

Powering In-Database Dynamic Model Slicing for Structured Data Analytics

Lingze Zeng¹, Naili Xing¹, Shaofeng Cai¹, Gang Chen², Beng Chin Ooi¹
Jian Pei³, Yuncheng Wu⁴

¹National University of Singapore ²Zhejiang University ³Duke University

⁴Renmin University of China

lingze@comp.nus.edu.sg, ooibc@comp.nus.edu.sg

ABSTRACT

Relational database management systems (RDBMS) are widely used for the storage and retrieval of structured data. To derive insights beyond statistical aggregation, we typically have to extract specific subdatasets from the database using conventional database operations, and then apply deep neural networks (DNN) training and inference on these respective subdatasets in a separate machine learning system. The process can be prohibitively expensive, especially when there are a combinatorial number of subdatasets extracted for different analytical purposes. This calls for efficient in-database support of advanced analytical methods.

In this paper, we introduce LEADS, a novel SQL-aware dynamic model slicing technique to customize models for subdatasets specified by SQL queries. LEADS improves the predictive modeling of structured data via the mixture of experts (MoE) technique and maintains inference efficiency by a SQL-aware gating network. At the core of LEADS is the construction of a general model with multiple expert sub-models via MoE trained over the entire database. This SQL-aware MoE technique scales up the modeling capacity, enhances effectiveness, and preserves efficiency by activating only necessary experts via the gating network during inference. Additionally, we introduce two regularization terms during the training process of LEADS to strike a balance between effectiveness and efficiency. We also design and build an in-database inference system, called INDICES, to support end-to-end advanced structured data analytics by non-intrusively incorporating LEADS onto PostgreSQL. Our extensive experiments on real-world datasets demonstrate that LEADS consistently outperforms baseline models, and INDICES delivers effective in-database analytics with a considerable reduction in inference latency compared to traditional solutions.

1 INTRODUCTION

Relational Database Management Systems (RDBMS) are extensively employed as the primary storage solution for structured data across various applications [30, 34, 41, 46]. They serve as a fundamental infrastructure for various domains and are critical to the operation of numerous businesses [23, 28, 64]. In the contemporary business landscape, structured data analytics via databases has become an indispensable component for driving business growth and success [23, 28, 37, 43, 64]. Traditional structured data analytics approaches rely on database-driven filtering or aggregation operations to derive insights. However, these insights only offer a limited statistical view, which often fails to capture the complexity and intricacies of the underlying patterns [21, 45]. Fortunately, recent advancements in

Deep Neural Networks (DNNs) open up new horizons for advanced analytics beyond simple statistical aggregation [8, 9, 19, 35].

At its core, exploiting DNNs for advanced structured data analytics comprises two main phases: training and inference [19]. The former primarily involves the construction of a DNN model and the training of this model on targeted data, while the latter utilizes the trained model to make predictions on new data. Notably, to deliver advanced DNN-driven analytics for informed decision-making, effectiveness and efficiency are the two most important metrics to optimize for [7, 14, 32, 52]. Specifically, effectiveness focuses on the inference phase, measuring the extent to which the predictions delivered by the model are accurate. Meanwhile, efficiency evaluates the requirements of the model in terms of *response time* and *computational resources* in both phases [32].

In real-world scenarios, analysts are often more interested in performing analytics on specific subsets of data. For instance, they may assess trends among patients diagnosed with a particular disease, or, study behaviors of consumers of a certain age group. Consider the scenario illustrated in Figure 1, where an analyst aims to evaluate the influence of education and city location on the incomes of different subdatasets, i.e., tuples grouped by gender and age. Naturally, the analyst seeks to build a predictive model that is effective, delivering accurate predictions for these subsets of tuples, and meanwhile, executes predictions efficiently with minimal response time and computational resources. However, there are two main challenges in achieving this objective.

First, achieving efficient training for effective predictive modeling across analyst-specified subdatasets is challenging. Conventionally, a single general model is trained to support inference across all data tuples [19, 23, 30]. This approach is efficient, which requires training only one model. However, such a model, optimized to capture the common patterns and general behaviors of the whole dataset, is likely not as effective in providing accurate predictions as a dedicated model trained on a specific subdataset of interest. Taking for example the scenario in Figure 1, a model trained explicitly for the group of the gender male and age 24 would probably identify finer-grained patterns and behaviors pertinent to this subdataset, given sufficient training tuples, this dedicated model could outperform the general model significantly. Nonetheless, training a separate model for each subdataset is computationally prohibitive due to the combinatorial nature of potential subdatasets.

Second, efficiently integrating the inference phase into an RDBMS while ensuring effectiveness is also challenging from a system perspective. One major obstacle is how to reconcile the practices of managing structured data within an RDBMS and the execution of inference on a separate ML system. Many existing solutions support

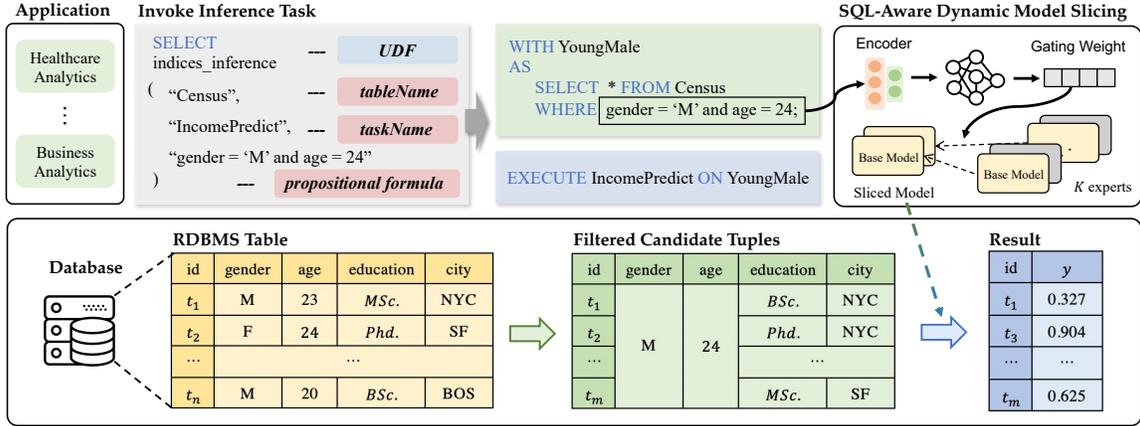


Figure 1: An illustration of in-database analytics on income via SQL-aware dynamic model slicing.

the inference process with two separate systems [13, 51], which requires transferring the inference data from an RDBMS, typically a subset of tuples, to another inference system. Such a process is time-consuming, susceptible to errors, and might also violate privacy and security requirements [55]. Recently, several preliminary attempts have been made to integrate the inference process directly into RDBMS via User-Defined Functions (UDFs) [5, 17, 24, 55, 58, 63], which improves user experience by enabling in-database inference through SQL statements. Specifically, running Python in UDFs can tap into its rich machine learning libraries [18], while it misses the opportunity to leverage the more efficient data retrieval APIs offered by RDBMS, e.g., server programming interface (SPI). A proposed enhancement involves utilizing multiple programming languages within UDFs, aiming to harness both data retrieval APIs and advanced ML libraries. However, this approach introduces extra overhead and affects inference efficiency, especially when conversions and copying of inference data between different language execution environments become necessary [18]. Therefore, achieving efficient and seamless integration of the inference process into RDBMS is an imperative problem to address.

To address the above challenges, we build an efficient and effective **IN-Database Inference System** (INDICES). The system is designed to produce effective predictions across subsets of data dynamically specified and retrieved by SQL queries. To this end, we propose a novel **SQL-aware dynamic model Slicing** (LEADS) technique, which enhances the effectiveness of the base model via the mixture of experts (MoE) technique, and maintains the inference efficiency using a SQL-aware gating network for dynamic model customization for subdatasets specified by SQL queries. Specifically, in LEADS, we propose to enhance the modeling capacity of the base model by constructing a general model that is composed of multiple replicas of this base model. These replicas, termed as *experts*, are trained to specialize in different problem subspaces for more effective predictive modeling. To enhance effectiveness via MoE without incurring reduced inference efficiency, we further introduce a SQL-aware gating network that dynamically generates sparse *gating weights* based on filter conditions in the SQL query to slice a subset of only necessary experts from the general model.

Such a *sliced model* is optimized for the corresponding SQL query during training, and is dedicated to the specified subdataset for enhancing inference effectiveness while maintaining efficiency.

To support end-to-end structured data analytics, our system INDICES seamlessly incorporates LEADS into PostgreSQL, an open-source RDBMS widely used in both industry and academia. For ease of use and inference efficiency, we divide the proposed in-database inference process into four separate stages and propose three optimization techniques to minimize the overhead of each stage: efficient execution allocation, memory sharing, and state caching. Given that all the stages of the inference process are supported within a single UDF, analysts can now conveniently invoke inference queries using a single SQL statement. This approach obviates the need to transfer and manage data in separate systems and reduce data copying overhead between executions of different programming languages. Additionally, while the current system is supported by PostgreSQL, INDICES can be readily integrated into other RDBMSs, e.g., MySQL.

We summarize the main contributions as follows.

- We formulate the SQL-aware structured data analytics problem, which requires efficient and effective predictive modeling on subdatasets specified by corresponding SQL queries. To the best of our knowledge, this is the first work that develops techniques and a system to address the problem.
- We propose a novel SQL-aware dynamic model slicing technique LEADS, which scales up the modeling capacity of the base model via MoE and devises a SQL-aware gating network for efficient and effective dynamic model customization for SQL-specified subdataset.
- We design and build an end-to-end in-database inference system INDICES for advanced structured data analytics, which non-intrusively incorporate LEADS onto PostgreSQL with three optimization techniques for further improving the inference efficiency.
- We conduct extensive experiments on four real-world datasets. The results confirm the effectiveness of LEADS, with up to 3.95% improvement in accuracy for given workloads of datasets compared with the baseline models, while INDICES achieves up to

2.06x speedup in terms of inference efficiency compared with the traditional solution.

In the remainder of this paper, we introduce preliminaries in Section 2. We formulate our problem in Section 3. We present LEADS with detailed descriptions of its modules and optimization schemes in Section 4. We discuss the integration of LEADS and INDICES with PostgreSQL in Section 5. Experimental results are presented in Section 6. We review related work in Section 7 and conclude the paper in Section 8.

2 PRELIMINARIES

In this section, we present two key techniques central to our system, namely Mixture of Experts (MoE) for scaling up the model capacity while maintaining its inference efficiency via conditional computation [48], and sparse softmax [38] for the informed selection of active experts for enhancing efficiency. Scalars, vectors and matrices are denoted by x , \mathbf{x} and \mathbf{X} respectively.

Mixture of Experts (MoE) [29, 48, 53] is a general ensemble learning and conditional computation technique to scale up the modeling capacity without incurring much computational overhead. In DNN-based MoE, a series of expert DNNs are adopted to divide problem space into different regions, where each expert specializes in handling a certain sub-region. MoE is particularly useful when the data exhibits complex patterns or variations [16, 47, 65] due to its capability of enhancing model capacity. There are two main components in an MoE layer: expert models and a gating network. For simplicity of construction, expert models can be composed of homogeneous models that share the same model architecture. During training, expert models are trained to specialize in different problem sub-spaces. The gating network is also trained to produce a set of gating weights dynamically, which determines the importance assigned to corresponding experts.

Denoting the gating weights and outputs of experts as $\mathbf{w} = [w_1, w_2, \dots, w_K]$ and $\mathbf{H} = [\mathbf{h}_1, \mathbf{h}_2, \dots, \mathbf{h}_K]$ respectively, where K is the number of experts, and \mathbf{h}_i is the output of the i -th expert, the output of the MoE for the current input is then a weighted average of these experts: $\hat{\mathbf{y}} = \sum_{i=1}^K w_i \mathbf{h}_i$. During training, the MoE model optimizes the gating network and experts simultaneously. The gating network learns to assign appropriate weight to experts, while the experts learn to make accurate predictions within their respective regions of expertise.

MoE has found extensive application in various domains, notably in the large language model GPT-4 [42] for texts and the large vision-language model MoE-LLaVA [36] for images, which combines the benefits of large model capacity with efficient computation, by only engaging a fraction of the model parameters for each input. In LEADS, we focus on the applicability of the MoE technique to structured data analytics, intending to harness its scalable modeling capacity for enhanced predictive accuracy and efficiency.

Sparse Softmax. Softmax transformation is a crucial function in the gating network, which maps an input vector \mathbf{z} into a probability distribution \mathbf{p} whose probabilities correspond proportionally to the exponential of its input values, i.e., $\text{softmax}(z_j) = \frac{\exp(z_j)}{\sum_i \exp(z_i)}$. The output of softmax can thus be subsequently used as the network output denoting the class probabilities or weights indicating the

importance of corresponding inputs. Specifically, denoting the d -dimension probability as $\Delta^d := \{\mathbf{p} \in \mathbb{R}^d : \mathbf{p} \geq 0, \|\mathbf{p}\|_1 = 1\}$, softmax can be interpreted in the variational form with entropy:

$$\text{softmax}(\mathbf{z}) = \underset{\mathbf{p} \in \Delta^d}{\text{argmax}} \mathbf{p}^T \mathbf{z} + H^S(\mathbf{p}) \quad (1)$$

where $H^S(\mathbf{p}) = -\sum_j p_j \log p_j$ is the Shannon entropy. The softmax function is extensively used in DNNs, largely due to its differentiable and convex properties. However, softmax always assigns dense probabilities to inputs, which is less interpretable and effective [8, 20], as compared with sparse credit assignment. To overcome this limitation, sparse softmax is proposed to produce sparse distributions, by assigning zero probability to certain outputs. Particularly, α -entmax [44] generalizes both dense and sparse softmax with Tsallis α -entropies $H_\alpha^T(\mathbf{p})$ [50]:

$$\alpha\text{-entmax}(\mathbf{z}) = \underset{\mathbf{p} \in \Delta^d}{\text{argmax}} \mathbf{p}^T \mathbf{z} + H_\alpha^S(\mathbf{p}) \quad (2)$$

where $H^S(\mathbf{p}) = -\sum_j (p_j = p_j^\alpha)$ if $\alpha \neq 1$, else $H_1^T(\mathbf{p}) = H^S(\mathbf{p})$. With a larger α , α -entmax tends to produce a sparser probability distribution. Another appealing property is that the hyper-parameter α , which controls the shape and sparsity of the mapping, can be learned adaptively to the predictive task in the training stage. Let $\mathbf{p}^* = \alpha\text{-entmax}(\mathbf{z})$ denote the distribution $\tilde{p}_i = (p_i^*)^{2-\alpha} / \sum_j (p_j^*)^{2-\alpha}$ and the Shannon entropy $h_i = -(p_i^*) \log(p_i^*)$. The gradient of α is derived as: $\frac{\partial \alpha\text{-entmax}(\mathbf{z})}{\partial \alpha} = \frac{(p_i^*) - \tilde{p}_i}{(\alpha-1)^2} + \frac{h_i - \tilde{p}_i \sum_j h_j}{\alpha-1}$, $\alpha > 1$, which can be optimized end-to-end together with the parameters of the predictive model [44]. We aim to adopt a learnable sparse softmax in LEADS to improve model training and further enhance the predictive efficiency.

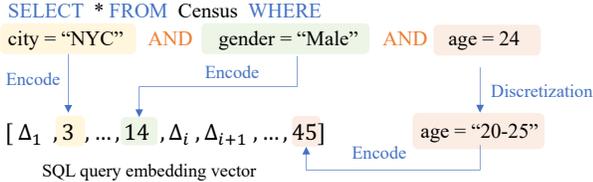
3 PROBLEM FORMULATION

Technically, structured data can be viewed as one logical table \mathbf{T} , which comprises N rows and M attributes within RDBMS. Each row, represented as a tuple $\mathbf{x} = (x_1, x_2, \dots, x_M)$, serves as a feature vector in predictive modeling, with x_i denoting the value of the i -th attribute. In structured data analytics, data analysts typically focus on specific subsets of data characterized by shared attributes. For example, analysts may assess the readmission rates among the patients diagnosed with a certain disease, or predict the e-commerce click-through rate (CTR) within a particular age group. Typically, for complex analytical queries that involve prediction, WHERE statement in a SQL query is executed first to select relevant tuples, to which DNNs are applied subsequently for prediction. In this paper, we refer to this process as SQL-aware predictive modeling.

Given a SQL query, denoted by q , there are two main steps in SQL-aware predictive modeling: data selection and model prediction. Utilizing relational algebra, a generalized SQL query selection q is expressed as $\sigma_\varphi(\mathbf{T})$, where σ is the unary operator for selection and φ is the propositional formula in q . Typically, φ consists of multiple predicates connected by logical operators. The selection $\sigma_\varphi(\mathbf{T})$ retrieves all tuples in table \mathbf{T} that satisfies φ , formally defined as $\sigma_\varphi(\mathbf{T}) = \{\mathbf{x} : \mathbf{x} \in \mathbf{T}, \varphi(\mathbf{x})\}$. For simplicity, the subset retrieved by the SQL query q is denoted as $\mathbf{T}_\varphi = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n\}$, where n is the number of tuples. Each tuple $\mathbf{x}_i \in \mathbb{R}^M$ in \mathbf{T}_φ comprises

SELECT * FROM Census WHERE city = "NYC" OR city = "BOS" ❌
 SELECT * FROM Census WHERE age > 25 AND gender = "Male" ❌
 SELECT * FROM Census WHERE edu = "MSc." AND gender = "Male" ✅

(a) Examples of a primitive SQL query.



(b) Process of Encoding SQL query.

Figure 2: SQL query encoder.

M attributes, and \mathbf{x}_i can be represented as a vector of categorical and/or numerical features, i.e., $\mathbf{x}_i = [x_{i,1}, x_{i,2}, \dots, x_{i,M}]$. DNNs are then applied to perform prediction on these selected tuples, e.g., to predict the labels $\mathbf{y} = \{y_1, y_2, \dots, y_n\}$, aiming to derive meaningful insights, such as patients readmission rates in healthcare analytics or CTR in e-commerce. Technically, SQL-aware predictive modeling refers to making predictions on a selected subset of tuples retrieved from a logical table \mathbf{T} based on a SQL query q with a propositional formula φ .

4 SQL-AWARE DYNAMIC MODEL SLICING

In contrast to conventional machine learning paradigms, SQL-aware predictive modeling makes use of constraints specified in SQL queries to provide more accurate predictions with respect to the data of interest. For example, the query outlined in Figure 2 is interested in data constrained based on age, location, and gender. Such selection constraints present optimization opportunities for prediction accuracy and efficiency. To this end, we propose SQL-aware dynamic model slicing technique, LEADS, to leverage the propositional formula φ from SQL queries as meta-information to customize the base model, as a means to improve the effectiveness and efficiency of the prediction of a specified datasubset.

In this section, we first introduce the SQL query encoder to translate φ into a vectorized format. Next, we present two key components of LEADS for scaling up the modeling capacity of the base model via the Mixture of Experts (MoE) technique and dynamic model slicing via a SQL-aware gating network. For optimization, we further design two regularization terms to strike a balance between effectiveness and efficiency.

4.1 SQL Query Encoder

In SQL-aware predictive modeling, the WHERE clause of SQL queries filters tuples according to a predefined propositional formula φ . This formula comprises one or more predicates, each setting a logical condition on a particular attribute. For instance, "gender = 'M'" mandates that the gender attribute of the filtered tuples must be 'M'. These predicates are interconnected via logical operators such as "AND" or "OR" to form a complete propositional formula. The exponential number of possible predicate combinations in SQL queries

renders the direct development of models for every conceivable subdataset, as in traditional machine learning approaches, both complex and intractable.

For the SQL query encoder, we focus on individual queries, referred to as primitive SQL query. Considering a table \mathbf{T} with M attributes, the j -th attribute denoted as A_j , each attribute is linked to either a numerical or categorical feature in predictive modeling. Particularly, each numerical feature needs to be converted into a corresponding categorical feature through discretization, which will be detailed subsequently. In a primitive SQL query, each attribute A_j may be associated with zero or one predicate, with predicates across attributes conjoined using the logical operator \wedge (AND), as depicted in Figure 2a. Technically, a predicate for attribute A_j in primitive SQL query can be expressed as $P_j : A_j = a_j$, where $a_j \in \mathcal{D}_j \cup \{\Delta_j\}$, \mathcal{D}_j represents the domain of possible values for A_j , and Δ_j denotes a default value assigned to A_j when it is not specified in the query. Figure 2a illustrates a valid primitive SQL query example, contrasting with two non-examples. Thus, the propositional formula φ can be represented as:

$$\varphi = P_1 \wedge P_2 \wedge \dots \wedge P_M.$$

The objective of the SQL query encoder is to generate a categorical feature vector \mathbf{q} for each primitive SQL query based on the meta-information φ , achieved by concatenating the attribute values of the predicates. Formally, the feature vector of the SQL query encoding can be obtained by:

$$\mathbf{q} = [q_1, q_2, \dots, q_M]$$

where q_j is the categorical attribute value for predicate P_j . Figure 2b demonstrates the transformation of a primitive SQL query into a feature vector. Notably, the numerical attribute "age" here is first discretized before being encoded alongside categorical attributes "city" and "gender", and columns lacking predicates are filled by the default value Δ_i .

Discretization. Discretization is essential for encoding numerical attributes, e.g., weight or salary. The infinite possible values of numerical attributes make direct encoding infeasible, hence requiring discretization. This process first partitions the domain \mathcal{D} of each numerical attribute into a fixed number of bins, akin to approximating a k-nearest neighbors classifier in predictive modeling. The goal of discretization is to preserve the key information in the embedding space for maintaining predictive modeling capacity.

To this end, we employ a supervised discretization approach that accounts for the correlation between numerical attributes and the target attribute. This aims to maximize information value (IV), which measures the reduction of uncertainty within each bin relative to the prediction target. Higher IV values indicate a significant decrease in uncertainty, thereby preserving the predictive capacity. In particular, we introduce the open-source *OptBinning* [40] implementation for discretization, which optimizes IV effectively while supporting constraints like the maximum bin count per attribute.

4.2 SQL-Aware Dynamic Model Slicing

The categorical feature vector \mathbf{q} , obtained from the SQL query encoder, captures key information that facilitates dynamic customization of a predictive model to an optimal configuration for

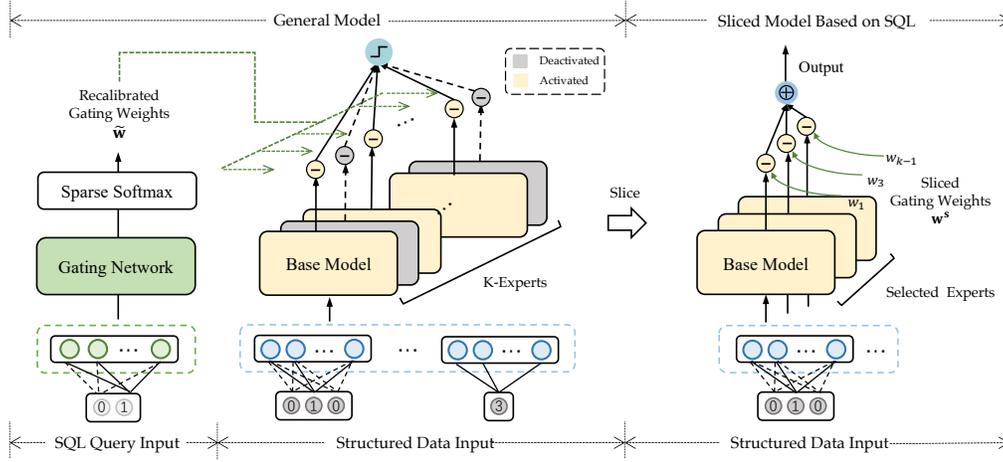


Figure 3: Overview of SQL-aware dynamic model slicing.

the targeted subdataset. The customization should significantly enhance predictive performance in SQL-aware predictive modeling. As illustrated in Figure 3, LEADS first scales up the modeling capacity via MoE by replicating the base model to construct a general model, and subsequently, LEADS integrates a SQL-aware gating network based on the SQL encoding vector \mathbf{q} to selectively activate experts in the general model to derive a sliced model for higher efficiency and effectiveness.

In this subsection, we will first introduce the preprocessing module that prepares the embedding vector \mathbf{q} for the predictive modeling, elaborate on the two key modules, i.e., the general model and the SQL-aware gating network, and finally, explain our SQL-aware dynamic model slicing technique in detail.

4.2.1 Preprocessing Module. There are two sets of input constructed for the SQL-aware prediction modeling given an input tuple. The first set of input is constructed for the gating network and can be uniformly represented as a categorical feature vector \mathbf{q} . $\mathbf{q} = [q_1, q_2, \dots, q_M]$ comprises M feature values from respective attribute fields, where numerical attributes need to be converted into categorical attributes via discretization, as discussed in the previous subsection. The second set is the attribute values of the input tuple $\mathbf{x} = [x_1, x_2, \dots, x_M]$, and each attribute value x_i can be either categorical or numerical.

For both \mathbf{q} and \mathbf{x} , each field of attribute value v_i (q_i/x_i) needs to be transformed into a corresponding embedding vector \mathbf{e}_i to participate the subsequent predictive modeling. Specifically, each categorical attribute is transformed via *embedding lookup*, i.e., $\mathbf{e}_i = \mathbf{E}_i[q_i]$, $\mathbf{e}_i \in \mathbb{R}^{n_e}$, where n_e is the feature embedding size, and \mathbf{E}_i is the embedding matrix of this categorical attribute. Note that different embedding vectors of \mathbf{E}_i correspond to their respective values of this attribute. As for each numerical attribute x_j of \mathbf{x} , the corresponding embedding vector is obtained by linearly scaling up a learnable embedding vector $\hat{\mathbf{e}}_j$ for this numerical attribute, namely $\mathbf{e}_j = x_j \cdot \hat{\mathbf{e}}_j$. In this way, we obtain fixed-size inputs, i.e., embedding vectors $\hat{\mathbf{q}} = [\mathbf{q}_1, \mathbf{q}_2, \dots, \mathbf{q}_M]$ and $\hat{\mathbf{x}} = [x_1, x_2, \dots, x_M]$.

4.2.2 General Model and SQL-aware Gating Network. The general model comprises a set of K replicated base models, denoted as

$\mathcal{F} = [\mathcal{F}_1, \mathcal{F}_2, \dots, \mathcal{F}_K]$, which are referred to as "expert models". These experts share the same model architecture but learn distinct model parameters during training, which take the same input $\hat{\mathbf{x}}$ and produce different outputs that need to be aggregated for final predictions. The output of the i -th expert for a given input \mathbf{x} is denoted as $\mathcal{F}_i(\hat{\mathbf{x}})$.

As for the SQL-aware gating network \mathcal{G} , it takes the SQL query embedding vectors $\hat{\mathbf{q}}$ as input to produce a K -dimensional vector, termed the gating weight \mathbf{w} , with $\mathbf{w} \in \mathbb{R}^K$. Specifically, a two-layer multilayer perceptron (MLP) is employed as the gating network following the practice [16, 42, 47]. We concatenate all embeddings in $\hat{\mathbf{q}}$ as the input of the gating network $\tilde{\mathbf{q}} = \mathbf{q}_1 \oplus \mathbf{q}_2 \dots \oplus \mathbf{q}_M$, where $\tilde{\mathbf{q}} \in \mathbb{R}^{M \cdot n_e}$, then feed $\tilde{\mathbf{q}}$ to \mathcal{G} , and obtain the gating weight \mathbf{w} by:

$$\begin{aligned} \mathbf{z} &= \phi(\mathbf{W}_1 \tilde{\mathbf{q}} + \mathbf{b}_1) \\ \mathbf{w} &= G(\mathbf{z}) = \mathbf{W}_2 \mathbf{z} + \mathbf{b}_2 \end{aligned} \quad (3)$$

where $\mathbf{W}_1 \in \mathbb{R}^{n_z \times M n_e}$, $\mathbf{W}_2 \in \mathbb{R}^{K \times n_z}$ and $\mathbf{b}_1 \in \mathbb{R}^{n_z}$, $\mathbf{b}_2 \in \mathbb{R}^K$ are the weights and biases respectively, n_z is the hidden layer size, and ϕ represents the ReLU activation function.

Given the gating weight \mathbf{w} , the α -entmax function [12, 44] is further applied to recalibrate \mathbf{w} to a probability distribution. As introduced in Section 2, the hyper-parameter α in α -entmax controls the level of sparsity, and a larger value of α sets more gating weights to zero and thus deactivates more experts for higher efficiency. The output of α -entmax $\tilde{\mathbf{w}}$ is thus:

$$\tilde{\mathbf{w}} = \alpha\text{-entmax}(\mathbf{w}), \quad \tilde{\mathbf{w}} \in \mathbb{R}^K \quad (4)$$

which is used to aggregate expert outputs. The final output of the general model is a weighted average of expert outputs:

$$\hat{\mathbf{y}} = \sum_{i=1}^K \tilde{w}_i \cdot \mathcal{F}_i(\hat{\mathbf{x}}) \quad (5)$$

where $\hat{\mathbf{y}}$ is the prediction given the input \mathbf{x} and the corresponding query \mathbf{q} of the SQL-aware predictive modeling.

4.2.3 Dynamic Model Slicing via Gating Network. For a given SQL query, all the retrieved data tuples share the same recalibrated

gating weight \tilde{w} . Further, $\tilde{w}_i = 0$ in Equation 5 indicates that the corresponding i -th expert is not required in the current predictive modeling, and thus, only a small fraction of experts \mathcal{F}_i need to be activated for prediction for much higher computational efficiency.

Denoting the set of indices of activated experts as $\{I_1, I_2, \dots, I_{n_o}\}$, where n_o is the current number of activated experts and $\tilde{w}_{I_j} \neq 0, \forall j \in \{1, 2, \dots, n_o\}$, and given the corresponding SQL query encoding \mathbf{q} , we index the activated experts to form a sliced model, i.e., $\mathcal{F}_{\mathbf{q}} = [\mathcal{F}_{I_1}, \mathcal{F}_{I_2}, \dots, \mathcal{F}_{I_{n_o}}]$. Therefore, the final output of the sliced model is as follows:

$$\hat{y} = \sum_{j=1}^{n_o} \tilde{w}_{[I_j]} \cdot \mathcal{F}_{I_j}(\mathbf{x}) \quad (6)$$

where the number of activated experts n_o directly affects the effectiveness and efficiency of the sliced model. A large n_o indicates larger model capacity while incurring higher computational overhead, and vice versa. In LEADS, n_o is determined by the gating network based on the SQL query encoding \mathbf{q} and the hyper-parameter α of the sparse softmax function. Notably, instead of predefining a fixed value, α in α -entmax is learnable and optimized based on the input tuples and corresponding queries during training. Subsequently, during inference, LEADS can dynamically adapt n_o based on the current SQL query, trading off between the effectiveness and efficiency of the predictive modeling.

4.3 Optimization

Our LEADS framework can be applied to different predictive tasks by configuring a proper objective function, such as mean squared error (MSE) for regression or cross-entropy for classification. For instance, in binary classification, the objective function employed is binary cross-entropy:

$$\text{LogLoss}(\hat{y}, \mathbf{y}) = -\frac{1}{N} \sum_i^N \{y_i \log \sigma(\hat{y}_i) + (1 - y_i) \log(1 - \sigma(\hat{y}_i))\} \quad (7)$$

where \hat{y} represents the prediction labels, \mathbf{y} denotes the ground truth labels, N is the number of tuples for prediction, and $\sigma(\cdot)$ is the sigmoid function.

To make the optimization more robust and effective, we introduce two regularization terms to the main loss function. The first term is the balance loss, \mathcal{L}_{baln} , which is introduced to address the issue of *imbalanced expert utilization*. This imbalance occurs when the gating network \mathcal{G} tends to favor a small subset of experts, leading to a skewed training process where these preferred experts are overutilized while others are underutilized. Such a scenario undermines the capacity of MoE and can detrimentally affect the model performance.

Let \mathbf{X} denote a mini-batch of training instances with n_b tuples, and $\tilde{\mathbf{W}} = [\tilde{w}_1, \tilde{w}_2, \dots, \tilde{w}_{n_b}]$ is the recalibrated gating weights of \mathbf{X} , where \tilde{w}_{ij} is the j -th weight of \tilde{w}_i . \mathcal{L}_{baln} is defined as:

$$\mathcal{L}_{baln} = cv(\Phi) = \sum_{j=1}^K \frac{\phi_j - \mathbb{E}(\Phi)}{\mathbb{E}(\Phi)^2} \quad (8)$$

$$\Phi = [\phi_1, \phi_2, \dots, \phi_K], \phi_j = \sum_{i=1}^{n_b} \tilde{w}_{ij}$$

where $\mathbb{E}(\Phi) = \frac{1}{K} \sum_{j=1}^K \phi_j$. The balance loss term \mathcal{L}_{baln} encourages uniform distribution of weights across experts within the mini-batch to ensure more balanced importance among all experts.

The balance loss term is designed to encourage a more balanced selection of experts, which however, empirically results in activating a large number of experts despite the introduction of sparse softmax. To maintain sparsity within the model and counteract this tendency for more efficient computation, we further introduce a sparsity loss term, \mathcal{L}_{sprs} :

$$\mathcal{L}_{sprs} = -\frac{1}{n_b} \sum_{i=1}^{n_b} (\tilde{w}_i)^2 \quad (9)$$

which encourages the gating network to allocate higher weights to select a few experts while minimal or zero weights to others. Both loss terms are scaled by their respective *regularization coefficient*, λ_1 and λ_2 , and then added to the main loss:

$$\text{Loss} = \text{LogLoss}(\hat{y}, \mathbf{y}) + \lambda_1 \mathcal{L}_{baln} + \lambda_2 \mathcal{L}_{sprs}. \quad (10)$$

With this objective function, LEADS can then be trained effectively with gradient-based optimizers, e.g., SGD or Adam [31].

5 IN-DATABASE MODEL INFERENCE

In this section, we present our in-database model inference system INDICES. We use PostgreSQL [61] as our underlying database system. By exploiting Postgres extension and User-Defined-Functions (UDFs), we seamlessly incorporate the SQL-aware dynamic model slicing technique LEADS onto PostgreSQL to enable in-database model inference.

The typical model inference pipeline consists of four stages: model loading, data retrieval, data preprocessing, and inference. A naive way to support LEADS is to decouple the database system and the inference system. That is, analysts retrieve data from the database via SQL query, preprocess the data, and perform inference in a dedicated inference system. However, this decoupled solution presents three drawbacks. First, moving the data out from the database can expose it to potential security risks and may not align with compliance standards. Second, it is troublesome for the users to maintain two separate systems with complicated data analytics workflow. For traditional data analysts accustomed to SQL queries, learning an additional inference framework represents an extra burden. Third, moving data from the database to the inference system may incur additional overhead and latency, particularly when dealing with large datasets being moved across the network to an external inference system. We conduct a profiling experiment to evaluate the time usage breakdown of the decoupled solution using LEADS with the PyTorch [26] runtime. As shown in Figure 4, data retrieval time occupies approximately 33% to 43% of the total inference time. This overhead primarily stems from the database connection, serialization, network communication, and deserialization for data movement. Therefore, we focus on creating an in-database inference system by integrating the inference procedure onto PostgreSQL via UDF to avoid transferring data out from the database and reduce data retrieval overhead.

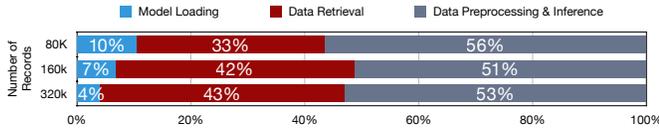


Figure 4: The breakdown response time of the inference stages for the decoupled approach.

5.1 Inference UDF Design

By utilizing a UDF that encompasses all stages of the inference process, users can initiate queries by specifying parameters such as ‘TableName’ and ‘WHERE’ conditions, as illustrated in Figure 5. The UDF then retrieves relevant subdatasets from the database by applying the ‘WHERE’ condition on ‘TableName’. Subsequently, the UDF dynamically loads the trained model and performs the designated inference task. Upon completion, the prediction results are returned to the users. However, relying solely on Python in UDFs for the entire model inference pipeline remains suboptimal due to its inefficient data retrieval process. Therefore, we introduce three optimizations to improve the UDF inference efficiency.

Efficient Execution Allocation. We utilize a multi-language strategy in UDFs, combining low-level languages like C or Rust for efficient data retrieval, and high-level languages like Python for model loading, data preprocessing, and inference with its extensive ML libraries, such as Pytorch and Sklearn. Rust is employed in our system due to its Postgres extension development library PGRX [1] which helps access advanced low-level data retrieval APIs in Postgres like the Server Programming Interface (SPI) for faster data retrieval. However, even with this approach, there are two main challenges for efficient model inference: (1) data copying overhead, arises from different execution environments and data representations between RDBMS and the inference runtime. It necessitates extensive copying and conversions. (2) state initializing overhead, comes from repeatedly loading and releasing the deep learning model when handling an inference request. To mitigate these overheads, we further design memory sharing and state caching techniques in INDICES to improve inference efficiency.

Memory Sharing. Due to the isolation between Python and Rust execution environments, data transfer between them requires two read-write operations: first, data is fetched from an RDBMS and stored in Rust’s environment, then it is transferred from Rust’s memory to Python’s memory. To mitigate data transfer inefficiencies, we leverage shared memory to bypass redundant read-write operations. Initially, data is filtered and retrieved within Rust’s environment using SPI. The data is then directly written to shared memory. This shared memory, allocated at the commencement of UDFs invocation, is accessible in both environments. Thus, the Python environment can directly access and extract data, eliminating the need for an additional copying step.

State Caching. In handling numerous inference requests, the frequent loading and releasing of the model during each inference execution incur significant overhead. To address it, we persistently cache the general model trained via the LEADS technique at the PostgreSQL session-level and maintain a state cache for the utilized sliced model. Specifically, when our inference UDF encounters an SQL query with a new filter condition, it first checks the cache for

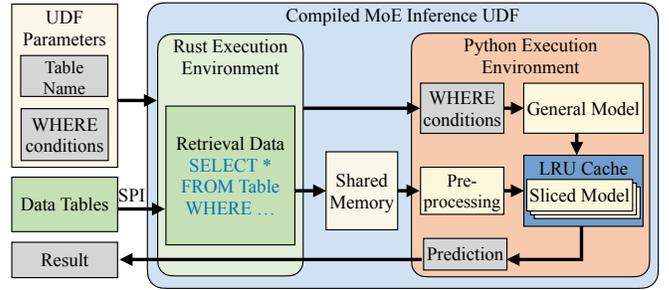


Figure 5: INDICES inference UDF execution.

Table 1: Dataset statistics.

Dataset	Tuples	Positive Ratio	Attributes	Features
Payment	30,000	21.4%	23	350
Credit	244,280	7.8%	69	550
Census	269,356	6.4%	41	540
Diabetes	101,766	46.8%	48	850

the existing sliced model related to this condition. If the corresponding dedicated model is not found, a new model is derived from the general model and stored in the cache. To ensure efficient memory utilization and achieve constant time complexity for management, we adopt the least recently used (LRU) caching policy to manage cached sliced models.

5.2 System Workflow

Next, we present INDICES’ workflow, which comprises training and in-database inference phases, as shown in Figure 6.

Training Phase. In the training phase, we construct a general model based on tables in RDBMS following LEADS for the prediction task. We collect SQL query logs from the real-world database to help construct the training workload. The frequent filter conditions in these logs reveal the attributes and features that data analysts consider most relevant and significant. Using these queries, we can extract the corresponding subdatasets as the training dataset. Both the SQL queries and the selected subdatasets are preprocessed, vectorized, and fed into the general model for iterative training (Step 1 in Figure 6). Once this well-trained general model is prepared, it is serialized and saved as a state dictionary (Step 2). When the associated UDF is invoked, the model is loaded into Postgres and dynamically sliced based on the SQL query, allowing for the handling of online inference requests.

In-Database Inference Phase. In the inference phase, we integrate the inference process into Postgres by implementing a UDF through extension installation. The UDF, named *indices_inference*, offers a SQL interface for issuing inference queries using the following statement:

```
SELECT indices_inference(<tableName>, <taskName>, <filter>);
```

which accepts three arguments, the *tableName* refers to the table in RDBMS from which the subdataset is selected, the *taskName* represents the prediction target (e.g., click-through-rate, readmission-rate). The final argument *filter* denotes the propositional formula following the “WHERE” clause.

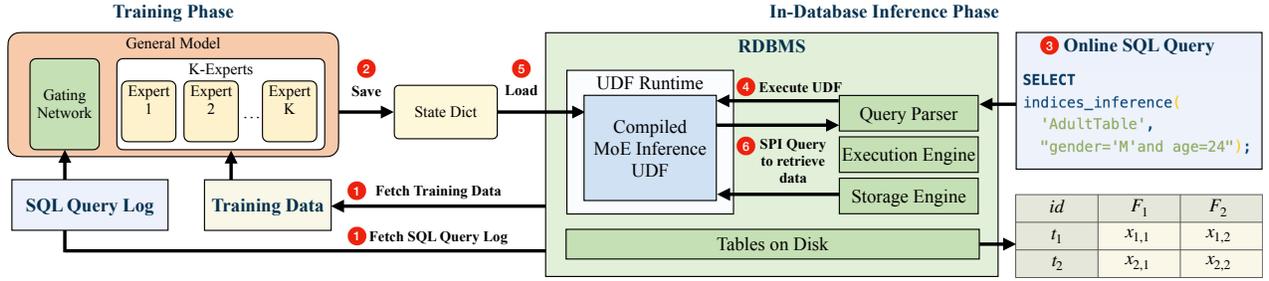


Figure 6: INDICES workflow: model training and in-database inference.

When the query parser receives the inference query (Step 3), it initiates the execution of the UDF within the PostgreSQL UDF runtime environment (Step 4), as illustrated in Figure 5. Upon activation, the UDF performs four main tasks for an online SQL query. First, using the initial two parameters, it identifies the prediction target and determines the required general model. It checks whether the model is already cached; if not, it locates and loads the trained general model (Step 5). Second, the UDF customizes the model via LEADS technique mentioned in Section 4 based on the *filter* in the query and subsequently caches it in the Least Recently Used (LRU) cache. Third, it retrieves the relevant data based on the filter conditions specified in the query via the server programming interface (SPI) and writes the selected data to a shared memory (Step 6). The model inference stage then reads the data from the shared memory and executes model inference. Finally, the UDF creates a view based on the original table, incorporating a new column that holds the predictive results.

6 EXPERIMENTS

In this section, we evaluate the effectiveness of LEADS and efficiency of our in-database inference system, INDICES, using four real-world datasets. Particularly, we devise the experiments to answer the following three key research questions (RQs):

- RQ1: Does the LEADS technique improve the SQL-Aware predictive modeling task compared with the original base models?
- RQ2: How effective is each component of LEADS in these prediction tasks?
- RQ3: Does the INDICES system improve the inference efficiency compared with the traditional decoupled approach?

We report our findings with regards to the above questions respectively in Sections 6.2, 6.3, and 6.4.

6.1 Experimental Setup

6.1.1 Datasets. We conduct experiments on four real-world datasets from the domains of finance, sociology, and healthcare. The statistics of the datasets are summarized in Table 1.

- (1) **Payment** [59, 60] consists of the profile of credit card clients and their past bill payments. The task is to predict whether the payment on a credit card will be in default in the next month.
- (2) **Credit** [6, 25] is gathered by Home Credit Group, focusing on the unbanked population. The task is to predict the repayment abilities of this population for better loan experience.
- (3) **Census** [3, 56] contains data from the Current Population Survey conducted by the U.S. Census Bureau. The task is to determine

Algorithm 1 Synthetic Workload Generation

Require: dataset D , the number of SELECT queries N , the maximum filter condition size max_col

Ensure: a synthetic workload W containing N SELECT queries

- 1: $W = \emptyset$
- 2: **for** $i \leftarrow 1$ to N **do**
- 3: Randomly select a data tuple $\mathbf{x} \in \mathbb{R}^M$ from D
- 4: Randomly sample the number of selected columns $m \in [1, \min(max_col, M)]$
- 5: Randomly sample m columns from data tuple \mathbf{x} along with their corresponding values
- 6: Form a SELECT query with a filter condition of size m based on the selected columns and values
- 7: Add the generated SELECT query to the workload W
- 8: **end for**
- 9: **return** synthetic workload W

whether a person’s annual income exceeds 50K based on their profile information, including age, class education, etc.

(4) **Diabetes** [10, 49] contains ten years of clinical care at 130 US hospitals. Each tuple pertains to hospital records of patients diagnosed with diabetes, including details like medications and laboratory results. The task is to predict the patient’s readmission.

6.1.2 Workloads. In SQL-aware predictive modeling, there is currently no benchmark that fulfills the criteria of having both SQL queries and supervised data. Traditional OLAP benchmarks, like TPC-DS [39] and YCSB [11], primarily focus on assessing query performance with complex operations such as JOIN and GROUPBY. However, they lack prediction tasks and labeled data. On the other hand, conventional datasets for evaluating deep learning algorithms do not incorporate OLAP queries. To bridge this gap, we opt to create synthetic inference queries as workloads based on deep learning datasets to evaluate LEADS and INDICES.

Our workload generation method is outlined in Algorithm 1, which employs a random strategy to generate a workload that comprises a set of synthetic SQL queries, with each query retrieving a subset of the dataset for prediction. The procedure begins by randomly selecting a data tuple \mathbf{x} from the dataset D (Step 3). Then, a value m is sampled from the range $[1, \min(max_col, M)]$ to determine the number of predicates in the SQL query, where M is the number of attributes in D (Step 4). max_col is a parameter that indicates the maximum number of predicates in any SQL query. Subsequently, m attributes are randomly chosen from \mathbf{x} , and the

Table 2: Evaluation of performance improvements with LEADS.

Datasets	Metric	DNN			CIN			AFN			ARMNet		
		w/o	w/	Imprv.									
Payment	Workload-AUC	0.7003	0.7089	+1.23%	0.7164	0.7189	+0.35%	0.7067	0.7143	+1.08%	0.7141	0.7212	+0.99%
	Worst-AUC	0.4733	0.5467	+15.51%	0.3836	0.4463	+16.35%	0.4467	0.6333	+41.77%	0.5267	0.6067	+15.19%
Credit	Workload-AUC	0.7145	0.7427	+3.95%	0.7234	0.7408	+2.41%	0.7171	0.7218	+0.66%	0.7231	0.7347	+1.60%
	Worst-AUC	0.3852	0.6000	+55.76%	0.3333	0.4074	+22.23%	0.3852	0.4074	+5.76%	0.4444	0.6264	+40.95%
Census	Workload-AUC	0.9157	0.9200	+0.47%	0.9187	0.9224	+0.40%	0.9151	0.9216	+0.71%	0.9196	0.9237	+0.45%
	Worst-AUC	0.7692	0.8041	+4.54%	0.7692	0.7845	+1.99%	0.7577	0.7892	+4.16%	0.7692	0.7962	+3.51%
Diabetes	Workload-AUC	0.8308	0.8375	+0.81%	0.8322	0.8419	+1.17%	0.8329	0.8390	+0.73%	0.8342	0.8402	+0.72%
	Worst-AUC	0.5495	0.6374	+16.00%	0.6264	0.7033	+12.28%	0.6484	0.6813	+5.07%	0.6044	0.6593	+9.08%

Table 3: Top-4 SQL queries in terms of AUC improvement due to LEADS.

query#	no. of tuples (test/train)	propositional formula in query
1	20/134	$change = \text{"No"} \ \&\& \ admission_type = 3$
2	20/153	$outpatient = 20 \ \&\& \ metformin.pio = \text{"Up"}$
3	55/451	$glipizide = \text{"Down"}$
4	61/481	$diag_1 = \text{"50"}$

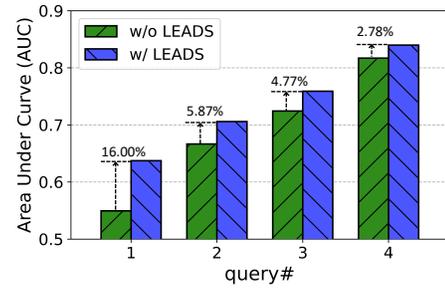
values of these selected attributes are collected to form a propositional formula for the generated SQL query (Steps 5-6). This query is put into the workload (Step 7). We repeat this process N times to create a complete workload. In the experiments, we set N to 30 and max_col to 3, and generate workloads for each dataset.

6.1.3 Baseline Methods. We select four kinds of base models designed for structured data and enhance these models via the LEADS technique. We evaluate LEADS’s effectiveness by comparing the performance of these base models with and without the integration of LEADS. We introduce each base model as follows.

- (1) **DNN** [19]: it is a perceptron with multiple linear and activation layers, representing the most fundamental neural network.
- (2) **CIN** [35]: it is a convolutional layer-based neural network, which models higher-order feature interactions through compressed interaction with input embeddings.
- (3) **AFN** [9]: it incorporates logarithm neurons in the network layer, aiding in capturing the feature interaction in arbitrary order.
- (4) **ARMNet** [8]: it introduces multi-head attention to adaptively extract the combination of features, demonstrating state-of-the-art performance in structured data prediction tasks.

Moreover, to evaluate the efficiency of our in-database inference system INDICES, we compare it with **INDICES-decoupled**, a variant of INDICES that follows the traditional inference approach mentioned in Section 5. For INDICES-decoupled, since data is retrieved from PostgreSQL through network communication based on *psycopg*, there is no data copy process between different execution environments as in INDICES. To ensure a fair comparison, we warm up both systems by caching the general model in advance.

6.1.4 Evaluation Metric. Notice that a workload contains a series of prediction queries, with each query representing a specific prediction task. We use the AUC (Area Under the ROC Curve) metric

**Figure 7: AUC improvement of SQL queries listed in Table 3.**

to evaluate the effectiveness of LEADS on a query, denoted by q . A higher value indicates a better performance. Then, we use two metrics to assess the overall performance of LEADS on the entire workload. The first is the average AUC value across all queries in the workload, denoted as *Workload-AUC*, which is calculated by:

$$\text{Workload-AUC}(W) = \frac{1}{N} \sum_{i=0}^N \text{AUC}(q_i), \quad (11)$$

where N is the number of SQL queries in the workload W . The second is the lowest AUC value among all prediction queries in the workload, termed *Worst-AUC*, which is calculated as follows:

$$\text{Worst-AUC}(W) = \text{Min}(\text{AUC}(q_1), \text{AUC}(q_2) \dots \text{AUC}(q_N)). \quad (12)$$

In fields like finance or healthcare, where prediction errors can result in significant losses, focusing on the lower bound of the performance is crucial. The Worst-AUC metric provides insights into the worst-case scenario, ensuring that the technique’s performance is reliable and does not lead to the worst decisions.

For model-level efficiency, we utilize the floating-point operations per second (FLOPs) metric to measure the computations during the inference phase of a workload. As for our system INDICES, we measure the performance using the end-to-end response time, which calculates the CPU time elapsed from the moment a user invokes an inference query to the moment when the user receives the prediction results.

6.1.5 Hyper-parameter Settings. For fair comparisons, we fix the feature embedding size at 10 and set the size of the hidden layer to 32 for all the base models. Given the ability to select multiple experts in LEADS, we reduce the hidden layer size of each expert to 16 for better efficiency. The depth of each base model is set to 3.

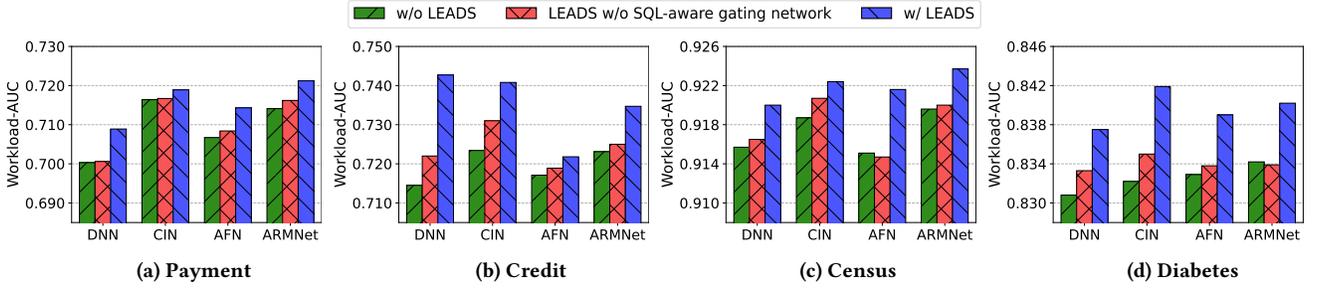


Figure 8: Effects of SQL-aware gating network on accuracy.

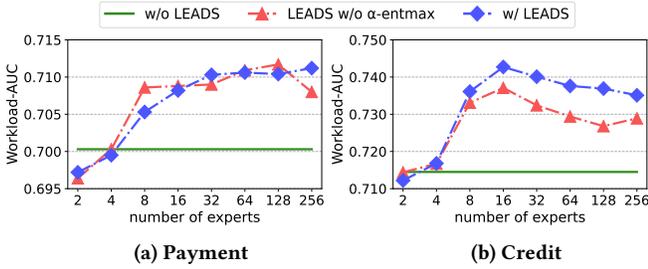


Figure 9: Effects of α -entmax and number of experts on accuracy.

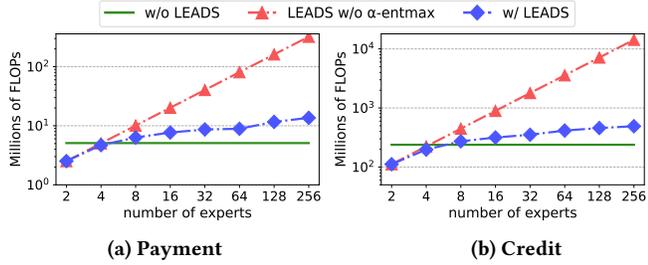


Figure 10: Effects of α -entmax and number of experts on efficiency.

For ARMNet, we specify the number of heads and the hidden size of the self-attention module as 8 and 16, respectively. The initial α in α -entmax is set to 2.5. The number of experts of LEADS is searched in 2-256 and fixed at 16. Also, the balance regularization factor λ_1 and the sparsity regularization factor λ_2 are searched in $1e-3$ - $5e-2$, and fixed at $1e-3$. We perform a sensitivity analysis on these hyper-parameters and report the best results.

6.1.6 Training Details. Since there are no specific SQL queries in the training dataset to update the parameters of the gating network, we simulate a SQL query for each input data following Steps 3-6 in Algorithm1. Additionally, we adopt the Adam [31] optimizer with a learning rate searched in $1e-3$ - 0.1 and a batch size of 1024 for all base models and datasets. All the experiments are conducted on a server equipped with Xeon(R) Silver 4114 CPU@2.2GHz (10 cores), 256G memory, and GeForce RTX 3090 Ti. We implement all the models with PyTorch 1.6.0 with CUDA 10.2.

6.2 SQL-aware Predictions

To evaluate the efficacy of LEADS, we investigate the performance improvement of four base models after integrating with LEADS for given workloads.

The experimental results are summarized in Table 2. The main observation is that the prediction performance w.r.t. both Workload-AUC and Worst-AUC consistently improve when utilizing LEADS for all base models and all workloads. Further, we note that the most significant improvement is in the Worst-AUC metric. For instance, when using DNN as the base model, LEADS achieves improvements of 55.76% and 16.00% on the Credit and Diabetes datasets, respectively. The reason for the base model’s low performance could be significant variability or nuances in the instances of the retrieved subset that are not well-represented in the training data. As a consequence, the trained base model fails to provide accurate predictions for these instances.

To further analyze the Worst-AUC improvement, we perform a breakdown analysis on the Diabetes dataset with a DNN base model. Table 3 describes the top-4 SQL queries in terms of AUC improvement due to LEADS, and Figure 7 presents the respective AUC improvement of these queries. We observe that the AUC values under these SQL queries are less than Workload-AUC (see Table 2), the average AUC value of all queries in the workload. In addition, the number of training/testing tuples in the selected subset for each SQL query is very small. For instance, there are only 134 training tuples for SQL query#1, while the whole training dataset contains 101,766 tuples (see Table 1). Without sufficient examples, the base model cannot generalize effectively in these subdatasets, resulting in a misleading prediction. In our LEADS, the SQL-aware network leverages the propositional formula from SQL queries as meta-information to assist the general model in learning associated patterns within these subdatasets. Therefore, a base model handled by LEADS yields better performance for queries with limited numbers of related training samples.

We note that the propositional formulas of these four queries in the Diabetes dataset all revolve around the drug’s status. For instance, in query #2, $glipizide = \text{“Down”}$ denotes that the drug “glipizide” has been prescribed and its dosage reduced. In healthcare records, the drug’s status is sparse, seen in only a few training samples. Despite its rarity, it significantly affects readmission rates, making it essential for analysts to gain insights into retrieving these subsets. This makes LEADS valuable for improving prediction models and preventing performance collapse, especially in healthcare,

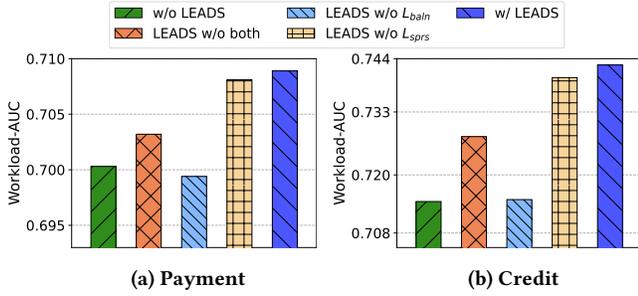


Figure 11: Effects of the regularization terms on accuracy.

where insights from specific subsets matter. Our experimental results demonstrate that LEADS effectively handles this scenario. The improvement results from dynamically integrating multiple expert models. When faced with challenging input samples, LEADS strategically allocates more experts for the sliced model, enhancing the network’s complexity and improving prediction capabilities.

6.3 Ablation Study

In this subsection, we shall conduct an ablation study to evaluate the effectiveness of each component in LEADS.

SQL-aware gating network. In this evaluation, we compare LEADS with two methods: *w/o LEADS* and *LEADS w/o SQL-aware gating network*. For the latter, we create a default SQL query vector by concatenating a set of default values for each attribute, denoted as $\mathbf{q}_d = [\Delta_1, \Delta_2, \dots, \Delta_M]$, indicating the absence of predicates in the SQL query for slicing a model. The comparison results are shown in Figure 8. There are two main observations. First, the *w/o LEADS* method achieves the lowest Workload-AUC because it simply uses the base model to handle the SQL queries. Second, the *LEADS w/o SQL-aware gating network* method results in a performance reduction across four base models compared to LEADS. For example, on the Credit dataset, this reduction can reach up to 0.02 in terms of Workload-AUC. It is because without the SQL-aware gating network, LEADS loses the ability for dynamic model customization based on SQL query vectors, leading to unsatisfactory results.

α -entmax. In this investigation, we evaluate the effect of the α -entmax function in LEADS. We compare LEADS to *w/o LEADS* and *LEADS w/o α -entmax*, where the latter is a variant that substitutes the α -entmax function with the softmax function. We use DNN as the base model and vary the number of experts from 2 to 256 to evaluate the performance w.r.t. Workload-AUC and FLOPs on Payment and Credit datasets.

The results are presented in Figures 9-10. We can observe from Figure 9 that as the number of experts increases from 2 to 32, there is a notable improvement in AUC. This is expected because the model is able to generate more accurate predictions with additional experts. However, when the number of experts exceeds 32, there is a reduction in the Credit dataset for *LEADS w/o α -entmax*, because the model becomes overly complex and results in overfitting and subsequently a decline in AUC performance. Figure 10 highlights the advantages of α -entmax in terms of FLOPs saving, particularly when the number of experts increases. Specifically, the FLOPs of *LEADS w/o α -entmax* show a linear increase with the growing number of experts. In contrast, *LEADS with α -entmax* experiences

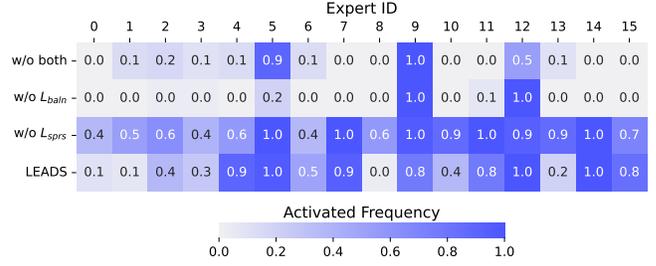


Figure 12: Analysis of the activated frequency for each expert w.r.t. the query workload. The x-axis denotes the expert ID, and the y-axis denotes the frequency (1.0 means that the expert is activated in every SQL query).

an initial increase with a much gentler incline due to the fact that α -entmax assigns small values to zero instead of retaining all experts when using softmax. In the slicing process, we then remove these unused experts with zero weights, effectively conserving computational resources.

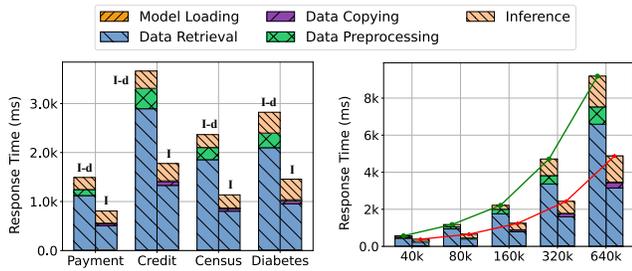
Regularization terms. For the balance term L_{baln} and the sparsity term L_{sprs} , we compare the performance of LEADS with three variants: without the balance term (*LEADS w/o L_{baln}*), without the sparsity term (*LEADS w/o L_{sprs}*), and without both terms (*LEADS w/o both*). We use DNN as the base model and conduct experiments on the Payment and Credit datasets.

Figure 11 presents the comparison results w.r.t. Workload-AUC, and Figure 12 provides a breakdown analysis of activated frequency for each expert during the execution of the query workload, where a higher frequency indicates more extensive usage of an expert in the workload. There are three main findings from the results. First, removing the balance term significantly reduces the Workload-AUC of LEADS, as shown in Figure 11. While the sparse term is designed to counteract the effect of the balance term, using it solely will lead to a much smaller number of experts being utilized for each SQL query, resulting in lower prediction accuracy. This phenomenon can be validated in Figure 12, where only two experts are predominately selected. Second, when only adding the balance term, the performance is slightly lower than that of LEADS (see Figure 11), but the model utilizes almost all experts for every SQL query (see Figure 12), increasing computational costs. This is because the balance term encourages an even expert selection and drives the LEADS to utilize as many experts as possible. Third, when enabling both terms in LEADS, we can observe that the experts are utilized in a balanced manner while achieving the best performance, which demonstrates the effectiveness of our regularization terms.

6.4 System Efficiency

In this subsection, we evaluate the efficiency of INDICES in terms of its end-to-end response time and compare it with the INDICES-decoupled baseline (see Section 6.1.3). We also measure the breakdown response time for each system.

Comparison with the baseline. We utilize a SQL query that selects 100k records for inference. We report the response time of INDICES and INDICES-decoupled on the four datasets, as presented in Figure 13a. Compared to INDICES-decoupled, INDICES achieves speedup of 1.94x, 2.06x, 2.00x, and 1.82x on the Census, Credit, Diabetes, and Payment, respectively. There are three main reasons



(a) Response time for predicting 100k records on four datasets. (b) Response time w.r.t. #predicting records on Payment dataset.

Figure 13: Efficiency evaluation of INDICES in terms of response time. In the sub-figures, the ‘Left Bar’ denotes INDICES-decoupled (I-d), and the ‘Right Bar’ denotes INDICES (I).

for such superior performance. First, INDICES reduces the costly data movement overheads between PostgreSQL and the inference system, with lower data retrieval time usage. Second, INDICES is further enhanced with the aforementioned optimizations: shared memory to reduce data copying overhead, and state caching to eliminate the cost of model loading during the entire inference UDF execution process. Last, within Frame’s Python execution, data from shared memory is directly read as a *numpy.ndarray*[22], which facilitates quicker conversion to tensors for inference tasks and results in lower data preprocessing time in INDICES.

Effects of the number of predicting records. We next evaluate the effect of the number of selected records in the SQL query in terms of the response time. We create SQL queries that select various numbers of records ranging from 40k to 640k from the Payment dataset. Figure 13b shows the response time of INDICES and INDICES-decoupled. We observe that INDICES consistently surpasses INDICES-decoupled across all record sizes, with performance improvement ranging from 1.47x to 1.93x. Moreover, the response time of INDICES increases more slowly than that of INDICES-decoupled. This is because the data movement overhead between the database and the inference system becomes more pronounced with more records.

Evaluation of optimization techniques in INDICES. Further, we experiment to evaluate the benefits of the optimizations in Section 5.1. Specifically, we compare INDICES with (i) INDICES without memory sharing; (ii) INDICES without SPI; (iii) INDICES without state caching; and (iv) INDICES without any optimizations. Figure 14 presents the comparison results w.r.t. the response time to predict 100k records on the Payment dataset. The absence of shared memory results in significant data copying overhead between Rust and Python execution environments. Likewise, without SPI, the data retrieval time is high. Besides, if we do not enable state caching, it results in a substantial model loading overhead. With all the optimizations enabled, INDICES can greatly reduce the in-database inference response time.

7 RELATED WORK

Mixture-of-Experts [15, 27, 62] integrates the outputs of different experts in an input-driven manner. In the case of Sparse MoE, only a small subset of experts is chosen for each input, facilitating

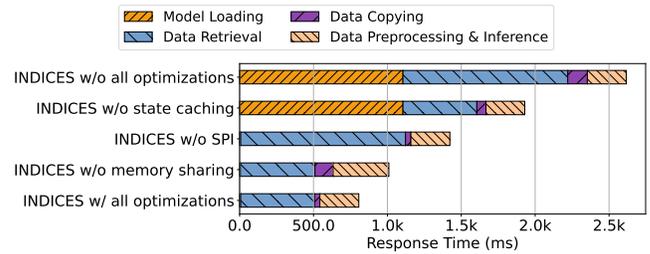


Figure 14: Effects of INDICES’ optimization techniques.

substantial model scaling without additional computation overhead. Sparse MoE has been used to build large language models [16, 33, 48] and applied to vision-related tasks[47, 57]. Our research delves into the potential of MoE in structured data analytics. We closely combine it with database data analytics, selecting experts based on the filter conditions in SQL queries.

In-Database Machine Learning involves running machine learning directly within the database. MADlib [24] is an open-source library providing SQL-based ML functions in PostgreSQL. Google ML library[2], and Microsoft’s SQL Server Machine Learning Services [4] offer SQL APIs for ML functions on Oracle, bigquery, and Microsoft SQL Server, respectively. Recently, [54] proposed to integrate neural architectural search (NAS) model selection into PostgreSQL as an extension, which is orthogonal to our proposal. In summary, these works incorporate existing ML algorithms into the database for analysts but typically lack optimization for specific data analysis scenarios and seldom support deep learning algorithms.

8 CONCLUSIONS

In this paper, we propose a novel SQL-aware dynamic model slicing technique called LEADS. We enhance the general model trained on the entire database with the Mixture of Experts (MoE) technique and devise a SQL-aware gating network to effectively customize a sliced model given the propositional formula in the user’s SQL query. Further, we integrate LEADS into our end-to-end in-database inference system INDICES. We build the system on top of a full-fledged and open-source RDBMS, PostgreSQL, and introduce three optimization techniques to reduce the response time of inference queries. Extensive experiments conducted on four real-world datasets demonstrate that LEADS consistently outperforms four baseline models and INDICES significantly reduces the inference time compared to the conventional approach.

REFERENCES

- [1] [n.d.]. gprx:Framework for developing PostgreSQL extensions in Rust. <https://github.com/pgcentralfoundation/gprx>. Accessed: October 10, 2023.
- [2] [n.d.]. machine learning models in BigQuery ML. <https://cloud.google.com/bigquery/docs/create-machine-learning-model>. Accessed: October 10, 2023.
- [3] 2000. Census-Income (KDD). UCI Machine Learning Repository. DOI: <https://doi.org/10.24432/C5N30T>.
- [4] 2023. Microsoft SQL MLS. <https://learn.microsoft.com/en-us/sql/machine-learning/sql-server-machine-learning-services?view=sql-server-2017>. Accessed: October 10, 2023.
- [5] Mahmoud Abo Khamis, Hung Q Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. 2018. In-database learning with sparse tensors. In *PODS*. 325–340.
- [6] KirillOdimtsov Martin Kotek Anna Montoya, inversion. 2018. Home Credit Default Risk. <https://kaggle.com/competitions/home-credit-default-risk>
- [7] Shaofeng Cai, Gang Chen, Beng Chin Ooi, and Jinyang Gao. 2019. Model Slicing for Supporting Complex Analytics with Elastic Inference Cost and Resource Constraints. *Proceedings of the VLDB Endowment* 13, 2 (2019), 86–99.
- [8] Shaofeng Cai, Kaiping Zheng, Gang Chen, HV Jagadish, Beng Chin Ooi, and Meihui Zhang. 2021. Arm-net: Adaptive relation modeling network for structured data. In *SIGMOD*. 207–220.
- [9] Weiyu Cheng, Yanyan Shen, and Linpeng Huang. 2020. Adaptive factorization network: Learning adaptive-order feature interactions. In *AAAI*, Vol. 34. 3609–3616.
- [10] Cios Krzysztof DeShazo Jon Clore, John and Beata Strack. 2014. Diabetes 130-US hospitals for years 1999–2008. UCI Machine Learning Repository. DOI: <https://doi.org/10.24432/C5230J>.
- [11] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *SoCC*, Joseph M. Hellerstein, Surajit Chaudhuri, and Mendel Rosenblum (Eds.). ACM, 143–154.
- [12] Gonçalo M Correia, Vlad Niculae, and André FT Martins. 2019. Adaptively sparse transformers. *arXiv preprint arXiv:1909.00015* (2019).
- [13] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System. In *NSDI*, Aditya Akella and Jon Howell (Eds.). 613–627.
- [14] Jesse Davis and Mark Goodrich. 2006. The relationship between Precision-Recall and ROC curves. In *ICML*. 233–240.
- [15] David Eigen, Marc'Aurelio Ranzato, and Ilya Sutskever. 2013. Learning factored representations in a deep mixture of experts. *arXiv preprint arXiv:1312.4314* (2013).
- [16] William Fedus, Barret Zoph, and Noam Shazeer. 2022. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *The Journal of Machine Learning Research* 23, 1 (2022), 5232–5270.
- [17] Xixuan Feng, Arun Kumar, Benjamin Recht, and Christopher Ré. 2012. Towards a unified architecture for in-RDBMS analytics. In *SIGMOD*. 325–336.
- [18] Yannis E. Foufoulas, Alkis Simitis, Eleftherios Stamatogiannakis, and Yannis E. Ioannidis. 2022. YeSQL: "You extend SQL" with Rich and Highly Performant User-Defined Functions in Relational Databases. *Proceedings of the VLDB Endowment* 15 (2022), 2270–2283.
- [19] Matt W Gardner and SR Dorling. 1998. Artificial neural networks (the multilayer perceptron)—a review of applications in the atmospheric sciences. *Atmospheric environment* 32, 14–15 (1998), 2627–2636.
- [20] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. 6.2. 2.3 softmax units for multinoulli output distributions. *Deep learning* 180 (2016).
- [21] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. 1997. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data mining and knowledge discovery* 1 (1997), 29–53.
- [22] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. <https://doi.org/10.1038/s41586-020-2649-2>
- [23] Xiangnan He and Tat-Seng Chua. 2017. Neural factorization machines for sparse predictive analytics. In *SIGIR*. 355–364.
- [24] Joe Hellerstein, Christopher Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, et al. 2012. The MADlib analytics library or MAD skills, the SQL. *arXiv preprint arXiv:1208.4165* (2012).
- [25] Xin Huang, Ashish Khetan, Milan Cvitkovic, and Zohar Karmin. 2020. Tabtransformer: Tabular data modeling using contextual embeddings. *arXiv preprint arXiv:2012.06678* (2020).
- [26] Sagar Imambi, Kolla Bhanu Prakash, and GR Kanagachidambaresan. 2021. Py-Torch. *Programming with TensorFlow: Solution for Edge Computing Applications* (2021), 87–104.
- [27] Robert A Jacobs, Michael I Jordan, Steven J Nowlan, and Geoffrey E Hinton. 1991. Adaptive mixtures of local experts. *Neural computation* 3, 1 (1991), 79–87.
- [28] Peng Jia, Shaofeng Cai, Beng Chin Ooi, Pinghui Wang, and Yiyuan Xiong. 2023. Robust and Transferable Log-based Anomaly Detection. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–26.
- [29] Michael I Jordan and Robert A Jacobs. 1994. Hierarchical mixtures of experts and the EM algorithm. *Neural computation* 6, 2 (1994), 181–214.
- [30] Mahmoud Abo Khamis, Hung Q Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. 2020. Learning models over relational data using sparse tensors and functional dependencies. *TODS* 45, 2 (2020), 1–66.
- [31] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).
- [32] Hyoukjun Kwon, Prasanath Chatarasi, Michael Pellauer, Angshuman Parashar, Vivek Sarkar, and Tushar Krishna. 2019. Understanding reuse, performance, and hardware cost of dnn dataflow: A data-centric approach. In *MICRO*. 754–768.
- [33] Dmitry Lepikhin, Hyoukjoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. 2020. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668* (2020).
- [34] Side Li, Lingjiao Chen, and Arun Kumar. 2019. Enabling and optimizing non-linear feature interactions in factorized linear algebra. In *SIGMOD*. 1571–1588.
- [35] Jianxun Lian, Xiaohuan Zhou, Fuzheng Zhang, Zhongxia Chen, Xing Xie, and Guangzhong Sun. 2018. xdeepfm: Combining explicit and implicit feature interactions for recommender systems. In *KDD*. 1754–1763.
- [36] Bin Lin, Zhenyu Tang, Yang Ye, Jiayi Cui, Bin Zhu, Peng Jin, Junwu Zhang, Munan Ning, and Li Yuan. 2024. MoE-LLaVA: Mixture of Experts for Large Vision-Language Models. *CoRR abs/2401.15947* (2024). <https://doi.org/10.48550/ARXIV.2401.15947>
- [37] Pingchuan Ma, Rui Ding, Shi Han, and Dongmei Zhang. 2021. MetaInsight: Automatic Discovery of Structured Knowledge for Exploratory Data Analysis. In *SIGMOD*, Guoliang Li, Zhanhui Li, Stratos Idreos, and Divesh Srivastava (Eds.). 1262–1274.
- [38] Andre Martins and Ramon Astudillo. 2016. From softmax to sparsemax: A sparse model of attention and multi-label classification. In *International conference on machine learning*. PMLR, 1614–1623.
- [39] Raghunath Othayoth Nambiar and Meikel Poess. 2006. The Making of TPC-DS. In *VLDB*, Vol. 6. 1049–1058.
- [40] Guillermo Navas-Palencia. 2020. Optimal binning: mathematical programming formulation. *arXiv preprint arXiv:2001.08025* (2020).
- [41] Milos Nikolic, Haozhe Zhang, Ahmet Kara, and Dan Olteanu. 2020. F-IVM: learning over fast-evolving relational data. In *SIGMOD*. 2773–2776.
- [42] OpenAI. 2023. GPT-4 Technical Report. *CoRR abs/2303.08774* (2023). <https://doi.org/10.48550/ARXIV.2303.08774>
- [43] Kwanghyun Park, Karla Saur, Dalitso Banda, Rathijit Sen, Matteo Interlandi, and Konstantinos Karanasos. 2022. End-to-end Optimization of Machine Learning Prediction Queries. In *SIGMOD*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 587–601.
- [44] Ben Peters, Vlad Niculae, and André FT Martins. 2019. Sparse Sequence-to-Sequence Models. In *ACL*. <https://www.aclweb.org/anthology/P19-1146>
- [45] S Brintha Rajakumari and C Nalini. 2014. An efficient data mining dataset preparation using aggregation in relational database. *Indian Journal of Science and Technology* 7 (2014), 44.
- [46] Steffen Rendle. 2013. Scaling factorization machines to relational data. *Proceedings of the VLDB Endowment* 6, 5 (2013), 337–348.
- [47] Carlos Riquelme, Joan Puigcerver, Basil Mustafa, Maxim Neumann, Rodolphe Jenatton, André Susano Pinto, Daniel Keyser, and Neil Houlsby. 2021. Scaling vision with sparse mixture of experts. *NeurIPS* (2021), 8583–8595.
- [48] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538* (2017).
- [49] Beata Strack, Jonathan P DeShazo, Chris Gennings, Juan L Olmo, Sebastian Ventura, Krzysztof J Cios, John N Clore, et al. 2014. Impact of HbA1c measurement on hospital readmission rates: analysis of 70,000 clinical database patient records. *BioMed research international* 2014 (2014).
- [50] Constantino Tsallis. 1988. Possible generalization of Boltzmann-Gibbs statistics. *Journal of statistical physics* 52 (1988), 479–487.
- [51] Wei Wang, Jinyang Gao, Meihui Zhang, Sheng Wang, Gang Chen, Teck Khim Ng, Beng Chin Ooi, Jie Shao, and Moaz Reyad. 2018. Rafiki: Machine Learning as an Analytics Service System. *Proceedings of the VLDB Endowment* 12, 2 (2018), 128–140.
- [52] Abdul Wasay, Subarna Chatterjee, and Stratos Idreos. 2021. Deep Learning: Systems and Responsibility. In *SIGMOD*, Guoliang Li, Zhanhui Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 2867–2875.
- [53] Steve Waterhouse, David MacKay, and Anthony Robinson. 1995. Bayesian methods for mixtures of experts. *NIPS* 8 (1995), 351–357.
- [54] Naili Xing, Shaofeng Cai, Gang Chen, Zhaojing Luo, Beng Chin Ooi, and Jian Pei. [n.d.]. Database Native Model Selection: Harnessing Deep Neural Networks

- in Database Systems. (n. d.).
- [55] Lijie Xu, Shuang Qiu, Binhang Yuan, Jiawei Jiang, Cedric Renggli, Shaoduo Gan, Kaan Kara, Guoliang Li, Ji Liu, Wentao Wu, et al. 2022. In-Database Machine Learning with CorgiPile: Stochastic Gradient Descent without Full Data Shuffle. In *SIGMOD*. 1286–1300.
- [56] Lei Xu and Kalyan Veeramachaneni. 2018. Synthesizing tabular data using generative adversarial networks. *arXiv preprint arXiv:1811.11264* (2018).
- [57] Fuzhao Xue, Ziji Shi, Futao Wei, Yuxuan Lou, Yong Liu, and Yang You. 2022. Go wider instead of deeper. In *AAAI*, Vol. 36. 8779–8787.
- [58] Zhihui Yang, Zuozhi Wang, Yicong Huang, Yao Lu, Chen Li, and X Sean Wang. 2022. Optimizing machine learning inference queries with correlative proxy models. *arXiv preprint arXiv:2201.00309* (2022).
- [59] I-Cheng Yeh. 2016. Default of credit card clients. UCI Machine Learning Repository. DOI: <https://doi.org/10.24432/C55S3H>.
- [60] I-Cheng Yeh and Che-hui Lien. 2009. The comparisons of data mining techniques for the predictive accuracy of probability of default of credit card clients. *Expert systems with applications* 36, 2 (2009), 2473–2480.
- [61] Andrew Yu and Jolly Chen. 1995. The POSTGRES95 user manual.
- [62] Seniha Esen Yuksel, Joseph N Wilson, and Paul D Gader. 2012. Twenty years of mixture of experts. *IEEE Transactions on Neural Networks and Learning Systems* 23, 8 (2012), 1177–1193.
- [63] Yuhao Zhang, Frank Mcquillan, Nandish Jayaram, Nikhil Kak, Ekta Khanna, Orhan Kislal, Domino Valdano, and Arun Kumar. 2021. Distributed deep learning on data systems: a comparative analysis of approaches. *Proceedings of the VLDB Endowment* 14, 10 (2021).
- [64] Kaiping Zheng, Shaofeng Cai, Horng Ruey Chua, Wei Wang, Kee Yuan Ngiam, and Beng Chin Ooi. 2020. Tracer: A framework for facilitating accurate and interpretable analytics for high stakes applications. In *SIGMOD*. 1747–1763.
- [65] Yanqi Zhou, Tao Lei, Hanxiao Liu, Nan Du, Yanping Huang, Vincent Zhao, Andrew M Dai, Quoc V Le, James Laudon, et al. 2022. Mixture-of-experts with expert choice routing. *NeurIPS* (2022), 7103–7114.