

# LearnedWMP: Workload Memory Prediction Using Distribution of Query Templates

Shaikh Quader  
York University, Canada  
shaikhq@yorku.ca

Andres Jaramillo  
York University, Canada  
andrew2jaramillo@gmail.com

Sumona Mukhopadhyay  
California Polytechnic State U, USA  
mukhopad@calpoly.edu

Ghadeer Abuoda  
York University, Canada  
gha@yorku.ca

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

David Kalmuk  
IBM Canada Ltd., Canada  
dckalmuk@ca.ibm.com

Marin Litoiu  
York University, Canada  
mlitoiu@yorku.ca

Manos Papagelis  
York University, Canada  
papaggel@eecs.yorku.ca

## Abstract

In modern database management systems (DBMS), *working memory* often serves as a critical bottleneck when executing in-memory analytic queries, including operations like joins, sorting, and aggregation. Traditional approaches to resource estimation in DBMSs predict query resource consumption by estimating the memory usage of individual database operators within a query execution plan. However, this method is both slow and prone to errors, as it depends on simplifying assumptions such as data uniformity and independence. Furthermore, existing approaches primarily concentrate on individual query optimization without considering the impact of executing multiple queries together as a batch.

In this research, we address query performance optimization in the context of batch execution of multiple queries (*a workload*). Rather than estimating memory requirements for each query separately, we focus on predicting the overall memory demand of an entire workload. To this end, we introduce the problem of *workload memory prediction* and formulate it as a *distribution regression problem*. To address the problem, we propose Learned Workload Memory Prediction (LearnedWMP), a method designed to enhance and streamline memory demand estimation for *a batch of queries*. LearnedWMP first learns low-rank query representations of queries by leveraging structural similarities in their execution plans. This enables queries with similar characteristics to be grouped or clustered and assigned *a query template* that reflects their associated memory demand. Then, for a given batch of queries, LearnedWMP generates a histogram representation of the query templates within the batch and employs a distribution regressor to predict its overall memory demand. Our extensive experimental evaluation demonstrates that LearnedWMP reduces memory estimation errors by up to 47.6% compared to state-of-the-art methods. Additionally, LearnedWMP and its variants outperform alternative single-query models, achieving 3x to 10x faster training and inference times. In most cases, LearnedWMP-based models were also at least 50% smaller in size. Overall, these results highlight the effectiveness of the LearnedWMP approach, underscoring its potential for broader applications in query performance optimization.

## Keywords

Query Performance Prediction, Workload Memory Prediction

## 1 Introduction

Estimating resource usage of database queries is critical to many database operations and decision-making tasks, such as admission control, workload management, and capacity planning [16, 20, 68, 72]. *Working memory* is a region of the system memory where DBMS performs in-memory operations, such as sort and aggregation, while executing queries. Using a limited system memory, the DBMS can decide the optimal time to schedule executing a finite number of in-memory operations (i.e., a set or a batch of queries). Inaccurate estimation of a query's working memory requirement causes the DBMS to either under- or over-commit the memory. This hinders the DBMS from achieving optimal query performance, which includes faster query execution and higher throughput, and could potentially result in query failures. To achieve high performance, the DBMS needs accurate query working memory estimations before grouping and admitting them for execution.

**Motivation.** The DBMS admits queries for execution based on system capacity. Queries are queued and grouped into batches based on similar execution patterns (e.g., query structure) or resource requirements (e.g., CPU, I/O, memory constraints). *A workload* is a set of queries executed under DBMS *batch* execution. Predicting query workload requirements is a critical aspect of workload-based optimization in DBMS, which aims to maximize performance for specific workloads [18, 23, 71]. Accurate memory predictions ensure that queries receive sufficient memory, preventing out-of-memory errors or query abortions [24, 25, 49, 54]. Moreover, the DBMS can delay or reorder execution based on batch memory prediction to avoid excessive memory pressure, select efficient execution plans, or make better scaling decisions for multi-tenant databases and cloud environments [60, 65]. These capabilities underpin self-tuning and self-driving databases, ensuring efficient and autonomous performance [22, 43, 47, 57].

**The State of the Practice & Limitations.** In modern DBMS, the query optimizer's cost model drives estimations for each query independently, without leveraging insights from batches of queries that collectively stress specific resource utilization, such as memory [16]. For example, processing two queries as a batch with each including a group by operation can require a

---

EDBT '26, Tampere (Finland)

© 2025 Copyright held by the owner/author(s). Published on OpenProceedings.org under ISBN 978-3-98318-102-5, series ISSN 2367-2005. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

higher working memory than executing each query separately.<sup>1</sup> Thus, estimation based on simple heuristics, such as calculating the memory demand for a single query and multiplying it by batch size, is not practical. If this collective memory requirement surpasses available system memory, it can increase query execution times, decreasing the DBMS's throughput. As another option, creating a general cost model not tied to specific resources and workloads is challenging because it relies on engineered features that are often tied to specific schemas and query structures, which limit its applicability across a diverse range of database schemas, query structures, and data distributions. Despite dedicated efforts over the past decade, the complexity arising from these variations makes the task difficult and non-generalizable (e.g., [6, 37, 56, 64, 66, 75, 76, 79]). In current practice, database administrators (DBAs) depend on extensive testing and their expertise to define heuristic features and rules for calculating runtime metrics of queries to optimize database configurations leading to significant time and effort investments [46, 74]. To our knowledge, there is currently no practical engine or optimizer capable of predicting a memory requirement with high accuracy for a workload of queries. Predicting memory needs for a *batch of queries* simultaneously carries the premise of delivering a more precise estimate of the overall memory demand, helping to avoid excessive or insufficient memory resource allocation, thereby enhancing system stability and performance. This necessitates the development of memory prediction techniques designed for query batches, enabling the generation of precise memory estimates suitable for DBMS integration [17, 38, 53].

**Our Approach.** We propose a novel approach for estimating the memory demand of a batch of database queries, a *workload*. Our approach is a departure from the existing approach of estimating the resource usage of each query separately [17, 21], especially for cardinality estimation [19, 21, 28, 30, 42, 77, 78], which is a distinct task from working memory estimation but often lead to inaccuracies, contributing to imprecise memory estimations [34, 35]. We exploit the observation that a DBMS admits queries, queues them for execution in batches, and processes each batch based on available resources, where these queries often compete for these resources. By modeling the resource demand of a batch of queries, we expect to achieve higher accuracy in estimating resources. Also, we expect our approach will reduce the development costs of the DBMS's resource estimator and speed up the computation of resource estimations. As an embodiment of our idea, we initially focused on estimating the working memory of database workloads. We design a Learned Workload Memory Prediction (LearnedWMP) model in three steps. *First*, we use an intuition that queries with similar plan characteristics and estimated cardinalities have similar memory demand. Based on this intuition, we use historical queries to learn *query templates* that serve as groups for queries with similar memory demands. *Second*, we randomly divide training queries into fixed-size training workloads and represent each workload as a *histogram – a distribution of query templates*. A histogram-based representation allows capturing the underlying statistical distribution of the queries by grouping them into bins or templates. Histograms have been used in different domains to aggregate multiple observations and obtain approximate data distributions [31]. To simplify the setup, the current design of LearnedWMP uses fixed-length workloads. However, the design

can be extended to work with variable-length workloads. In *the final step*, using training workloads and their historical collective actual memory usage, we train a regression model that learns to estimate the memory usage of an unseen workload based on the distribution of query templates. As the model learns from diverse training workloads, its accuracy at estimating the memory of workloads will improve with time.

**Contributions.** We summarize our key contributions as follows:

- We propose LearnedWMP, a novel prediction model that can estimate the working memory demand of a batch of SQL queries in a workload at once. 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. To our knowledge, this is the first attempt to predict memory demand at the workload level utilizing machine learning (ML) techniques.<sup>2</sup>
- 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 ML to learn the regressor without relying on hand-crafted query-level or operator-level features.
- 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 extensively evaluate our LearnedWMP model employing three database benchmarks. These experimental results demonstrate the merit of our proposed technique in workload-based query processing and resource estimation.

**Summary of Experimental Results.** Our experiments demonstrate that LearnedWMP substantially decreased memory estimation errors compared to current state-of-the-art practices, achieving an impressive improvement of 47.6%. Our experiments are conducted over transactional and analytical database benchmarks to train and evaluate our model. LearnedWMP accepts as input a workload and returns the workload's estimated working memory demand. Our findings indicate that, during training and inferencing, the LearnedWMP model and its variant models were 3x to 10x faster compared to alternative machine learning models. Also, LearnedWMP-based models were at least 50% smaller in most cases. Furthermore, our study delved into the performance and impact assessment of different components and parameters within the model. These included various learning models, workload sizes, numbers of query templates, and techniques for learning query templates.

**Paper Organization.** The rest of the paper is organized as follows. Section 2 reviews the related work. Section 3 introduces key terminology, notations, and the problem formally. Section 4 provides an overview of LearnedWMP, including details of the training and inference stages. Section 5 presents an experimental evaluation of LearnedWMP. Section 6 concludes the paper.

## 2 Related Work

Our research relates to ML methods for (i) database query optimization [8, 26, 27], (ii) database query resource estimation [37], (iii) query-based workload analysis [43], and (iv) distribution regression problems. Each of these areas has extensive research literature, and we discuss some of the most relevant research efforts to our work.

<sup>1</sup>While some DBMS process queries in parallel within a batch, this aspect is beyond the scope of our research.

<sup>2</sup>The source code of our model is available for reproducibility: <https://github.com/shaikhq/learnedwmp>

**ML for Database Query Optimization** In the broader topic of query optimization, besides query resource estimation, many recent works have explored ML techniques to learn various tasks related to query optimization. Some of the key tasks include cardinality estimation [17, 28, 42], query latency prediction [6, 80], index selection [12, 62]. A recent cardinality estimation benchmark [17] evaluated eight ML-based cardinality estimation methods—MSCN, LW-XGB, LW-NN, UAE-Q, NeuroCard, BayesCard, DeepDB, and FLAT. Zhou et al. [80] proposed a graph-based deep learning method to predict the execution time of concurrent queries. Akdere et al. [6] used SVM and linear regression to predict the execution time of analytical queries. Marcus and Papaemmanouil [46] proposed a plan-structured neural network architecture, which uses custom neural units designed at the level of query plan operators to predict query execution time. Ahmad et al. [5] proposed an ML-based method for predicting the execution time of batch query workloads. They relied upon DBAs to define a set of query types used to create simulated workloads and model interactions among queries. Contender [14] is a framework for predicting the execution time of concurrent analytical queries that compete for I/O. Ding et al. [12] applied classification techniques to compare the relative cost of a pair of query plans and use that insight in index recommendations.

**ML for Database Query Resource Estimation** Closer to our problem are methods that estimate resources—such as memory, CPU, and I/O—for executing database queries. Tang et al. used ML to classify queries into low, medium, and high resource consumption. They employed separate models for memory and CPU, utilizing a bag-of-words approach to extract query keywords as input features. XGBoost proved the most accurate among the three compared algorithms [69]. Ganapathi et al. [16] tried five ML techniques to predict run-time resource metrics of individual queries. They achieved the best results with the kernel canonical correlation analysis (KCCA) algorithm. Li et al. [37] applied boosted regression trees to predict individual queries' CPU and I/O costs. They modeled the requirements of each database operator separately by computing a different set of features for each operator type. All the above three approaches [16, 37, 69] generate resource predictions at the level of individual queries. In contrast, our proposed method estimates memory at the level of workload query batches, which we found more accurate and computationally efficient.

**ML for Query-based Workload Analysis** There is a line of research that considers query-based workload analysis. Higginson et al. [20] have applied time series analysis, using ARIMA and Seasonal ARIMA (SARIMA), on database workload monitoring data to identify patterns such as seasonality (recurring patterns), trends, and shocks. They used these time series patterns in database capacity planning. Kipf et al. [30] use Multi-Set Convolutional Network (MSCN) for workload cardinality estimation, representing query features as sets of tables, joins, and predicates. Similar methods have been used for constructing query-based cardinality estimators (e.g., [53]). However, these approaches involve significant sampling overhead or rely on static data featurization, unsuitable for modern databases. Additionally, they lack explanations for the relationships between data, queries, and actual cardinalities. Other workload-based analysis is Flux [40], which focuses on system-level resource management by separating short- and long-running queries into independently scalable clusters based on performance metrics and workload forecasts. Ortiz et al. [55] proposed a PSLA-based

method that generates synthetic queries from schema statistics to model runtimes and guide SLA planning. In contrast, LearnedWMP provides query-level memory prediction by learning query templates from real workloads and execution plans, enabling accurate resource provisioning and efficient scheduling. Unlike DBSeer [50], which uses clustering (e.g., DBSCAN) to group transactions and predict resource demand, LearnedWMP does not rely on clustering; instead, it learns from simple features in query plans that exhibit a stronger correlation with memory usage. Our experiments demonstrate that  $k$ -means clustering outperforms DBSCAN for resource prediction. While Flux and PSLA address infrastructure scaling and SLA guarantees, LearnedWMP focuses on fine-grained, data-driven memory estimation at the query processing level.

**ML for Distribution Regression Problems** Distribution regression has emerged as a popular ML approach for mapping complex input probability distributions to real-valued responses, particularly in supervised tasks that require handling input and model uncertainty [32], emerging as a promising alternative to traditional techniques like random forests and neural networks [39]. To our knowledge, we are the first to apply distribution regression to model resource demand forecasting of database workloads. Outside the database domain, some of the illustrative use cases of distribution regression include predicting health indicators from a patient's list of blood tests [45], solar energy forecasting, and traffic prediction [39]. Many recent papers have offered approaches and optimization techniques for solving distribution regression tasks (e.g., [32, 39, 45]).

### 3 Preliminaries and the Problem

In this section, we introduce notation and preliminaries to assist in defining the *workload memory prediction problem*. Next, we formally present our novel approach to representing and solving the problem as a *distribution regression problem*.

**Query** Let  $q = (e, p, m)$  be a single SQL query where (i)  $e$  is a query expression received from a database user, (ii)  $p$  is a query execution plan generated by the DBMS optimizer for evaluating  $e$ , and (iii)  $m$  is the actual highest working memory usage of the query for  $p$ .  $m$  is available only for training queries that the DBMS has already executed. For unseen queries,  $m$  is unknown.

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

$$y = \sum_{i=1}^{|Q|} m_i \quad (1)$$

$y$  value of a workload (eq. 1) is only present for the training workloads executed by the DBMS. In the inference phase, LearnedWMP receives only  $Q$ , a collection of queries without  $y$ .

**Workload Memory Prediction** Let us assume a training corpus of  $n$  workloads as follows:

$$\{(w_1, y_1), \dots, (w_n, y_n)\} \quad (2)$$

Here, each tuple,  $(w_i, y_i)$  corresponds to the highest historical working 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 highest working memory usage  $y$ :

$$\hat{f}(w) = y \quad (3)$$

**Table 1: Summary of key notations**

Symbol	Description
$w$	A workload.
$Q$	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 $\mathcal{T}$ .
$f(\hat{\mathcal{H}})$	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 $Q$ in a workload $w$ .
$\hat{y}$	The predicted collective memory demand of all queries $Q$ in an unseen workload $w$ as estimated by $f(\hat{\cdot})$ .

**Working Memory** is a memory region used to store intermediate results, temporary data structures, and execution context information while executing database operations such as sorting and aggregation. A query may use the working memory up to a limit that is controlled by a DBMS setting. Working memory size in a DBMS varies based on system configuration, workload, and available resources. In the rest of this paper, for brevity, we may refer to working memory as only memory.

We formulate estimating memory usage of an unseen workload as a distribution regression problem [58, 67], where the estimate is computed from an input probability distribution - the distribution of queries  $Q$  among templates  $\mathcal{T}$ .

**Query Templates** Let  $\mathcal{T} = \{t_1, \dots, t_k\}$  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. Any query  $q$  can be mapped to a query template  $t_i \in \mathcal{T}$ .

**Workload Histogram** Let  $w$  be a workload consisting of a set of  $Q$  queries.  $c_i$  is the number of queries in  $Q$  that can be mapped to query template  $t_i \in \mathcal{T}$ . The counts of queries in  $Q$  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,  $\mathcal{H} = [c_1, \dots, c_k]$  and

$$\sum_{i=1}^k c_i = |Q| \quad (4)$$

From such an input distribution, encoded in a *workload histogram*, a distribution regression function computes as estimated memory usage for the workload. Assume we have a training corpus of  $n$  workload histograms, one histogram per workload, as follows:

$$\{(\mathcal{H}_1, y_1), \dots, (\mathcal{H}_n, y_n)\} \quad (5)$$

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 the following:

- (1) The distribution of queries among the query templates (i.e., the workload histogram bins) is uniform.
- (2) 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,  $\mathcal{H}$ .

$$f(\mathcal{H}) = y \quad (6)$$

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

$\hat{f}(\cdot)$ , an approximation of  $f(\cdot)$ , using the distribution of regression. 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  $y$ .

$$\hat{f}(\mathcal{H}) = \hat{y} \quad (7)$$

Using training workloads labeled with their actual memory usage,  $f(\cdot)$  learns to estimate the memory usage of unseen workloads. We expect that the larger and more diverse the training data set of workload examples is, the more precise the predictor  $f(\hat{\cdot})$  will be. Table 1 provides a summary of the key notations.

## 4 The LearnedWMP Model

LearnedWMP comprises two stages: training and inference. The training stage employs a machine learning pipeline and dataset to build the model, while the inference stage utilizes the trained model to predict memory usage for unseen workloads. Fig. 1 offers an overview of the workflow, and we subsequently outline and delve into the technical details of each step.

### 4.1 Overview of the LearnedWMP's pipeline

**Users and the Database.** The top left section of Fig. 1 illustrates user-database interaction, with the database interacting with two LearnedWMP stages. The users and applications send SQL queries to the database. To calculate the memory utilization of a workload, we sum the highest memory usage for queries in that workload. The LearnedWMP training pipeline (right of Fig. 1) periodically retrains the model using the latest query log dump.

**Training Stage.** In Fig. 1 (top), TR1 through TR6 are the steps of the training pipeline. Training begins with a set of training queries,  $Q_{train}$ , collected from a dump of the DBMS query log. At TR1, from  $Q_{train}$ , the pipeline extracts training queries, their final execution plans, and the actual highest working 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  $|Q_{train}| \times m$  feature matrix. At TR3, the pipeline learns  $\mathcal{T}$ , a set of  $k$  query templates, from the query feature matrix. The value of  $k$  can be determined experimentally (cf. Section 4.2.1). At TR4, the pipeline equally divides the training queries of  $Q_{train}$  into a set of  $n$  workloads,  $\mathcal{W} = |Q_{train}|/s$ . Each workload contains  $s$ , a constant, queries. We found a value of  $s$  experimentally (cf. Section 5.3). At TR5, the pipeline generates a workload histogram  $\mathcal{H}$  for each training workload  $w = (Q, y)$  in  $\mathcal{W}$ .  $\mathcal{H}$  represents the distribution of queries of  $Q$  among  $k$  query templates of  $\mathcal{T}$ . In addition, the collective actual highest memory utilization  $y$  of the workload  $w$  is computed by summing up the working memory utilization of each query  $q_i \in Q$ . Each  $(\mathcal{H}, y)$  pair represents a training example for training a regression model. At TR6, the model receives as input a 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 working memory demand,  $y$ . At the end of TR6, the training pipeline produces a trained LearnedWMP model.

**Inference Stage.** In Fig. 1 (bottom), IN1 through IN5 are the steps of the inference pipeline, which generates estimated memory usage of an unseen workload  $w$ , consisting of  $Q$  queries. Step IN1 collects the query plans of  $Q$  queries in  $w$ ; step IN2 generates the feature vectors for these plans. Step IN3 assigns each query  $q_i \in Q$  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

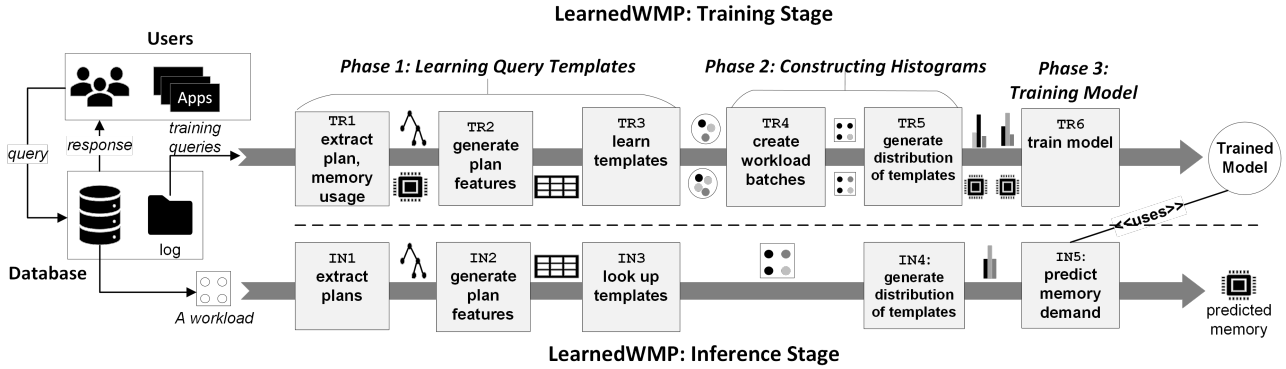


Figure 1: Overview of the LearnedWMP model. Left: Users send queries to a DBMS; the DBMS processes and sends responses to the queries. Right-upper: the training steps of the LearnedWMP model. Right-bottom: the steps of the inference stage.

to the LearnedWMP and predicts the memory usage  $\hat{y}$  of the workload,  $w$ .

## 4.2 LearnedWMP: Training Stage

The LearnedWMP model is trained in six steps (TR1 - TR6), grouped in three phases (see Fig. 1). The first phase uses historical queries from a DBMS’s log to learn a set of *query templates*. The second phase prepares the *training dataset* for the learning algorithm. The third phase uses the training dataset to train a *regression model*, which can predict the memory usage of an input workload.

**4.2.1 Phase 1: Learning Query Templates.** We assign queries to templates based on the intuition that queries with similar query-plan characteristics and cardinality estimates have similar execution-time memory usage [33]. In each workload, by grouping similar queries in the same templates or clusters, we expect to model the memory requirements of the queries more efficiently and speed up the computation of training and inference stages. We are not looking for an optimal template assignment for each query, which will require computation of operator-level features, increase computation overhead, and outweigh the acceleration we hope to gain from compressing queries into templates. Also, estimating the cost of individual queries is a separate research problem [16, 37] 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 template assignment needs to be just good enough, not highly accurate. This allows for designing simple but efficient methods for assigning queries to templates that neither jeopardize the runtime cost nor the machine learning approach. This is not an algorithmic simplification but rather an algorithmic design choice.

To assign queries to templates, we experimented with various classes of techniques. We evaluated each method performance (Section 5.3). Algorithm 1 describes the steps of GETTEMPLATES function that uses training queries of  $Q_{train}$  to learn a set of  $k$  query templates,  $\mathcal{T}$ . For each query in  $Q_{train}$ , GETTEMPLATES() first extracts the query plan (line 5). A query plan is a tree-like structure where each node corresponds to a database operator. The plan’s execution begins at the leaf nodes and completes at the root node. The root’s output is the result of the entire query’s executing. Each node has an input and an output. When applicable, a node includes statistics, such as estimated pre-cardinality and post-cardinality of the operator. For each operator type in a query plan, GETTEMPLATES() counts its frequency and aggregate

cardinalities of all its instances. The frequency count and aggregated cardinality of each operator type are retrieved (line 6) and used to represent the *query features*. Finally,  $k$ -means algorithm [44], or any clustering technique, uses these query feature vectors to learn  $k$  clusters, each one representing a query template  $t_i \in \mathcal{T}$  (line 8). We use the elbow method to tune the value of  $k$ . Fig. 2 provides an illustrative example query, where its associated query plan tree (below) has five unique operators: TBSCAN, HSJOIN, INDEX SCAN, SORT, and GROUP BY. Since each operator type provides a (count, cardinality) pair of features (# of operators, total cardinality), this sample query plan has 10 features - 2 features for each of the 5 operators. GETTEMPLATES() encodes these features in a 1- $d$  vector as follows: [4, 139532.48, 3, 50224.6, 1, 3201, 1, 134179, 1, 48873.6]. The approach of featurizing queries is inspired by earlier research [13, 16] as the authors identified this set of features as effective for predicting query performance.

**4.2.2 Phase 2: Constructing Histograms from Workloads.** In this phase, LearnedWMP performs two tasks: (i) partitions training queries of  $Q_{train}$  into a set of workloads,  $\mathcal{W}$  and (ii) from each training workload,  $w_i \in \mathcal{W}$ , constructs a histogram that represents the distribution of its queries among the query templates set  $\mathcal{T}$ . Histograms have played a significant role in estimating query plans cost [7]. LearnedWMP randomly divides training queries from  $Q_{train}$  into  $m$  training workloads, where  $m = |Q_{train}|/s$  with  $s$  being a constant number of training queries per workload. The value for  $s$  depends on the application domain and can be empirically found (cf. Section 5.3). As defined in *definition 2.2*, each training workload,  $w_i$ , is a tuple  $(Q, y)$ , where  $Q$  is a collection of queries and  $y$  is their collective memory usage from the past execution.

Algorithm 2 describes the steps of phase 2. It takes as input a training workload,  $w$ , and a set of query templates,  $\mathcal{T}$ , which were learned in phase 1. In lines 6-7, for each query  $q_i \in Q$ , the algorithm extracts the query execution plan and the features from the plan. Since phase 1 already computed these features, line 7 reuses the values from the previous computation. Using these features, line 8 looks up the query template,  $t_j \in \mathcal{T}$ , for  $q_i$ . After assigning each query,  $q_i \in Q$ , to a template, the algorithm counts the number of queries in  $Q$  in each template,  $t_j \in \mathcal{T}$  (line 10) and stores the counts in a histogram, a 1- $d$  count vector,  $\mathcal{H} = [c_1, \dots, c_{k=|\mathcal{T}|}]$ . The length of  $\mathcal{H}$  is  $k$ , corresponding to the number of query templates in  $\mathcal{T}$ . Each count  $c_j \in \mathcal{H}$  is the number of queries in  $w$  that are associated with the query template  $t_j \in \mathcal{T}$ . These counts add up to  $s = |Q|$ , the workload

**Algorithm 2** Histogram construction from training workloads

```

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

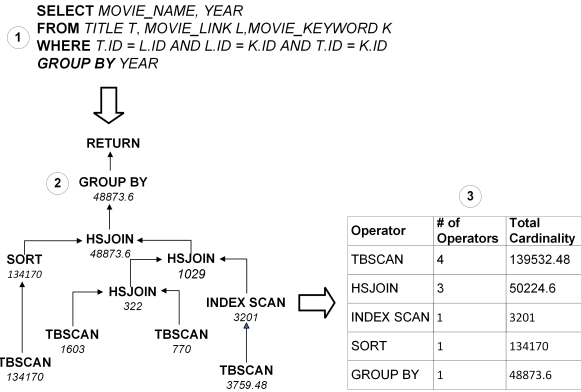
```

**Algorithm 1** Learning query templates with  $k$ -means clustering

```

1:  $Q_{train} \leftarrow \{q_1, q_2, \dots, q_n\}$  ▷  $Q_{train}$  is a set of historical training queries collected from a DBMS.
2: function GETTEMPLATES( $Q_{train}$ )
3:    $Array Z \leftarrow []$  ▷ feature matrix for  $Q_{train}$ 
4:   for  $q_i \in Q_{train}$  do
5:      $plan_i = \text{getQueryPlan}(q_i)$ 
6:      $features_i = \text{getFeatures}(plan_i)$ 
7:      $Z.insert(features_i)$ 
8:   end for
9:    $\mathcal{T} \leftarrow kmeans(Z, k)$  ▷ learns templates using kmeans
return  $\mathcal{T}$  ▷  $k$  learned query templates
10: end function

```



**Figure 2: Extracting query features from a query.** An example query (top) is executed by the query plan (left), resulting in the extraction of query features (right). The query features are used to learn a set of query templates  $\mathcal{T}$ , of size  $k$ .

batch size:

$$\sum_{j=1}^{k=|\mathcal{T}|} c_j = s \quad (8)$$

The histogram  $\mathcal{H}$  is the distribution of queries in workload  $w$  among the query templates set  $\mathcal{T}$ . The histogram will be sparse with many zeros as a workload is not expected to contain queries that belong to each query template of  $\mathcal{T}$ . At the final step (line 11), the algorithm returns a pair  $(\mathcal{H}, y)$ , where  $y$  is the collective memory usage of all  $q \in Q$ .  $(\mathcal{H}, y)$  becomes a labeled example for training a supervised ML model in Phase 3.

Fig. 3 shows an example of constructing a histogram  $\mathcal{H}$  of 4 bins,  $k = |\mathcal{T}| = 4$ . The example uses an input workload,  $w$ , with 9 queries,  $s = |Q| = 9$ . 3 of 4 histogram bins are populated with nonzero values. The remaining bin has a zero value as its corresponding query template has no queries in  $w$ . The histogram vector is  $[3, 4, 0, 2]$ . The value of  $y$  is the total actual memory usage of all 9 queries in  $w$ . Let's assume  $y = 125$  MB. The output pair for this example workload is  $([3, 4, 0, 2], 125)$ .

**4.2.3 Phase 3: Training a Distribution Regression Deep Learning Model.** In this phase, we train a regression model for predicting workload memory usage. The trained model takes an input workload, represented as a histogram of query templates, and computes the workload's memory usage. We explored several ML and deep learning (DL) techniques to train the model. In this subsection, we present the design and implementation of a DL model for our regression model. Section 4.2.4 describes the other ML algorithms we explored for the model training. DL recently 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 [63]. Additionally, DL 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.

**Multilayer Perceptron (MLP) Model.** In our case, the input vector for each workload is structured and has a fixed length corresponding to the number of query templates. Each vector element represents the workload queries of a specific template. Since we want to learn a regression function from fixed-length input vectors, the multilayer perceptron (MLP) is a suitable choice for learning a regression function from fixed-length input vectors due to its assumption of fixed input dimension and flexibility in architecture [51]. In our case, from  $n$  training examples  $(\mathcal{H}_1, y_1), (\mathcal{H}_2, y_2), \dots, (\mathcal{H}_n, y_n)$ , a MLP model learns a function  $f(\cdot) : R^k \rightarrow R^o$ , where  $k$  is the number of dimensions for input  $\mathcal{H} = [c_1, \dots, c_{k=|\mathcal{T}|}]$  and  $R^o$  is the scalar output  $y$ .

**Activation Function.** The activation function in each layer determines how the input is transformed. We explored linear and Rectified Linear Unit (ReLU). For simpler datasets with fewer query templates, linear activation performed better, while ReLU proved more effective for complex datasets with more query templates.

**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:

$$Loss(\hat{y}, y, W) = \frac{1}{2N} \sum_{i=1}^N \|\hat{y}_i - y_i\|_2^2 + \frac{\alpha}{2N} \|W\|_2^2 \quad (9)$$

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. The MLP begins with random weights and iteratively updates them to minimize the loss by propagating the loss backward from the output layer to the preceding layers and updating the weights in each layer. The training uses stochastic gradient descent (SGD), where the gradient  $\nabla Loss_W$  of the loss with respect to the weights

**Algorithm 3** Predict workload memory by rained MLP

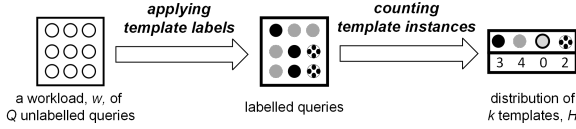
---

```

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

```

---



**Figure 3: An example of generating a histogram  $\mathcal{H}$  from a training workload  $w = (Q, y)$ , where  $|Q| = 9$  and  $k = 4$ .**

is computed and subtracted from  $W$ . More formally,

$$W^{i+1} = W^i - \epsilon \nabla \text{Loss}_W^i \quad (10)$$

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

**Optimizer.** We compared L-BFGS [41] and Adam [29] 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[61].

**Hyperparameter Tuning of the MLP Model.** We tuned hyperparameters, including the number of hidden layers, the number of nodes in each layer, the optimizer, and the dropout rate. Given the large dataset and parameter search space, we employed randomized search using *scikit-learn* library [2]. We found a neural network architecture with eight layers (input, six hidden, and output), and the hidden layers contained 48, 39, 27, 16, 7, and 5 nodes, while the input layer received workload histograms or query distribution, and the output layer provided estimated memory demand.

**Model Complexity.** Assume  $n$  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.

**4.2.4 Other machine learning methods.** Besides deep learning networks, for a comparative analysis, we explored additional ML techniques to train LearnedWMP models. They include a *linear* and three *tree-based* techniques. 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)**. DT uses a single tree for predictions, while Random Forest employs an ensemble of trees that consider random feature subsets, resulting in better generalization and outlier handling capabilities [59]. Finally, XGBoost [9], a gradient boosting tree technique that has achieved high performance for many ML tasks based on tabular data [63].

### 4.3 LearnedWMP: Inference Stage

Algorithm 3 describes the steps of PREDICTMEMORY() function that estimates the memory demand of an unseen workload,  $w$ . The function operates with two models: a trained  $k$ -means clustering model,  $\mathcal{T}$ , which represents a set of learned query templates, and a trained predictive model for estimating workload memory demand. PREDICTMEMORY() receives as input an unseen workload whose memory demand needs to be estimated. At line 5, the function generates a histogram vector,  $\mathcal{H}$ , a distribution of query templates. This step reuses the BINWORKLOAD() function of algorithm 2. Next, line 6 estimates working memory demand  $\hat{y}$  of  $w$  using  $\mathcal{H}$ .

## 5 Experimental Evaluation

We would like to reiterate that LearnedWMP is an innovative model addressing the novel problem of predicting memory for a workload. In this section, we experimentally evaluate LearnedWMP through this prism. In our first set of experiments, we demonstrate how LearnedWMP estimates memory for a workload using different ML models and metrics, revealing a significant reduction in estimation error compared to alternative baselines for predicting query memory demand. Second, we show that LearnedWMP is more efficient in terms of training, inference time, and model size when compared to single-based query models. Finally, we conduct an analysis to study the impact of the major parameters of the LearnedWMP model and the choice of the template learning method.

**Baselines.** In the workload memory prediction problem, we aim to estimate the aggregate memory demand for a query workload. As there are no existing libraries or methods for this novel problem, we develop and evaluate different variants of LearnedWMP and compare them with the state-of-the-art single-based models for predicting query memory demand and assessing their performance.

- **LearnedWMP-based Methods.** LearnedWMP accepts as input a workload  $w$  and returns the workload's estimated working memory demand,  $\hat{y}$ . As described in Section 4.2.4, besides the proposed MLP-based deep neural network (DNN) method, we can use other ML techniques to learn the memory estimation regression function. Thus, we explored additional ML techniques, such as Ridge, DT, RF, and XGB, in this experiment. We refer to the LearnedWMP-based models trained with different ML techniques as *LearnedWMP-DNN*, *LearnedWMP-Ridge*, *LearnedWMP-DT*, *LearnedWMP-RF*, and *LearnedWMP-XGB*.
- **SingleWMP-based Methods.** An alternative approach to estimate the workload's working memory is to rely on single-query-based methods. In this approach, first, the highest working memory requirement for each query in the workload is estimated separately. This estimation depends on the cost model and heuristics implemented in the DBMS engine. Then, these individual estimates are summed up to produce the aggregate working memory estimation for the workload. We refer to this method as Single-query based Workload Memory Prediction (SingleWMP). Formally, given a workload  $w$ , consisting of a set of  $Q$  queries, the estimated memory demand  $\hat{y}$  of  $w$  is:

$$\hat{y} = \sum_{i=1}^{|Q|} \hat{y}_{q_i} \quad (11)$$

where  $\hat{y}_{q_i}$  is the estimated memory demand of a single query  $q_i \in Q$ . In this approach, we use query plan features of each

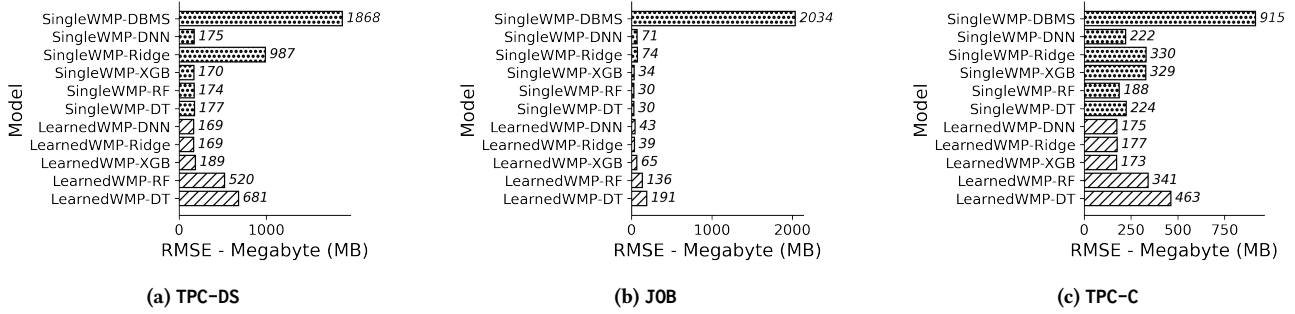


Figure 4: Root Mean Squared Errors (smaller is better)

query as direct input to an ML algorithm. During the training, the algorithm additionally receives the historical memory usage of each training query. Using the pairs of query plan features and memory usage of many individual training queries, the algorithm learns a function to estimate the memory demand of individual queries. Unlike the LearnedWMP approach, the SingleWMP approach does not learn query templates from historical queries. Like LearnedWMP, we use DNN, Ridge, DT, RF, and XGB techniques to train five variations of the SingleWMP models. We refer to these variants of SingleWMP as *SingleWMP-DNN*, *SingleWMP-Ridge*, *SingleWMP-DT*, *SingleWMP-RF*, and *SingleWMP-XGB*. An additional method of the SingleWMP approach is *SingleWMP-DBMS*, which obtains the estimated memory usage for each query directly from a DBMS’s query optimizer. SingleWMP-DBMS does not use ML in the memory estimation; instead, it relies on heuristics written by database experts. SingleWMP-DBMS represents the *current state of practice* in the commercial DBMSs.

**Datasets.** We used three popular database benchmarks: TCP-DS[52] Join Order Benchmark or JOB[34], and TPC-C[36]. TPC-DS and JOB are benchmarks for analytical workloads, whereas TPC-C is for transactional workloads. We used either its query generation toolkit or the seed query templates for each benchmark to generate queries for our experiment. We generated 93000 queries for TPC-DS, 2300 queries for JOB, and 3958 queries for TPC-C. For each benchmark, we randomly divided the queries into training ( $Q_{train}$ ) and test ( $Q_{test}$ ) partitions, with 80% of the queries belonging to  $Q_{train}$  and the remaining 20% to  $Q_{test}$ . We grouped queries into workload batches from training and test partitions. We experimented with different batch sizes and found 10 to be a decent size to improve the memory estimation of our experimental workloads. We discuss our experiment with batch size parameter in Section 5.3.

**Evaluation metrics.** To evaluate the accuracy performance of various models, we use two accuracy metrics:

- **Root Mean Squared Error (RMSE):** For measuring the accuracy performance of the LearnedWMP and SingleWMP models. We use  $L_2$  loss or root mean squared error (RMSE), as follows:

$$RMSE = \sqrt{\frac{\sum_{i=1}^N (y_i - \hat{y}_i)^2}{N}} \quad (12)$$

We seek to find an estimator that minimizes the RMSE.

- **IQR and Error Distribution:** While RMSE is convenient to use, it does not provide insights into the distribution of prediction errors of a model. Two models can have similar RMSE scores but different distributions of errors. For each benchmark, we compute the signed differences between the actual and the predicted memory estimates - the residuals of errors. We use the

residuals to generate violin plots [11], which help us compare the interquartile ranges (IQR) and the error distributions of different models. IQR is defined as follows.

$$IQR = q_n(0.75) - q_n(0.25) \quad (13)$$

Here,  $q_n(0.75)$  is the 75th percentile or the upper quartile, and the  $q_n(0.25)$  is the 25th percentile or the lower quartile. The range of values that fall between two quartiles is called the interquartile range (IQR). IQR is shown as a thick line inside the violin in a violin plot. A white circle on the IQR represents the median. When a model’s violin is closer to zero and has a smaller violin, it is more accurate.

In addition to computing accuracy, for each ML-based LearnedWMP and SingleWMP model, we measured the *model size* in kilobyte ( $kB$ ), the *training time* in millisecond ( $ms$ ), and the *inference time* in microsecond ( $\mu$ ).

**Experiments Design.** In Section 5.1, we evaluate the performance of LearnedWMP-based models compared to that of SingleWMP-based models. The computational overhead of the LearnedWMP model is discussed in Section 5.2. This includes the model size and runtime cost of training and inference of LearnedWMP-based models and how it compares to SingleWMP-based models’. Section 5.3 performs a sensitive study for the parameters and design choices of the LearnedWMP model. 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 LearnedWMP Accuracy Performance

We report on LearnedWMP’s accuracy performance in terms of RMSE and the distribution of error residuals presented as violin plots for completeness.

**RMSE.** Fig. 4 reports the RMSEs of SingleWMP-based and LearnedWMP-based models. SingleWMP-DBMS represents the state of practice in commercial DBMSs. LearnedWMP models and ML-based SingleWMP models significantly outperformed the SingleWMP-DBMS model. For the TPC-DS, SingleWMP-DBMS’s RMSE was 1868. In comparison, LearnedWMP-DNN and LearnedWMP-Ridge, the two best models, achieved an RMSE of 169, representing a 90.95% reduction in estimation error compared to SingleWMP-DBMS.

On RMSE, ML-based methods – LearnedWMP-based models and SingleWMP-based ML models – were significantly more accurate than SingleWMP-DBMS method. Using heuristics, SingleWMP-DBMS couldn’t accurately capture the complex interactions between database operators within a query plan and produced large estimation errors. In contrast, using ML, LearnedWMP-based, and SingleWMP-based ML models learned to estimate the

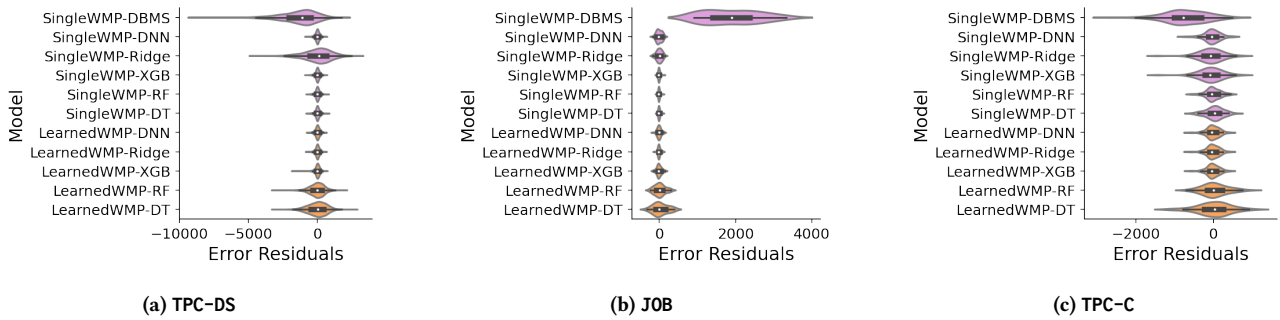


Figure 5: Estimation Error Residuals Distributions

memory requirements of complex database workloads more accurately.

**IQR and Error Distribution.** Fig. 5 compares the violin plots of different models. The violins of the SingleWMP-DBMS are wider and far from zero. DBMS’s estimations are skewed towards either underestimation or overestimation and span a larger region. In contrast, ML-based estimates are balanced between over and under-estimations and are not skewed. The violins of ML-based models span a smaller range than SingleWMP-DBMS’s. For TCP-DS, LearnedWMP-DNN’s violin is centered at zero and small. For the same dataset, SingleWMP-DBMS’s violin is skewed toward underestimation and larger when compared to the IQR of LearnedWMP-DNN. We see a similar pattern when comparing the violins of SingleWMP-DBMS models with the violins of other ML models. Using human-crafted rules, SingleWMP-DBMS’s estimation errors are not distributed between overestimations and underestimations. These static rules skew the estimations toward one direction. In contrast, ML-based models learn from real-world workloads – which include examples of both overestimation and underestimation – and learn to compute memory predictions that are not skewed in one direction. For instance, the memory estimation errors of the DNN and XGboost methods for both SingleWMP-based and LearnedWMP-based models are smaller and balanced.

## 5.2 Computational Overhead

We evaluate the LearnedWMP-based and SingleWMP-based models overhead in terms of training time, inference time, and model size.

**Training and Inference Time.** Fig. 6 reports the training time of all models<sup>3</sup>. For each dataset, LearnedWMP-based and SingleWMP-based methods use the same set of training queries as input. The singleWMP-based method uses individual training queries directly as input to the model. LearnedWMP batches the training queries into workloads, represents workloads as histograms, and uses the histograms as input to the models. Compared to SingleWMP-based models, the training of LearnedWMP-based models was significantly faster. For instance, with the TPC-DS dataset, SingleWMP-XGB was trained in 912.7 ms, whereas LearnedWMP-XGB was trained in 404 ms – which is more than 2x faster. For all three datasets, we observe a similar trend: the training of a LearnedWMP-based model was faster than that of the equivalent SingleWMP-based model. The Ridge is the only algorithm that did not demonstrate a significant difference in training time between the LearnedWMP and SingleWMP approaches. This

<sup>3</sup>Note that for this set of experiments, we do not consider LearnedWMP-DBMS as it is not an ML model, and it does not have a training and an inference cost. It is expected as Ridge is a linear algorithm with no sophisticated learning method.

Fig. 7 compares the inference time of SingleWMP-based and LearnedWMP-based models. The LearnedWMP-based models achieved between 3x and 10x acceleration compared to their equivalent SingleWMP-based models. As an example, for inference of TPC-DS workloads, LearnedWMP-DNN took 87.3  $\mu$ s as compared to 870.5  $\mu$ s needed by SingleWMP-DNN. Similarly, with JOB, LearnedWMP-XGB needed 313.3  $\mu$ s for inference, while SingleWMP-XGB took 1115.6  $\mu$ s. Accelerated training and inference performance of the LearnedWMP models can be attributed to our approach of formulating the training and inference task at the level of workloads, not at the level of individual queries. LearnedWMP-based models process batches of queries at the same time and, therefore, speed up the computation during both training and inference. In contrast, SingleWMP-based models require a longer time for training and inference as they work one query at a time.

**Model Size.** The model size largely depends on the learning algorithm and the feature space complexity of the training set. Fig. 8 shows the size of LearnedWMP-based and SingleWMP-based models. LearnedWMP-based models were significantly smaller when compared with equivalent SingleWMP-based models. For example, when compared with SingleWMP-DNN, LearnedWMP-DNN is 59% TPC-DS, 72% JOB, and 97% TPC-C smaller than SingleWMP-DNN. We see a similar pattern with XGBoost, RF, and DT when comparing their LearnedWMP-based models with equivalent SingleWMP-based models. Batching training queries as workloads compress the information a LearnedWMP model needs to process during training. This compression helps the LearnedWMP approach produce smaller models compared to those of the SingleWMP approach. Ridge is an exception to this observation. The size of a LearnedWMP-Ridge model is larger than its equivalent SingleWMP-Ridge model. This was expected as Ridge learns a set of coefficients, one for each input feature in the training dataset. In our training datasets, each LearnedWMP training example has more input features, one per query template, than the number of features in a SingleWMP training example. As a result, LearnedWMP-Ridge learns more coefficients during training and produces larger models.

## 5.3 Sensitivity Analysis

In the next set of experiments, we investigate the impact of various parameters of LearnedWMP, such as the batch size parameter  $s$ , the number of query templates  $K$ , the effect of variable batch size  $s$ , the choice of the learning query templates techniques, and their effect on the memory prediction accuracy.

**5.3.1 Learning Query Templates.** In the first phase of LearnedWMP, queries are assigned to templates based on their similarity in query plan characteristics and estimates, with the expectation

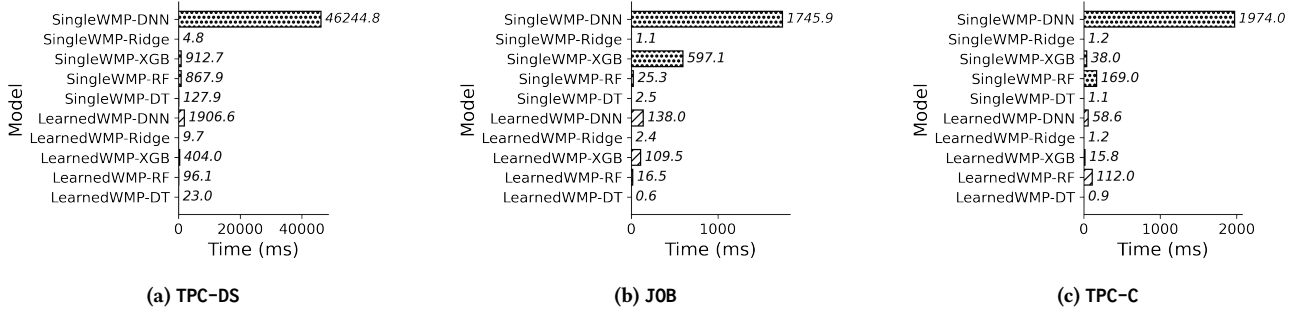


Figure 6: ML model training time

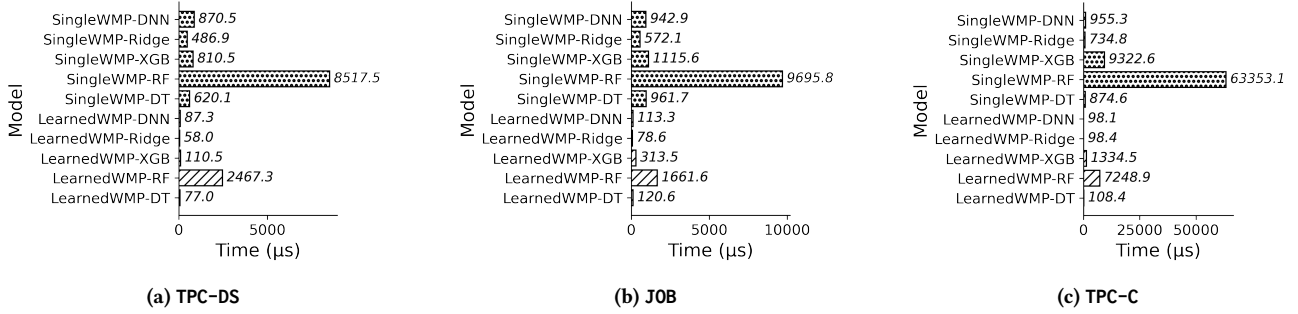


Figure 7: ML model inference time

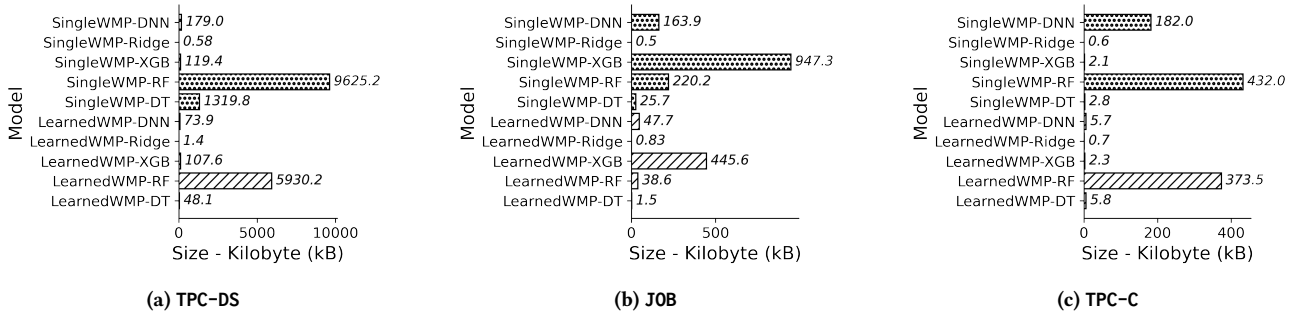
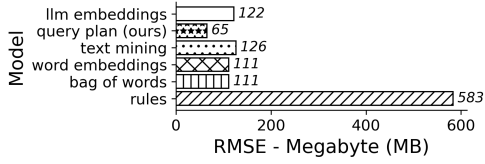


Figure 8: ML model size

that queries in the same template exhibit similar memory usage (See Section 4.2.1). We evaluated our method against four other approaches for learning query templates:

- (1) **Query plan based:** Our proposed LearnedWMP model assigns queries to query templates by extracting features from the query plan and then employing a  $k$ -means clustering algorithm. Details can be found in Section 4.2.1.
- (2) **Heuristic Rule based:** We create a set of rules, one per template, to classify a query statement into one of the pre-defined query templates. Subject matter experts, such as DBAs, may need to be involved in defining these rules [80].
- (3) **Bag of Words based:** We extract unique keywords from the entire query corpus to build a vocabulary. Each query expression generates a feature vector representing the count of each vocabulary word in the query. The  $k$ -means clustering algorithm then assigns these feature vectors to different query templates.
- (4) **Text mining based:** This is a variation of the bag of words approach, which indiscriminately extracts unique keywords from the training corpus. In contrast, in this approach, the vocabulary includes only those keywords that are either database object names (e.g., a Table name) or SQL clauses (e.g., group by). After vocabulary building, we generate a feature vector for each query then  $k$ -means clustering to assign them to templates.
- (5) **Word embeddings based:** Word Embeddings addresses two limitations of bag-of-words methods: dealing with numerous keywords and capturing keyword proximity. Using word embeddings, we construct a vocabulary from the query corpus and generate a feature vector for each query expression. Applying  $k$ -means clustering assigns these feature vectors to templates.
- (6) **SQL Embeddings using Foundation Models:** Since SQL queries consist of meaningful textual statements, we leverage pre-trained foundation models to embed these statements into a high-dimensional space, following established approaches in the literature [70, 73]. We use Sentence-Transformers [15], a fine-tuned model based on a pre-trained Microsoft mpnet-base model[48]. The generated query embeddings are then used in the LearnedWMP to form templates for the workloads.

To evaluate the performance of the five alternative methods for learning templates, we used the LearnedWMP-XGB model with JOB workloads. We trained five LearnedWMP-XGB models, each using a different method for learning templates. Fig. 9 compares the accuracy of these models. The model—labeled query plan (ours) in the figure—that uses our original method for learning templates outperformed the four alternatives. Compared with the alternatives, the LearnedWMP method uses features from the query plan. The query plans include estimates that are strong indicators of the query’s resource usage. A prior research [16]



**Figure 9: LearnedWMP’s accuracy performance achieved by different methods for learning query templates.**

made a similar observation. In contrast, the alternative methods extract features directly from the query expression, which does not provide insights into the query’s memory usage. A major limitation of the rules-based method is that creating effective rules requires knowledge from human experts, which can be both a slow and challenging process.

**5.3.2 Effect of the number of query templates.** As described in Section 4.2.1, LearnedWMP assigns queries into templates, such as queries with similar query plan characteristics in the same groups. This experiment aimed to assess the effects of the number of templates on the LearnedWMP model’s accuracy. In the experiment, we tested the performance of the LearnedWMP-XGB model on three datasets using 10 to 100 templates, comparing the model’s performance across different template sizes.

We used the Mean Absolute Percent Error (MAPE) [10] to evaluate and compare these models, each using a different number of templates. The scale of the error can vary significantly when changing the number of templates. MAPE is unaffected by changes in the error scale when comparing models trained with different numbers of templates because it calculates a relative error. We used equation (14) to compute the MAPE of the LearnedWMP-XGB model.

$$MAPE = \frac{1}{N} \sum_{i=1}^N \frac{|y_i - \hat{y}_i|}{y_i} \times 100 \quad (14)$$

We computed the relative estimation error between the actual and predicted memory usage by dividing the absolute difference between them by the actual value and then averaged the relative estimation errors across all workloads. The resulting average was multiplied by 100 to obtain the MAPE, ranging from 0 to 100 percent. Fig. 11 shows the performance of LearnedWMP-XGB as a factor of the number of templates. Fig 10 shows the performance of the LearnedWMP-XGB model for each template size for each dataset. For the TPC-DS dataset, performance improved as the number of templates increased. The best performance was observed at 100 templates. However, for the JOB and TPC-C datasets, performance varied as the number of templates increased, with optimal performance achieved within the 20 to 40 templates. We argue that this correlation between the number of queries and the optimal number of templates is due to the characteristics of each dataset. The larger TPC-DS dataset benefits from a greater diversity of queries (93,000) queries generated from (99) templates, which allows clustering with a higher number of templates. However, the other datasets lack the same level of query variation, which explains why the best performance was achieved with a moderate number of templates.

**5.3.3 Effect of the batch size parameter.** The experiments we discussed so far used a constant workload batch size of 10. The batch size,  $s$ , is a tunable hyperparameter of the LearnedWMP model. We tried different workload batch sizes and compared their impact on the LearnedWMP’s accuracy. We used the TPC-DS dataset and the LearnedWMP-XGB model for this experiment. We used 12 values for the batch size: [1, 2, 3, 5, 10, 15, 20, 25,

30, 35, 40, 45, 50]. For each value, we created TPC-DS training and test workloads, which we used for training and evaluating a LearnedWMP-XGB model. We computed and used MAPE to compare the relative accuracy performance of these models. Fig. 11 shows the performance of LearnedWMP-XGB as a factor of the batch size. We can see that as the batch size increases, the accuracy of the memory estimation improves. The improvement is more rapid initially, gradually slowing down, as expected of any learning algorithm as it approaches perfect prediction. For example, at batch size 2, the estimation error was 10.4% then at batch size 10, the error was reduced to 3.8%. We have seen a similar improvement in prediction accuracy with the other experimental datasets. This observation supports our position that batch estimation of workload memory is more accurate than estimating one query’s memory at a time. Additionally, we compared the MAPE of the LearnedWMP model with batch size 1 with the SingleWMP model’s MAPE. At batch size 1, LearnedWMP’s MAPE is 10.2, whereas SingleWMP’s MAPE is 3.6.

The SingleWMP model outperformed the LearnedWMP batch 1 model, as expected, since SingleWMP was directly trained with query plan features, which provided strong signals for individual query memory usage. In contrast, the LearnedWMP model mapped queries into templates and generated predictions based on collective memory usage, lacking the ability to learn from individual query features. While LearnedWMP may have weaker signals for single-query predictions, it consistently outperforms SingleWMP when predicting memory demand for batches of queries, as demonstrated in Section 5.1.

**5.3.4 Effect of variable batch size.** To evaluate the generalizability of LearnedWMP across varying workload sizes, we trained a model using a mixed-batch setup that included workloads with batch sizes of 5, 10, 15, 20, and 25—each contributing 20% of the training data. We then evaluated the model on five separate test sets, each corresponding to a different batch size. As shown in Fig. 12, the mixed-batch model demonstrates consistently strong performance across all batch sizes. When compared to models trained on a single batch size, the mixed-batch model achieves comparable accuracy. For example, the model trained exclusively on a batch size of 15 achieved a MAPE of 2.9 on its corresponding test set, while the mixed-batch model obtained a MAPE of 3.71 on the same set. Although there is a slight increase in error, the performance remains competitive, underscoring the robustness and flexibility of the mixed-batch training approach.

**5.3.5 DBMS Integration & Broader Impact.** A DBMS vendor can train a LearnedWMP model on sample workloads and ship it with the DBMS product to enable memory prediction before query execution. During deployment, the model can be refined using real query traces, allowing it to adapt to workload-specific patterns and dynamically determine batch sizes. LearnedWMP can be integrated without modifying the optimizer or execution engine through various strategies: (1) as an external estimator providing predictions to the resource manager, (2) as a middleware component analyzing query batches pre-execution, (3) as a query scheduler plugin for memory-aware scheduling, or (4) as a workload management module guiding batch formation and provisioning. Since LearnedWMP relies on features from query execution plans, commonly available in modern DBMSs, such integration is practical. Moreover, many DBMSs now offer built-in ML infrastructure to host, retrain, and serve models, such as LearnedWMP [1, 3, 4].

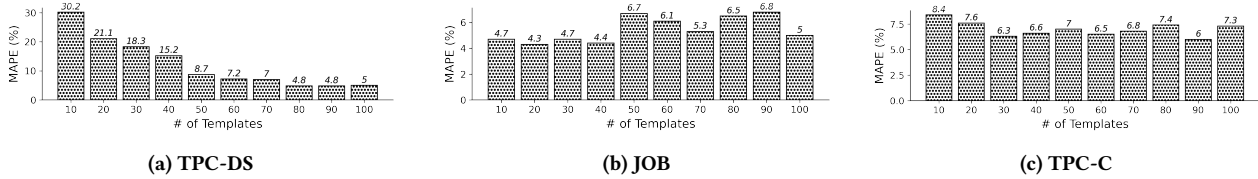


Figure 10: MAPE at different template sizes with LearnedWMP XGBoost for datasets: (a) TPC-DS, (b) JOB, and (c) TPC-C

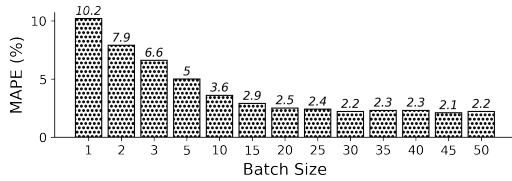


Figure 11: TPC-DS MAPE at different batch sizes with LearnedWMP XGBoost Model.

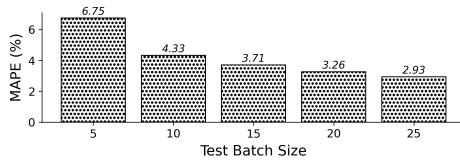


Figure 12: TPC-DS MAPE evaluated for mixed batch sizes with LearnedWMP XGBoost Model.

## 6 Conclusions

We proposed a novel approach to predicting the memory usage of database queries in batches. Our approach is a paradigm shift from the state of the practice and the state-of-the-art methods designed to estimate resource demand for single queries. As an embodiment of our approach, we presented LearnedWMP, a method for estimating the working memory demand of a batch of queries, a workload. The LearnedWMP method operates in three phases. First, it learns query templates from historical queries. Second, it constructs histograms from the training workloads. Third, using training workloads, it trains a regression model to predict the memory requirements of unseen workloads. 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 a contemporary DBMS, multiple sensible baselines, and state-of-the-art methods. Our analysis demonstrates that our proposed method significantly improves the memory estimation of the current state of the practice. Additionally, LearnedWMP matches the performance of advanced ML-based methods trained with a single-query approach. It generates smaller models, enabling faster training and predictions. We conducted parameter sensitivity analysis and explored various strategies for learning query templates from historical DBMS queries. Our novel LearnedWMP model presents an alternative perspective on a crucial DBMS problem, easily integratable with major DBMS products.

## 7 Artifacts

The source code and resources for our research are available for reproducibility: <https://github.com/shaikhq/learnedwmp>

## Acknowledgments

This work was supported by grants from the IBM Centre of Advanced Studies (CAS) and the Natural Sciences and Engineering Research Council of Canada (NSERC).

## References

- [1] 2022. *SQL machine learning documentation*. <https://learn.microsoft.com/en-us/sql/machine-learning/?view=sql-server-ver16> Accessed on 2023-11-21.
- [2] 2023. *Comparing Randomized Search and Grid Search for Hyperparameter Estimation*. [https://scikit-learn.org/stable/auto\\_examples/model\\_selection/plot\\_randomized\\_search.html](https://scikit-learn.org/stable/auto_examples/model_selection/plot_randomized_search.html) Accessed on 2023-11-21.
- [3] 2023. *In-database Machine Learning*. <https://www.ibm.com/docs/en/db2/11.5?topic=content-in-database-machine-learning> Accessed on 2023-11-21.
- [4] 2023. *Machine Learning in Oracle Database*. <https://www.oracle.com/ca-en/artificial-intelligence/database-machine-learning/> Accessed on 2023-11-21.
- [5] Mumtaz Ahmad, Songyun Duan, Ashraf Aboulnaga, and Shivnath Babu. 2011. Predicting Completion Times of Batch Query Workloads using Interaction-aware Models and Simulation. In *Proceedings of the International Conference on Extending Database Technology*. 449–460.
- [6] Mert Akdere, Ugur Cetintemel, Matteo Riondato, Eli Upfal, and Stanley B Zdonik. 2012. Learning-based Query Performance Modeling and Prediction. In *IEEE 28th International Conference on Data Engineering*. 390–401.
- [7] Nicolas Bruno and Surajit Chaudhuri. 2002. Exploiting statistics on query expressions for optimization. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. 263–274.
- [8] Surajit Chaudhuri. 1998. An overview of Query Optimization in Relational Systems. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. 34–43.
- [9] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A Scalable Tree Boosting System. In *Proceedings of the ACM SIGKDD international conference on Knowledge Discovery and Data Mining*. 785–794.
- [10] Arnaud De Myttenaere, Boris Golden, Bénédicte Le Grand, and Fabrice Rossi. 2016. Mean Absolute Percentage Error for Regression Models. *Neurocomputing* 192 (2016), 38–48.
- [11] Frederik Michel Dekking, Cornelis Kraaikamp, Hendrik Paul Lopuhaä, and Ludolf Erwin Meester. 2005. *A Modern Introduction to Probability and Statistics: Understanding why and how*. Vol. 488. Springer.
- [12] Bailu Ding, Sudipto Das, Ryan Marcus, Wentao Wu, Surajit Chaudhuri, and Vivek R Narasayya. 2019. Ai meets ai: Leveraging query executions to improve index recommendations. In *Proceedings of the 2019 International Conference on Management of Data*. 1241–1258.
- [13] Jennie Duggan, Ugur Cetintemel, Olga Papaemmanouil, and Eli Upfal. 2011. Performance prediction for concurrent database workloads. In *Proceedings of International Conference on Management of data*. 337–348.
- [14] Jennie Duggan, Olga Papaemmanouil, Ugur Cetintemel, and Eli Upfal. 2014. Contender: A Resource Modeling Approach for Concurrent Query Performance Prediction. In *EDBT*. 109–120.
- [15] Hugging Face. [n. d.]. Sentence Transformers. <https://huggingface.co/sentence-transformers>. Accessed: April 2025.
- [16] Archana Ganapathi, Harumi Kuno, Umeshwar Dayal, Janet L Wiener, Armando Fox, Michael Jordan, and David Patterson. 2009. Predicting Multiple Metrics for Queries: Better Decisions enabled by Machine Learning. In *IEEE International Conference on Data Engineering*. 592–603.
- [17] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, et al. 2021. Cardinality Estimation in DBMS: A Comprehensive Benchmark Evaluation. *Proceedings of VLDB Endowment* (2021).
- [18] Haitian Hang, Xiu Tang, Jianling Sun, Lingfeng Bao, David Lo, and Haoye Wang. 2024. Robust Auto-Scaling with Probabilistic Workload Forecasting for Cloud Databases. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 4016–4029.
- [19] Shohedul Hasan, Saravanan Thirumuruganathan, Jeess Augustine, Nick Koudas, and Gautam Das. 2020. Deep learning Models for Selectivity Estimation of Multi-attribute Queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1035–1050.
- [20] Antony S Higginson, Mihaela Dediu, Octavian Arsene, Norman W Paton, and Suzanne M Embury. 2020. Database Workload Capacity Planning using Time series Analysis and Machine Learning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 769–783.
- [21] Benjamin Hilprecht, Andreas Schmidt, Moritz Kullessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2019. Deepdb: Learn from Data, not from Queries! *Proceedings of the VLDB Endowment* (2019).
- [22] Marc Holze, Ali Haschimi, and Norbert Ritter. 2010. Towards Workload-aware Self-management: Predicting Significant Workload Shifts. In *IEEE 26th International Conference on Data Engineering Workshops (ICDEW)*. IEEE, 111–116.
- [23] Hanxian Huang, Tarique Siddiqui, Rana Alotaibi, Carlo Curino, Jyoti Leeka, Alekh Jindal, Jishen Zhao, Jesús Camacho-Rodríguez, and Yuanyuan Tian. 2024. Sibly: Forecasting Time-Evolving Query Workloads. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–27.

- [24] IBM. 2024. *Db2 11.5: Self-tuning Memory*. <https://www.ibm.com/docs/en/db2/11.5?topic=overview-self-tuning-memory> Accessed: 2024-09-20.
- [25] IBM. 2024. *IBM Informix*. <https://www.ibm.com/products/informix> Accessed: 2024-11-17.
- [26] Yannis E Ioannidis. 1996. Query Optimization. *ACM Computing Surveys (CSUR)* 28, 1 (1996), 121–123.
- [27] Matthias Jarke and Jurgen Koch. 1984. Query Optimization in Database Systems. *ACM Computing surveys (CSUR)* 16, 2 (1984), 111–152.
- [28] Kyoungmin Kim, Jisung Jung, In Seo, Wook-Shin Han, Kangwoo Choi, and Jaehyok Chong. 2022. Learned Cardinality Estimation: An in-depth Study. In *Proceedings of the International Conference on Management of Data*. 1214–1227.
- [29] Diederik P Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. *CoRR* (2014).
- [30] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2018. Learned cardinalities: Estimating Correlated Joins with Deep Learning. *arXiv preprint arXiv:1809.00677* (2018).
- [31] Georgia Kolomiari, Yannis Petrakis, Evaggelia Pitoura, and Thodoris Tsotsos. 2005. Query Workload-Aware Overlay Construction Using Histograms. In *Proceedings of the ACM International Conference on Information and Knowledge Management*. 640–647.
- [32] Ho Chung Leon Law, Danica J Sutherland, Dino Sejdinovic, and Seth Flaxman. 2018. Bayesian approaches to distribution regression. In *International Conference on Artificial Intelligence and Statistics*. PMLR, 1167–1176.
- [33] Kukjin Lee, Arnd Christian König, Vivek Narasayya, Bolin Ding, Surajit Chaudhuri, Brent Ellwein, Alexey Eksarevskiy, Manbeen Kohli, Jacob Wyant, Pra-neeta Prakash, et al. 2016. Operator and Query Progress Estimation in Microsoft SQL Server Live Query Statistics. In *Proceedings of the International Conference on Management of Data*. 1753–1764.
- [34] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good are Query Optimizers, Really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215.
- [35] Viktor Leis, Bernhard Radke, Andrey Gubichev, Alfons Kemper, and Thomas Neumann. 2017. Cardinality Estimation Done Right: Index-Based Join Sampling. In *Cidr*.
- [36] Scott T Leutenegger and Daniel Dias. 1993. A Modeling Study of the TPC-C Benchmark. *ACM Sigmod Record* 22, 2 (1993), 22–31.
- [37] Jiexing Li, Arnd Christian König, Vivek Narasayya, and Surajit Chaudhuri. 2012. Robust estimation of resource consumption for SQL queries using statistical techniques. *Proceedings VLDB Endowment* 5, 11 (2012), 1555–1566.
- [38] Pengfei Li, Wenqing Wei, Rong Zhu, Bolin Ding, Jingren Zhou, and Hua Lu. 2023. ALECE: An Attention-based Learned Cardinality Estimator for SPJ Queries on Dynamic Workloads. *Proceedings of the VLDB Endowment* 17, 2 (2023), 197–210.
- [39] Rui Li, Howard D Bondell, and Brian J Reich. 2019. Deep Distribution Regression. *arXiv preprint arXiv:1903.06023* (2019).
- [40] Wei Li, Jiachi Zhang, Ye Yin, Yan Li, Zhanyang Zhu, Wenchao Zhou, Liang Lin, and Feifei Li. 2024. Flux: Decoupled Auto-Scaling for Heterogeneous Query Workload in Alibaba AnalyticDB. In *Companion of the 2024 International Conference on Management of Data*. 255–268.
- [41] Dong C Liu and Jorge Nocedal. 1989. On the Limited Memory BFGS Method for Large Scale Optimization. *Mathematical programming* 45, 1 (1989), 503–528.
- [42] Henry Liu, Mingbin Xu, Zitong Yu, Vincent Corvinelli, and Calisto Zuzarte. 2015. "Cardinality Estimation using Neural Networks". In *Proceedings of the International Conference on Computer Science and Software Engineering*. 53–59.
- [43] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J Gordon. 2018. Query-based Workload Forecasting for Self-Driving Database Management Systems. In *Proceedings of the International Conference on Management of Data*. 631–645.
- [44] J MacQueen. 1967. Classification and Analysis of Multivariate Observations. In *5th Berkeley Symp. Math. Statist. Probability*. 281–297.
- [45] Yuan Mao, Lei Shi, and Zheng-Chu Guo. 2022. Coefficient-based Regularized Distribution Regression. *arXiv preprint arXiv:2208.12427* (2022).
- [46] Ryan Marcus and Olga Papaemmanouil. 2019. Plan-structured Deep Neural Network Models for Query Performance Prediction. *Proceedings of VLDB Endowment* (2019).
- [47] Patrick Martin, Said Elnaffar, and Ted Wasserman. 2006. Workload models for Autonomic Database Management Systems. In *International Conference on Autonomic and Autonomous Systems (ICAS)*. IEEE, 10–10.
- [48] Microsoft. [n. d.]. MPNet-Base Model on Hugging Face. <https://huggingface.co/microsoft/mpnet-base>. Accessed: April 2025.
- [49] Microsoft. 2024. *SQL Server technical documentation*. <https://learn.microsoft.com/en-us/sql/sql-server/?view=sql-server-ver16> Accessed: 2024-11-17.
- [50] Barzan Mozafari, Carlo Curino, Alekh Jindal, and Samuel Madden. 2013. Performance and Resource Modeling in Highly-Concurrent OLTP Workloads. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 301–312.
- [51] Kevin P Murphy. 2022. *Probabilistic machine learning: an introduction*. MIT press.
- [52] Raghunath Othayoth Nambiar and Meikel Poess. 2006. The Making of TPC-DS. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, Vol. 6. 1049–1058.
- [53] Parimarjan Negi, Ziniu Wu, Andreas Kipf, Nesime Tatbul, Ryan Marcus, Sam Madden, Tim Kraska, and Mohammad Alizadeh. 2023. Robust Query Driven Cardinality Estimation under Changing Workloads. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1520–1533.
- [54] Oracle. 2023. *About Automatic Memory Management*. <https://docs.oracle.com/en/database/oracle/oracle-database/19/admin/managing-memory.html#GUID-8F54391B-D42A-4FDA-9D12-E1F81FD113EA> Accessed on 2024-11-28.
- [55] Jennifer Ortiz, Victor Teixeira De Almeida, and Magdalena Balazinska. 2015. Changing the Face of Database Cloud Services with Personalized Service Level Agreements. In *CIDR*.
- [56] Debjyoti Paul, Jie Cao, Feifei Li, and Vivek Srikumar. 2021. Database Workload Characterization with Query Plan Encoders. *Proceedings VLDB Endowment* (2021).
- [57] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C Mowry, Matthew Perron, Ian Quah, et al. 2017. Self-Driving Database Management Systems. In *CIDR*, Vol. 4. 1.
- [58] Barnabas Poczos, Aarti Singh, Alessandro Rinaldo, and Larry Wasserman. 2013. Distribution-Free Distribution Regression. In *Artificial Intelligence and Statistics*. 507–515.
- [59] Sebastian Raschka. 2019. *Python Machine Learning Ed. 3*. Packt Publishing.
- [60] Redshift. 2024. *Workload Management Automatic WLM*. <https://docs.aws.amazon.com/redshift/latest/dg/cm-c-implementing-workload-management.html> Accessed: 2024-11-17.
- [61] Scikit-learn. 2024. *Sklearn.neural\_network.MLPRegressor*. [https://scikit-learn.org/stable/modules/generated/sklearn.neural\\_network.MLPRegressor.html](https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPRegressor.html) Accessed on 2024-11-21.
- [62] Ankur Sharma, Felix Martin Schuhknecht, and Jens Dittrich. 2018. The case for Automatic Database Administration using Deep Reinforcement Learning. *arXiv preprint arXiv:1801.05643* (2018).
- [63] Ravid Shwartz-Ziv and Amitai Armon. 2021. Tabular Data: Deep Learning is Not All You Need. *arXiv preprint arXiv:2106.03253* (2021).
- [64] Tarique Siddiqui, Alekh Jindal, Shi Qiao, Hireen Patel, and Wangchao Le. 2020. Cost models for Big Data Query Processing: Learning, Retrofitting, and our Findings. In *Proceedings of the SIGMOD International Conference on Management of Data*. 99–113.
- [65] Snowflake. 2024. *Optimizing Performance in Snowflake*. <https://docs.snowflake.com/en/guides-overview-performance> Accessed: 2024-11-17.
- [66] Ji Sun and Guoliang Li. 2019. An End-to-End Learning-based Cost Estimator. *Proceedings of VLDB Endowment* (2019).
- [67] Zoltán Szabó, Bharath K. Sriperumbudur, Barnabás Póczos, and Arthur Gretton. 2016. Learning Theory for Distribution Regression. *J. Mach. Learn. Res.* 17, 1 (2016), 5272–5311.
- [68] Jian Tan, Tieying Zhang, Feifei Li, Jie Chen, Qixing Zheng, Ping Zhang, Honglin Qiao, Yue Shi, Wei Cao, and Rui Zhang. 2019. iBTune: Individualized Buffer Tuning for Large-scale Cloud Databases. *Proceedings of the VLDB Endowment* (2019), 1221–1234.
- [69] Chunxu Tang, Beinan Wang, Zhenxiao Luo, Huijun Wu, Shajan Dasan, Maosong Fu, Yao Li, Mainak Ghosh, Ruchin Kabra, Nikhil Kantibhai Navadiya, et al. 2021. Forecasting SQL query Cost at Twitter. In *2021 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 154–160.
- [70] Xiu Tang, Sai Wu, Mingli Song, Shanshan Ying, Feifei Li, and Gang Chen. 2022. PreQR: pre-training representation for SQL understanding. In *Proceedings of the International Conference on Management of Data*. 204–216.
- [71] Quoc Trung Tran, Konstantinos Morfonios, and Neoklis Polyzotis. 2015. Oracle Workload Intelligence. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1669–1681.
- [72] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning through Large-scale Machine Learning. In *Proceedings of the ACM International Conference on Management of Data*. 1009–1024.
- [73] Jiayi Wang, Tianyi Li, Anni Wang, Xiaozhe Liu, Lu Chen, Jie Chen, Jianye Liu, Junyang Wu, Feifei Li, and Yunjun Gao. 2023. Real-Time Workload Pattern Analysis for Large-Scale Cloud Databases. *Proc. VLDB Endow.* 16, 12 (2023), 3689–3701.
- [74] Zhijie Wei, Zuohua Ding, and Jueliang Hu. 2014. Self-tuning Performance of Database Systems based on Fuzzy Rules. In *The International Conference on Fuzzy Systems and Knowledge Discovery (FSKD)*. IEEE, 194–198.
- [75] Wentao Wu, Yun Chi, Hakan Hacigümüş, and Jeffrey F Naughton. 2013. Towards Predicting Query Execution Time for Concurrent and Dynamic Database Workloads. *Proceedings of the VLDB Endowment* 6, 10 (2013), 925–936.
- [76] Wentao Wu, Xi Wu, Hakan Hacigümüş, and Jeffrey F Naughton. 2014. Uncertainty aware Query Execution Time Prediction. *Proceedings of the VLDB Endowment* (2014).
- [77] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. 2020. NeuroCard: one cardinality estimator for all tables. *arXiv preprint arXiv:2006.08109* (2020).
- [78] Jintao Zhang, Chao Zhang, Guoliang Li, and Chengliang Chai. 2023. AutoCE: An Accurate and Efficient Model Advisor for Learned Cardinality Estimation. In *IEEE International Conference on Data Engineering (ICDE)*. 2621–2633.
- [79] Yue Zhao, Gao Cong, Jiachen Shi, and Chunyan Miao. 2022. QueryFormer: a Tree Transformer Model for Query Plan Representation. *Proceedings of the VLDB Endowment* 15, 8 (2022), 1658–1670.
- [80] Xuanhe Zhou, Ji Sun, Guoliang Li, and Jianhua Feng. 2020. Query Performance Prediction for Concurrent Queries using Graph Embedding. *Proceedings of the International Conference on Very Large Databases (VLDB)* 13, 9 (2020), 1416–1428.