 Transformer-Forward

1: **Input:** S, tokenized query plan templates
2: $l \leftarrow len(S)$
3: for $t \in [l]$: $\mathbf{e}_t \leftarrow \textsc{emb}_{token}(S[t]) + \textsc{emb}_{pos}(t)$
4: $X \leftarrow [\mathbf{e}_1, \mathbf{e}_2, ..., \mathbf{e}_l]$
5: **for** $i \in [N_{encoders}]$ **do**
6: $\quad X \leftarrow X + \text{MHAttention}(X|W^i_{MHA})$
7: $\quad \textsc{layer-norm}(X)$
8: $\quad X \leftarrow X + \text{MLP}(X|W^i_{MLP})$
9: $\quad \textsc{layer-norm}(X)$
10: **end for**
11: **return:** Matrix of contextual operator template embeddings X

2.2 Graph Contrastive Learning Embeddings

As discussed in Section 2.1, effectively representing QEPs is critical for automatic database tuning, as tuning decisions rely on extracting meaningful structural and cost-based features from execution plans. While `QBERT` effectively encodes QEPs using a transformer-based architecture, it has two potential limitations. First, QBERT requires linearizing the graph structure of QEPs into a sequence, potentially discarding relational information between operators and hiding feature information through templating. Although `QBERT` mitigates structural issues by using a self-attention mechanism capable of reasoning about input ordering, feature information is still lost through mapping the operators to templates based on ranges of estimated operator costs. Second, although `QBERT` is relatively small for a transformer model, it still

requires a GPU for efficient training. These computational requirements may be prohibitive when GPU resources are unavailable, making fast retraining or fine-tuning impractical on the CPU-only systems common in database environments.

To address these limitations, we propose *graph contrastive learning* (GCL) as an alternative approach to learning QEP embeddings while maintaining their inherent graph structure. Recently, graph representation learning has been explored for learning low-dimensional embeddings of nodes and graphs through *graph neural networks* (GNNs). However, most GNNs are trained through supervised learning, requiring labeled datasets that are often unavailable, undefined, or expensive in the database tuning domain. GCL provides a self-supervised training objective, enabling the GNN model to extract general knowledge about QEPs without explicit labels similar to QBERT's masked template prediction task.

In contrastive learning, the idea is to learn an embedding space where semantically similar instances are close together, while dissimilar instances are pushed apart [23]. In GCL, multiple graph views are generated via stochastic augmentations, and a GNN model is trained to contrast positive pairs (augmented versions of the same QEP) against negative pairs (QEPs from different queries) [24]. The model learns representations such that different augmentations of the same QEP produce similar embeddings while remaining distinct from embeddings of other QEPs. By applying the GCL paradigm to embedding QEPs, we aim to generate embeddings that preserve both structural and cost-related characteristics of query plans, providing the information to inform downstream tuning tasks.

Our `GNN` embedding approach for QEPs, illustrated in Figure 2.4, consists of three steps: (1) a data augmentation pipeline that perturbs QEPs through both attribute-

level and structure-level augmentations, (2) a GNN-based encoder that learns to compute QEP embeddings, and (3) a contrastive loss function that optimizes the similarity of representations in the learned space. We define a QEP as a directed graph $G = (V, E)$, where nodes $v_i \in V$ correspond to query plan operators (e.g., HSJOIN, TBSCAN), and edges $(v_i, v_j) \in E$ represent execution dependencies between operators. Each node is associated with a feature vector x_i, capturing operator attributes such as operator type, estimated cardinality, and cost metrics (e.g., CPU, I/O, memory).

We start by performing stochastic augmentation to generate multiple positive views from the same QEP. Specifically, we sample two augmentation functions $t1, t2 \in T$ to generate two augmented views $\tilde{G}_1 = t_1(G)$ and $\tilde{G}_2 = t_2(G)$. Following prior work in GCL [24]–[27], we use a bi-level augmentation scheme that perturbs both the attributes and structure of the QEP. At the attribute level, we inject Gaussian noise on operator cost estimates and cardinality, as well as apply feature masking, randomly zeroing out node attributes [25]–[27]. On the structural level, we randomly extract subplans within the graph and perform random walk subgraph sampling [25]–[27]. This combination of augmentations ensures that the model captures both structural and cost-related information, allowing embeddings to generalize across QEPs.

To compute embeddings from the augmented QEPs, we employ a graph isomorphism network (GIN) [28] $f(\cdot)$, chosen for its expressiveness in distinguishing different graph structures. Given an input graph G, the GIN updates node representations as in Equation 2.1:

$$h_k = MLP((1 + \epsilon) \cdot x_i + \sum_{j \in \mathcal{N}_i} x_j) \tag{2.1}$$

Where $\mathcal{N}_i$ denotes the neighbors of the ith node, and MLP is a multi-layer percep-

tron. GIN aggregates information across multiple k layers to capture hierarchical execution dependencies. This process encodes node-level representations; a global pooling function aggregates node embeddings into a graph-level embedding h_G as in Equation 2.2:

$$h_G = \sum_{i=0}^{N} h_i^0 \| \cdots \| \sum_{i=0}^{N} h_i^k \tag{2.2}$$

where node embeddings are first summed together at each layer and concatenated.

During training, a projection head $g(\cdot)$ is applied to transform the QEP embeddings into a low dimensional space $z_{\tilde{G}_1} = g(h_{\tilde{G}_1})$, $z_{\tilde{G}_2} = g(h_{\tilde{G}_2})$ before applying the contrastive loss. The projection head filters out features that encode the applied augmentations, leaving the encoder to learn content-based features and improve generalization [23], [29]. The contrastive loss function we use is the InfoNCE loss [30] as described by Equation 2.3:

$$\mathscr{L} = -\mathbb{E}_z \left[log \frac{exp(cos(z, z^+))}{\sum_{j=1}^{N-1} exp(cos(z, z_j))} \right] \tag{2.3}$$

Where z^+ is the positive pair embedding and z_j are negative pairs sampled from other QEPs. The loss encourages the embeddings of two augmented views of the same training example to be closer while pushing the embeddings of other training examples apart.

After training, the GNN encoder $f(\cdot)$ is retained and the projection head $g(\cdot)$ is discarded. This is because the encoder captures the generalizable QEP features, while the projection head functions as a contrastive loss facilitator rather than an essential component of the model [23], [29].

By leveraging GCL, our `GNN` method generates QEP embeddings that preserve

structural and cost-related properties while remaining computationally efficient. Unlike `QBERT`, this approach avoids the need to linearize the QEP into templates, maintaining the graph topology and attributes.

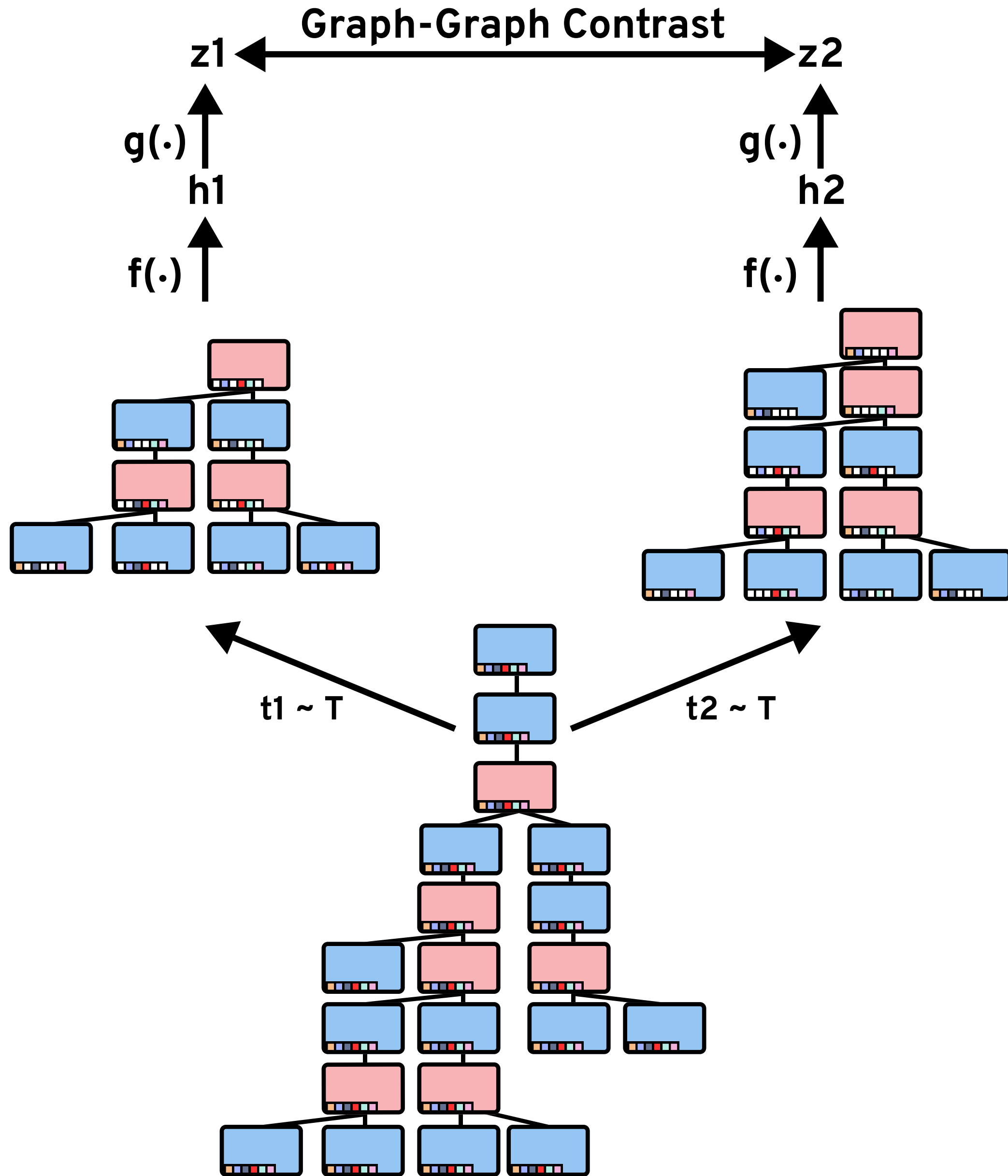


Figure 2.4: A Generalization of The GCL Model for Embedding Query Plans

2.3 Automatic Tuning System

The complex, correlated nature of database-system tuning parameters makes manual exploration of the configuration space both challenging and expensive. To navigate the complexity of the tuning configuration space, we employ deep reinforcement learning (DRL), using *actor-critic* networks, as illustrated in Figure 2.1. 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 2.3.3) is subsequently calculated based on the changes in performance metrics (Section 2.3.2), 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.

2.3.1 Stability-Oriented, On Policy Tuning

Prior work in DRL for database knob tuning [6], [7], [11], [31]–[33] has employed *off-policy* algorithms such as *deep deterministic policy gradient* (DDPG) [14]. 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 [6] and are sensitive to the quality of training data [34]. DDPG specifically exhibits instability and is brittle to hyperparameter selection [35], [36]. 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 [37]. This is a problem in a knob-tuning scenario where query responsiveness varies greatly across configurations.

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 [13], [38]. A potential drawback of on-policy algorithms is that they discard past experiences after each update. This requires fresh data for each iteration, a limiting factor in environments where sampling is expensive. As sampling latency and throughput from the database is so costly, it has been difficult to make use of these algorithms.

Our system mitigates these drawbacks in two ways. First, we pre-train the policy using quickly obtainable cost estimates instead of using runtimes (Section 2.3.2). Second, we employ the *proximal policy optimization* (PPO) on-policy algorithm [13], recognized for its stability and adaptability. It can also handle discrete and continuous action spaces, making it particularly well-suited for knob tuning.

The fundamental concept behind PPO is to enhance training stability by limiting the magnitude of policy updates within 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 2.4:

$$f_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \tag{2.4}$$

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 $f_t(\theta) > 1$, this indicates 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` [11] and `CDBTune` [31]. 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 2.5 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.

$$\widehat{A}(s_t, a_t) = Q(s_t, a_t) - V(s_t) \tag{2.5}$$

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 can rewrite the advantage as in Equation 2.6.

$$\widehat{A}(a_t, s_t) = \sum_{i=t}^{T-1} \gamma^{i-t} r_i + \gamma^{T-t} V(s_T) - V(s_t) \tag{2.6}$$

PPO's clipped surrogate objective function, denoted by L^{clip} in Equation 2.7, 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. This prevents large shifts between policies, ensuring stability throughout the tuning process.

$$L^{clip} = \widehat{\mathbb{E}}[\min(f_t(\theta)\widehat{A}_t, \text{clip}(f_t(\theta), 1-\epsilon, 1+\epsilon)\widehat{A}_t)] \tag{2.7}$$

The objective function in Equation 2.7 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 [13]. 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.

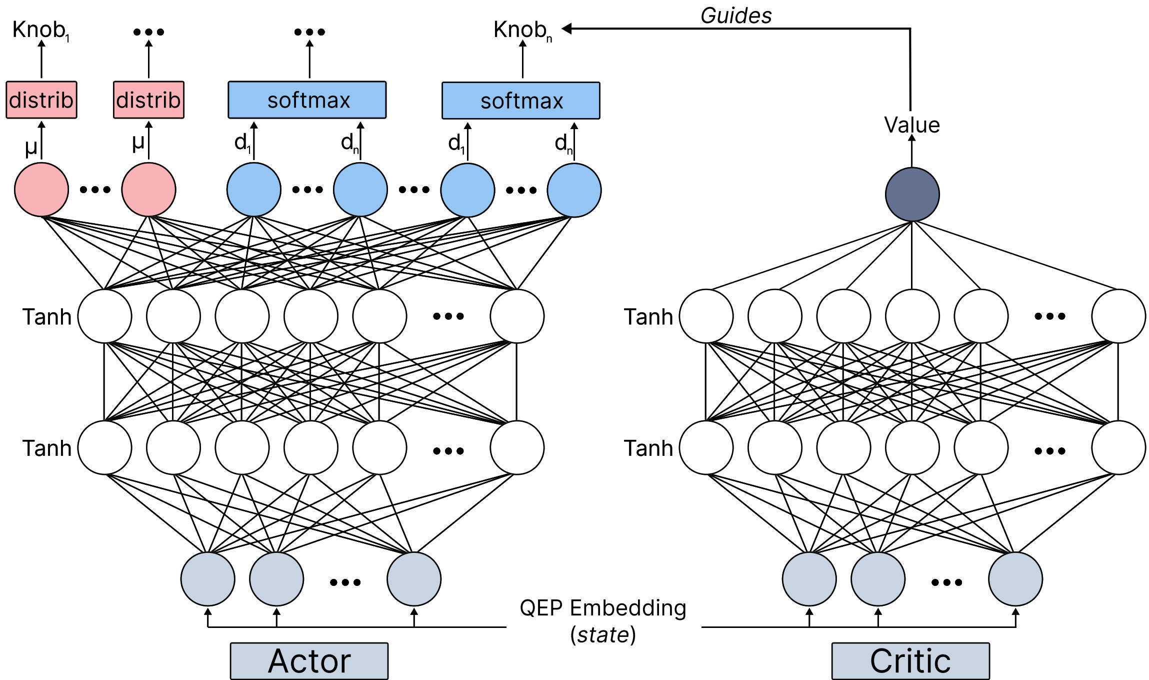


Figure 2.5: Tuning agent based on advantage actor-critic.

Figure 2.5 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 represent 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 akin to the actor-network to compute the state's V-value, $V(s)$. The V-values and the rewards are used to calculate the advantage (Equation 2.5), which, in

turn, guides the actor-network to minimize the objective function (Equation 2.7).

2.3.2 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.

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 [39], [40], recent advancements in machine learning-based cardinality estimation have significantly improved these estimates, offering promising enhancements for tuning efforts [41], [42]. 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.

The benefit of using cost estimates lies in the scalability of tuning training. 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 or even days 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, having a significant impact on training. In Section 4, 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 to estimate the `IBM Db2` cost estimates themselves. We sample and store QEPs initially, along with their associated costs and knob settings. Then using the sampled costs at given settings, we create a function to interpolate linearly between the sampled costs. Figure 2.6 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 4.2, 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, a subset of knobs stand out as the most influential.

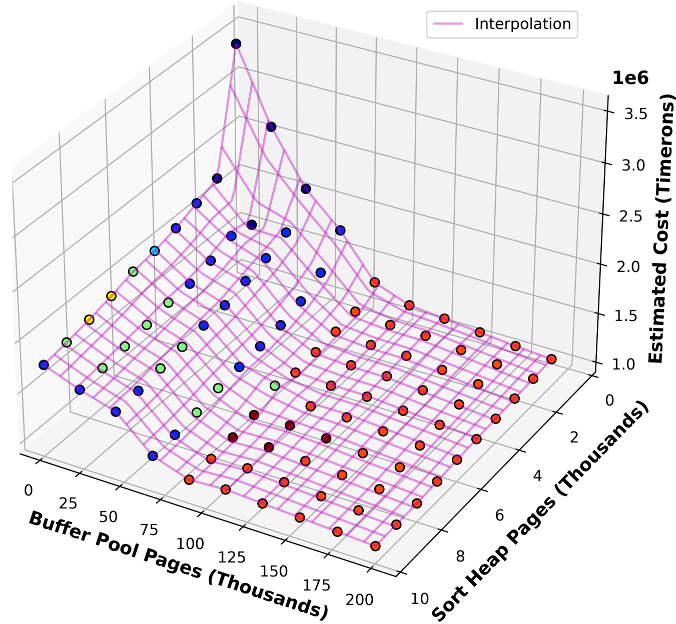


Figure 2.6: Example of interpolation for $\mathbf{Q}_{64}$.

Meta-Data Driven Training. A significant challenge for tuning database systems is *downtime*, as the database is typically taken offline during training. While cloning the database is a workaround, it introduces potential concerns regarding space, resources, data duplication time and synchronization, and network transfer costs. To address this issue, we propose a novel method to train over database meta-data instead of the full physical database. We use 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. DDLs are used for defining and managing the structure of the database

schema. DDL commands can create, modify, and delete database objects such as tables, indexes, views, and constraints. UPDATE STATISTICS commands are used to update statistics on tables or indexes in a database. It helps the database system optimize QEPs by providing updated statistical information about data distribution in tables or indexes.

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 [43]. The `db2fopt` command is used during training to 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, as 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 tuning model solely on the database statistics, even when our test system lacks the production system's resources. This method scales well for large databases with terabytes of data (Section 4.2)

2.3.3 Performance and Back Pressure Reward

The reward signal that the agent receives from the environment for recommending certain knobs at a given state is vital for 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 $\delta \in \{0, 1\}$.

$$r_t = r_{perf} + \delta r_{res} \tag{2.8}$$

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 2.3.2). 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 [11], [31]. This allows our system to receive instant feedback by assessing the quality of the current recommendation against an initial baseline, to gauge whether its recommendations improve or worsen performance during an episode. Thus, r_{perf} is defined as in Equation 2.9.

$$r_{perf} = \begin{cases} ((1 + r_{prev})^2 - 1)|1 + r_{init}| & \text{if } r_{init} > 0 \\ -((1 - r_{prev})^2 - 1)|1 - r_{init}| & \text{if } r_{init} \leq 0 \end{cases} \tag{2.9}$$

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 maintain a controlled scaling.

Back Pressure. Our design of r_{res} is motivated by the application of automated tuning systems in multi-tenant environments, where resources are shared, or must be rented, as in the cloud. In such environments, 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 [11] or are constrained by upper-bound resource limits [12], [15]. 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. We track the best performance seen so far and the corresponding settings for a given query during training. 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 2.10.

$$r_{res}^{(i)} = \beta^{(i)} \frac{r_{rec}^{(i)} - r_{best}^{(i)}}{r_{max}^{(i)}} \tag{2.10}$$

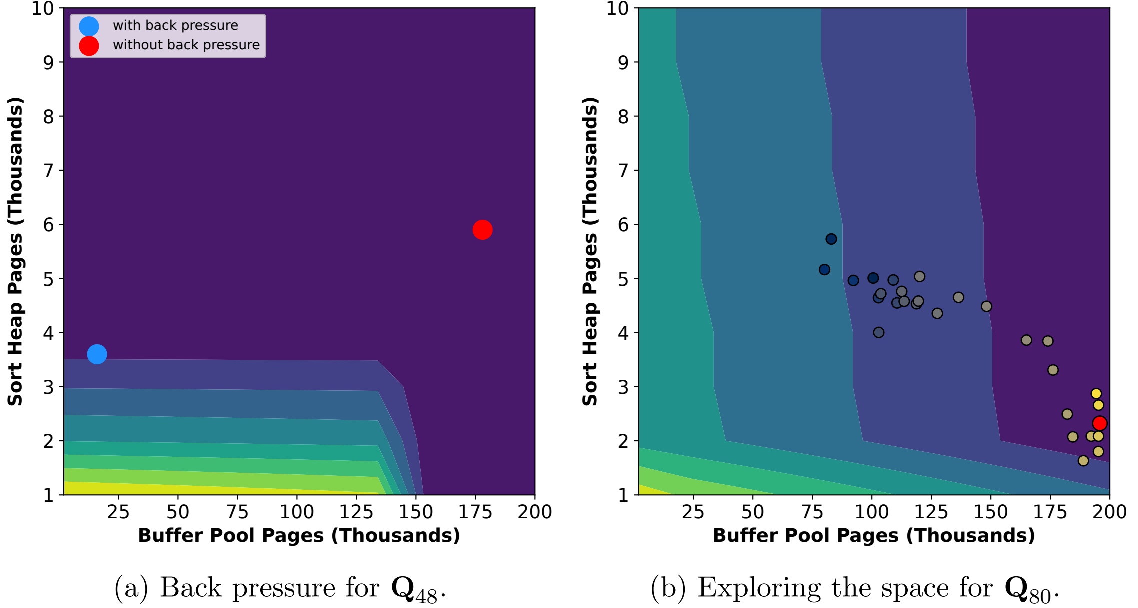
For knobs that allocate shared resources—for instance, such as buffer pool and sort heap do for memory—the resource reward is calculated by subtracting the current number of recommended pages of memory and the pages allocated to achieve the best-seen performance, normalized as per Equation 2.10. Back 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 2.10. 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 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 2.7 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 2.7a, 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 2.7b 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.



(a) Back pressure for $\mathbf{Q}_{48}$.

(b) Exploring the space for $\mathbf{Q}_{80}$.

Figure 2.7: Db2une setting knobs.

Back Pressure Variations. We consider three different 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 2.8, 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 4.3), we observed that split back pressure offers the best trade-off between efficiency and resource allocation.

2.3.4 The Training Process

Algorithm 3 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 (*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, and 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 respective objective functions (*Lines* 7–8), where R_t is the sum of discounted intermediate rewards.

In Figure 2.7b, 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

Algorithm 3 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**

system's exploration process, ultimately recommending a setting in the lowest cost region.

Chapter 3

System Demonstration

To showcase the Db2une system we developed an interactive web application that lets users tune data systems to boost performance and see how Db2une's query representation models aid users in understanding the query plans generated from complex analytical workloads.

The backend of Db2une is written in Python and uses PyTorch for the machine-learning components. The system leverages Celery and Redis to execute model training as a background task, enabling concurrent training and tuning recommendations. Db2une can connect to local and server instances of IBM Db2. The interactive front end is web-based and was built using the Django framework and JavaScript libraries. The system comes preloaded with TPC-DS and TPC-H benchmarks sized at 100GB for testing. An architectural overview of Db2une's web application is shown in Figure 3.1.

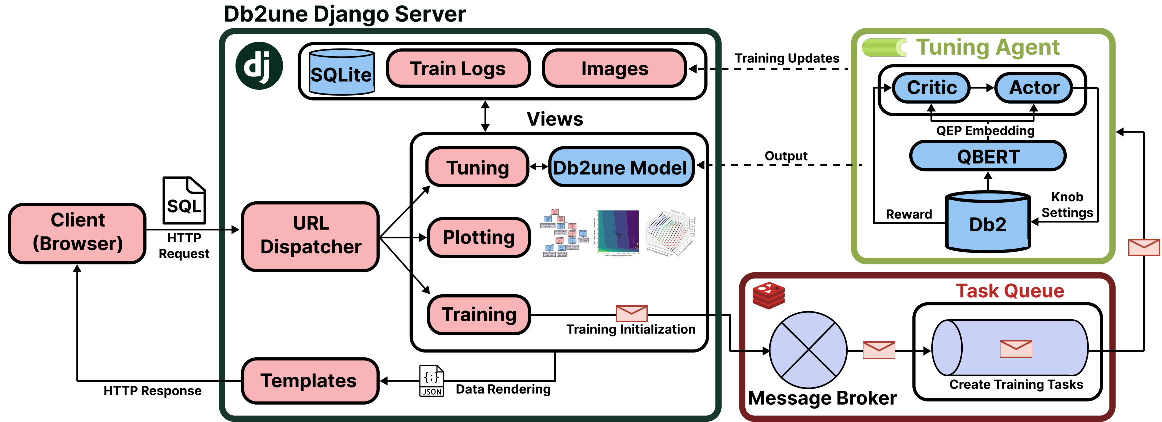


Figure 3.1: Architecture Overview of The Db2une Web System © IEEE 2025

3.1 Demonstration of Knob Tuning Interface

Figure 3.2 shows the interface for training the tuning model. This screen exposes numerous configuration parameters and registry variables available in IBM Db2 that are available for tuning. Users can tune all of the database knobs at once or they can select a specific subset to tune for their workload. Users also specify the query workload they want to optimize the database for, the mode for which the backpressure module is applied, the cost metric, and finally, whether they want to load a pre-trained model. This gives users flexibility in how they train their models, giving them the ability to train on one metric and then reload the model later for finetuning on additional queries or on a different training metric. Once a tuning request is made, users of the system can watch in real-time as Db2une's tuning agent explores the knob search space through the use of contour plots as shown in Figure 2.7b.

After Training a model, users can submit a tuning request through the tuning interface as shown in Figure 3.3. By selecting a model, the user can input any

query workload they want to tune, including queries they did not train the model on. Pressing the tune button will pass the workload to the Db2une model and display the tuning recommendation in the right pane. Users can load the tuning recommendations for the different queries through the use of a drop-down menu. For example, Figure 3.3 shows the tuning recommendation for TPC-DS $\mathbf{Q}_3$ along with the resulting QEP of using these settings for compiling the query. The ability to view the resulting QEP is of extra help to DBAs when analyzing the effects of the knobs and validating recommendations.

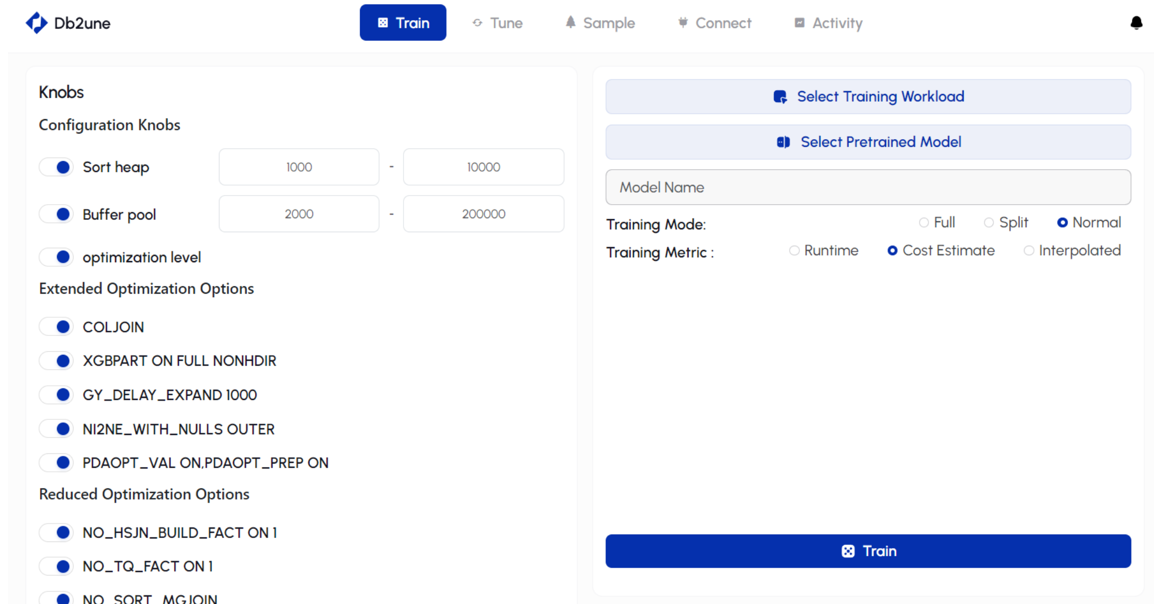


Figure 3.2: Db2une Training Screen © IEEE 2025

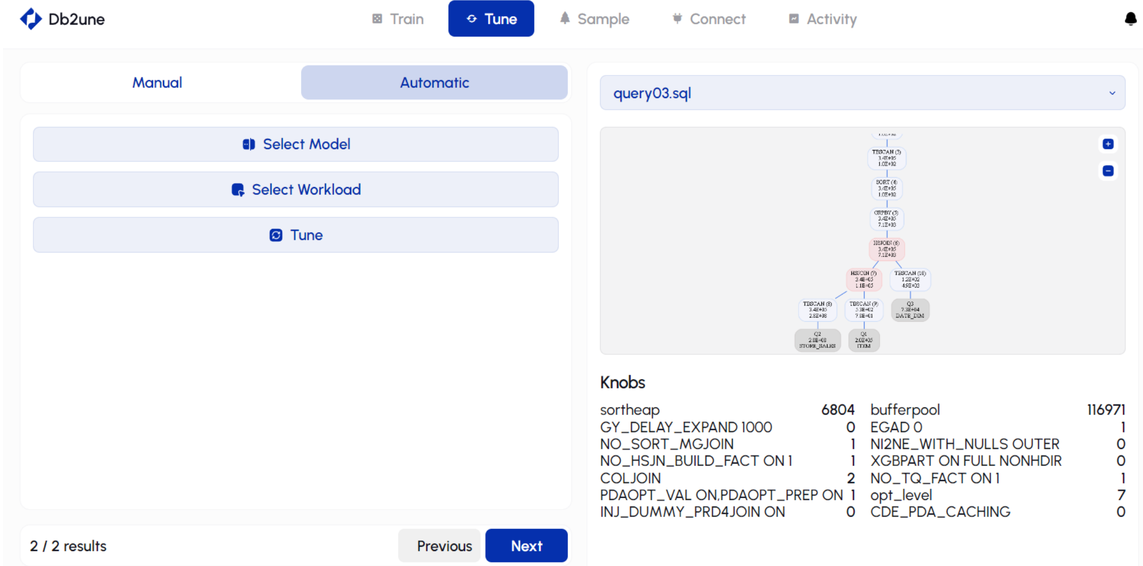


Figure 3.3: `Db2une` Tuning Screen © IEEE 2025

3.2 Query Aware Tuning and Transfer Learning

With the tool, users can attempt to manually tune the database and observe how their configurations affect query plans. They can compare their results with those from the trained `Db2une` model. This hands-on approach highlights the complexities of tuning and the benefits of an automated system like `Db2une`. After selecting their trained model and target workload in the tuning screen (Figure 3.3), users may submit a tuning request and review the knob settings and query plans.

The application also allows us to highlight the transfer learning capability of `Db2une`. Users can load a tuning model trained on one workload to make tuning requests for another. For example, `Db2une` can be trained using the TPC-DS schema and tested on the TPC-H schema or vice-versa. We can also demonstrate `Db2une`'s transferability by testing models trained with different metrics and environments.

Users may load a model pre-trained on cost estimates and fine-tuned on runtime metrics, then compare its recommendations to a model trained solely on cost estimates.

Lastly, the tool allows us to compare the tuning recommendations from two `Db2une` models: one trained for performance optimization and another with back pressure to balance performance and resource usage. Users can see that the model with backpressure can recommend more efficient settings for the resource-related knobs while sacrificing minimal performance.

Chapter 4

Experimental Evaluation

We perform 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 an RTX3070 GPU, running IBM Db2 Enterprise Edition. We evaluate Db2une on the TPC-DS [8] and TPC-H [44] benchmarks, and on an IBM client-inspired analytical workload, referred to as IBM Client. Each database is column-organized and sized at 100GB. 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. All experimental results for Db2une are with respect to using QBERT embeddings unless stated otherwise.

4.1 Query and Workload Level Tuning

We evaluate the Db2une system against BLUTune [12], [15], which uses QEP2Vec embeddings, and against QTune [11], 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 in tuning for each query. As illustrated in Figure 4.1, Db2une's recommended knob configurations result in high performance gains over the TPC-DS, TPC-H, and IBM Client workloads, On TPC-DS, Db2une reduces total execution time by 23.3% compared to BLUTune, and by 30.9% compared to QTune-F. Similarly, on the IBM Client workload, Db2une achieves reductions of 46.8% compared to BLUTune, and 50.6% compared to QTune-F. On TPC-H, Db2une further excels, outperforming BLUTune by 50.1%, and QTune-F by 60.4%. To demonstrate Db2une's effectiveness in re-using 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. These results highlight Db2une's effectiveness in identifying suitable cross-workload tuning requirements.

Exp-2: Workload Level. We next evaluate how the Db2une system tunes once per workload. As shown in Figure 4.2, Db2une outperforms BLUTune and QTune-F over the TPC-DS, TPC-H, and IBM Client workloads, On TPC-DS, Db2une reduces execution time by 25% (vs. BLUTune) and 31.2% (vs. QTune-F). These improvements are even more pronounced on the IBM Client workload (42.6% vs. BLUTune, 46.2% vs. QTune-F) and TPC-H (48.5% vs. BLUTune, 56.3% vs. QTune-F). XTune's transfer-

ability is further demonstrated by training the model on one workload and applying it to another. `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 $\rightarrow$ TPC-H and TPC-H $\rightarrow$ IBM Client, respectively.

4.2 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 2.3.2). 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 with 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 greatest impact on cost estimates and runtime.

Table 4.1 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*

[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 offline cost is not included in the table.

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.

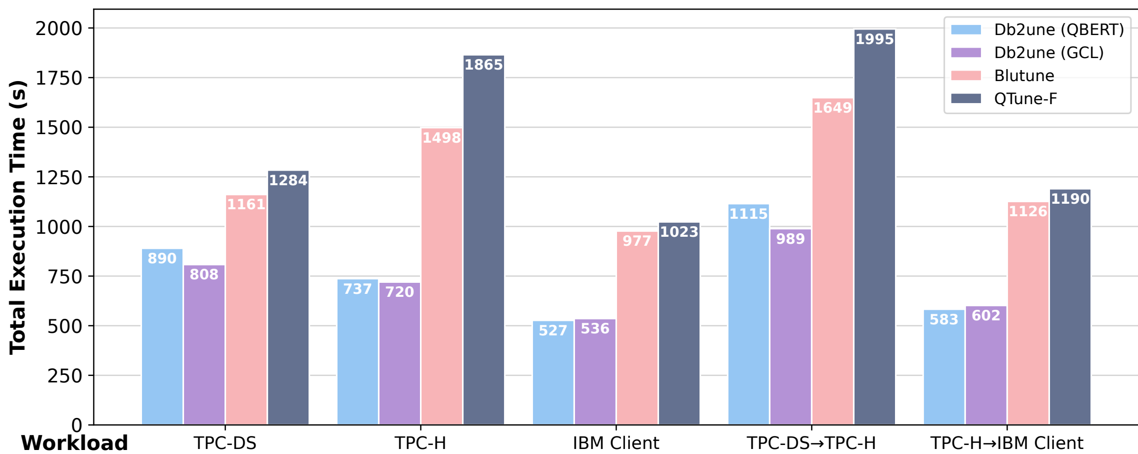


Figure 4.1: Query level tuning.

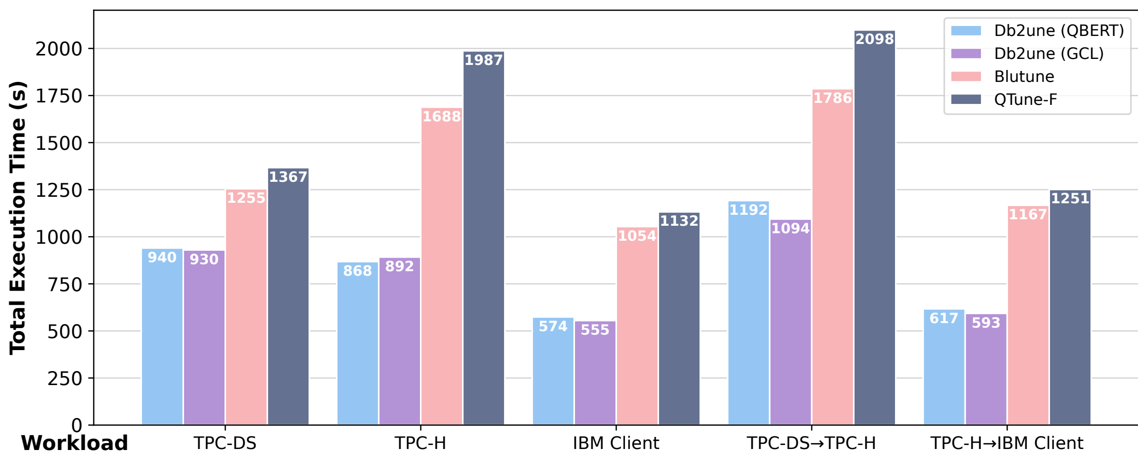


Figure 4.2: Workload level tuning.

Figure 4.3: 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 without completing. We conclude that training solely on runtime is impractical for large workloads. We observed similar trends over the TPC-H and IBM Client workloads as reported in Table 4.1. Our experiments confirm that we can greatly reduce training time while not drastically 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.

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

Table 4.1: 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

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 auto-

Table 4.2: 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%

matically, alleviating the human burden to tune manually, which IBM experts report can take hours, even days.

4.3 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 4.4a plots the estimated cost of each test query as provided by the `IBM Db2` optimizer at `Db2une`'s recommended settings. Figure 4.4b 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 4.4c shows the pages of memory allocated by each model, revealing a large memory-resource reduction for both the split- and full-back pressure models, just as anticipated.

Table 4.2 summarizes the models' performance. Split shows a slight 3% increase in total runtime (and 7% in estimated cost), in exchange for a notable 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 large increases in runtime while economizing on resource allocation, we advocate for the split back-pressure approach.

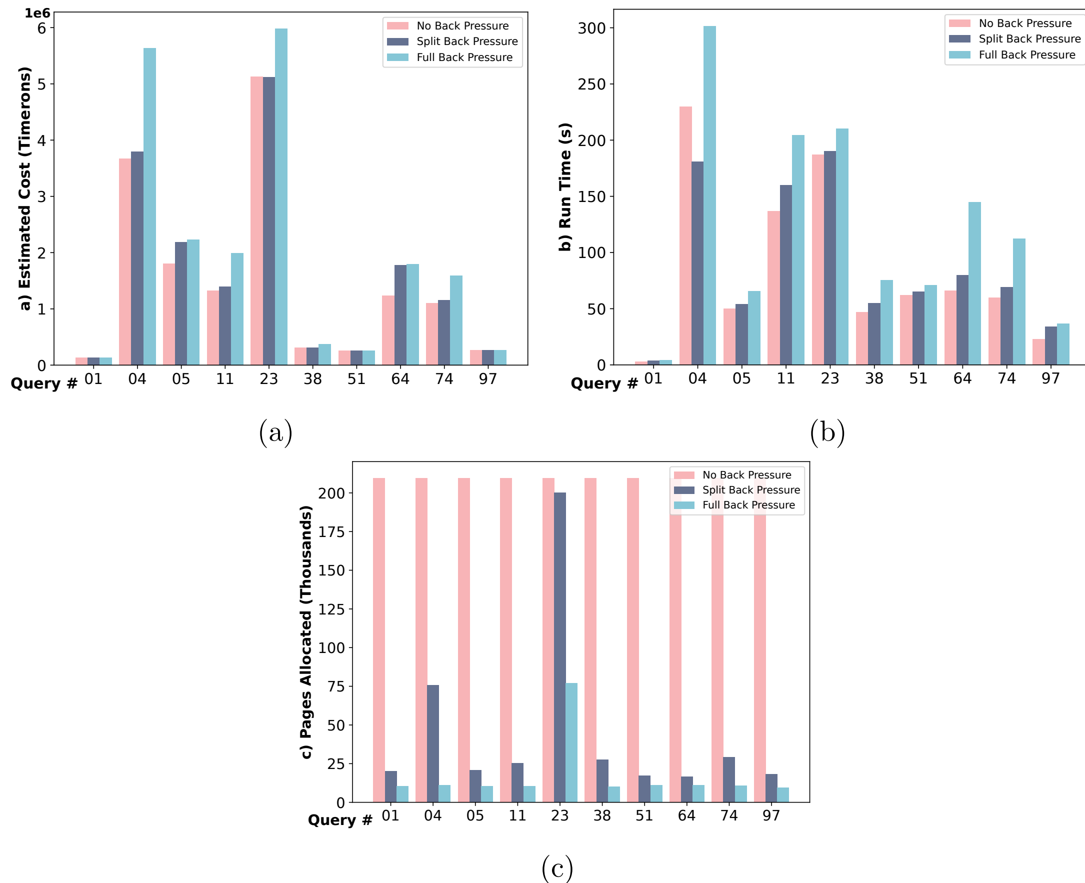


Figure 4.4: Comparison of `Db2une` with and without back pressure over test queries.

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 4.3, 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 a 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, requiring an increase. 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.

4.4 Embedding Quality

Exp-8: IBM Expert Case Study. To gauge the quality of query representation methods, we conducted a case study with IBM experts on the TPC-DS workload to measure how similarly experts evaluate query plans with how `Db2une`'s `QBERT` and `GNN`, `BLUTune`'s `QEP2Vec`, and `QTune`'s `Featurization` do. The study involves four cases, each with a designated reference plan and an evaluation set of five plans. For each case, the experts ranked the five plans in the evaluation set from most to least similar to the reference plan to establish a ground truth. The IBM experts' rankings were based on their knowledge and experience, considering factors such as associated QEP estimated costs, cardinalities, and the structure of the subplans. `QBERT`, `GNN`, `QEP2Vec`, and `Featurization` ranked the plans based on their embedding similarities.

Across the four cases, the ranking of QEPs based on the embeddings of `QBERT` and `GNN` aligned most closely with the IBM experts' judgments. To quantify the agreement between experts and models, we use the Kendall rank correlation coefficient [45]. The Kendall correlation between two rankings is *high* when the observations are ranked similarly (1 when identical), and low when the observations have a dissimilar ranking

(-1 when fully dissimilar). We report the average of the correlations. GNN achieved the highest average correlation of 0.70, followed by QBERT with an average correlation of 0.65. This greatly outperforms QEP2Vec with 0.10 and QTune with -0.25. Thus, Db2une's embeddings better capture the nuances of query plans as do the experts.

Figure 4.5 illustrates the case with the QEP for Query $\mathbf{Q}_7$ as the reference. Db2une's QBERT, GNN and the IBM experts agreed the QEP for $\mathbf{Q}_{79}$ from the evaluation set to be the most similar to $\mathbf{Q}_7$'s. This choice reflects the shared recognition of the plans' similar table sizes, comparable join costs, and highly similar operator structure. A minor difference is that the GROUP BY operator in the QEP for $\mathbf{Q}_7$ is pushed down to be before the last hash join in the QEP for $\mathbf{Q}_{79}$. In contrast, BLUTune's QEP2Vec selected the QEP for $\mathbf{Q}_6$ as most similar to $\mathbf{Q}_7$'s. While sharing some cost similarities and a common substructure, QEP2Vec's bag-of-words approach lacks sensitivity to operator order, so it does not find the closer match of $\mathbf{Q}_{79}$'s, due to the join ordering and the position of the GROUP BY. Meanwhile, QTune's Featurization chose the QEP for $\mathbf{Q}_{25}$ as most similar to $\mathbf{Q}_7$'s. These plans exhibit greater structural and cost differences compared to $\mathbf{Q}_7$'s and $\mathbf{Q}_{79}$'s, as identified by the IBM experts and Db2une's embeddings. The plans for $\mathbf{Q}_7$ and $\mathbf{Q}_{25}$ also differ greatly in cardinality, likely due to that QTune does not encode cardinality in its featurization. One factor in Featurization's selection is its over-emphasis on table overlap: $\mathbf{Q}_7$ and $\mathbf{Q}_{25}$ share three tables in common, while $\mathbf{Q}_7$ and $\mathbf{Q}_{79}$ just share two. We verified this by modifying $\mathbf{Q}_{25}$ to change one of the overlapping table names, upon which Featurization no longer considered its QEP as most similar.

Exp-9: Comparison of QBERT and GNN based embedding approaches.

To evaluate the efficiency and practicality of our two proposed embedding meth-

Metric	GNN	QBERT
Training Time (CPU)	20s	21m (templating) + 132m (training)
Training Time (GPU)	15s	21m (templating) + 8m (training)
Inference Time (100 Queries CPU)	0.1178s	0.8247s
Inference Time (100 Queries GPU)	0.0926s	1.3624s
Model Size	0.047mb	14.25mb
Parameters	12,192	3,836,856

Table 4.3: GNN and QBERT Comparison

ods, we compare their training time, inference speed, and model size, as shown in Table 4.3. Our results demonstrate that while both approaches achieve comparable tuning performance (Section 4.1), they present different trade-offs in terms of computational efficiency and resource usage.

GNN outperforms QBERT in training and inference speed, making it well-suited for both GPU and CPU environments. Training the GNN model takes only 15s on GPU and 20s on CPU, whereas QBERT requires 21 minutes for templating and an additional 8 minutes of training on GPU—or over two hours on CPU. For inference across 100 queries, GNN achieves a latency of 0.1178s on CPU and 0.0926s on GPU, compared to 0.8247s and 1.3624s for QBERT, respectively. While both inference speeds are fast, GNN is a more practical alternative for CPU-only tuning environments. However, QBERT offers advantages in resource-aware tuning. While the GNN model tends to allocate more resources, QBERT achieves similar tuning performance while maintaining lower resource usage. This suggests that QBERT may be preferable in cloud-based or shared-resource environments, where minimizing resource allocation is critical.

Beyond performance, the compact nature of the GNN model makes it highly scalable. With only 12,192 parameters and a 47KB model size, it is substantially smaller than QBERT, which has 3.8M parameters and a 14.25MB model size. This lightweight

architecture makes `GNN` more efficient for deployment in database systems without dedicated GPUs or high-memory constraints.

Overall, `GNN` provides a faster and more scalable approach to QEP embedding, whereas `QBERT` remains a strong choice for optimizing tuning performance while controlling resource consumption. The choice between the two depends on the target environment and whether the user wants to prioritize speed and scalability (`GNN`) or balancing performance and resource efficiency (`QBERT`).

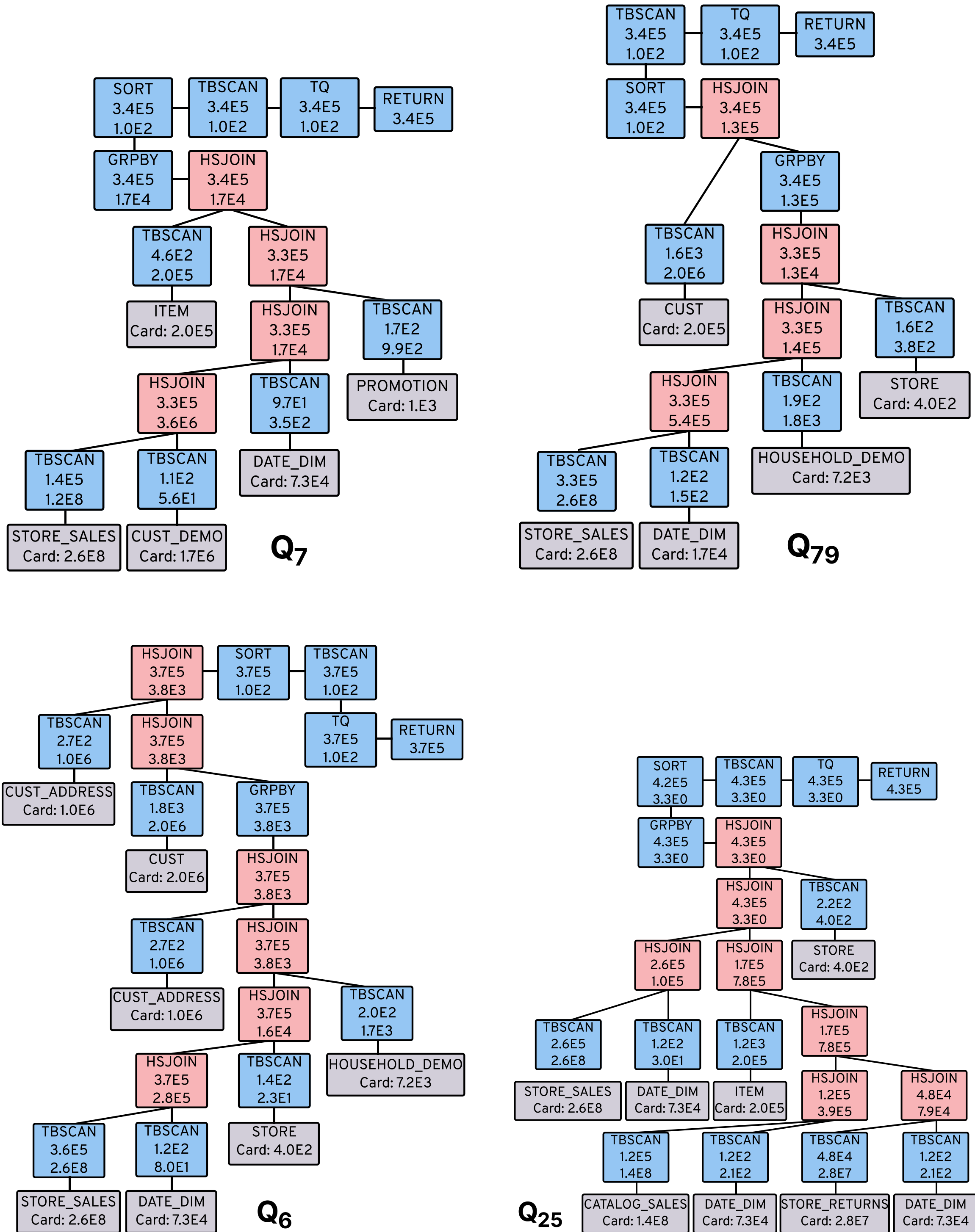


Figure 4.5: The QEPs for $\mathbf{Q}_7$, $\mathbf{Q}_{79}$, $\mathbf{Q}_6$, and $\mathbf{Q}_{25}$ in the IBM expert case study.

Chapter 5

Related Work

Many systems for automatic database system knob tuning using machine learning have been developed over the last several years; e.g., [7], [11], [31], [32], [46]–[49]. In this section, we focus on the ones most relevant to the `Db2une` system.

`OtterTune` [6], [46] 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 [50], [51], which may limit its use for tuning complex analytical workloads.

`CDBTune` [31] 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. [33] 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` [11] 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` [12] 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 [52]. 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. Additionally, the GNN used in `SQLBERT` requires a supervised training scheme, unlike our GCL approach. 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 [53]. Tree-LSTM architectures have also been used to encode query plans for cost estimation [54], 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.

Chapter 6

Conclusion

6.1 Conclusion

In this work, we created the Db2une system to automatically tune database knobs for optimal performance while conserving hardware resources. 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 provides query-aware tuning configurations via our QBERT and GNN embeddings that capture the intricacies of query execution plan structures, operators, and costs. These tuning configurations are not just performant, but also resource-conscious through the use of our back pressure reward function that is specifically designed with cloud environments in mind. A key innovation in Db2une is its multi-phased training strategy that utilizes quickly obtainable metrics like cost estimates for initial exploration and refines performance through runtime-based tuning. This approach allows for faster training than traditional automatic knob tuning systems while ensuring comprehensive performance

enhancements. Additionally, `Db2une`'s meta-data-driven training allows scalable tuning without direct query execution, further enhancing its practical applicability. Our experimental evaluations have demonstrated significant improvements by `Db2une` over previous state-of-the-art query-aware tuning systems and IBM experts.

6.2 Future Work

`Db2une` is designed as one piece of a greater self-tuning database system. When there is a problem in workload performance, say when the workload changes, `Db2une` is an effective way for DBAs to get a tuning configuration. However, if `Db2une` could predict how the workload would change in the future, it could take proactive measures to tune the database in anticipation. There is a history of work in query workload forecasting, where machine learning is used to analyze historical workload traces and then taught to predict how that workload will evolve in the future [55], [56]. By incorporating the techniques developed in this field to `Db2une` our system could be even more versatile in its recommendations.

Modern database systems include a variety of features that impact performance beyond knobs, including indexes, stat views, materialized views, etc. In future work, we would be interested in developing a unified approach to tuning, providing recommendations for all the tunable features of a database beyond knobs.

Finally, although we have shown the speed of training `Db2une` we believe we can further improve. Modifications of PPO [57] allow for distributed training through the use of multiple workers and a parameter server. In this setup, a shared model is updated every time a certain number of workers are ready to send their updates. Currently, `Db2une` processes queries one by one during training. Using a distributed

training algorithm, we could process multiple queries at once before sending the results to the parameter server to update the model. This would allow for even faster training on top of our cost metrics.

Copyright

In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of York University's products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications standards/publications/rights/rights link.html to learn how to obtain a License from RightsLink. If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

References

[1] X. Zhao, X. Zhou, and G. Li, "Automatic database knob tuning: A survey," *IEEE Trans. Knowl. Data Eng.*, vol. 35, no. 12, pp. 12 470–12 490, 2023. DOI: 10.1109/TKDE.2023.3266893. [Online]. Available: https://doi.org/10.1109/TKDE.2023.3266893.

[2] P. Belknap, B. Dageville, K. Dias, and K. Yagoub, "Self-tuning for SQL performance in oracle database 11g," in *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009*, IEEE Computer Society, 2009, pp. 1694–1700. DOI: 10.1109/ICDE.2009.165. [Online]. Available: https://doi.org/10.1109/ICDE.2009.165.

[3] I. D. Documentation, *Configuration parameters summary*, Accessed: 2023-12-07, 2022. [Online]. Available: https://www.ibm.com/docs/en/db2/11.5?topic=parameters-configuration-summary.

[4] I. D. Documentation, *Db2 registry and environment variables*, Accessed: 2023-12-07, 2022. [Online]. Available: https://www.ibm.com/docs/en/db2/11.5?topic=variables-registry-environment.

[5] G. Damasio, V. Corvinelli, P. Godfrey, *et al.*, "Guided automated learning for query workload re-optimization," *Proc. VLDB Endow.*, vol. 12, no. 12, pp. 2010–

2021, 2019. DOI: 10.14778/3352063.3352120. [Online]. Available: http://www.vldb.org/pvldb/vol12/p2010-damasio.pdf.

[6] D. V. Aken, D. Yang, S. Brillard, *et al.*, "An inquiry into machine learning-based automatic configuration tuning services on real-world database management systems," *Proc. VLDB Endow.*, vol. 14, no. 7, pp. 1241–1253, 2021. DOI: 10.14778/3450980.3450992. [Online]. Available: http://www.vldb.org/pvldb/vol14/p1241-aken.pdf.

[7] B. Cai, Y. Liu, C. Zhang, *et al.*, "HUNTER: an online cloud database hybrid tuning system for personalized requirements," in *Proceedings of the 2022 International Conference on Management of Data*, ser. SIGMOD '22, ACM, 2022, pp. 646–659. DOI: 10.1145/3514221.3517882. [Online]. Available: https://doi.org/10.1145/3514221.3517882.

[8] M. Pöss, R. O. Nambiar, and D. Walrath, "Why you should run TPC-DS: A workload analysis," in *Proceedings of the 33rd International Conference on Very Large Data Bases, VLDB 2007*, ACM, 2007, pp. 1138–1149. [Online]. Available: http://www.vldb.org/conf/2007/papers/industrial/p1138-poess.pdf.

[9] A. Bianchi, A. Chai, V. Corvinelli, P. Godfrey, J. Szlichta, and C. Zuzarte, "Db2une: Tuning under pressure via deep learning," *Proc. VLDB Endow.*, vol. 17, no. 12, pp. 3855–3868, 2024. DOI: 10.14778/3685800.3685811. [Online]. Available: https://doi.org/10.14778/3685800.3685811.

[10] A. Bianchi, A. Chai, R. Dolores, *et al.*, "Db2une: Tuning ibm db2 with deep learning," *2025 IEEE 41st International Conference on Data Engineering (ICDE)*, pp. 1–4, 2025.

[11] G. Li, X. Zhou, S. Li, and B. Gao, "Qtune: A query-aware database tuning system with deep reinforcement learning," *PVLDB*, vol. 12, no. 12, pp. 2118–2130, 2019.

[12] C. Henderson, S. Bryson, V. Corvinelli, *et al.*, "Blutune: Query-informed multi-stage ibm db2 tuning via ml," in *Proceedings of the 31st ACM International Conference on Information and Knowledge Management*, ser. CIKM '22, 2022, pp. 3162–3171, ISBN: 9781450392365. DOI: 10.1145/3511808.3557117. [Online]. Available: https://doi.org/10.1145/3511808.3557117.

[13] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *CoRR*, vol. abs/1707.06347, 2017. arXiv: 1707.06347. [Online]. Available: http://arxiv.org/abs/1707.06347.

[14] T. P. Lillicrap, J. J. Hunt, A. Pritzel, *et al.*, "Continuous control with deep reinforcement learning," in *International Conference on Learning Representations, ICLR 2016*, 2016. [Online]. Available: http://arxiv.org/abs/1509.02971.

[15] C. Henderson, V. Corvinelli, P. Godfrey, P. Mierzejewski, J. Szlichta, and C. Zuzarte, "Blutune: Tuning up IBM db2 with ML," in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, IEEE, 2023, pp. 3615–3618. DOI: 10.1109/ICDE55515.2023.00281. [Online]. Available: https://doi.org/10.1109/ICDE55515.2023.00281.

[16] S. Jain and B. Howe, "Query2vec: NLP meets databases for generalized workload analytics," *CoRR*, vol. abs/1801.05613, 2018. arXiv: 1801.05613. [Online]. Available: http://arxiv.org/abs/1801.05613.

[17] Q. V. Le and T. Mikolov, "Distributed representations of sentences and documents," in *Proceedings of the 31th International Conference on Machine Learning, ICML 2014*, vol. 32, 2014, pp. 1188–1196. [Online]. Available: http://proceedings.mlr.press/v32/le14.html.

[18] S. Chaudhuri and V. R. Narasayya, "Autoadmin 'what-if' index analysis utility," in *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '98, ACM Press, 1998, pp. 367–378. DOI: 10.1145/276304.276337. [Online]. Available: https://doi.org/10.1145/276304.276337.

[19] I. D. Documentation, *Explain information for data operators*, Accessed: 2024-07-10, 2024. [Online]. Available: https://www.ibm.com/docs/en/db2/11.5?topic=information-explain-data-operators.

[20] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, "Attention is all you need," in *Advances in Neural Information Processing Systems, NeurIPS 2017*, 2017, pp. 5998–6008. [Online]. Available: https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

[21] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "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, 2019, pp. 4171–4186. DOI: 10.18653/V1/N19-1423. [Online]. Available: https://doi.org/10.18653/v1/n19-1423.

[22] Y. Liu, M. Ott, N. Goyal, *et al.*, "Roberta: A robustly optimized BERT pretraining approach," *CoRR*, vol. abs/1907.11692, 2019. arXiv: 1907.11692. [Online]. Available: http://arxiv.org/abs/1907.11692.

[23] T. Chen, S. Kornblith, M. Norouzi, and G. E. Hinton, "A simple framework for contrastive learning of visual representations," in *Proceedings of the 37th International Conference on Machine Learning, ICML 2020*, ser. Proceedings of Machine Learning Research, vol. 119, PMLR, 2020, pp. 1597–1607. [Online]. Available: http://proceedings.mlr.press/v119/chen20j.html.

[24] Y. Zhu, Y. Xu, Q. Liu, and S. Wu, "An empirical study of graph contrastive learning," *NeurIPS*, 2021. [Online]. Available: https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/0e01938fc48a2cfb5f2217fbfb00722d-Abstract-round2.html.

[25] Y. Zhu, Y. Xu, F. Yu, Q. Liu, S. Wu, and L. Wang, "Deep graph contrastive representation learning," *CoRR*, 2020.

[26] Y. You, T. Chen, Y. Sui, T. Chen, Z. Wang, and Y. Shen, "Graph contrastive learning with augmentations," *NeurIPS*, 2020.

[27] Y. Zhu, Y. Xu, F. Yu, Q. Liu, S. Wu, and L. Wang, "Graph Contrastive Learning with Adaptive Augmentation," in *Proceedings of The Web Conference 2021*, ser. WWW '21, Association for Computing Machinery, Apr. 2021, pp. 2069–2080, ISBN: 9781450370233. DOI: 10.1145/3442381.3449802. [Online]. Available: https://doi.org/10.1145/3442381.3449802.

[28] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" *ICLR*, 2019. [Online]. Available: `https://openreview.net/forum?id=ryGs6iA5Km`.

[29] K. Gupta, T. Ajanthan, A. van den Hengel, and S. Gould, "Understanding and improving the role of projection head in self-supervised learning," *CoRR*, vol. abs/2212.11491, 2022. DOI: `10.48550/ARXIV.2212.11491`. arXiv: `2212.11491`. [Online]. Available: `https://doi.org/10.48550/arXiv.2212.11491`.

[30] A. van den Oord, Y. Li, and O. Vinyals, "Representation learning with contrastive predictive coding," *CoRR*, vol. abs/1807.03748, 2018. arXiv: `1807.03748`. [Online]. Available: `http://arxiv.org/abs/1807.03748`.

[31] J. Zhang, Y. Liu, K. Zhou, *et al.*, "An end-to-end automatic cloud database tuning system using deep reinforcement learning," in *Proceedings of the 2019 International Conference on Management of Data*, ser. SIGMOD '19, ACM, 2019, pp. 415–432. DOI: `10.1145/3299869.3300085`. [Online]. Available: `https://doi.org/10.1145/3299869.3300085`.

[32] J. Ge, Y. Chai, and Y. Chai, "Watuning: A workload-aware tuning system with attention-based deep reinforcement learning," *J. Comput. Sci. Technol.*, vol. 36, no. 4, pp. 741–761, 2021. DOI: `10.1007/S11390-021-1350-8`. [Online]. Available: `https://doi.org/10.1007/s11390-021-1350-8`.

[33] Y. Gur, D. Yang, F. Stalschus, and B. Reinwald, "Adaptive multi-model reinforcement learning for online database tuning," in *Proceedings of the 24th International Conference on Extending Database Technology, EDBT 2021*, OpenPro-

ceedings.org, 2021, pp. 439–444. DOI: 10.5441/002/EDBT.2021.48. [Online]. Available: https://doi.org/10.5441/002/edbt.2021.48.

[34] Z. Wang, V. Bapst, N. Heess, *et al.*, "Sample efficient actor-critic with experience replay," in *International Conference on Learning Representations, ICLR 2017*, OpenReview.net, 2017. [Online]. Available: https://openreview.net/forum?id=HyM25Mqel.

[35] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "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, 2018, pp. 1856–1865. [Online]. Available: http://proceedings.mlr.press/v80/haarnoja18b.html.

[36] Y. Duan, X. Chen, R. Houthooft, J. Schulman, and P. Abbeel, "Benchmarking deep reinforcement learning for continuous control," in *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016*, vol. 48, JMLR.org, 2016, pp. 1329–1338. [Online]. Available: http://proceedings.mlr.press/v48/duan16.html.

[37] O. Anschel, N. Baram, and N. Shimkin, "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, 2017, pp. 176–185. [Online]. Available: http://proceedings.mlr.press/v70/anschel17a.html.

[38] J. Schulman, S. Levine, P. Abbeel, M. I. Jordan, and P. Moritz, "Trust region policy optimization," in *Proceedings of the 32nd International Conference on*

Machine Learning, ICML 2015, vol. 37, 2015, pp. 1889–1897. [Online]. Available: http://proceedings.mlr.press/v37/schulman15.html.

[39] V. Leis, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann, "How good are query optimizers, really?" *Proc. VLDB Endow.*, vol. 9, no. 3, pp. 204–215, 2015. DOI: 10.14778/2850583.2850594. [Online]. Available: http://www.vldb.org/pvldb/vol9/p204-leis.pdf.

[40] B. Ding, S. Das, W. Wu, S. Chaudhuri, and V. R. Narasayya, "Plan stitch: Harnessing the best of many plans," *Proc. VLDB Endow.*, vol. 11, no. 10, pp. 1123–1136, 2018. DOI: 10.14778/3231751.3231761. [Online]. Available: http://www.vldb.org/pvldb/vol11/p1123-ding.pdf.

[41] P. Negi, Z. Wu, A. Kipf, *et al.*, "Robust query driven cardinality estimation under changing workloads," *Proc. VLDB Endow.*, vol. 16, no. 6, pp. 1520–1533, 2023. DOI: 10.14778/3583140.3583164. [Online]. Available: https://www.vldb.org/pvldb/vol16/p1520-negi.pdf.

[42] W. Tan, M. Alhamid, M. Kalil, *et al.*, "Query predicate selectivity using machine learning in db2®," in *Proceedings of the 31st Annual International Conference on Computer Science and Software Engineering*, ser. CASCON '21, ACM, 2021, pp. 143–152. DOI: 10.5555/3507788.3507808. [Online]. Available: https://dl.acm.org/doi/10.5555/3507788.3507808.

[43] S. Das, M. Grbic, I. Ilic, *et al.*, "Automatically indexing millions of databases in microsoft azure SQL database," in *Proceedings of the 2019 International Conference on Management of Data*, ser. SIGMOD '19, ACM, 2019, pp. 666–

679. DOI: 10.1145/3299869.3314035. [Online]. Available: https://doi.org/10.1145/3299869.3314035.

[44] A. Simitsis, P. Vassiliadis, U. Dayal, A. Karagiannis, and V. Tziovara, "Benchmarking ETL workflows," in *Performance Evaluation and Benchmarking, First TPC Technology Conference, TPCTC 2009*, vol. 5895, Springer, 2009, pp. 199–220. DOI: 10.1007/978-3-642-10424-4_15. [Online]. Available: https://doi.org/10.1007/978-3-642-10424-4%5C_15.

[45] M. G. Kendall, "A new measure of rank correlation," *Biometrika*, vol. 30, no. 1/2, pp. 81–93, 1938. [Online]. Available: https://doi.org/10.2307/2332226.

[46] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang, "Automatic database management system tuning through large-scale machine learning," in *SIGMOD*, 2017, pp. 1009–1024.

[47] J. Wang, I. Trummer, and D. Basu, "UDO: universal database optimization using reinforcement learning," *Proc. VLDB Endow.*, vol. 14, no. 13, pp. 3402–3414, 2021. DOI: 10.14778/3484224.3484236. [Online]. Available: http://www.vldb.org/pvldb/vol14/p3402-wang.pdf.

[48] X. Zhang, H. Wu, Z. Chang, *et al.*, "Restune: Resource oriented tuning boosted by meta-learning for cloud databases," in *Proceedings of the 2021 International Conference on Management of Data*, ser. SIGMOD '21, ACM, 2021, pp. 2102–2114. DOI: 10.1145/3448016.3457291. [Online]. Available: https://doi.org/10.1145/3448016.3457291.

[49] I. Trummer, “DB-BERT: A database tuning tool that ”reads the manual”,” in *Proceedings of the 2022 International Conference on Management of Data*, ser. SIGMOD ’22, ACM, 2022, pp. 190–203. DOI: 10.1145/3514221.3517843. [Online]. Available: https://doi.org/10.1145/3514221.3517843.

[50] D. Eriksson and M. Jankowiak, “High-dimensional bayesian optimization with sparse axis-aligned subspaces,” in *Proceedings of the Thirty-Seventh Conference on Uncertainty in Artificial Intelligence, UAI 2021*, ser. Proceedings of Machine Learning Research, vol. 161, AUAI Press, 2021, pp. 493–503. [Online]. Available: https://proceedings.mlr.press/v161/eriksson21a.html.

[51] R. Moriconi, M. P. Deisenroth, and K. S. S. Kumar, “High-dimensional bayesian optimization using low-dimensional feature spaces,” *Mach. Learn.*, vol. 109, no. 9-10, pp. 1925–1943, 2020. DOI: 10.1007/S10994-020-05899-Z. [Online]. Available: https://doi.org/10.1007/s10994-020-05899-z.

[52] X. Tang, S. Wu, M. Song, S. Ying, F. Li, and G. Chen, “Preqr: Pre-training representation for SQL understanding,” in *Proceedings of the 2022 International Conference on Management of Data*, ser. SIGMOD ’22, ACM, 2022, pp. 204–216. DOI: 10.1145/3514221.3517878. [Online]. Available: https://doi.org/10.1145/3514221.3517878.

[53] C. Tsapelas and G. Koutrika, “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, 2024, pp. 307–319. DOI: 10.48786/EDBT.2024.27. [Online]. Available: https://doi.org/10.48786/edbt.2024.27.

[54] J. Sun and G. Li, "An end-to-end learning-based cost estimator," *Proc. VLDB Endow.*, vol. 13, no. 3, pp. 307–319, 2019. DOI: 10.14778/3368289.3368296. [Online]. Available: http://www.vldb.org/pvldb/vol13/p307-sun.pdf.

[55] L. Ma, D. V. Aken, A. Hefny, G. Mezerhane, A. Pavlo, and G. J. Gordon, "Query-based workload forecasting for self-driving database management systems," in *Proceedings of the 2018 International Conference on Management of Data, SIGMOD*, ACM, 2018, pp. 631–645. DOI: 10.1145/3183713.3196908. [Online]. Available: https://doi.org/10.1145/3183713.3196908.

[56] H. Huang, T. Siddiqui, R. Alotaibi, *et al.*, "Sibyl: Forecasting time-evolving query workloads," *Proc. ACM Manag. Data*, vol. 2, no. 1, 53:1–53:27, 2024. DOI: 10.1145/3639308. [Online]. Available: https://doi.org/10.1145/3639308.

[57] N. Heess, D. TB, S. Sriram, *et al.*, "Emergence of locomotion behaviours in rich environments," *CoRR*, vol. abs/1707.02286, 2017. arXiv: 1707.02286. [Online]. Available: http://arxiv.org/abs/1707.02286.