# Batch Query Memory Prediction Using Deep Query Template Representations

**NICOLAS ANDRES JARAMILLO DURAN**

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF APPLIED SCIENCE

GRADUATE PROGRAM IN ELECTRICAL AND COMPUTER ENGINEERING

YORK UNIVERSITY
TORONTO, ONTARIO

MAY 2023

# Abstract

Resource demand prediction is vital to many database optimization tasks such as admission control, database monitoring, query scheduling, resource management, and more [1], [2]. Current techniques are based on statistical models for each operator in the current processing query's plan. Resource demands focus on estimating the demand of a single query *in isolation*, which often fails to capture the net resource demand of a batch of queries, denoted as *a workload.* Estimating the net resource demand of a collection of queries is a challenging task due to the difficulty of capturing the correlation between different queries [3].

We introduce the novel problem of *workload memory prediction* and formalize it as a *distribution regression problem.* A methodology for estimating the collective resource demand of a batch of queries, we call it LearnedWMP. Specifically, we focus on predicting the memory cost demand of a batch of queries. LearnedWMP makes use of existing components of the optimizer engine, such as the query plan. We group queries with similar query plan characteristics into a set of learned query templates. Ideally, these templates are pre-built such that each template represents a group of queries with similar resource demands. Thereafter, the system generates a histogram representation of the query templates in the workload. Lastly, a regressor uses the histogram representation of the workload to predict the resource demand(e.g., Memory cost) of the workload. In our experimental settings, we use three database benchmarks to demonstrate a 47.6% improvement of the memory estimation in comparison to the existing state-of-the-art. We also compare our approach to different machine and deep learning techniques, which learn the resource demand of individual queries.

Through our experiment comparison, we demonstrate that our model was 3x to 10x faster and at least 50% smaller when compared to a single query prediction approach.

# Acknowledgments

I want to express my profound gratitude to my supervisors, Dr. Marin Litoiu and Dr. Manos Papagelis, for their support and guidance throughout my graduate studies. I was very fortunate to have had such great supervisors with such deep and extensive knowledge. Thank you for your continuous patience, encouragement, and guidance over these past years.

I am also grateful to Ph.D. student Shaikh Quader; it was a fantastic experience and a privilege to have worked and learned from you.

I wish to thank the members of my supervisory committee: Dr. Marin Litoiu, Dr. Manos Papagelis, Dr. Hamzeh Khazaei, and Dr. Xiaohui Yu for volunteering their time to read and examine this thesis.

Lastly, I would like to thank the Lassonde faculty for their teachings and guidance; specifically, I would like to extend my gratitude to Aijun An, Jarek Gryz, and Ruth Urner. They made their classes fascinating, and their passion for their field of expertise opened up different avenues of interest for me.

# Co-Authorship

This thesis is based on my upcoming published work listed below. In these publications, I contributed in collaboration with Shaikh Quader in the following ways: materializing the initial research idea, researching related work, conducting experiments and analyzing the resulting data (developing required tools), writing the paper's drafts, and contributing to the final paper. Parts of this thesis will be published as follows:

- Shaikh Quader, Nicolas Andres Jaramillo Duran, Sumona Mukhopadhyay, Calisto Zuzarte, David Kalmuk, Marin Litoiu, Manos Papagelis, LearnedWMP: Database Workload Memory Prediction Using Distribution of Query Templates

It is noteworthy to mention that the thesis in question has been granted a patent [4], indicating that the research carried out is recognized as possessing novelty and innovation. Patents are legal documents that afford inventors exclusive rights to their inventions for a defined duration. In this case, the patent pertains to the particular technique or technology that was developed during the research process. The granted patent information is listed below:

- Quader, Shaikh Shahriar, Nicolas Andres Jaramillo Duran, Sumona Mukhopadhyay, Emmanouil Papangelis, Marin Litoiu, David Kalmuk, and Piotr Mierzejewski. "Learning-based workload resource optimization for database management systems." U.S. Patent 11,500,830, issued November 15, 2022.

# Table of Contents

# List of Tables

# List of Figures

# Abbreviations & Acronyms

| | |
|---|---|
| **BOW** | Bag of Words. |
| **CCAEO** | Cardinality Cost Aggregation for Aach Operator. |
| **CPU** | Central Processing Unit. |
| **DBA** | Database Administrators. |
| **DBMS** | Database Management Systems. |
| **DBSCAN** | Density-Based Spatial Clustering of Applications With Noise. |
| **DFS** | Depth-First Search. |
| **DL** | Deep Learning. |
| **DNN** | Deep Neural Network. |
| **IQR** | Interquartile Range. |
| **JOB** | Join Order benchmark. |
| **LearnedWMP** | Learned Workload Memory Prediction. |
| **MAPE** | Mean Absolute Percent Error. |
| **ML** | Machine Learning. |
| **MLP** | Multilayer Perceptron. |

| | |
|---|---|
| **NLP** | Natural Language Processing. |
| **NN** | Neural Network. |
| **OLAP** | Online Analytical Processing. |
| **OLTP** | Online Transactional Processing. |
| **QPEWMTS** | Query Plan Encoding While Maintaining Tree Structure. |
| **RELU** | Rectified Linear Unit. |
| **RL** | Reinforced Learning. |
| **RMSE** | Root Mean Square Error. |
| **SGD** | Stochastic Gradient Descent. |
| **SingleWMP** | Single Workload Model Predictor. |
| **SQL** | Structured Query Language. |
| **tanh** | Hyperbolic tangent. |
| **TPC-C** | Transaction Processing Performance Council Benchmark C. |
| **TPC-DS** | Transaction Processing Performance Council Decision Support. |
| **XGBoost** | Extreme Gradient Boosting. |

# Chapter 1

# Introduction and Background

## 1.1 Introduction

Today, numerous companies in different industries are now using data for decision-making, accelerating innovation, improving customer experience, and much more. For example, banks generate an astronomical amount of data for everyday activities such as banking transactions, behavior patterns, customer information, and more. The Internet of Things (IoT) devices collect and share data for optimizing performance to aid us in everyday life. E-commerce data plays an essential key role in tracking consumer shopping behavior. Self Driving Cars combine a variety of sensors to perceive their surroundings, such as radar, computer vision, sonar, Global Positioning System (GPS), and more. The car's data must be stored and analyzed to determine the best present or future course of action. To fully leverage their data, these industries require a dependable database system that can quickly store and collect data for future accurate and fast access. However, achieving fast query processing, which is an essential requirement for any DBMS, can be a challenging task for Database Administrators (DBAs) who need to manually tune the system to improve query performance. As a result, researchers have proposed various approaches to optimize query performance and alleviate the burden on DBAs, including machine learning-based approaches.

This research aims to predict resource demand, with a specific focus on memory cost. Accurately predicting memory usage is critical for optimizing database performance and resource allocation. Currently, state-of-the-art DBMS typically uses static resource estimation models for each query's operator in the query plan. When a SQL query arrives in the system, it is decomposed into low-level database operators (e.g., GROUP BY). A static model computes many low-level features for each operator and then estimates the operator's resource utilization. At the query level, all the unit operators' resource estimations are aggregated to come up with the resource demand for an individual query. These models use a set of heuristics that make assumptions about the system and therefore do not adapt well to the database configuration and system changes.

An active commercial database management system continuously receives an extensive stream of query requests. For commercial database management systems to be efficient, they must process the maximum number of SQL queries while using all of the available computing resources in the system. For this purpose, the database management system must find an efficient method of deriving the resource estimation of the database workload - the collection of queries that are to be executed concurrently on the system. For example, a DBMS can only simultaneously execute a finite number of queries due to its limited system memory. Inaccurate memory estimations can produce a performance bottleneck in the system when the system under or over-commits its memory's capabilities. To achieve low query latency and high throughput, the DBMS needs an accurate estimation of the memory utilization of the queries before admitting them for execution.

In this research, we address the problem of predicting resource demand for database queries, specifically within the context *memory cost of a workload.* Current methods for estimating memory demand for individual queries often fail to capture the net memory demand of multiple queries in a batch. We formalize this problem as a *distribution regression problem* and propose a novel approach, called LearnedWMP, for *Learned Workload Memory Prediction* . Unlike single-query based estimation approaches, LearnedWMP does not require memory estimations of individual database operators. Instead, it groups queries within a workload based on query plan similarity, generating a histogram representation of the query templates. This representation is then used to train a regressor that predicts the memory demand of the workload. In an experimental evaluation using three database benchmarks, LearnedWMP was able to reduce the memory estimation errors of state-of-the-art practices by 47.6%. In addition, we conducted a comparison between LearnedWMP and an alternative machine learning method that involves training memory estimation models for single queries. Our results indicate that the LearnedWMP models were significantly faster, with 3x to 10x faster training and inferencing times, as well as being at least 50% smaller in most cases. These findings demonstrate the benefits of the LearnedWMP approach and its potential to have a broader impact in the field of database optimization.

## 1.2 The State of the Practice & Limitations

In contemporary DBMS systems, the query optimizer's cost model relies on simplified assumptions for deriving cardinality estimates. These assumptions include assuming that distinct column values are uniformly distributed for join cardinality estimation and that columns are independent of each other without advanced statistics. However, these assumptions are often inaccurate, leading to inaccurate memory estimations [5]. Furthermore, traditional DBMS methods derive estimations for each query separately [1], without considering the fact that concurrent queries may stress certain resources, such as memory, on the system. For example, when a DBMS processes two concurrent queries, each with a `group by` operation, their collective memory demand will be higher than if they were executed separately. When the collective memory requirement exceeds the available system memory, some of the queries may take longer to finish, resulting in increased query execution time and decreased DBMS throughput. Currently, human experts are relied upon to define the features and rules for calculating runtime metrics of queries [6]. However, this approach is expensive, and these estimation rules do not generalize well to unseen workloads and new operational environments of the DBMS.

## 1.3 The State of the Art & Limitations

In recent years, there has been a growing body of research that proposes the use of deep learning-based methods for addressing query optimization problems [7], [8], particularly for cardinality estimation [8]–[12], which is closely related to memory usage estimation. These methods show promise as they can learn hidden patterns in the data and understand relationships between interdependent variables. However, there is still a lack of evidence that these methods are suitable for enterprise-grade DBMS instances [7]. This is because they are typically evaluated on simplified benchmark datasets that may not accurately reflect the characteristics of complex enterprise datasets [7]. Furthermore, these methods often require complex hyperparameter tuning, which can limit their reproducibility and interpretability. Kim et al [13] provides a comprehensive comparative analysis of learned cardinality estimators for single queries and their limitations. It should be noted that deep learning-based models designed for single query memory estimation may not generalize well to the workload memory prediction problem.

## 1.4 Our Approach

In this study, we propose a novel approach for estimating the resource usage of a batch of database queries, referred to as a workload. This approach departs from the traditional method of estimating the resource usage of each query separately and instead focuses on modeling the resource demand of concurrent queries. By doing so, we aim to achieve higher accuracy in estimating resources and provide additional benefits, such as reducing development and maintenance costs of the DBMS's resource estimators and speeding up the computation of resource estimation. For this research, we initially focus on estimating memory usage, an important resource type, of database workloads. To this end, we design a Learned Workload Memory Prediction (LearnedWMP) model in three steps.

*First*, we use historical queries to learn query templates that serve as groups for queries with similar memory demands. We based this on the intuition that queries with similar plan characteristics and estimated cardinalities have similar memory demands.

*Second*, we randomly divide training queries into fixed-size training workloads and represent each workload as a *histogram* or *distribution over query templates*. This histogram-based representation allows us to capture the underlying statistical distribution of the queries by grouping them into bins or templates. Histograms have been utilized in various domains to aggregate multiple observations and obtain approximate data distributions [14]. To reduce the complexity of the experiment setup, the current design of LearnedWMP uses fixed-length workloads. However, the design can easily be extended to work with variable-length workloads.

*In the third and final step*, using training workloads and their historical collective actual memory usage, we learn a regression function that can estimate the memory usage of an unseen workload based on its distribution of query templates. As the learning algorithm has access to diverse training workloads, its accuracy at estimating the collective workload memory will improve over time. A production DBMS can use a trained LearnedWMP model to predict the memory demand of unseen workloads.

## 1.5 Contributions

We summarize our key contributions as follows:

- We introduce a novel problem of *workload memory prediction*. To the best of our knowledge, this is the first attempt to predict a workload's memory demand using machine learning techniques.

- We formulate the problem of workload memory prediction as *a distribution* *regression problem*, which learns a regressor function from workloads represented as distributions of query templates. We use machine learning techniques to learn the regressor without relying on the hand-crafted query-level or operator-level features.

- We propose LearnedWMP, a novel prediction model that can estimate the memory demand of a batch of SQL queries called a workload. This is a departure from the state of the practice and the state-of-the-art methods, which estimate the memory demand of each query separately.

- We devise unsupervised machine learning methods to group queries of similar memory needs to reduce the computational overhead of workload memory usage estimation significantly.

- We present a thorough experimental evaluation of our LearnedWMP model employing three database benchmarks, including two OLTP transactional workloads and one OLAP analytical workload. In the evaluation, our model reduced the memory estimation errors of DBMS by at least 47.16%. These experimental results prove the merit of our proposed technique in workload-based query processing and resource estimation.

# Chapter 2

# Literature Review

The Application of machine learning (ML) techniques for improving databases has become more popular due to advances in large-scale data, novel methods, and high processing power [2]. These improvements have provided additional opportunities for the research community to improve the autonomic capabilities of current approaches [15]. Tuning a database management system (DBMS) to increase run-time performance can be costly and rigorous as it requires extensive knowledge of the system and experts such as Database administrators (DBAs) to carry out the task. This endeavor has been made considerably more difficult by the complexity of implementing new data-driven applications and services online. Many of the complexities and expenses of DBMS tuning may be reduced if the DBMS could optimize itself without human interaction, resulting in an autonomous DBMS [16].

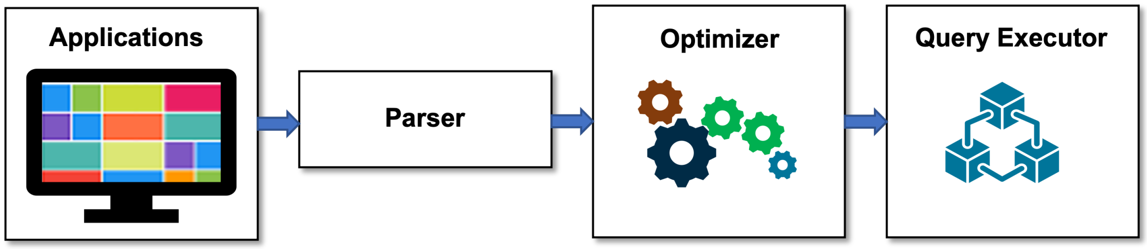


Figure 2.1: DBMS

Standard DBMS consists of a parser, query optimizer, and query execution. The parser is in charge of parsing the submitted SQL query and checking if the syntax is correct. If the submitted SQL query is correct, it generates the parser tree and submits it to the query optimizer. The query optimizer is responsible for creating an optimal process for executing the query called a query plan. After the plan is created, it is sent to the query execution engine, where the query will be executed and the data retrieved. Even today, current DBMS can select a poor execution plan[17].

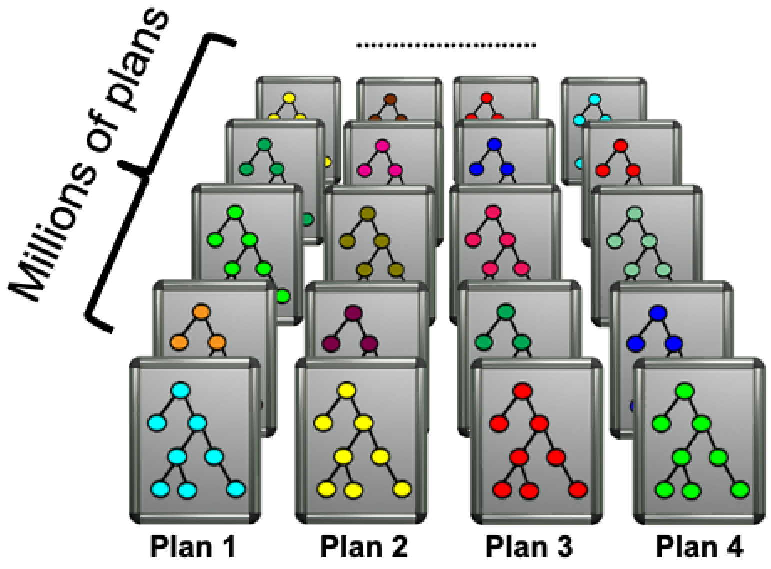


Figure 2.2: Query Plans

Due to their importance, much research has been conducted to improve the parser and the query optimizer. The parser is important because writing these SQL queries can be challenging, require expertise, and is prone to mistakes that can hinder performance [18] [2]. The query optimizer is responsible for selecting the most efficient way to execute a given query. The optimizer functions by evaluating different query plans and choosing the plan with the lowest estimated cost, but this can be a challenging task as the search space for picking the optimum query plan can be in the millions. To generate the optimal execution plan, the query optimizer takes into account factors such as available indexes, table size, distribution of data, query complexity, and more.

## 2.1 ML for Database

Research into the query optimizer for producing a query plan focused on a number of areas, including but not limited to 1) *cardinality estimation* 2) *database configuration tuning* 3) *database indexing selection* 4) *arrival rate prediction* 5) *query performance prediction* 6) *resource cost estimation* 7) *Query-based Workload Analysis* and more. This work focuses on examining the potential of machine learning and deep learning techniques to enhance the query optimizer's capability to predict memory resource demand for queries.

### 2.1.1 ML for Cardinality Estimation

One of the critical aspects of the optimizer is the ability to predict the execution cost of the query, and an accurate cardinality estimation at all stages of a query plan is needed to predict this cost [10]. Cardinality estimation is the process of correctly estimating the number of tuples each sub-query will produce. It refers to how unique a specific attribute's values are in the database table instance. This means that the greater the cardinality of an attribute, the more unique the value is. However, estimating the cardinality can be challenging, especially if the query contains numerous join operators [19]. Recently, the research community has used different machine learning techniques to predict the cardinality estimation of the query. Kipf et al [11] uses the query features from the query plan to predict the query's cardinality estimation. Specifically, they use a CNN that learns to predict the join crossing correlations in the data while managing known weak points of existing sampling-based techniques. Ortiz et al [19] uses encoding techniques to encode a query into a single-dimension vector that is used as the input to a deep neural network model. It encodes the query into a vector containing three sections representing the query's selection, join, and relation. For the model, they leverage a Recurrent Neural Network's (RNN) ability to handle sequential data. Normally, the input to an RNN is a sequence of time steps, so they model the queries as a set of action steps where each action represents a query operation. It then goes on to experiment with the trade-offs between the estimation error and model size. Liu et al [10] uses a 3-layer augmented Neural Network (NN) to learn the selectivity function, in which the input is a bound range on each column. This model can effectively estimate the selectivity of all relational operators. A query optimizer uses the selectivity metric to justify applying an index to specific table attributes. A general optimization guide is to apply an index when the selectivity is high, meaning the cardinality is high. Hasan et al [12] presents two complementary approaches for the problem of estimating query selectivity in databases which are effective for multi-attribute queries with a large number of predicates and low selectivity. The first approach models selectivity estimation as a density estimation problem and uses techniques from neural density estimation to build an accurate estimator. The second approach formulates selectivity estimation as a supervised deep learning problem. The authors also address practical challenges when adapting deep learning for relational data, such as query/data featurization, query workload information, and the dynamic scenario where both data and workload queries could be updated. Hilprecht et la [8] introduce Relational Sum-Product Networks (RSPNs), a new deep probabilistic modeling approach for databases aimed at capturing key characteristics of a database. The model captures the joint probability distribution of the data and its correlations across attributes while supporting direct updates to the database. The model devises a probabilistic query compilation approach that translates database queries into probabilities and expectations for RSPNs. The authors evaluate the performance of the data-driven approach by implementing the DeepDB DBMS architecture.

### 2.1.2 ML for Database Configuration Tuning

Database systems require hundreds of knob-turning configurations to achieve optimum results and high-performance [20]. It is an NP-hard problem where current existing methods cannot solve the problem because of certain limitations [21]. Moreover, traditional methods leverage DBAs to manually tune and configure these knobs, requiring extensive experience and knowledge of the system. Additionally, these knob-tuning settings can take days to weeks to set up, which can hinder productivity, and it is not optimal when DBAs have to set up many database instances on the cloud. Due to this, The research community has been trying to leverage machine learning techniques to automate this database knob tuning for this problem[2].

Van et al [22] implemented OtterTune, which uses both supervised and unsupervised machine learning techniques to select and recommend best knob-tuning configurations. OtterTune collects running statistics which are recorded once the query is executed. Next, they use K-means clustering to narrow down the space of the metrics collected during run-time. Lastly, OtterTune uses Lasso (Liner Regression) model to chose the knobs that have the strongest correlation to the query's performance. Li et al [21] created a query-aware database tuning system called QTune. It uses reinforced learning, deep learning, and unsupervised learning to predict the database tuning requirements. Qtune clusters the queries such that all the queries in the cluster have similar tuning properties. For this task, Qtune uses the query plan to vectorize the query, which becomes the input to a reinforced learning model. The model learns a configuration pattern for each input query. Next, the system uses deep learning to build a second vector that maps the configuration patterns' queries. Once this new vector has been created, the authors use DBSCAN to cluster the queries together.

### 2.1.3 ML for Database Indexing Selection

The goal of the index selection process is to determine what available attributes of the database schema will be selected for creating secondary indexes. Index selection can result in a significant reduction of a query's execution cost. However, most state-of-the-art index tuning systems depend on accurate cost estimation from the query optimizer for good index recommendations, which are subject to well-known limitations such as cardinality estimation inaccuracy [23].

Index selection has been an active research area where machine learning has been used for this recommendation. Query2Vec from Jain et al [24] uses supervised and unsupervised machine learning techniques to achieve workload summarization for index selection. In this work, workload summarization refers to summarizing a large set of query logs. Query2Vec first vectorizes the query for the input to the clustering algorithm. Query2Vec uses LSTM to learn the vector representations of the SQL queries; this has the added benefit of constructing vectors where similar vectors are closer to each other. Once the vectors are built, Query2Vec uses k-means to cluster similar queries together. The process of workload summarization is beneficial as it enables the identification of a representative subset of queries from a given workload. This subset can then be used to optimize the creation of indexes that improve the performance of these queries and, in turn, enhance the overall performance of the workload. Ding et al [23] improves index selection by iteratively generating hypothetical configurations and checking if the new configuration improved the previous configuration through the query plans generated by the configurations. This query plan comparison works by turning a query plan into a vector representation of the query and training a classifier to compare the cost of two query plans. When the model learns on pairs of plans, it can minimize the errors that lead to comparison errors. Therefore, training a classifier to predict which of the two query plans has a cheaper execution cost can derive a higher accuracy for comparing costs.

### 2.1.4 ML for Arrival Rate Prediction

Proper query workload scheduling can significantly help improve performance. For this task, the system must be able to estimate future query workloads correctly [25]. Traditional approaches focus on rule-based methods that use metrics from current workload characteristics. Additionally, these statistics often need to be changed once the workload or physical design of the database varies[25][23][26].

Different works focus on predicting workload arrival rates by using machine learning techniques. Ma et al [25] created the framework Querybot5000 that predicts the expected arrival rate of queries in the future. Their approach uses the logical composition of queries in the workload rather than the amount of physical resources used for query execution to support complex optimization planning decisions. Querybot5000 uses a joined supervised and unsupervised approach for predicting future query arrival rates based on historical arrival rate logs. First, the authors templatize the queries by removing literals and replacing them with the symbol "?". It next uses DBSCAN to cluster the queries based on their arrival rate, such that queries with similar arrival rates are clustered together. The goal of this step is to minimize the space of queries the system has to look at. Lastly, Querybot5000 uses RNN to predict the future arrival rate for the queries in the clusters.

### 2.1.5 ML for Query Performance Prediction

Query performance prediction focuses on predicting the query's latency before execution, meaning how long it will take to process the query at hand [6]. Existing approaches rely upon DBAs to understand database workload characteristics. DBAs need to be able to understand the execution features of queries and apply data mining techniques to understand the nature of the workload [27].

Current works focus on using machine learning techniques to predict the queries performance for one or multiple queries. Marcus et al [6] use a tree-structure neural network architecture where the network's structure matches that of the query plan. Here, the entire query plan is modeled as a tree of neural units where each neural unit is an operator-level neural network. The task of each neural unit is to predict the performance of each operation type and interesting features of the operator, which the parent neural unit can use. Overall, the plan-level neural network is trained to predict the query plan's latency. Zhou et al [3] also try to predict the query performance, but take it one step further by attempting to predict concurrent query performance. The authors aim to capture the correlation between concurrent queries' buffer sharing and lock conflicts. It uses a graph model to encode query features by representing a query plan as a graph where each node is a value of the query plan, and the edge is the correlation between these two values. For example, the shared data which is accessed by both nodes. Once the graphs have been built, the authors use this information to predict the query performance through a graph-embedded network.

### 2.1.6 ML for Query Resource Cost Estimation

Query resource cost estimation focuses on predicting physical resources used to execute a query. These resources can include memory, I/O, CPU, memory, logical page reads, etc. Traditional methods focus on statistical methods for resource estimation or manual cost models. These models fail to capture the improvements made by the optimizer to the query execution plan or capture the physical characteristics of the DBSM [28]. The physical accurate resource estimations of SQL queries are central to many key DBMS operations and goals, including admission control [10], resource management, query optimization [10], and managing user expectations. Sub-optimal resource estimations of the database queries can affect business outcomes, such as losing customers from a slow online storefront. Ganapathi et al [1] goal is to predict the resource usage characteristics (e.g. CPU, memory, disk, and message bytes sent) of queries executed. The study aims to make these predictions using information available before query execution starts, such as the SQL statement or query plan. The study categorizes the queries into three categories (feather, golf ball, and bowling ball). It uses the kernel canonical correlation analysis (KCCA) method to build the predictive model by creating feature vectors for the query and performance data. Performance data is the data collected from executing the data, and it is only used during training. The Gaussian kernel is used as the distance metric in KCCA. Sun et la [29] focuses on an end-to-end solution for cost estimation, which uses a tree structure model. They use feature extractions and encoding techniques that consider both queries and physical operations. Moreover, they extract the features from the query plan and encode them into a tree-structure vector that is taken as an input to the model. Lastly, a tree-structured neural network learns to predict the cost of the query from these vectors. Li et la [28] focuses on constructing a regression tree modules for each type of physical database operator. Each model is trained to predict the resource cost of the operators. Additionally, there is a scaling function that is trained for different resource/operator combinations. Tang et al [30] propose a SQL query cost predictor service that uses machine learning techniques to forecast the CPU and memory resource usage of online queries. The service aims to improve query scheduling by relieving imbalanced online analytical processing workloads in SQL engine clusters. Two machine learning models are trained from historical query request logs for CPU time and peak memory prediction. The raw data is cleaned, discretized, and vectorized using Natural Language Processing techniques, such as Bag of Words (BOW) and TF-IDF (term frequency-inverse document frequency ). Then three types of classifiers (Random Forest, XGBoost, and Logistic Regression) are trained on the transformed data. The evaluation results show that the XGBoost model with the TF-IDF approach outperforms other CPU models.

### 2.1.7 ML for Query-based Workload

Higginson et al [31] describes a method for applying time series analysis techniques, specifically ARIMA and Seasonal ARIMA (SARIMA), to database workload monitoring data with the aim of identifying patterns and trends, including seasonality (reoccurring patterns) and shocks. These findings were then utilized for database capacity planning purposes. Mozafari et al [32] created DBSeer, a system for analyzing transaction logs in a database management system (DBMS) to predict resource utilization. The system first aligns and joins the logs, then categorizes transactions into types based on their SQL statements and table access patterns. These transactions are then clustered using the DBSCAN algorithm and summarized into a workload summary. DBSeer uses linear regression, decision trees, and feedforward neural networks to predict demand for CPU, I/O, and memory.

### 2.1.8 Comparative Analysis of ML Techniques for Databases and LearnedWMP

The present study builds upon prior research that has shown the effectiveness of several techniques used to analyze database queries. Specifically, these techniques involve representing queries using their text or query plans and utilizing both supervised and unsupervised machine learning methods to enhance the database optimizer. It is important to note that previous research in machine learning for databases has focused on addressing different problems, and when resource demand prediction is considered, it has been performed at the level of individual queries. In contrast, the approach proposed in this study estimates resource demand at the level of workload query batches. This change in analysis provides several potential benefits, including increased accuracy and computational efficiency.

Our approach differs from previous query-based workload methods as it does not cluster transactions based on query expressions or table access patterns. Instead, we utilize a set of query templates that are learned from simple features of the query plan.

Our experiments have shown that these features have a stronger correlation with the runtime memory usage of the workload when compared to other methods. Furthermore, we compared the performance of our approach with that of the DBSCAN-based templates and found that $k$-means resulted in a more accurate prediction of resource usage.

## 2.2 ML for Distribution Regression Problems

Recently, there has been substantial attention paid to distribution regression as a general approach for addressing the challenge of supervised learning in situations where labels are provided at the group level, rather than the individual level [33]. For supervised machine learning problems in which accounting for uncertainty in the inputs and model is crucial, distribution regression emerges as a potentially superior alternative to traditional techniques, such as random forests and neural networks [34]. To the best of our knowledge, we are the first instance in which distribution regression has been utilized for modeling resource demand forecasts of database workloads. Distribution regression is a statistical method used in supervised learning for situations where labels are available at the group level, rather than at the individual level. It involves modeling the distribution of the target variable for each group rather than modeling the individual target values. This allows the model to account for uncertainty in the observations and estimate the distribution of the target variable rather than just a single-point estimate. Distribution regression has been used in various applications, such as predicting health indicators from a patient's list of blood tests [35], solar energy forecasting [34], and traffic prediction [34]. Law et al [33] proposes a set of Bayesian methods for distribution regression. The work uses point estimates for the input variables and accounts for uncertainty in the regression model through Bayesian linear regression. It considers uncertainty in the input variables and uses a Bayesian mean shrinkage model to make predictions based on a sparse representation of the target function. lastly, it combines the two previous methods into a fully Bayesian model and uses Hamiltonian Monte Carlo for inference. Li et al [34] proposes an ensemble learning method for conditional density estimation. It transforms the problem into a multi-class classification task using distribution regression, where different machine learning algorithms can be used, such as neural networks. Mao et al [35] specifies a coefficient-based regularized method for learning distribution regression functions using indefinite kernels.

# Chapter 3

# Preliminaries and Problem Definition

In this section, we introduce notation and preliminaries to help to define the novel *workload memory prediction problem.* Next, we formally present our approach to model and solve the problem as *a distribution regression problem.*

**Definition 1** ***(Query)*** *Let $q = (e, p, m)$ be a single SQL query where (*i*) $e$ is a query expression, provided by either a user or an application, (*ii*) $p$ is a query execution plan generated by the query optimizer of a DBMS for evaluating $e$, and (*iii*) $m$ is the actual highest memory usage of $e$ after the execution of $p$ in the DBMS. $m$ is available only for training queries that the DBMS has already executed. We assume that each query is executed in isolation. For unseen queries, $m$ is unknown.*

**Definition 2** ***(Workload)*** *Let $w = (\mathcal{Q}, y)$ be a workload, which consists of (*i*) $\mathcal{Q}$, a set of queries where $q_i \in \mathcal{Q}$ is a tuple $(e_i, p_i, m_i)$, as per def. 2.1, and (*ii*) $y$ is the sum of actual memory utilization of all queries in $\mathcal{Q}$ after the DBMS executes them.*

$$y = \sum_{i=1}^{|\mathcal{Q}|} m_i \tag{3.1}$$

$y$ value of a workload (eq. 3.1) is available only during the training phase, as part of a labeled training set. During the inference phase, LearnedWMP will receive only Q, a collection of queries, but not $y$.

**Problem 3 (Workload Memory Prediction)** *Let us assume a training corpus of n workloads, as follows:*

$$\{(w_1, y_1), \ldots, (w_n, y_n)\} \tag{3.2}$$

*Here, each tuple, $(w_i, y_i)$ corresponds to the collective historical memory utilization $y_i$ of all queries in the workload $w_i$. Now, given an unseen workload $w$, we wish to learn a predictor function $\hat{f}(\cdot)$ that can accurately estimate the workload $w$'s collective memory usage $y$:*

$$\hat{f}(w) = y \tag{3.3}$$

## 3.1 Modelling of the Problem as a Distribution Regression Problem

**Definition 4** ***(Query templates)*** *Let $\mathcal{T} = \{t_1,...,t_k\}$ be a set of $k$ query templates. A query template $t_i \in \mathcal{T}$ represents a class of queries with similar plan characteristics and memory requirements. As such, any query $q$ can be mapped to a query template $t_i \in \mathcal{T}$.*

**Definition 5** ***(Workload histogram)*** *Let $w$ be a workload, which consists of a set of $\mathcal{Q}$ queries. $c_i$ is the number of queries in $\mathcal{Q}$ that can be mapped to query template $t_i \in \mathcal{T}$. The counts of queries in $\mathcal{Q}$ that map to different query templates in $\mathcal{T}$ are recorded in a 1-d vector of length $k = |\mathcal{T}|$. We call this vector a* workload histogram *$\mathcal{H}$. Here, $\mathcal{H} = [c_1, ..., c_k]$ and*

$$\sum_{i=1}^{k} c_i = |\mathcal{Q}| \tag{3.4}$$

We formulate estimating memory usage of an unseen workload as a distribution regression problem **Gretton**, [36], where the estimate is computed from an input probability distribution - the distribution of queries $\mathcal{Q}$ among templates $\mathcal{T}$. From such an input distribution, encoded in a *workload histogram*, a distribution regression function computes as estimated memory usage for the workload. Let us assume we have a training corpus of $n$ workload histograms, one for each workload, as follows:

$$\{(\mathcal{H}_1, y_1), \dots, (\mathcal{H}_n, y_n)\} \tag{3.5}$$

Here, each tuple, $(\mathcal{H}_i, y_i)$, corresponds to a single workload; $\mathcal{H}_i$ is the workload histogram and $y_i$ is the collective historical memory utilization of all queries in the workload. On the workload histogram, we assume:

1. The distribution of queries among the query templates (i.e., the workload histogram bins) is uniform.

2. The query templates are independently and identically distributed.

3. An underlying function, $f(\cdot)$, exists that can accurately compute any workload's memory usage, $y$, from the workload histogram, $\mathcal{H}$.

   $$f(\mathcal{H}) = y \tag{3.6}$$

   We, however, neither know $f(\cdot)$ nor have access to the set of all possible workload examples to derive $f(\cdot)$.

Using distribution regression, we wish to learn a function, $\hat{f}(\cdot)$, an approximation of $f(\cdot)$. From the input *workload histogram*, $\mathcal{H}$, of a workload, $\hat{f}(\cdot)$ can compute $\hat{y}$, an accurate estimate of the actual memory usage $y$.

$$\hat{f}(\mathcal{H}) = \hat{y} \tag{3.7}$$

Using training workloads labeled with their actual memory utilization, $f\hat{(\cdot)}$ learns to estimate the memory demand of unseen workloads. We expect that the larger and more diverse the training data set of workload examples are, the more precise the predictor $f\hat{(\cdot)}$ will be. Table 3.1 provides a summary of the key notations.

Table 3.1: Summary of key notations

| **Notation** | **Definition** |
|---|---|
| $w$ | A workload. |
| $n$ | The number of workloads. |
| $\mathcal{Q}$ | The set of queries in a workload. |
| $\mathcal{T}$ | The set of query templates in the DBMS. |
| $\mathcal{H}$ | $\mathcal{H} \in \mathbb{R}^k$ is a workload histogram, representing the distribution of queries in a workload $w$ over the $k$ query templates $\mathcal{T}$. |
| $\hat{f(\mathcal{H})}$ | The learned function (predictor) that predicts the memory demand of an input workload histogram $\mathcal{H}$. |
| $c_i$ | The number of queries in a workload $w$ that are mapped to a query template $t_i \in \mathcal{T}$. |
| $y$ | The actual collective historical memory utilization of all queries $\mathcal{Q}$ in a workload $w$. |
| $\hat{y}$ | The predicted collective memory demand of all queries $\mathcal{Q}$ in an unseen workload $w$ as estimated by $\hat{f(\cdot)}$. |

# Chapter 4

# Methodology

This section provides an overview of how our system Learned Workload Memory Prediction (LearnedWMP) can predict the memory resource demand of a batch of queries. Our methodology is formulated into three main steps.

1. **Phase 1: Learning Query Templates.** A commercial database system's query log can have diverse types of large volumes of queries. Storing and processing these queries for building a predictive model can be expensive. We look at templatizing the queries to reduce our model's time and space complexity.

2. **Phase 2: Constructing Histograms from Workloads.** The input to the model is a vector histogram representation of the query distribution in the workload. Therefore, we look at binning the workload's queries to represent the query workload as a histogram - a distribution of query templates.

3. **Phase 3: Training a Distribution Regression Deep Learning Model.** The third step of our methodology focuses on training a deep learning model to predict the resource cost of the workload.

## 4.1 High-level Overview of the ML Pipeline

The LearnedWMP model has two stages: training and inference. The training stage uses a machine learning training pipeline and a training dataset to build a Learned-WMP model. The inference stage uses the trained LearnedWMP model to predict memory usage for unseen workloads. Figure 4.1 and 4.2 shows an overview of the LearnedWMP workflow.

### 4.1.1 Users and the Database

The left section of figure 4.2 shows the interaction between users and a database and the database's interaction with two LearnedWMP stages. The users, human users and applications, send SQL queries to the database. The database processes their queries and returns responses. During this excecution, the database records the query expression, the query execution plan, and the actual memory utilization in a query log. Periodically, the LearnedWMP training pipeline (Left section of figure 4.1) uses the latest dump of the query log for re-training the model. In this section, we first briefly describe the steps of this workflow and then discuss the technical details of each step.

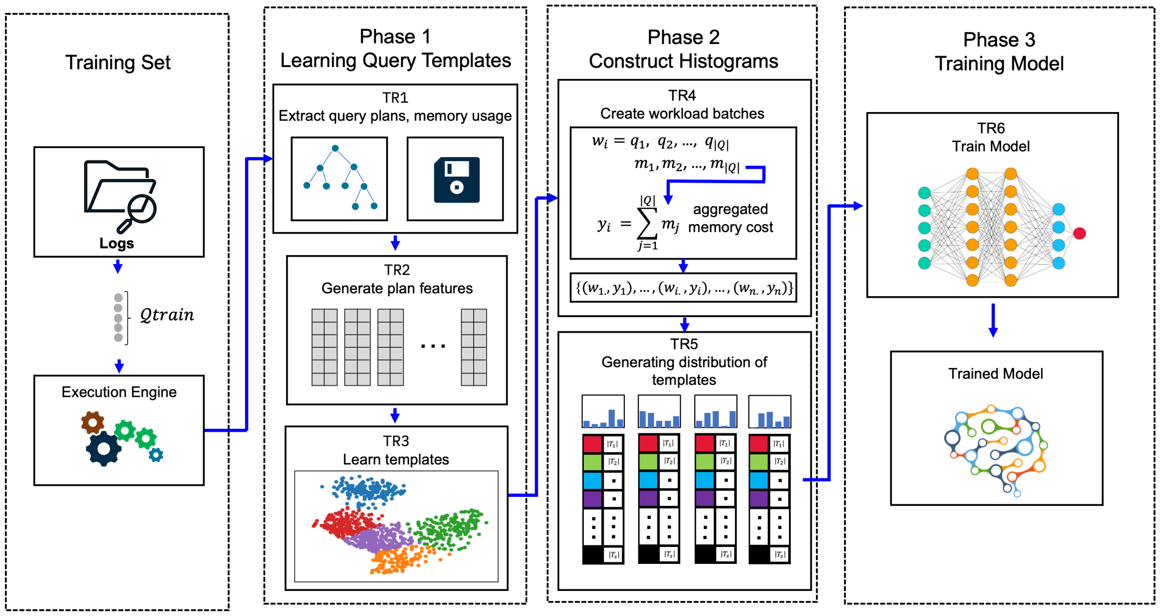


Figure 4.1: LearnedWMP: Training Stage

### 4.1.2 Training Stage

In figure 4.1, `TR1` through `TR6` are the steps of the training pipeline. Training begins with a set of training queries, $\mathcal{Q}_{train}$, collected from a dump of the DBMS query log. At `TR1`, the pipeline extracts training queries, their final execution plans, and the actual memory usage from the past execution. At `TR2`, from the query plans, the pipeline generates a set of $m$ features to represent the training queries as a $|\mathcal{Q}_{train}| \times m$ feature matrix. Here we use the cardinality cost aggregation for each operator to extract the $m$ features (cf. section 4.2.2). At `TR3`, the pipeline learns $\mathcal{T}$, a set of $k$ query templates, from the query feature matrix. We use $K$-Means to learn the templates and determine $k$ experimentally (cf. section 4.2.2). At `TR4`, the pipeline equally divides the training queries of $\mathcal{Q}_{train}$ into a set of $n$ workloads, $\mathcal{W} = |\mathcal{Q}_{train}|/s$. Each workload contains $s$, a constant, queries. We found a value of $s$ experimentally (cf. subsection 5.2.3). At `TR5`, the pipeline generates a workload histogram $\mathcal{H}$ for each training workload $w = (\mathcal{Q}, y)$ in $\mathcal{W}$. $\mathcal{H}$ represents the distribution of queries of $\mathcal{Q}$ among $k$ query templates of $\mathcal{T}$. In addition, the collective actual memory utilization

$y$ of the workload $w$ is computed by summing up the memory utilization of each query $q_i \in \mathcal{Q}$. Each ($\mathcal{H}$, $y$) pair represents a supervised training example for training a regression model. At `TR6`, the model receives as input a large collection of training examples of the form ($\mathcal{H}$, $y$). From these examples, the model learns a regression function $\hat{f}(\mathcal{H})$ to map an input histogram $\mathcal{H}$ to its memory demand, $y$. At the end of `TR6`, the training pipeline produces a trained LearnedWMP model.

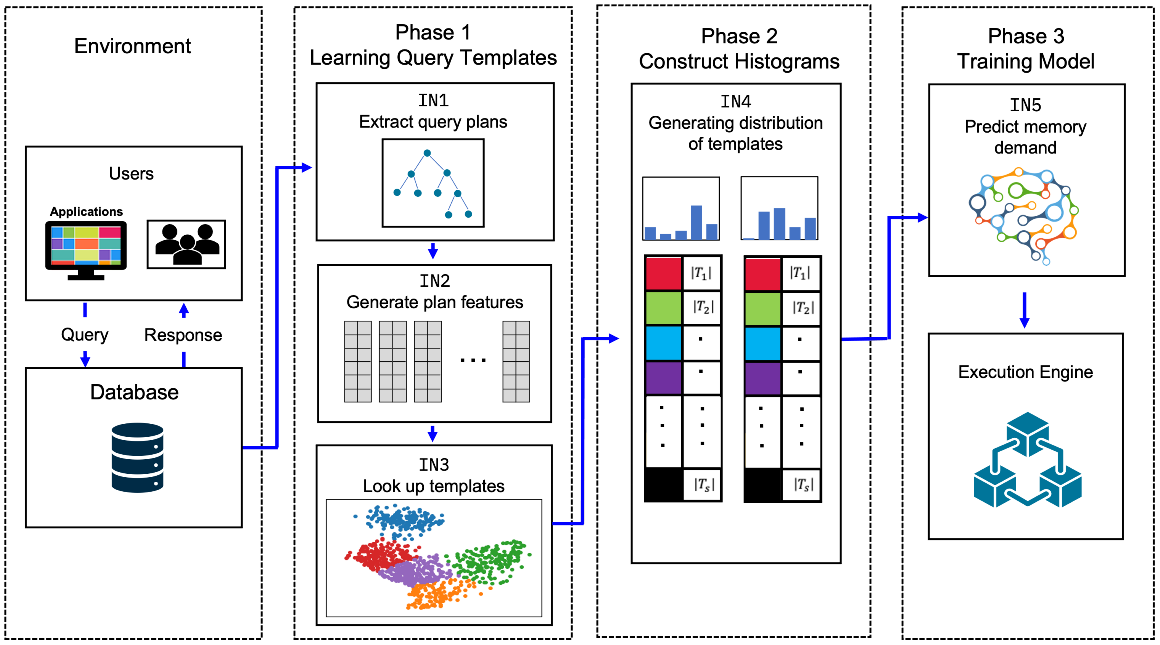


Figure 4.2: LearnedWMP: Inference Stage

### 4.1.3 Inference Stage

In figure 4.2, `IN1` through `IN5` are the steps of the LearnedWMP inference pipeline, which generates estimated memory usage of an unseen workload $w$, consisting of $\mathcal{Q}$ queries. Step `IN1` collects the query plans of the queries $\mathcal{Q}$ in $w$; step `IN2` generates the feature vectors for these plans. Step `IN3` assigns each query $q_i \in \mathcal{Q}$ to a template $t_i \in \mathcal{T}$, from which `IN4` constructs a workload histogram $\mathcal{H}$. The final step, `IN5`, uses the histogram $\mathcal{H}$ as input to the LearnedWMP model and predicts the memory usage $\hat{y}$ of the workload, $w$.

## 4.2 Training Stage Phase 1: Learning Query Templates

Collecting database access logs is standard practice and can be used in many settings; however, logs from DBMS can be very large. Therefore, building Machine or Deep Learning models on these logs would require a considerable computing footprint. In data processing systems such as online transactional processing (OLTP) and online analytical processing (OLAP), queries are frequently created by applications that interact with the user rather than the user themselves. OLTP queries typically involve executing the same queries while varying input parameters. Meanwhile, OLAP queries are generally constructed by users who engage with dashboards or reporting tools. These tools create queries using different parameters and predicates. As a result, both OLTP and OLAP queries tend to require similar levels of resource utilization within the system [25]. In phase 1, we aim to identify the $|\mathcal{T}|$ number of templates from all queries in the database access logs. By intuition, we expect queries associated with the same template to have similar resource requirements during execution. However, the templatization scheme doesn't need to be highly accurate. Rather, a fast and decent performance in finding loosely close queries regarding resource requirements is sufficient for the downstream resource prediction task. We are not looking for an optimal template assignment, which will require computation of operator-level features, increase computation overhead, and outweigh the acceleration we hope to gain from compressing queries into templates. Overall, estimating the cost of individual queries is a separate research problem [1], [28] that we are not addressing in our current research. Instead, we rely on a best-effort algorithmic principle for assigning each query to a template.

The database community has widely used query log optimization for different database optimization tasks. For example, index selection, query processing, histogram tuning, and statistics selection [37] [38], and there are different approaches to accomplishing these tasks. This work explores two main approaches:

1. Learning rule-based query templates

2. Learning clustering-based templates

Both of these methods fall under the category of query compression techniques that are commonly employed by the database community. This section aims to provide a comprehensive explanation of these two primary approaches and outline their respective features. Afterward, in the experimental section, we will present a comparative analysis of their results and an analysis of their capabilities.

### 4.2.1 Learning Rule-based Query Templates

To build the rule-based query templates, one or more rules are associated with a query statement of a predefined template. Typically, database benchmarks come with a predetermined collection of templates that we can use to create queries. Each template has a fixed query corpus where only the constants and literals are randomly generated. If a benchmark has a large set of predefined templates, we map the generated queries back to their corresponding templates. For database benchmarks where the queries are generated based on a limited set of query templates. We change our strategy as following the same approach would result in a limited number of templates, which could negatively impact the prediction accuracy. Therefore, for these types of benchmarks, we choose to group these queries based on their estimated cardinality similarity, where each template consists of queries with similar cardinality estimation. Overall, learning rule-based query templates are often created by DBAs, database researchers, and experiments [3].

### 4.2.2 Learning Clustering-based Query Templates

Clustering algorithms belong to the class of unsupervised machine learning methods. They work by dividing a set of input data points into homogeneous groups called clusters [39]. Data points within the same clusters are expected to be similar, while data points in different clusters are expected to be dissimilar. In the training phase of our approach, we utilize a clustering algorithm to construct templates. During the inference phase, we utilize the same algorithm to map new queries to the appropriate cluster. By doing so, we are able to efficiently categorize new data points based on their similarity to existing data points within a given cluster.

#### Clustering Features from SQL Query

To identify if a set of points belong to the same cluster, the clustering algorithm needs to use a set of features to compute the distance between the data points. In this section, we focus on features that can be collected before execution time and not post-execution. Post-execution features are called *physical* features; they can provide accurate information about how a query will behave in execution but depend on the DBMS hardware, and content [25]. For our case, we have considered two sources of features from pre-execution time: *1) query text 2)query execution plans*

**Query text feature encoding:** It refers to the query's expressions that the user or application sends to the database. To extract information from the query text, we mainly focus on three forms of feature encoding: Bag of words, Word Embedding, and Text Mining.

1. **Bag of words (BoW):** BoW is used to keep track of the frequency of a word in a document. BoW first constructs a dictionary vector containing all the words from the documents in the training set. Each slot in the dictionary vector represents a unique word found in one or more documents, and the value assigned to each slot represents the frequency count of each word in the

document. In this work, we apply BoW to SQL queries by representing each query as a document. BoW builds dictionary vector $L = \{E_1, E_2, ..., E_F\}$ where the size of the vector represents the $F$ tokens that were located from all of the $\mathcal{Q}_{train}$ queries. Additionally, we keep track of the tokens using Term Frequency ($TF$) which is the frequency of a term $i$ in a query $j$ such that $TF = t_{i,j}$. $TF$ is a common method used to represent text data. However, relying solely on $TF$ can lead to biased results due to the high frequency of common terms that may not hold significant meaning. To mitigate this issue, the Term Frequency-Inverse Document Frequency (TF-IDF) is often employed as a normalization technique to account for the importance of each term. TF-IDF $= t_{i,j} \times \log \frac{M}{n_i}$ where $n_i$ is the number of queries (e.g., documents) containing term $t_i$, and $M$ is the number of queries (e.g., documents) in the training dataset. Tokens are removed if they appear too many or too few times across all $M = \mathcal{Q}_{train}$ queries.

2. **Word embedding encoding:** Word embedding is widely used in natural language processing (NLP). Early approaches of NLP use one-hot encoding to represent words in a document. Given a document where the word $a_i$ occurs, it is represented as a vector the size of the dictionary with all zeros except for a single position with a one representing the word $a_i$. However, this approach does not provide any correlation between words $a_i$ and $a_j$. For example, the words queen and king or cat and mouse have a correlation that cannot be pinpointed by the one-hot encoding method. On the other hand, word embedding provides meaning to a word depending on its surrounding words. Meaning, semantically related words are close to each other in the embedding space. Mikolov et al. [40], [41] focuses on learning the representation of a word in a document by predicting the following word based on previous words. Mikolov expands this approach by setting the sentence, paragraph, or document as a window and learning the window's representation of its given parent paragraph, document,

or documents. Devlin et al. [42] focus on learning the word representation to the document based on the words before and after the targeted word we wish to learn. We can apply this approach to vectorizing our queries by learning the representation of each token to the query itself.

3. **Text mining approach:** has shown to work remarkably well for extracting features from SQL queries [43], [44]. Text mining focuses on identifying useful information through pattern recognition in the data. It concentrates on semistructured or unstructured data such as emails, text documents, or websites. We can map this process to an SQL query by viewing the query as a small document. In this step, we follow the process designed in Makiyama's work [43], where we focus on extracting the terms in the selection, from, joins, projection, and order-by operators. The terms are then joined to the table they are applied to. Once the new tokens are created, we follow similar steps to BoW and construct a vector dictionary where each slot contains the query token and its frequency count.

---

```
SELECT u.username, u.date_purchase
FROM user u, accounts a
WHERE u.id = a.userid
```

---

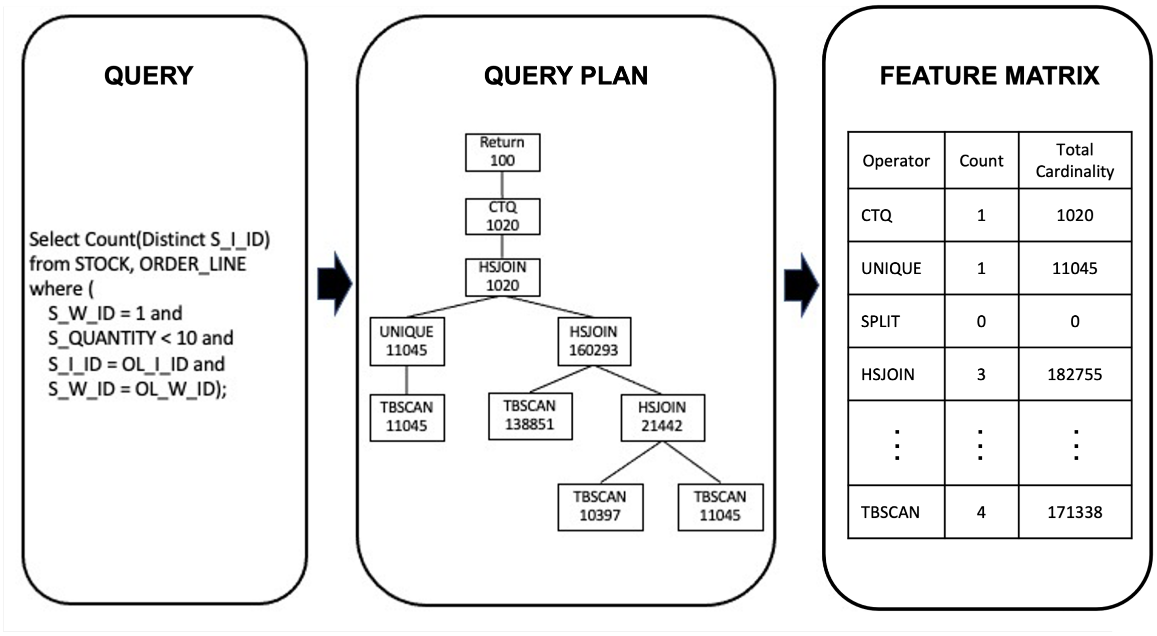
```
SELECT_u 2
FROM_user 1
FROM_accounts 1
WHERE_id=userid 1
```

---

As seen in the above example, we join the operators *select*, *from*, *where* to the table with the symbol "_". Then, a feature vector is created using the token frequency as weights to calculate the pairwise similarity.

### Query plan base feature encoding

In this section, we use the query plan generated by the DBMS. When the DBMS processes a query, the optimizer generates a query plan with the optimum execution process. The query plan takes the form of a tree where each node contains the operator and supplementary information. Generating the query plan is a standard process and can be obtained in milliseconds or seconds [1]. Previous research has utilized features extracted from the query plan for database clustering tasks [45]. For extracting the features from the query plan, we mainly focus on two methods Ganapathi et al. [1], and Sun et al. [29].



| Operator | Count | Total Cardinality |
|---|---|---|
| CTQ | 1 | 1020 |
| UNIQUE | 1 | 11045 |
| SPLIT | 0 | 0 |
| HSJOIN | 3 | 182755 |
| ⋮ | ⋮ | ⋮ |
| TBSCAN | 4 | 171338 |

Figure 4.3: Query Plans Feature Vector

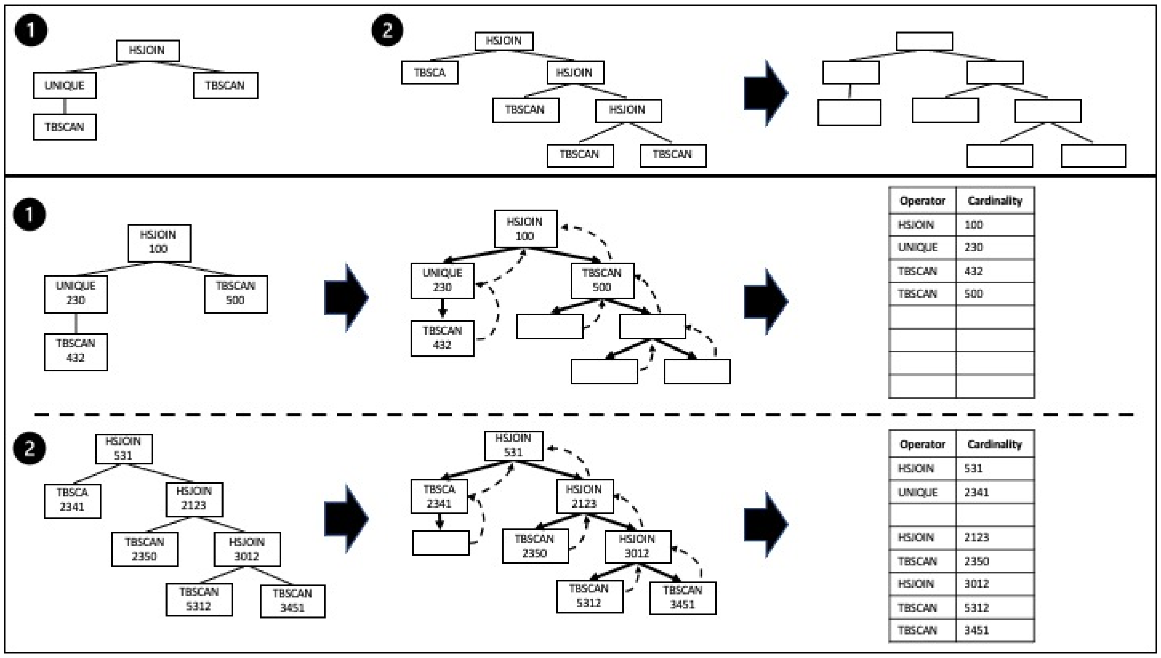
1. **Cardinality cost aggregation for each operator (CCAEO)**

   follows the steps from Ganapathi et al. [1]. At each node of the query plan tree, we can extract the operator type and the cardinality estimation. The cardinality estimation is the process of estimating the number of rows that will be returned by a particular operation in the query plan during its execution.

For each operator in the query plan, we track the aggregate cardinality cost and its instances in the query plan tree. As shown in figure 4.3, we use the query plan generated by the DBMS's optimizer to build our feature matrix. The feature matrix contains two columns: one column for the frequency count of an operator and a second column for the total cardinality cost of the operator in a given query plan. Given that the database management system (DBMS) being used is known, the set of operators that could potentially appear in the query plan can also be determined. As a result, the feature matrix can be constructed such that each row represents an operator, so the matrix ends up with a fixed length size. If an operator does not appear in the query plan, then its frequency count and total cardinality cost are recorded as zero in the matrix.

In figure 4.3, the query plan contains four main operators: CTQ, UNIQUE, HSJOIN, and TBSCAN. Once the unique operators have been extracted from the query plan, we track the frequency of each operator: CTQ: 1, UNIQUE: 1, HSJOIN: 3, and TBSCAN: 4. Finally, we aggregate the cardinality cost for all of the occurrences of each operator: CTQ: 1020, UNIQUE: 11045, HSJOIN: 1020 + 21442 + 160293 = 182755, and TBSCAN: 11045 + 138851 + 10397 + 11045 = 171338. Note that the operator "SPLIT" does not appear in the query plan; therefore, its count and total cardinality are recorded as zero.

2. **Query plan encoding while maintaining tree structure (QPEWMTS)** this approach follows the steps presented by Sun et al. [29], which focuses on feature encoding a query while maintaining the structure of the query plan tree. We start by constructing an empty skeleton tree. As seen in section one of figure 4.4, we build the skeleton tree based on the trees we see in the training query corpus. For example, in figure 4.4, queries 1 and 2 construct the skeleton tree in the top right of the figure. If there were a third query in the training corpus, we



| Operator | Cardinality |
|---|---|
| HSJOIN | 100 |
| UNIQUE | 230 |
| TBSCAN | 432 |
| TBSCAN | 500 |
| | |
| | |
| | |
| | |

| Operator | Cardinality |
|---|---|
| HSJOIN | 531 |
| UNIQUE | 2341 |
| | |
| HSJOIN | 2123 |
| TBSCAN | 2350 |
| HSJOIN | 3012 |
| TBSCAN | 5312 |
| TBSCAN | 3451 |

Figure 4.4: Query Plans Feature Vector

would add the missing section from the new query's plan tree onto the skeleton tree. Next, each query's plan tree is mapped to the skeleton tree, as seen in figure 4.4. The nodes that are not filled are left blank. Lastly, we traverse the tree using depth-first search (DFS) and fill the feature vector accordingly, where each node represents a slot in the vector. If the node is empty, then the vector slot representation for this node will also be empty.

**Clustering Algorithms**

In the clustering step, we aim to identify query templates $\mathcal{T}$ such that all queries $q \in \mathcal{Q}$ that map to $t \in \mathcal{T}$ have similar resource demands. We explore three popular clustering algorithms to accomplish this objective: 1) K-means, 2) K-Medoids, and 3) DBSCAN. For all three clustering algorithms, we use the Euclidean Distances:

$$d(x,y)^2 = \sum_{i=1}^{p} (\mathbf{x_i} - \mathbf{y_i})^2 \tag{4.1}$$

1. **K-Means clustering:** K-Means is a distance-based clustering algorithm widely applied in different domains due to its simplicity and speed [46]. $K$-means works by partitioning a set of $n$ observation points $(x_1, x_2, ..., x_n)$ into $K$ sets $C = C_1, C_2, ..., C_3$. It initiates by randomly selecting $k$ mean $\mu$ values and assigning each point to the closest $\mu$ centroid. Once all of the points have been assigned, the algorithm focuses on re-selecting the $k$ centroid by re-calculating $\mu$ of each cluster. The algorithm re-assigns the points to their closet new $\mu$ centroid. This process continues until K-means converges or reaches the stopping criteria. Formally, if $\mu_i$ is the means of the points in $C_i$ then the overall goal of the algorithm is to minimize the within clusters sum of the squares:

$$argmin_S \sum_{i=1}^{K} \sum_{x \in C_i} ||X = \mu_i||^2 \tag{4.2}$$

**Algorithm 1** Kmeans

```
Int K                                                    ▷ Number of clusters
Data D ← $x_1, x_2, ..., x_n$                                   ▷ Input data
function KMEANS(D, K)
    Initialize: pick cluster centroids $\mu_i, \ldots, \mu_k$ randomly.
    while until coverage do
        Compute clustering $C_1, \ldots, C_k$ such that $C^i = argmin_j \, (x_i - \mu_j)^2$
        Update the centroids $\mu_j = \frac{1}{|C_i|} \sum_{x \in C_i} x$
    return centers $\mu_i, \ldots, \mu_k$
end
```

2. **K-Medoids clustering:** $K$-Medoids works very similar to K-means by also partitioning a set of $n$ observation points $(x_1, x_2, ..., x_n)$ into $K$ sets $C = C_1, C_2, ..., C_3$. However, K-Medoids does not use the mean value $\mu$ as the cluster center; instead, it uses a medoid which is normally the most centrally located point in the cluster. This allows K-Medoids to be less affected by outliers in the dataset due to the effect that an extremely large value can have on the data's distribution. K-Medoids, start by selecting $k$ medoid points at random, let $M$ be the set of medoids such that $M = \{m_1, m_2, ..., m_k\}$. The algorithm assigns the remaining points to their closest medoid. Once the clusters have been created, the algorithm chooses at random a non-medoid point $x_i \notin M$ and computes the new cost $P$ if $x_i$ replaces $m_i$ as the new medoid. If $O$, the old cost, and $P < O$ then the algorithm leaves $x_i$ as the new medoid. K-Medoids continues the process until the convergence criteria is reached. The cost function is calculated as follows:

$$\sum_{j=1}^{K} \sum_{x \in C_j} |x - m_i| \tag{4.3}$$

**Algorithm 2** KMedoids

```
1: $Data\ D \leftarrow x_1, x_2, ..., x_n$                                ▷ Input data
2: **function** KMEDOIDS($D$)
3:     Initialize: pick cluster medoids $m_i, \ldots, m_k$ randomly.
4:     **while** until coverage **do**
5:         Compute clustering $C_1, \ldots,\ C_k$ such that $C^i\ =\ argmin_j\,(x_i - m_j)^2$
6:         Update the medoids $m_j = argmin_j \left(\sum_{x_j \in C_i, x_z \in C_i, x_j \neq x_z} d(x_j, x_z)^2\right)$
7:     **return** medoids $m_i, \ldots, m_k$
8: end
```

3. **DBSCAN** is a density-based clustering algorithm. DBSCAN mines dense regions of points that are separated by low-density of regions from other high-density of regions. It can be useful when the clusters' are of arbitrary shapes and not spherical. It can also handle noise and does not require the user to choose the number of clusters K, and it has also been used to classify queries [21][25]. DBSCAN starts by arbitrarily selecting an unprocessed point $p$. $p$ is a core point if there are at least a minimal number of $minPts$ points in its $\epsilon$ neighborhood. Point $q$ is density-reachable from $p$ if point $q$ is within $\epsilon$ distance from $p$. Additionally, a point $q$ is density-reachable by $p$ if there exist core points $e_1, ..., e_n$ such that $q$ is density-reachable to $e_1$, $p$ is density-reachable to $e_n$, and each $e_{i+1}$ is density-reachable by $e_i$. A cluster is formed from $p$ with all of its density-reachable core and non-core points, and these points are marked as processed. If p is not a core point, then DBSCAN moves on to the next non-process arbitrary point and repeats the above process until all of the points have been processed.

**Algorithm 3** DBSCAN

```
1: $Int\ minPts$                                  ▷ Number of clusters
2: $Long\ \epsilon$                               ▷ Number of clusters
3: $Data\ D \leftarrow x_1, x_2, ..., x_n$        ▷ Input data
4: function DBSCAN($minPts$, $\epsilon$)
5:     for $i = 1; i < D.size; i++$ do
6:         $Point \leftarrow D.get(i)$
7:         $Point.visited \leftarrow True$
8:         if $isCorePt(Point)$ then
9:             $C \leftarrow Cluster$
10:            $explanCluster(\ Point,\ C,\ \epsilon,\ minPts\ )$
11: end
```

In this section, we proposed different methods for grouping similar queries into templates in order to efficiently model the concurrent memory requirements of the queries and speed up the computation of the training and inference stages. We emphasized that our approach does not aim for an optimal template assignment, but rather a

**Algorithm 4** Learning query templates with clustering algorithm

```
1: $\mathcal{Q}_{train} \leftarrow \{q_1,\ q_2,\ \ldots\ ,\ q_n\}$ ▷ $\mathcal{Q}_{train}$ is a set of historical training queries collected from a DBMS.
2: function GetTemplates($\mathcal{Q}_{train}$)
3:     $Array\ Z \leftarrow [\,][\,]$                         ▷ feature matrix for $\mathcal{Q}_{train}$
4:     for $q_i$ in $\mathcal{Q}_{train}$ do
5:         $feature\_type \leftarrow$ "CCAEO"
6:         $features_i$ = getFeatures($q_i$, $feature\_type$)
7:         Z.insert($features_i$)
8:     $alg\_type \leftarrow$ "$K$-Means"
9:     $\mathcal{T} \leftarrow Clustering(Z,\ alg\_type)$  ▷ learns templates using clustering algorithm
10:    return $\mathcal{T}$                                  ▷ learned query templates
11: end
```

best-effort algorithmic principle for assigning each query to a template. Adding a few incorrect assignments can add randomness and noise to the model, which can help reduce overfitting and improve the model's generalization [47], [48]. Our approach represents a trade-off between accuracy and computational efficiency, and is intended to provide a simple but efficient solution for assigning queries to templates.

Algorithm 4 outlines the steps involved in the function `GetTemplates` for the creation of a set of query templates, $\mathcal{T}$, from the training queries in $\mathcal{Q}_{train}$. The process starts by using the `getFeatures()` function (line 6) to extract features from each query in $\mathcal{Q}train$. The feature extraction process is guided by the `feature_type` parameter, which is set to `CCAEO` (Line 7), meaning that the features will be based on Cardinality Cost Aggregation for Each Operator. The extracted features are then used as inputs for the clustering process performed by `Clustering()`. The type of clustering algorithm used, either `K-Means`, `K-Mediods`, or `DBSCAN`, is specified through the `alg_type` parameter. After the clustering process, each cluster represents a query template $t_i \in \mathcal{T}$ (line 9).

## 4.3 Training Stage Phase 2: Constructing Histograms from Workloads

This step presents how to create the input vector $\mathcal{H}$ for the ML model. The input vector $\mathcal{H}$ is a one-dimensional fixed-size vector of length $|\mathcal{T}|$ where each slot represents a template. Here, $|\mathcal{T}|$ is defined by the number of templates generated in phase 1.

Consider $n$ number of workloads where each workload comprises of $|Q|$ number of queries. In each workload, queries are binned into a vector $\mathcal{H} \in R^D$, where $D = |\mathcal{T}|$ is the number of unique clusters. Each bin slot $c_d \in \mathcal{H}$ represents the count of the queries belonging to it such that $\sum_{j=1}^{K} c_j = |Q_i|$. The vector $\mathcal{H} = \{c_1, \ldots, c_{|\mathcal{T}|}\}$ becomes an unknown distribution $P$ which is associated by a label $y \in R$. Consequently, $y$ denotes the aggregated resource cost of the workload. For ${w_i}^{th}$ workload, the histogram representation of the workload becomes $(\mathcal{H}_i, y_i)$ where $y_i = \sum_{j=1}^{|Q_i|} m_j$.

**Algorithm 5** Histogram construction from training workloads

```
 1:  $w \leftarrow (\mathcal{Q}, y)$                          ▷ A training workload
 2:  $\mathcal{T} \leftarrow \{t_1, ..., t_k\}$              ▷ A set of $k$ query templates
 3:  function BinWorkload($w$, $\mathcal{T}$)
 4:      $Array\ \mathcal{H}[0\ ...\ (k-1)] \leftarrow 0$
 5:      for $q_i$ in $\mathcal{Q}$ do
 6:          $features_i$ = getFeatures($q_i$)
 7:          $q_i.template$ = findTemplate($features_i$)
 8:      for $t_j$ in $\mathcal{T}$ do
 9:          $\mathcal{H}[j] = countTemplateInstances(t_j, w)$
10:      return $(\mathcal{H}, y)$
11:  end
```

Algorithm 5 describes the steps of the binning of queries phase. It takes as input a training workload, $w$, and a set of query templates, $\mathcal{T}$, which were learned in section 4.2.2. In `lines 5-7`, for each query $q_i \in \mathcal{Q}$, the algorithm extracts the features from the query using the method selected in section 4.2.2. `findTemplate()` (line 7) identifies the query template $t_j \in \mathcal{T}$ that $q_i$ belongs to. `countTemplateInstances()` counts how many queries in $\mathcal{Q}$ map to each template and stores the counts in vector

$\mathcal{H} = [c_1, ..., c_{|\mathcal{T}|}]$. This binning process allows for the characterization of the training workload through the histogram representation of the distribution of query templates.

The histogram $\mathcal{H}$ represents the distribution of queries in the workload $w$ across the set of query templates $\mathcal{T}$. It is important to note that the histogram will typically be sparse, with many zeros, as not all query templates in $\mathcal{T}$ are expected to be present in workload $w$. The final step of the algorithm (line 10) returns a pair $(\mathcal{H}, y)$, where $y$ is the cumulative memory usage of all queries in $\mathcal{Q}$. The combination of the histogram and memory usage $(\mathcal{H}, y)$ becomes a labeled sample for the training of a supervised machine learning and deep learning model in section 4.4.

## 4.4 Training Stage Phase 3: Training a Distribution Regression Deep Learning Model

In this step, we train a regression model to predict the workload's memory usage. The trained model takes an input workload, represented as a histogram of query templates, and computes the workload's memory usage. For training the model, we explored several machine learning and deep learning techniques. In this section, we present the design and implementation of a deep learning (DL) model for our regression model. Section 4.4.6 describes the other machine learning algorithms we explored for the model training. Recently, deep learning has had several algorithmic breakthroughs and has been highly successful with many learning tasks over unstructured data. For example, DL models for image recognition and language translation are now highly accurate [49]. Through training, DL models are excellent at extracting hidden relationships between input and output pairs [50]. They can be useful in learning a non-linear mapping function between input and output without requiring low-level feature engineering. In our case, we have dual complexities: the input is a complex distribution of query templates, and there is a complex relationship between the distribution of query templates and its collective memory demand. We wanted to explore the effectiveness of deep learning for the problem.

### 4.4.1 Multilayer Perceptron (MLP) Model

Many DL networks exist for learning from unstructured data, such as images and text. For example, convolution neural network (CNN) [51] is suitable for working with images; recurrent neural network (RNN) [52] and transformers [53] work with sequential data; graph neural networks (GNN) are available for graphs [54]. With unstructured data, the dimension of the input vector has a variable length, and each element in the vector usually has no meaning in isolation [54]. In our case, the input vector for each workload is structured and has a fixed length corresponding to the number of query templates. Each element of the vector represents the number of workload queries belonging to a specific template. Since we want to learn a regression function from fixed length input vectors, multilayer perceptron (MLP) is a good place to begin with as it assumes the input data has a fixed dimension [54].

As a learning problem, we use the template vector $\mathcal{H} \in \mathbb{R}^D$, for some $D$, and label set $Y$. Remember that $D$ is identified in the templatization step, which represents the number of $|\mathcal{T}|$ templates, and $y \in Y$, where $y$ is the aggregated resource cost of workload $w$. We aim to learn function $f(\cdot) : \mathbb{R}^D \to \mathbb{R}$ that best approximates the relationship between $\mathcal{H}$ and $y$.

A DNN can be represented as a directed graph, denoted as $G = (V, E)$, where each node in the graph represents a neuron. The neuron is modeled as a scalar function $\sigma : \mathbb{R} \to \mathbb{R}$. The connections between neurons in the graph are represented by edges, and each edge contains a weighted function $w : E \to \mathbb{R}$. In this way, the neural network can be viewed as a graph with nodes and edges, where each node is a neuron and each edge represents the connection between neurons. In this research, we focus on an MLP, which is a feedforward DNN where the underline graph has no cycles.

### 4.4.2 Activation Function

The activation function of each hidden layer controls how the layer's input is transformed. It is used to introduce nonlinearity into the output of a neuron. The activation function takes the weighted sum of the inputs to the neuron and applies a mathematical function to it, producing an output. This output is then fed as input to the next layer of neurons. In choosing the activation function, a nonlinear function is essential since, without it, the DNN model can collapse into a simple linear model [55]. Popular choices for nonlinear functions include sigmoid and hyperbolic tangent ($tanh$) functions. The sigmoid activation function $\sigma(x) = 1/(1 + exp(-x))$ transforms the input value into a value between $[0, 1]$, and the tanh function $tanh(x)$ transforms an input to a value between $[-1, 1]$. However, both of these functions saturate through most of their domain [48]. Meaning, that very large numbers are converted to 1, and very small numbers are converted to 0 for *sigmoid* and -1 for *tanh*. This saturation can become problematic as the network becomes deeper. MLPs are trained using stochastic gradient descent (SGD), which involves calculating and descending the slope of the function. At each step, the prediction error of the function is computed and used to calculate the gradient. The gradient is then used to tune the weights so that the error is reduced in the next iteration. In a DNN, the error used to update the weights is propagated backward via the network. The saturation causes the error to decrease largely as it propagates each layer. This phenomenon is called the vanishing gradient problem, which prevents DNNs from learning effectively. To solve this problem, the activation function ReLU is employed to tackle this problem

$$f(e) = \begin{cases} 0 & if\ e < 0 \\ e & otherwise. \end{cases} \tag{4.4}$$

The ReLU activation function is effective in addressing the issue of vanishing gradient because it replaces all negative values in the input with zero when $e \leq 0$, effectively creating a threshold that prevents the gradient from becoming too small. This means that the gradient will always be non-zero for positive input values ($e > 0$), which helps to propagate the gradient through the network and prevents it from vanishing. It acts as a linear function when the error is greater than zero ($e > 0$). This helps maintain many of the desirable properties of a linear activation function, but it cannot be considered a true linear function since the error is zero when $e \leq 0$. Empirically, this correlates with our experimentation as the data complexity grows our model performs better with *ReLu* in comparison to *sigmoid* and *tahn*.

### 4.4.3 Loss Function

Depending on the problem type, an MLP uses different loss functions. For the regression task, we use the mean squared error loss function as follows:

$$Loss(\hat{y}, y, W) = \frac{1}{2N}\sum_{i=1}^{N} ||\hat{y}_i - y_i||_2^2 + \frac{\alpha}{2N}||W||_2^2 \tag{4.5}$$

where $y_i$ is the target value; $\hat{y}_i$ is the estimated value produced by the MLP model; $\alpha||W||_2^2$ is an L2-regularization term (i.e., penalty) that penalizes complex models; and $\alpha > 0$ is a non-negative hyperparameter that controls the magnitude of the penalty. Starting with an initial set of random weights, the MLP minimizes the loss by iteratively updating these weights. After computing the loss each time, the MLP propagates the loss backward — from the output layer to the previous layers; it updates the weights in each layer to decrease the loss. For training, MLP uses stochastic gradient descent (SDG), where the gradient $\nabla Loss_W$ of the loss, with respect to the weights, is computed and deducted from $W$. More formally,

$$W^{i+1} = W^i - \epsilon \nabla Loss_W^i \tag{4.6}$$

where $i$ is the iteration step, and $\epsilon$ is the learning rate with a value larger than 0. The algorithm stops either after completing a preset number of iterations or when the loss doesn't improve beyond a threshold.

### 4.4.4 Optimizer

We compared L-BFGS [56] and Adam [57] optimizers using two datasets — a small dataset and a relatively large one.

For the small dataset, L-BFGS was more effective than Adam as it ran faster and learned better model coefficients. In contrast, Adam worked better with the large dataset. Our observation is consistent with *scikit-learn*'s `MLPRegressor`[1] documentation.

### 4.4.5 Hyperparameter Tuning of the MLP Model

To find an optimum final model, we tuned several hyperparameters, including *the number of hidden layers*, *the number of nodes in each layer*, *the optimizer*, and *the*

[1]https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPRegressor.html

*dropout rate*. Since we operate on a large dataset and the parameter search space is large (i.e., the combination of different parameter values), we used randomized search to reduce the time for optimizing hyperparameters [58]. The randomized search, which is similar to the grid search, performs a randomized scan of the parameter space. Compared to grid search, randomized search is faster with a slight sacrifice of the model performance. We used the `RandomizedSearchCV` [2] method of the `scikit-learn` library for random search. For instance, using randomized search with one of our experiment datasets, we came up with a tuned neural network architecture that had eight layers: an *input layer*, six *hidden layers*, and *an output layer*. The input layer receives an input workload, represented as a histogram or a distribution of its queries over query templates; the output layer generates an estimated memory demand for the input workload. Left to right, the hidden layers have 48, 39, 27, 16, 7, and 5 nodes, respectively.

**Model complexity**. Suppose there are $n$ training samples, $k$ features, $l$ hidden layers, each containing $h$ neurons — for simplicity, and $o$ output neurons. The time complexity of backpropagation is $O(n \cdot k \cdot h^l \cdot o \cdot i)$, where $i$ is the number of iterations.

### 4.4.6 Other Machine Learning Methods

Besides deep learning networks, for a comparative analysis, we explored four additional ML techniques to train LearnedWMP models. They include a *linear* and three *tree-based* techniques. For the linear model, we picked **Ridge**, a popular method for learning regularized linear regression models [59], which can help reduce the overfitting of the linear regression models. From the tree-based approaches, we used **Decision Tree (DT)**, **Random Forest (RF)**, and **XGBoost (XGB)**.

[2]https://scikit-learn/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html

Similar to section 4.4.1, as a learned problem, from $n$ training examples $(\mathcal{H}_1, y_1), (\mathcal{H}_2, y_2), \ldots, (\mathcal{H}_n, y_n)$, the model learns a function $f(\cdot) : R^D \rightarrow R$, where $D$ is the number of dimensions for input $\mathcal{H} = [c_1, ..., c_{D=|\mathcal{T}|}]$ and $R$ is the scalar output $y$. We aim to learn function $f(\cdot)$ that best approximates the relationship between $\mathcal{H}$ and $y$.

**Ridge**

is a common tool used by the machine learning community for identifying the relationship between an explanatory variable and some valued outcome [60]. For Ridge regression, the Hypothesis class is the set of linear functions. In the following equations below, $w$ denotes the weight vector used by the models.

$$L_d = \{x \rightarrow \langle w, x\rangle + b : w \in \mathbb{R}^D, b \in \mathbb{R}\} \tag{4.7}$$

Next, we need to define a loss function. A loss function measures how well $f(\cdot)$ correctly predicts $y$, and it assigns a penalty for not predicting correctly. For Ridge Regression we minimize the square-loss function with $L_2$ regularisation.

$$argmin_{w\in\mathbb{R}^D} \left( \frac{1}{m} \sum_{i=1}^{m} \frac{1}{2} (\langle w, X_i\rangle - y_i)^2 + \lambda ||w||_2^2 \right) \tag{4.8}$$

**Decision Tree**

is a predictor, $f : \mathcal{H} \rightarrow y$, which functions by predicting instance $\mathcal{H}$ by traversing the path of a tree from the root node to a leaf node. Normally, a decision tree algorithm is used as a classification. However, decision trees can be applied to both classification and regression tasks [55]. Therefore, we use decision trees as a regression task in our problem and $MSE$ to identify how much the prediction $\hat{y}$ differed from the target $y$.

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i) \tag{4.9}$$

**Random Forest**

works by building a group of decision trees based on bootstrapped samples of data from the training set. We create $R$ number of trees $\hat{f}^1(x_1), \hat{f}^2(x_2), ..., \hat{f}^R(x_B)$ using $B$ random sample training datasets such that $R = B$. To retrieve the model's prediction, we average the results of all of the trees to obtain a single output, given by.

$$\hat{f}_{rf}(x) = \frac{1}{R}\sum_{r=1}^{R}\hat{f}_r(x) \tag{4.10}$$

When building each decision tree $\hat{f}^r$, we use a sample of $m$ predictors out of $p$ predictors as a candidate when considering a split. Each split can use only one of the $m$ candidates, and $m$ new sample candidates are picked at each split, $m \approx \sqrt{p}$. Allowing only $m$ selectors at each split helps to create a mixture of trees. If there is a very strong feature, then all decision trees will pick this feature as their root split, and all trees will look very similar and highly correlated.

**XGBoost**

is a scalable machine-learning approach for tree boosting [61]. For our $n$ workloads and $D$ templates. $N = \{(x_i, y_i)\}(|N| = n, x_i \in \mathbb{R}^D, y_i \in \mathbb{R})$ XGBoost uses $K$ additive functions to predict the output.

$$\hat{y}_i = \sum_{k=1}^{k} f_k(x_i),\ \ f_k \in F, \tag{4.11}$$

where $F$ is the space of regression trees. To learn the appropriate weights, XGBoost works to minimize the loss function.

$$\begin{gathered} L_{XGBoost} = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(f_k) \\ where\ \Omega(f_k) = \gamma T + \frac{1}{2}\lambda||w||^2 \end{gathered} \tag{4.12}$$

$l$ is a convex function that measures how close the target $y$ is to the predicted $\hat{y}$. $\frac{1}{2}\lambda||w||^2$ helps regularize the model, and $\gamma T$ helps to prune the tree. Both of these sections help smooth the final learned weights, so the model does not overfit. If $\lambda$ and $\gamma$ are both set to zero, then the model falls back to traditional gradient tree boosting.

## 4.5 LearnedWMP: Inference Stage

Algorithm 6 outlines the steps of LearnedWMP Inference Stage. The PredictMemory function, which is used to estimate the memory demand of an unseen workload, $w$ utilizes two models: a trained clustering model which is trained to map a query $q_i \in \mathcal{Q}$ to its corresponding template $t_i \in \mathcal{T}$, and a trained predictive model for estimating workload memory demand. Using the BinWorkload function from Algorithm 5, The function takes an unseen workload as input and generates a histogram vector $\mathcal{H}$, which is a distribution of query templates. With the histogram $\mathcal{H}$ in hand, the function then estimates the memory demand of the workload, which is $\hat{y}$.

**Algorithm 6** Use of the trained MLP for workload memory prediction

```
1: $\mathcal{T} \leftarrow \{t_1, ..., t_k\}$                ▷ A set of $k$ query templates
2: $\hat{f}$                                   ▷ a workload memory estimation function
3: $w \leftarrow (\mathcal{Q})$                         ▷ An unseen input workload
4: function PredictMemory($w$, $\mathcal{T}$, $\hat{f}$)
5:     $\mathcal{H} = BinWorkload(w, \mathcal{T})$
6:     $\hat{y} = \hat{f}(\mathcal{H})$
7:     return $\hat{y}$
8: end
```

# Chapter 5

# Experimental Evaluation

In this section, we experimentally evaluated the performance of our proposed LearnedWMP model for the workload memory prediction problem. We also evaluated variants of LearnedWMP's phase 1 approach of learning query templates (see section 4.2). We'll first describe the experimental setup and then present the experiment results.

## 5.1 Experimental Setup

### 5.1.1 Environment

We conducted the experiments using a commercial DBMS instance running on a Linux system with 8 CPU cores, 32 GB of memory, and 500 GB of disk space.

### 5.1.2 Datasets

For the experiments, we acquire different database instances and a collection of queries from different databases to evaluate our methodology. We used the data and the queries from the TPC-DS and JOB Benchmarks. These benchmarks are very popular and widely used in the database research community and have been created to model online analytical processing (OLAP) workloads. Additionally, we also use the TPC-C benchmark, which is used to model a medium complexity online transaction processing (OLTP) workloads [62]. TPC-DS is a modern DSS benchmark that has been widely used in the industry as it emulates real-world datasets and queries. TPC-DS contains 24 tables with an average of 18 columns, 99 distinct SQL query templates with random substitution, and a more representative skewed database content [63]. JOB is a real-world movie-related benchmark constructed based on information from IMDB. JOB queries focus on join ordering which is a very important problem for query optimization [17]. Lastly, TPC-C is similar to TPC-DS as it is also a Transaction Processing Performance Council (TPC) Benchmark. All TPC benchmarks must undergo stringent testing and approvals before they are accepted as TPC Benchmarks. TPC-C is built to simulate an order entry workload with two types of transactions. First, it contains simple transactions that emulate a debit or credit workload. Second, it contains a medium complexity transaction which ranges from two to fifty times the number of simple transaction calls [62].

### 5.1.3 Workload

To generate the queries, we use the tool kits provided by each benchmark. Altogether, we generated 93,000 TPC-DS queries, 2300 JOB queries, and 3958 TPC-C queries. After the queries and database are created, the ground truth is generated by executing one query at a time through the database instance. The data collected is the maximum memory cost of the given query during execution. It is important to note that in our experimental setup, we assume that each query is executed in isolation, and therefore, we do not consider concurrency. While this might not accurately reflect real-world scenarios, it is sufficient for training our model and evaluating its performance. We also collect the queries' query plan. For the workload, the queries are randomly grouped into a collection of $s$ queries per group, where $s = 10$. Additionally, their individual resource cost (e.g., Memory) is aggregated to compute the workload's total resource cost.

## 5.2 Experimental Evaluation and Discussion

In the experimental evaluation and discussion section, we examine the following elements of our system.

- **Learning query templates**. What are the advantages of LearnedWMP's method for learning query templates over alternatives with respect to improving workload memory estimation accuracy?
- **LearnedWMP accuracy performance**. What are the accuracy comparisons between LearnedWMP-based models and SingleWMP-based models?
- **LearnedWMP training and inference cost**. How does learning the LearnedWMP-based models compare to SingleWMP-based models in terms of runtime cost?
- **LearnedWMP model size**. What is the difference in size between LearnedWMP-based models and SingleWMP-based models?
- **Effect of the batch size parameter** $s$. In the LearnedWMP model, how does batch size $s$ affect memory estimation accuracy?

### 5.2.1 Learning Query Templates Performance

The objective of phase 1, learning query templates, is to divide queries into sub-groups called templates such that all queries belonging to the same template have a similar resource demand (e.g., Memory). Our focus for experimenting with the learned query templates is primarily on the TPC-DS and JOB benchmarks, not TPC-C. This is because we are not looking for an optimal template assignment, but rather a best-effort algorithmic principle for assigning each query to a template. As mentioned in section 4.2, we use two main approaches for our process: the rule-base and the clustering approach. To measure the performance, we fix a machine learning algorithm and compare the model's performance based on what template learning step is used to build the workload histogram vector $\mathcal{H}^{|\mathcal{T}|}$, where $\mathcal{H} = [c_1, ..., c_{|\mathcal{T}|}]$. Remember that in the binning step, each bin $c_i \in \mathcal{H}^{|\mathcal{T}|}$ is the aggregation of all queries in the workload that map to the template's bin $c_i$.

**Rule-based Templatization**

In the Rule-base Templatization, one or more rules associate a query statement with a predefined template. Database administrators (DBA) or researchers often create these rules [3]. In this work, we investigated simple rules for identifying templates. TPC-DS queries are based on a set of templates. In each template, the literals and constants are generated randomly from a fixed query corpus. Consequently, because TPC-DS contains a large set of predefined query templates, we map the TCP-DS queries back to their original templates. JOB queries are generated based on a smaller set of query templates. We cannot follow the same approach as the TPC-DS queries, as this will result in a very small number of templates which can affect the prediction accuracy. Thus, we opt to group the JOB queries based on similar cardinality estimation.

**Clustering Templatization**

To evaluate our clustering techniques, we construct the input vector for K-Means, K-Medoids, and DBSCAN algorithms by using different query plan features from the query expression or query plan. Overall, we experiment with five methods for constructing the input feature vector to the clustering algorithms. These methods from the query expression are: 1) *bag of words* 2) *word embedding* 3) *text mining* and from the query plan: 4) *Cardinality cost aggregation for each operator (CCAEO)* 5) *Query plan encoding while maintaining tree structure (QPEWMTS).*

**Bag of words (BoW):** We focus on transforming the input query into a bag of words by constructing a dictionary vector $L = \{E_1, E_2, ..., E_L\}$ containing all $L$ tokens from the $\mathcal{Q}_{train}$ queries. Meaning, each slot in the dictionary vector represents a unique token $E_i$ found in the training queries $\mathcal{Q}_{train}$, and the value assigned to each slot represents the frequency count of each token $E_i$ in the query $q_i \in \mathcal{Q}_{train}$. We use Term Frequency $TF = t_{i,j}$ to keep track of the frequency of token $K_i$ in query $x_j$. Lastly, we use the Term Frequency-Inverse Document Frequency TF-IDF $= t_{i,j} \times \log \frac{M}{n_i}$ to normalize the term frequencies and remove common terms which can produce unsatisfactory results.

**Word embedding:** As a first step, we construct a vocabulary based on all the unique keywords found in the training queries. We assign a unique ID to each term in the vocabulary so that a set of IDs represents each query. Each query's ID set becomes the input to the embedding layer, which learns a vector representation for each keyword. Our query text features were represented using word embeddings with a vocabulary of 1500 words. We chose this value based on analyzing the unique number of keywords in the query corpus. Keywords may be SQL syntaxes, such as the GROUP BY operator, or schema information, such as table and column names. On average, the number of keywords ranges from 311 to a maximum of 1750 for each query. For queries with fewer than 1750 keywords, we padded them with zero cells to make their length 1750. The goal was to have all queries be 1750 characters long.

Using a word embedding model, we projected each keyword from the vocabulary into a dense vector of length 16. Each query that had 1750 keywords now becomes a 2D-Vector. After flatting the 2D-Vectors, we represented each query as a 1-D vector of length 28000 (1750x16).

**Text mining approach:** Text mining is very similar to the bag of words approach, except that the tokens are constructed differently. Rather than tracking keyword frequency in a query's dictionary representation, we now track operators and tables operated on. Next, to construct the text-mining feature vectors, we follow the same steps as bag of words.

---

SELECT u.username, u.date_purchase

FROM user u, accounts a

WHERE u.id = a.userid

---

SELECT_u 2

FROM_user 1

FROM_accounts 1

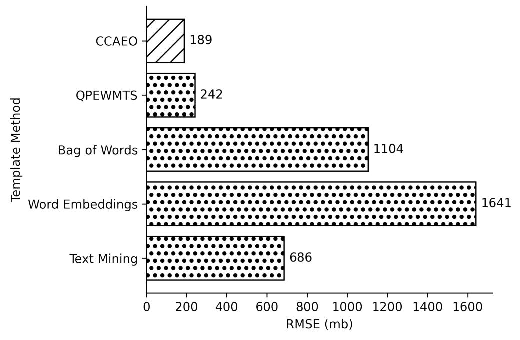
WHERE_id=userid 1

---

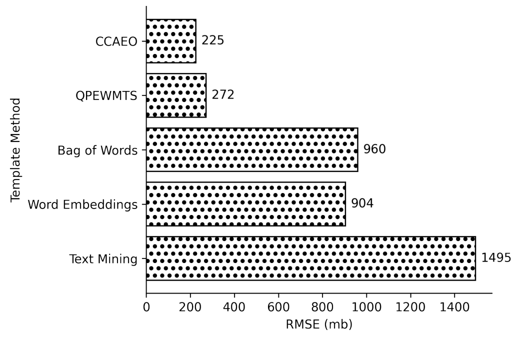
Overall, this technique reduces query log space [44] and attempts to discover the correlation between operators and resource demand [43].

**Cardinality cost aggregation for each operator (CCAEO):** Prior research [1] has seen query plan-based features as good predictors of the run-time metrics during query execution. Additionally, generating the query execution plans is a fast process that can take milliseconds or seconds [1]. Using a state-of-the-art RDBMS, we collect the query plans for all queries in the training dataset. We run each query through the DBMS and collect the query plans produced, which are then used to build the feature vectors. We parse the query plans and collect all of the operators in the query plans. With these operators, we construct the query vector where each unique operator contains two slots in the feature vector: the aggregated cardinality estimation and the operator's frequency. If an operator is not located in the given query, then the operator's aggregated cardinality estimation and the operator's frequency value are set to zero.

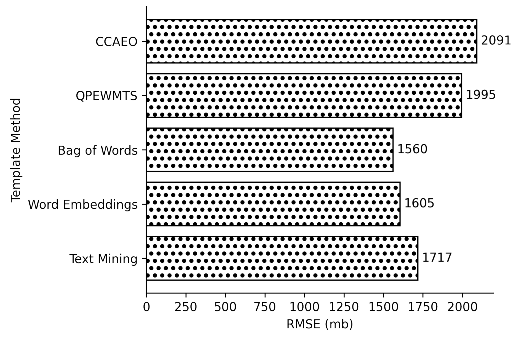
**Query plan encoding while maintaining tree structure (QPEWMTS):** focuses on feature encoding a query while maintaining the structure of the query plan tree. Here, we follow the same steps as CCAEO for collecting the query plans. Next, we use all of the query plans to construct the empty skeleton tree. Afterward, The feature vector is filled based on the depth-first search (DFS) traversal of the tree, where each node represents a position in the vector. In the case of an empty node, the vector slot representation of the node will also be empty.



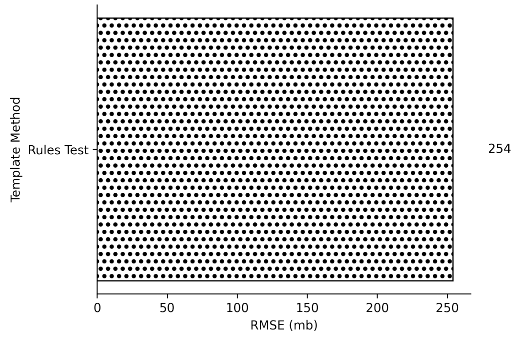
(a) K-Means



(b) K-Medoid



(c) DBSCAN

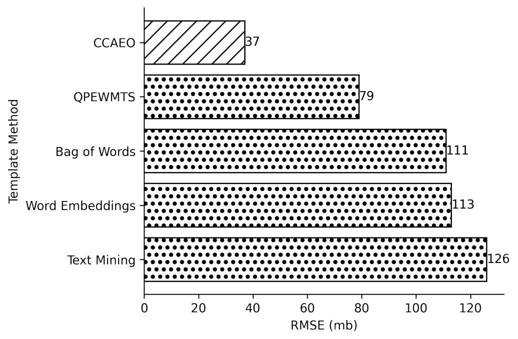


(d) Rules

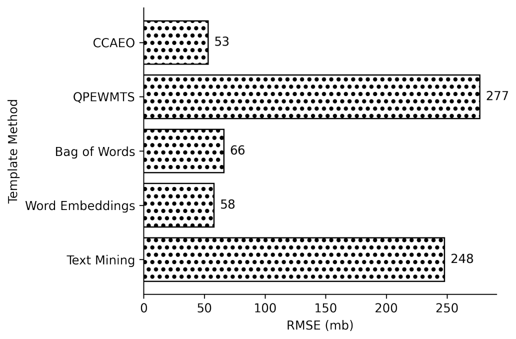
Figure 5.1: Query Templatization Influence for Achieving Workload Memory Estimation Accuracy for TPC-DS Benchmark

### 5.2.2 An Evaluation of the Learning Query Template Strategies

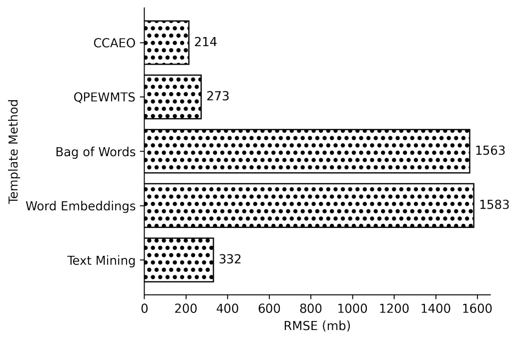
To assess the effectiveness of the learning query templates step, we developed a total of sixteen pipelines. Out of the sixteen pipelines, fifteen pipelines are created using the clustering approach, and one pipeline is created using the rule-based approach. The memory prediction section uses a fixed model algorithm (XGBoost) in all pipelines. The only variation between the pipelines is the learning query template procedure, and we evaluate the performance of the pipelines based on their RMSE (Root Mean Squared Error) scores.



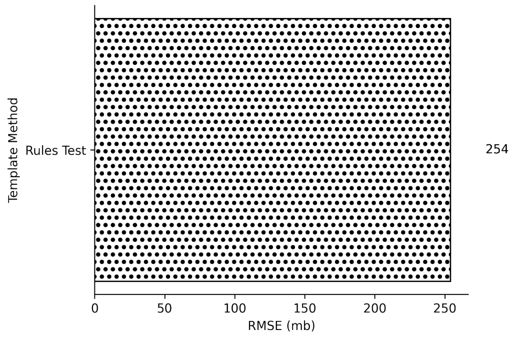
(a) K-Means



(b) K-Medoid



(c) DBSCAN



(d) Rules

Figure 5.2: Query Templatization Influence for Achieving Workload Memory Estimation Accuracy for JOB Benchmark

Figure 5.1 and 5.2 summarize how the XGBoost model performed based on the learning query template step used to construct the input vector $\mathcal{H}$ to the mode. Overall, it displays the RMSE performance of all sixteen template pipelines: Rule base, K-Means CCAEO, k-Medoids CCAEO, DBSCAN CCAEO, K-Means QPEWMTS, k-Medoids QPEWMTS, DBSCAN QPEWMTS, K-Means Bag of Words, k-Medoids Bag of Words, DBSCAN Bag of Words, K-Means Word Embedding, k-Medoids Word Embedding, DBSCAN Word Embedding, K-Means Text Mining, k-Medoids Text Mining, and DBSCAN Text Mining.

**Clustering algorithm results.** The results illustrate that the clustering algorithm approach outperforms Rule base approach. In the clustering approach, $K$-means outperformed $K$-Medoids and DBSCAN clustering algorithms. *Specifically, $K$-means CCAEO surpasses all the other clustering pipelines.*

We opt for the clustering approach instead of the rule-based approach due to the limitations of the latter. One major drawback of the rule-based method is that creating effective rules may require the expertise of human professionals, which can be time-consuming, expensive, and challenging. In the case of the TCP-DS benchmark, the rule-based method performed well because the TPC-DS queries were constructed based on pre-existing 99 query templates loosely associated with a set of rules. However, the rule-based method did not perform well in the JOB benchmark due to the lack of access to pre-existing templates.

When selecting the clustering algorithm, we made sure to test different clustering algorithms that have been previously used on SQL queries. Overall, in comparing other clustering algorithms, $K$-Means outperformed $K$-medoids and DBSCAN. $K$-medoids did perform fairly close to $K$-means; however, the $K$-Medoids training time was much larger than $K$-Means. $K$-Medoids can be expensive as it requires the computation of all pairwise distances. On the other hand, $K$-means works to find the cluster's centroid by calculating the average of all data points in the cluster, making it a simple and fast algorithm [46], and its iterative approach can be easily scaled to tackle large data [64]. Additionally, for our purpose of clustering queries with similar resource costs, it is more helpful to use the centroid means approach because it considers the distances of all data points from the centroid rather than just focusing on one point. This way, we can group queries with similar resource costs even if they are not necessarily similar to one particular query.

While DBSCAN has been used successfully in clustering queries based on certain features, our encoding process involves generating a high-dimensional feature set. Unfortunately, DBSCAN tends to underperform when it comes to high-dimensional data. As a result, we found that DBSCAN had the lowest performance compared to other clustering algorithms we evaluated for our needs.

**Feature Encoding Results.** We tested five feature encoding methods CCAEO, QPEWMTS, Bag of Words, Word Embedding, and Text Mining. The results illustrate that the query plan methods CCAEO and QPEWMTS outperform the query text methods: Bag of Words, Word Embedding, and Text Mining. In general, extracting features from the query text results in poorer performance when compared to the query plan features. This is because extracting features from the text does not consider how the optimizer is built or how the database is constructed. Bag of Words only relies on the text of the query, which cannot be relied upon as different operations to the database can have similar query syntax but entirely different resource costs. One example is the join operator, the cost of a join operation depends entirely on the tables that are being joined. Word Embedding uses the idea of the vector representation of one token based on the relationship to the document (query) it appears on, which has been shown to work exceptionally well for clustering tasks. However, even if two queries have similar query expressions, they can still have different resource costs. This is because the query optimizer may select different query plans for each query based on the indexing strategy used to improve performance [24]. Lastly, the Text Mining approach tries to build a relationship between the operator and the table operated on. Again, this approach relies solely on the query expressions and does not consider the query plan selected by the optimizer.

Overall, the query plan CCAEO and QPEWMTS clustering approach outperformed the other query text methods, with CCAEO being the best. The performance of CCAEO and QPEWMTS was very close; however, the QPEWMTS did generate a large vectors, an almost 600-length vector for the TPC-DS benchmark.

### 5.2.3 Learning Batch Memory Performance

This work focuses on predicting a batch of query memory estimation (i.e., a workload). Consequently, there are no existing state-of-the-art techniques or literature that we could utilize for comparative performance analysis. Therefore, we must design, develop, and evaluate different variants of LearnedWMP and available baselines for a well-founded comparison analysis.

**LearnedWMP-based methods.**

As a learning problem, LearnedWMP accepts the input workload $w$ and returns the workloads memory prediction $y$. Because LearnedWMP utilizes different ML and DL models, we build for the experiment five different pipelines with the models described in chapter 4. These five modes pipelines are called *LearnedWMP-MLP*, *LearnedWMP-Ridge*, *LearnedWMP-DT*, *LearnedWMP-RF*, and *LearnedWMP-XGB*.

**Evaluation Metrics.**

We compare our model with two different memory resources estimation approaches *Single Workload Model Predictor (SingleWMP)* and a state-of-the-art *SingleWMP-DBMS*.

1. *SingleWMP* follows the steps from et al. Kipf [11]. It predicts the resource demand of each individual query based on the features of the query plan. We construct five different pipelines with the same models used to construct the

LearnedWMP and name them: *SingleWMP-MLP*, *SingleWMP-Ridge*, *SingleWMP-DT*, *SingleWMP-RF*, and *SingleWMP-XGB*.

2. *SingleWMP-DBMS* uses the state-of-the-art DBMS's built-in optimizer to predict the resource demand of each query.

The optimization goal of the resource prediction model is to minimize loss, and the loss is defined as the difference between the actual and the predicted value. In our settings, this is the difference between actual and predicted resource demands cost. To identify the loss, we use Root Mean Square Error (RMSE). Let's assume our test set contains n test workloads represented as $\{(w_1, y_1), \ldots, (w_i, y_i), \ldots, (w_n, y_n)\}$. Then the absolute difference between the predicted resource demands $\hat{y}_i$ and the actual resource demand $y_i$ is computed. Now, we compute the root mean absolute error *(RMSE)* for the loss of the data set.

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n} (y_i - \hat{y}_i)^2}{n}}$$

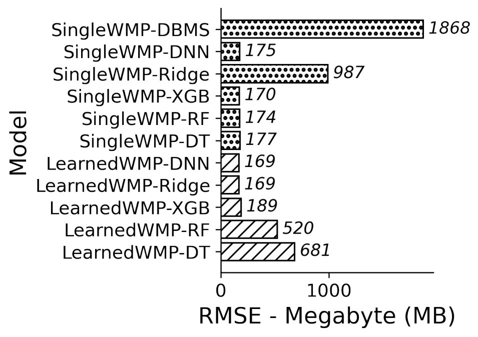
For interpretability, we use *RMSE* to compare our approach to *SingleWMP* and *SingleWMP-DBMS* by running the set of $\mathcal{Q}_i$ queries from workload $w_i$ through each model. Let's look at *SingleWMP* first; given the workload $w_i$, consisting of a set of $\mathcal{Q}_i$ queries, we run each query $q_j \in \mathcal{Q}_i$ through *SingleWMP* and collect the $\hat{y}_j$ prediction. Next, we aggregate all the $\hat{y}_j$ predictions to get $\hat{y}_i = \sum_{j=0}^{|\mathcal{Q}_i|} \hat{y}_j$. We do the same for the actual $y_j$ cost and get $y_i = \sum_{j=0}^{|\mathcal{Q}_i|} y_j$. Once $\hat{y}_i$ and $y_i$ are calculated for all workloads, we then use RMSE to calculate the *SingleWMP's* loss and follow the same steps for collecting *SingleWMP-DBMS* loss.

**LearnedWMP accuracy performance.**

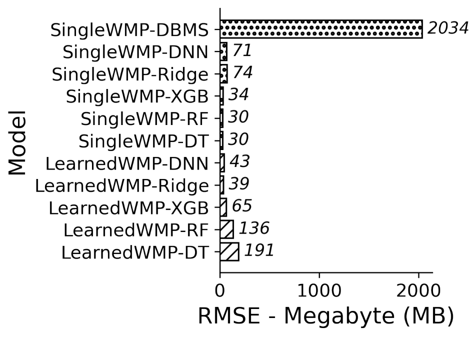
In this section, we compare the performance of all the LearnedWMP and SingleWMP pipelines. In these pipelines, we use the CCAEO K-means approach for locating the templates from the queries. Figure 5.3 illustrates the RMSE results for the different variants of LearnedWMP, SingleWMP, and SingleWMP-DBMS. The results illustrate how the SingleWMP-DBMS in the TPC-DS benchmark had the largest RMSE score compared to the other LearnedWMP and SingleWMP pipelines. This observation detail how the ML and DL method outperformed the current state-of-the-art DBMS prediction. We also observed SingleWMP-DBMS having the highest RMSE error in the JOB and TPC-C benchmarks. The scores for SingleWMP-DBMS are 1868 RMSE score for TPC-DS, 2034 RMSE score for JOB, and 915 RMSE score for TPC-C. In comparison, the overall best model LearnedWMP-DNN had an RMSE score of 169 for TPC-DS, 43 for JOB, and 175 for TPC-C. Overall, we see an improvement of TPC-DS 90.95%, JOB 97.88%, and TPC-C 80.87% when compared to SingleWMP-DBMS.

Figure 5.3 also illustrates the comparison between LearnedWMP and SingleWMP. Here, Ridge, XGBoost, and Deep Neural Network models performed best. Although XGBoost and Ridge performed very well, DNN outperformed XGBoost in the TPC-DS and JOB Benchmark. Ridge did perform well in the LearnedWMP approach but did not do so well in the SingleWMP. Therefore, we conclude that linear algorithms, such as Ridge, may not be the most effective strategy for modeling memory requirements associated with individual queries, which are based on low-level database operations and their estimated cardinalities. Lastly, we observe from Figure 5.3 that LearnedWMP-DNN outperformed SingleWMP-DNN in the TPC-DS, JOB, and TPC-C benchmarks.

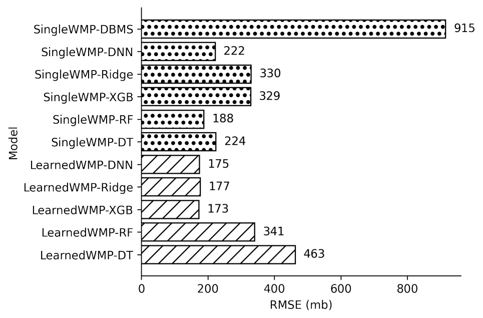
Table 5.1 illustrates the Interquartile range (IQR) of the RMSE residual errors. For each pipeline, IQR helps measure the spread of the errors and identify the skew-



(a) TPC-DS



(b) JOB



(c) TPC-C

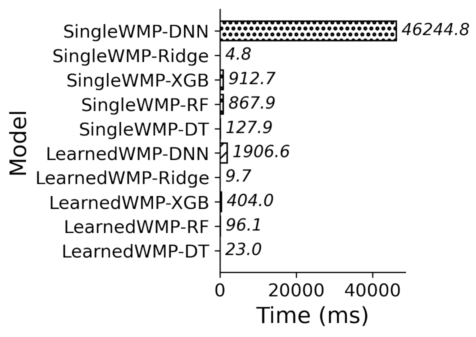
Figure 5.3: ML Model Root Mean Squared Errors (smaller is better)

ness of the errors. Table 5.1 indicates that SingleWMP-DBMS is skewed towards underestimating when predicting the memory demand of the workload for TPC-DS and TPC-C, and it is skewed towards overestimating the JOB benchmark. When compared with ML-based estimates, the results reveal a fairly even distribution between overestimates and underestimates; they do not have a marked skew in either direction. Furthermore, The IQR range for ML-based models is significantly smaller when compared with singleWMP-DBMS. For Example, when glancing at the TPC-DS Benchmark, the IQR range span of the singleWMP-DBMS is [-2063.8, -461], which is quite larger and skewed compared with LearnedWMP-DNN [-92.4, 102.8]. JOB singleWMP-DBMS IQR range span is [1394, 2367], which is also larger and skewed when compared to LearnedWMP-DNN [-32.9, 24.3]. SingleWMP-DBMS consist of human static rules which are not evenly distributed between overestimations and underestimations. These rules are intentionally skewed in one direction. In contrast,

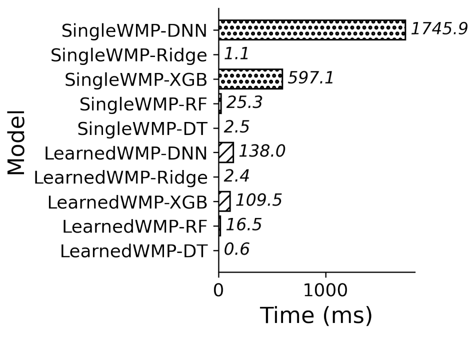
Table 5.1: Interquartile range (IQR) of residual errors in estimating workload memory by different models and datasets

| **Model** | **TPC-DS** | **JOB** | **TPC-C** |
|---|---|---|---|
| SingleWMP-DBMS | [-2063.8, -461] | [1394, 2367] | [-1025, -301.5] |
| SingleWMP-DNN | [-29.6, 158.5] | [-57.3, 64.7] | [-119.9, 77] |
| SingleWMP-Ridge | [-571.0, 665.2] | [-52.0, 48.9] | [-264.9, 113.6] |
| SingleWMP-XGB | [-83.4, 93.6] | [-18.2, 7.2] | [-249.2, 114] |
| SingleWMP-RF | [-83.2, 90.7] | [-17.5, 5.3] | [-87.8, 119.4] |
| SingleWMP-DT | [-83.6, 94.5] | [-17.5, 7.6] | [-102.7, 156.7] |
| LearnedWMP-DNN | [-92.4, 102.8] | [-32.9, 24.3] | [-94 , 78.8] |
| LearnedWMP-Ridge | [-80.2, 90.3] | [-22.6, 8.9] | [-94, 78.8] |
| LearnedWMP-XGB | [-85.5, 101.6] | [-36.1, 41.6] | [-98.8, 80.6] |
| LearnedWMP-RF | [-352.6, 310.1] | [-79.8, 89.2] | [-175.5, 235.3] |
| LearnedWMP-DT | [-460.5, 433.8] | [-94.6, 173.6] | [-231.7, 267.8] |

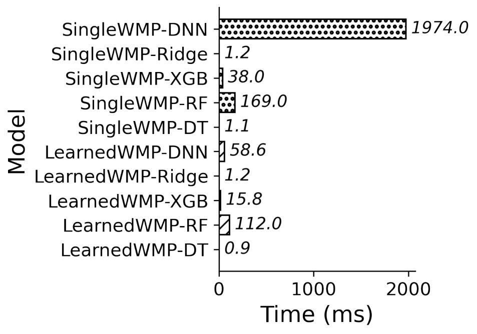
ML-based models learn from historical workflows with instances of both overestimating and underestimating. Therefore, they learn to calculate memory predictions that are not skewed in one direction.



(a) TPC-DS



(b) JOB

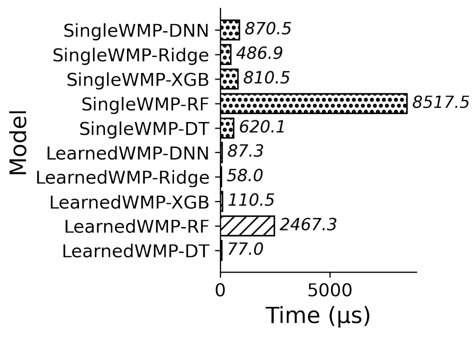


(c) TPC-C

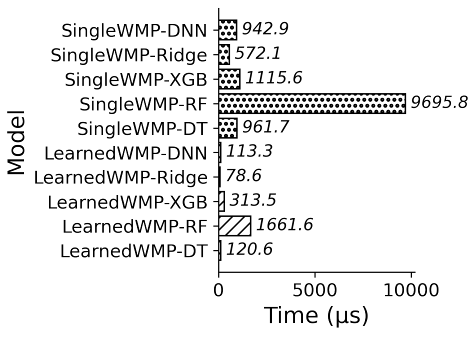
Figure 5.4: ML Model Training Latency

**LearnedWMP training and inference latency.**

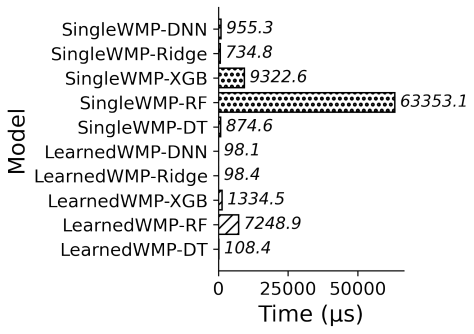
Figure 5.4 and 5.5 report the training and inference time for all learned base modes. Both SingleWMP and LearnedWMP use the same training and testing queries. Specifically, LearnedWMP methods use a histogram representation of the workload as an intake. In comparison, SingleWMP uses the queries themself as the input. Figure 5.4 conveys that LearnedWMP training time was faster than SingleWMP. To illustrate, LearnedWMP-XGB for TPC-C training time was 15.8 ms. A 2x improvement in the training time when compared with SingleWMP-XGB 38 ms time. We notice similar results when we compare the training time in LearnedWMP versus SingleWMP methods on the additional two benchmarks, TPC-DS and JOB. The Ridge model was the only algorithm that demonstrated no significant improvement in the training time between the LearnedWMP and SingleWMP; However, we do see



(a) TPC-DS



(b) JOB

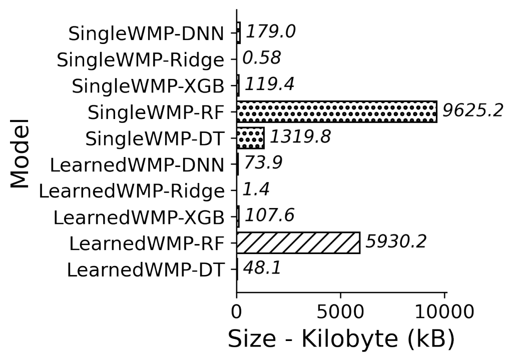


(c) TPC-C

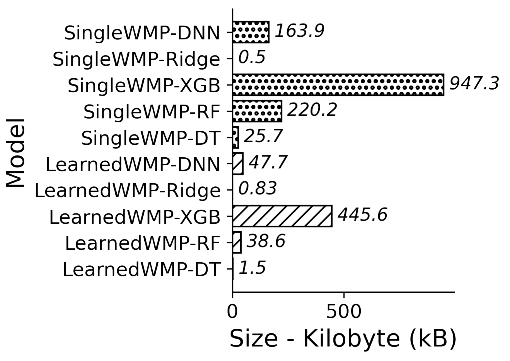
Figure 5.5: ML Model Inference Latency

a significant improvement in the inference time. Figure 5.5 illustrates the inference time results of LearnedWMP and SingleWMP. Similar to the training time results, we also see a significant improvement in LearnedWMP's inference time compared to SingleWMP's inference time. Specifically, we see an improvement of 3x to 10x acceleration when compared to SingleWMP. For example, when examining the TPC-DS benchmark, the inference time for LearnedWMP-DNN was 87.3 $\mu$s. A 10x improvement when compared to the 870.5 $\mu$s required by SingleWMP-DNN. Additionally, LearnedWMP-Ridge had an inference time of 58 $\mu$s. Again, a 10x improvement to the SingleWMP-Ridge 486.9 $\mu$s time. We observe similar results across the other benchmarks.

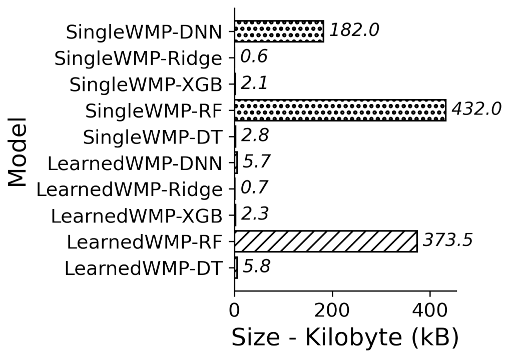
We can attribute the improvements in training and inference latency to our approach of developing a prediction process for a workload of queries instead of single queries. SingleWMP-based models only process one query at a time, which requires a longer computation time for all the queries in a workload. On the other hand, LearnedWMP-based models process batches of queries simultaneously at once. Therefore, it can speed up the computation during both training and inference.



(a) TPC-DS



(b) JOB



(c) TPC-C

Figure 5.6: ML Model Size

**LearnedWMP model size.**

The model size normally depends on the algorithm itself and the feature space complexity of the training data. Figure 5.6 describes the ML model size of LearnedWMP and SingleWMP. LearnedWMP-based models performed quite well in all three benchmarks in comparison to SingleWMP-based models. The LearnedWMP-DNN is 59 percent smaller TPC-DS, 72 percent smaller JOB, and 97 percent smaller TPC-C when compared to SingleWMP-DNN. We observe the same trend for XGBoost, RF, and DT.

The improvement of LearnedWMP's model size, when compared to SingleWMP, is due to how LearnedWMP functions. LearnedWMP focuses on accepting a workload as input, whereas SingleWMP has to process one individual query at a time. The representation of processing the queries at a workload level allows LearnedWMP to reduce the amount of information the model needs to learn during the training process. In principle, this allows LearnedWMP models to be smaller than SingleWMP models. The exemption to this conclusion was the pipeline with the Ridge model. Here, LearnedWMP-Ridge was actually larger than the SingleWMP-Ridge. Still, this outcome was expected. As a linear model, Ridge learns a set of coefficients for each input feature in the training dataset, and LearnedWMP's training examples contain more input features than SingleWMP. LearnedWMP's input is the distribution of query templates. Therefore, Ridge has to learn more coefficients when compared to SingleWMP, causing the model to be larger.

**Effect of the batch size parameter s**

Parameter $s$ is a tunable parameter that represents the size of the batch. In our experiments, we set the batch size to 10, so $s = 10$. In this section, we experiment with different values for $s$ on the TPC-DS dataset. Overall, we test a total of 12 different values for $s$ : [2, 3, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50].

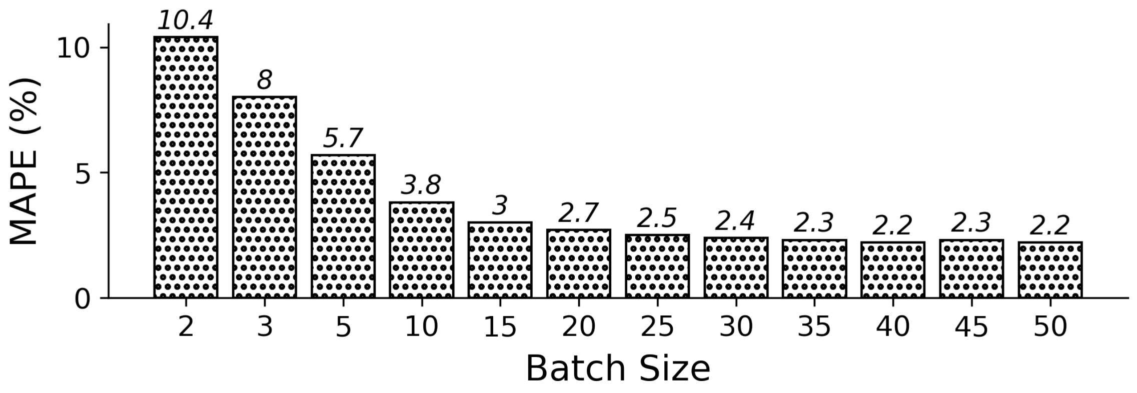


Figure 5.7: TPC-DS Dataset: Mean Absolute Percent Error (MAPE) at different batch sizes

For this test, we generated a training and testing dataset with workloads of different batch sizes $s$, and we used the LearnedWMP-XGB model to evaluate the performance of each batch size. Additionally, we use Mean Absolute Percent Error (MAPE)$^{5.1}$ to compare the errors generated by the 12 different batch sizes.

$$MAPE = \frac{1}{N}\sum_{i=1}^{N}\frac{\mid y_i - \hat{y}_i \mid}{y_i} \times 100 \tag{5.1}$$

We choose MAPE over RMSE for this step as RMSE can be impacted by the scale of the error value. We needed a method to compare the relative performance of models trained with different batch sizes, and MAPE is more suitable for this task as it is not impacted by the change in the error scale. Figure 5.1 illustrates the results from the LearnedWMP-XGB model as a factor of batch size. The figure displays a decreasing error that sharply decreases and then rounds off where there is not much improvement afterward. This is typical behavior for a learning algorithm that is reaching optimum prediction. We choose $s = 10$ for the optimum batch size, as we still get a significant decrease in MAPE before ten and a small improvement afterward. We have also seen similar patterns for TPCC and JOB benchmarks. This observation corroborates our position that batch size prediction is more accurate than single query prediction.

# Chapter 6

# Conclusion and Future Work

## 6.1 Conclusion

DBMS needs to be able to process the maximum number of SQL queries while using all available computing resources in the system. Therefore, being able to accurately predict the resource demand of a batch of queries is highly advantageous for the DBMS. In this research, we designed, developed, and evaluated the method LearnedWMP which focuses on predicting the resource demand of query workloads (e.g., batches of queries). LearnedWMP is a paradigm shift from the state-of-the-art methods, which focus on estimating the resource demand of only one query at a time. Our proposed method operates in three main phases. In the first phase, LearnedWMP learns the query templates from historical queries. Next, In the second phase, LearnedWMP uses the query templates that it learned in phase one to construct the histograms from the workloads. Lastly, in the third and last phase, LearnedWMP uses the histograms to train a predictive model that estimates the memory requirement of a workload. We model the prediction task as a distribution regression problem. We performed a comprehensive experimental evaluation of the LearnedWMP model against the state-of-the-practice method of contemporary DBMSs, multiple sensible baselines, and state-of-the-art methods. Our experiments extend into two OLTP benchmarks and one OLAP benchmark. Our analysis provides evidence that our proposed method can significantly improve the memory estimation of the current state-of-the-art techniques. Additionally, LearnedWMP performs on par with methods that follow a single-query-based approach, produces smaller models that are faster to train, and predicts memory usage during inference to a great extent. We also evaluated different strategies for learning query templates from historical DBMS queries and performed parameter sensitivity analysis. Our proposed LearnedWMP model is novel, offers an alternative perspective to a popular and challenging problem, and can be easily integrated with major DBMS products. Our proposed LearningWMP method is a novel approach that offers an alternative perspective on a challenging problem that can easily be integrated with major database management systems.

## 6.2 Future Work

A number of directions can be taken for future research in this area. Here we briefly discuss some of these ideas.

### 6.2.1 Dataset

Currently, we have used three state-of-the-art benchmarks for LearnedWMP experiments and evaluation. However, there is still the need to test LearnedWMP on an enterprise database system with real queries. We are currently working with industry partners to procure the needed dataset and queries.

### 6.2.2 Resources Prediction

Currently, LearnedWMP is only trained to predict the resource demand of a workload's memory consumption. Future extensions entail being able to predict different resource demands, such as CPU and I/O. Additionally, LearnedWMP could be trained to predict multiple resources simultaneously.

### 6.2.3 Generating Templates

For the generating query templates step, we experimented with two different techniques called cardinality cost aggregation for each operator (CCAEO) and query plan encoding while maintaining tree structure (QPEWMTS). These two approaches used the query plan to extract features for the input to the clustering method. Overall, CCAEO outperformed QPEWMTS in the experimental evaluation due to QPEWMTS generating large vectors, a 600-length vector for the TPC-DS benchmark. QPEWMTS holds great promise as it can maintain the structure of the query plan. Future works will focus on an improved approach for bringing vectors of different lengths, generated by the depth-first search (DFS) approach, to same-length vectors. Such approaches could include Natural Language Processing (NLP) techniques, as they are very useful for encoding vectors of different lengths. Future work in finding query templates could significantly improve the performance of the model.

### 6.2.4 Deep Learning Model

in our LearnedWMP method, we experiment with five different models for machine learning, deep learning, and reinforced learning. Future work would entail exploring additional models such as Recurrent Neural Networks (RNNs), Convolutional Neural Networks (CNNs), Long Short-Term Memory Networks (LSTMs), Deep Q-Networks, Deep Deterministic Policy Gradients (DDPG), and more.

# Bibliography

[1] A. Ganapathi, H. Kuno, U. Dayal, *et al.*, "Predicting multiple metrics for queries: Better decisions enabled by machine learning," in *2009 IEEE 25th International Conference on Data Engineering*, IEEE, 2009, pp. 592–603.

[2] X. Zhou, C. Chai, G. Li, and J. Sun, "Database meets ai: A survey,"

[3] X. Zhou, J. Sun, G. Li, and J. Feng, "Query performance prediction for concurrent queries using graph embedding," *Proceedings of the VLDB Endowment*, vol. 13, no. 9, pp. 1416–1428, 2020.

[4] S. S. Quader, N. A. J. Duran, S. Mukhopadhyay, *et al.*, *Learning-based workload resource optimization for database management systems*, US Patent 11,500,830, Nov. 2022.

[5] V. Leis, B. Radke, A. Gubichev, A. Kemper, and T. Neumann, "Cardinality estimation done right: Index-based join sampling.," in *Cidr*, 2017.

[6] R. Marcus and O. Papaemmanouil, "Plan-structured deep neural network models for query performance prediction," *arXiv preprint arXiv:1902.00132*, 2019.

[7] Y. Han, Z. Wu, P. Wu, *et al.*, "Cardinality estimation in dbms: A comprehensive benchmark evaluation," *arXiv preprint arXiv:2109.05877*, 2021.

[8] B. Hilprecht, A. Schmidt, M. Kulessa, A. Molina, K. Kersting, and C. Binnig, "Deepdb: Learn from data, not from queries!" *arXiv preprint arXiv:1909.00607*, 2019.

[9] Z. Yang, A. Kamsetty, S. Luan, *et al.*, "Neurocard: One cardinality estimator for all tables," *arXiv preprint arXiv:2006.08109*, 2020.

[10] H. Liu, M. Xu, Z. Yu, V. Corvinelli, and C. Zuzarte, "Cardinality estimation using neural networks," in *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering*, 2015, pp. 53–59.

[11] A. Kipf, T. Kipf, B. Radke, V. Leis, P. Boncz, and A. Kemper, "Learned cardinalities: Estimating correlated joins with deep learning," *arXiv preprint arXiv:1809.00677*, 2018.

[12] S. Hasan, S. Thirumuruganathan, J. Augustine, N. Koudas, and G. Das, "Deep learning models for selectivity estimation of multi-attribute queries," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 1035–1050.

[13] K. Kim, J. Jung, I. Seo, W.-S. Han, K. Choi, and J. Chong, "Learned cardinality estimation: An in-depth study," in *Proceedings of the 2022 International Conference on Management of Data*, 2022, pp. 1214–1227.

[14] G. Koloniari, Y. Petrakis, E. Pitoura, and T. Tsotsos, "Query workload-aware overlay construction using histograms," in *Proceedings of the 14th ACM international conference on Information and knowledge management*, 2005, pp. 640–647.

[15] W. Wang, M. Zhang, G. Chen, H. Jagadish, B. C. Ooi, and K.-L. Tan, "Database meets deep learning: Challenges and opportunities," *ACM SIGMOD Record*, vol. 45, no. 2, pp. 17–22, 2016.

[16] G. Lanfranchi, P. Della Peruta, A. Perrone, and D. Calvanese, "Toward a new landscape of systems management in an autonomic computing environment," *IBM Systems journal*, vol. 42, no. 1, pp. 119–128, 2003.

[17] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann, "How good are query optimizers, really?" *Proceedings of the VLDB Endowment*, vol. 9, no. 3, pp. 204–215, 2015.

[18] S. Chu, K. Weitz, A. Cheung, and D. Suciu, "Hottsql: Proving query rewrites with univalent sql semantics," *ACM SIGPLAN Notices*, vol. 52, no. 6, pp. 510–524, 2017.

[19] J. Ortiz, M. Balazinska, J. Gehrke, and S. S. Keerthi, "An empirical analysis of deep learning for cardinality estimation," *arXiv preprint arXiv:1905.06425*, 2019.

[20] J. Lu, Y. Chen, H. Herodotou, S. Babu, *et al.*, "Speedup your analytics: Automatic parameter tuning for databases and big data systems," *Proceedings of the VLDB Endowment*, 2019.

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

[22] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang, "Automatic database management system tuning through large-scale machine learning," in *Proceedings of the 2017 ACM international conference on management of data*, 2017, pp. 1009–1024.

[23] B. Ding, S. Das, R. Marcus, W. Wu, S. Chaudhuri, and V. R. Narasayya, "Ai meets ai: Leveraging query executions to improve index recommendations," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 1241–1258.

[24] S. Jain, B. Howe, J. Yan, and T. Cruanes, "Query2vec: An evaluation of nlp techniques for generalized workload analytics," *arXiv preprint arXiv:1801.05613*, 2018.

[25] L. Ma, D. Van 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*, 2018, pp. 631–645.

[26] S. Das, F. Li, V. R. Narasayya, and A. C. König, "Automated demand-driven resource scaling in relational database-as-a-service," in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 1923–1934.

[27] D. Paul, J. Cao, F. Li, and V. Srikumar, "Database workload characterization with query plan encoders," *arXiv preprint arXiv:2105.12287*, 2021.

[28] J. Li, A. C. König, V. Narasayya, and S. Chaudhuri, "Robust estimation of resource consumption for sql queries using statistical techniques," *arXiv preprint arXiv:1208.0278*, 2012.

[29] J. Sun and G. Li, "An end-to-end learning-based cost estimator," *arXiv preprint arXiv:1906.02560*, 2019.

[30] C. Tang, B. Wang, Z. Luo, *et al.*, "Forecasting sql query cost at twitter," in *2021 IEEE International Conference on Cloud Engineering (IC2E)*, IEEE, 2021, pp. 154–160.

[31] A. S. Higginson, M. Dediu, O. Arsene, N. W. Paton, and S. M. Embury, "Database workload capacity planning using time series analysis and machine learning," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 769–783.

[32] B. Mozafari, C. Curino, A. Jindal, and S. Madden, "Performance and resource modeling in highly-concurrent oltp workloads," in *Proceedings of the 2013 acm sigmod international conference on management of data*, 2013, pp. 301–312.

[33] H. C. L. Law, D. J. Sutherland, D. Sejdinovic, and S. Flaxman, "Bayesian approaches to distribution regression," in *International Conference on Artificial Intelligence and Statistics*, PMLR, 2018, pp. 1167–1176.

[34] R. Li, B. J. Reich, and H. D. Bondell, "Deep distribution regression," *Computational Statistics & Data Analysis*, vol. 159, p. 107 203, 2021.

[35] Y. Mao, L. Shi, and Z.-C. Guo, "Coefficient-based regularized distribution regression," *arXiv preprint arXiv:2208.12427*, 2022.

[36] B. Póczos, A. Singh, A. Rinaldo, and L. Wasserman, "Distribution-free distribution regression," in *Artificial Intelligence and Statistics*, PMLR, 2013, pp. 507–515.

[37] S. Chaudhuri, A. K. Gupta, and V. Narasayya, "Compressing sql workloads," in *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, 2002, pp. 488–499.

[38] T. Xie, V. Chandola, and O. Kennedy, "Query log compression for workload analytics," *Proceedings of the VLDB Endowment*, vol. 12, no. 3,

[39] G. Gan, C. Ma, and J. Wu, *Data clustering: theory, algorithms, and applications.* SIAM, 2020.

[40] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[41] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *Advances in neural information processing systems*, vol. 26, 2013.

[42] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[43] V. H. Makiyama, M. J. Raddick, and R. D. Santos, "Text mining applied to sql queries: A case study for the sdss skyserver.," in *SIMBig*, 2015, pp. 66–72.

[44] G. Kul, D. T. A. Luong, T. Xie, V. Chandola, O. Kennedy, and S. Upadhyaya, "Similarity metrics for sql query clustering," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 12, pp. 2408–2420, 2018.

[45] A. Ghosh, J. Parikh, V. S. Sengar, and J. R. Haritsa, "Plan selection based on query clustering," in *VLDB'02: Proceedings of the 28th international conference on very large databases*, Elsevier, 2002, pp. 179–190.

[46] M. Dash, H. Liu, and X. Xu, "'1+ 1¿ 2': Merging distance and density based clustering," in *Proceedings Seventh International Conference on Database Systems for Advanced Applications. DASFAA 2001*, IEEE, 2001, pp. 32–39.

[47] C. M. Bishop, "Training with noise is equivalent to tikhonov regularization," *Neural computation*, vol. 7, no. 1, pp. 108–116, 1995.

[48] I. Goodfellow, Y. Bengio, and A. Courville, "Deep learning (adaptive computation and machine learning series)," 2016.

[49] R. Shwartz-Ziv and A. Armon, "Tabular data: Deep learning is not all you need," *arXiv preprint arXiv:2106.03253*, 2021.

[50] R. Caruana and A. Niculescu-Mizil, "An empirical comparison of supervised learning algorithms," in *Proceedings of the 23rd international conference on Machine learning*, 2006, pp. 161–168.

[51] Y. LeCun, B. Boser, J. S. Denker, *et al.*, "Backpropagation applied to handwritten zip code recognition," *Neural computation*, vol. 1, no. 4, pp. 541–551, 1989.

[52] H. Salehinejad, S. Sankar, J. Barfett, E. Colak, and S. Valaee, "Recent advances in recurrent neural networks," *arXiv preprint arXiv:1801.01078*, 2017.

[53] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[54] K. P. Murphy, *Probabilistic machine learning: an introduction.* MIT press, 2022.

[55] G. James, D. Witten, T. Hastie, and R. Tibshirani, "An introduction to statistical learning, vol 112 springer," *New York*, 2013.

[56] D. C. Liu and J. Nocedal, "On the limited memory bfgs method for large scale optimization," *Mathematical programming*, vol. 45, no. 1, pp. 503–528, 1989.

[57] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[58] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization.," *Journal of machine learning research*, vol. 13, no. 2, 2012.

[59] S. Raschka, *Python Machine Learning Ed. 3*. Packt Publishing, 2019.

[60] S. Shalev-Shwartz and S. Ben-David, *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.

[61] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 2016, pp. 785–794.

[62] S. T. Leutenegger and D. Dias, "A modeling study of the tpc-c benchmark," *ACM Sigmod Record*, vol. 22, no. 2, pp. 22–31, 1993.

[63] R. O. Nambiar and M. Poess, "The making of tpc-ds.," in *VLDB*, vol. 6, 2006, pp. 1049–1058.

[64] B. Bahmani, B. Moseley, A. Vattani, R. Kumar, and S. Vassilvitskii, "Scalable k-means++," *Proceedings of the VLDB Endowment, Vol. 5, No. 7*, 2012.