Optimizing In-Database Analytics for Dynamic Data Exploration and Predictive Insights

Meet Amin[1], Maharshi Shukla[2]*
[1]Department of Information Systems, Rider University, New Jersey, USA
[2]Department of Data Analytics Engineering, Northeastern University, Vancouver, Canada

Emails: aminmeet23@gmail.com, maharshishukla19@gmail.com
*Corresponding author: maharshishukla19@gmail.com

Abstract. Extracting actionable insights from large structured datasets is often a significant challenge in data analytics. Common practices for implementing advanced analytical models are to move data around. This limits agility and real-time exploration. This paper proposes a computing infrastructure for on-database data analytics capable of providing correct prediction models through dynamic generation in a relational database management system. Analytics solutions can be customised and deployed according to user requirements or specific data views. This scheme has undergone thorough empirical validation. As a result, complex analysis workflows have been more efficient, more accurate and more responsive. In this research, I lay a foundation to advance the in-situ data analytics paradigms within an organisation and offer agile, powerful and scalable capabilities for data-driven discovery and decision-making.

Keywords: In-database analytics, SQL-aware predictive modeling, Dynamic model slicing, Mixture of Experts, Query-driven analytics

1 Introduction

Relational Database Management Systems (RDBMS) are essential software solutions that store structured data across a vast thrust of applications. They are used widely for a variety of business functions. Due to the rise of the digital economy, it is important to perform analytics on the structured data of systems to create operational insights.

SQL-based operations like filtering and aggregation are typically employed by traditional methods for analyzing. These operations generate statistical sum-maries, but they do not adequately capture the relationships within the data. The rise of Deep Neural Networks (DNNs) has given rise to complex analytics that go beyond statistical inference.[1,2,3]

The process of using DNNs for structured data analytics typically involves building and training a predictive model, followed by deploying the model to generate inferences. The goal of optimisation here is to ensure effectiveness (prediction precision) and efficiency (cost). How well a model predicts an outcome determines effectiveness, and latency and processing load determine efficiency.

In practice, domain experts often rummage through specific parts of the data. For instance, an analysis of the financial characteristics of a target population can be useful in determining loan approval criteria or market pricing, as shown inFig. 1. As a result, it is essential to generate models efficiently and accurately for these subsets.

P. Johri et al. (eds.), *Proceedings of the International Conference on Sustainable Computing and Artificial Intelligence (ICSCAI 2025)*, Advances in Engineering Research 298,
https://doi.org/10.2991/978-94-6239-674-6_6

The main challenge lies in training separate models for all possible data subsets, which is computationally impossible if not infeasible due to the exponential growth in combinations of subdatasets. Often a single model that can generalise well does not fit the specific subset well. A focused model could outperform a more general model if it was trained for a specific group (men, in NYC, for example). However, the costs incurred for training such a model could be very high.

Another concern relates to the latency in inference for subdataset prediction when different analytics/DB systems are used. When the RDBMS sends data to an external inference engine, a delay may happen, and the probability of errors increases. In the case of data sending, the security of the transmitted data will also be compromised. All this added overhead can hurt inference responsiveness significantly.

In order to counter these limitations a new scheme titled.**LEADS** (*SQL-aware dynAmic moDel Slicing*) is presented. LEADS understands the semantics of SQL queries to dynamically construct a high-capacity predictive model created from several copies of a base model. A gating module that is sensitive to SQL is used to activate only those experts that are relevant to the input SQL vector, which maintains the efficiency of inference.

In furtherance of the above, LEADS directly plugs into the RDBMS as a UDF for in-database inference. This implementation avoids data movement between systems and comprises three improvements: execution schedule optimization, shared memory usage, and inference caching.

Key Contributions

- The SQL-driven predictive analytics challenge requires an accurate solution for answers on subdatasets.
- The design of the LEADS framework involves using expert replication and SQL-based gating to modify models for current analytics.
- The goal is to develop an in-database inference mechanism within PostgreSQL with memory and execution. Using five real datasets, we validated our approach. The results show an AUC improvement of up to 3.95% and inference acceleration of 2.06x against standard baselines.

2 Preliminaries

Structured datasets (a.k.a. tabular data) are usually represented as a table with defined rows and columns and stored in a relational manner . SQL queries manipulate this data via selection, projection and aggregation operations. Formally, we model the data as a logical table T which has N records and M features. Each record $\mathbf{x}$ is in the form of $\mathsf{X} = (x_1\, x_2 \cdot \cdot \cdot x_M)$.

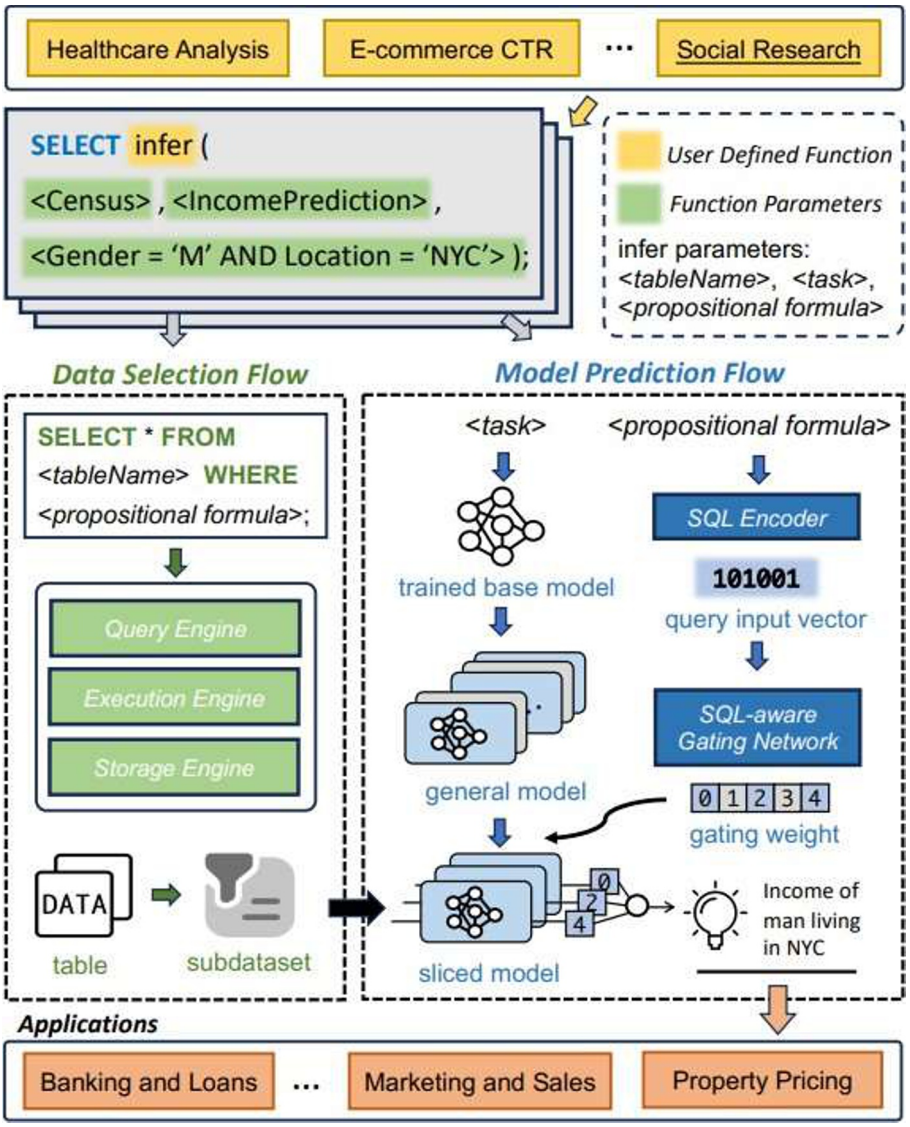


Fig. 1. Illustrative pipeline of SQL-aware dynamic slicing for income analytics within an RDBMS.

2.1 Mixture of Experts (MoE)

MoE is a scalable learning architecture, allowing conditional use of computational effort across its experts Each expert focuses on a slightly different aspect of the problem space, and a gating function decides which expert participates. The combined response $\hat{y}$ of K experts with response h_i and gating weight w_i is computed as:

$$\hat{y} = \sum_{i=1}^{K} w_i h_i \tag{1}$$

Optimising the gating and expert networks together enables adjustment to specific contexts for each query, which strikes a balance between size and prediction cost. Mixture of Experts (MoE) has been the key technique behind many high-impact applications, from language generation to vision-text fusion. In this work, we adapt MoE for tabular prediction.[7,8]

2.2 Sparse Activation via Sparsemax

Unlike softmax, which yields dense probability vectors, sparsemax offers a controlled sparsity by projecting logits onto a probability simplex. The sparsemax function is defined by:

$$\text{sparsemax}(\mathbf{z}) = \arg\max_{\mathbf{p} \in \Delta^d} \left\{ \mathbf{p}^\top \mathbf{z} - \frac{1}{2}\|\mathbf{p}\|_2^2 \right\} \tag{2}$$

This sparsification reduces active experts, enhancing interpretability and computational efficiency.

2.3 SQL-Aware Modeling Problem

Given a structured dataset T and a SQL query q with condition ϕ, the subset $\mathsf{T}_\phi = \{\mathbf{x}_i \in \mathsf{T} : \phi(\mathbf{x}_i)\}$ is selected. For this subset, a DNN $f(\cdot)$ infers predictions:

$$\hat{y}_i = f(\mathbf{x}_i), \quad \forall \mathbf{x}_i \in \mathsf{T}_\phi \tag{3}$$

This framework forms the basis of SQL-aware predictive analytics, where tailored models aim to provide both accuracy and low-latency responses.[10,11]

3 SQL-Aware Dynamic Model Slicing

The LEADS framework proposed by the authors aims at enabling predictive inference right within the traditional relational database systems like PostgreSQL. To achieve that, the authors are proposing a SQL-context-aware dynamic slicing methodology. LEADS constructs predictive pipelines throughout SQL queries instead of leveraging standard predictive models over attributes. It uses the structural properties of queries over the data. According to fig. 2, Common characteristics such as age group, location, and gender represented in query predicates can be encoded to help learn better.

3.1 SQL Query Embedding

A new encoding module can capture filtering semantics and is added to the SQL queries. The dense vector that the encoder generates during inference, which represents the logical conditions of the WHERE clause, is often a conjunction of these attribute-value pairs. Operators generally make use of phrases like (=, !=, ¡, ¿, ¡=, ¿=, LIKE, BETWEEN, [MAX IN] along with typical conjunctions. For example, for males, you may embed the condition as gender='M'. Next, for every predicate Pj on attribute Aj we find Pj: Aj=aj with aji=off Dj else aj= ij Dj. Together they form the complete formula:

$$\varphi = P_1 \wedge P_2 \wedge \cdots \wedge P_M \tag{4}$$

This categorical feature vector $\mathbf{q} = [q_1, q_2, \ldots, q_M]$ serves as the representation for the input query and the j_{th} entry of $\mathbf{q}$ represents the value of A_j. In a supervised discretisation mechanism, numerical fields are binned to optimise information value (IV) while keeping an interpretable signal for prediction.

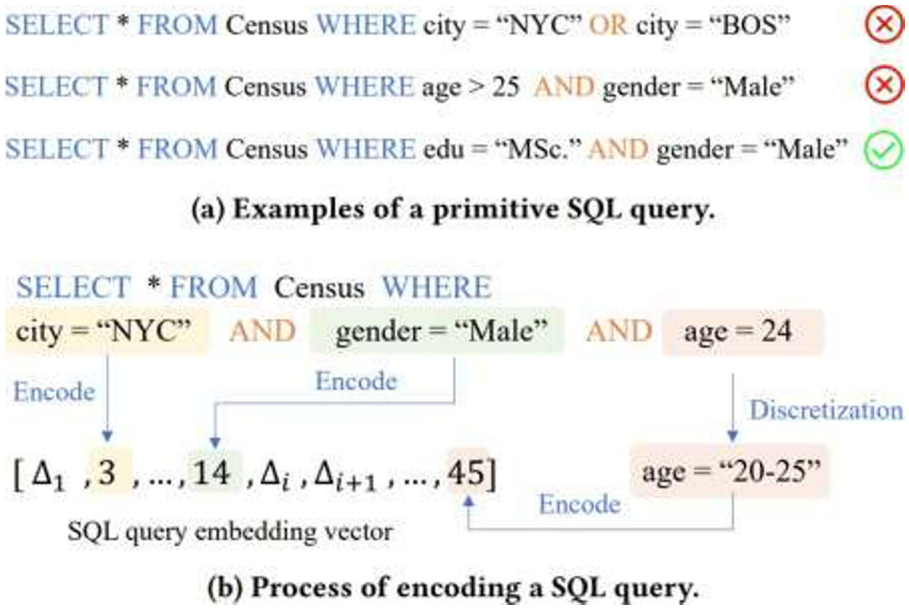


Fig. 2. SQL query encoder for generating query-based feature vectors.

3.2 Customized Model Slicing Based on SQL Context

Once vector $\mathbf{q}$ is constructed, LEADS personalises prediction through a combination of a general model F and a SQL-aware gating mechanism. The general

model consists of K expert sub-models $\mathsf{F} = \{F_1, F_2, \ldots, F_K\}$, each trained to specialize in distinct data segments. These experts receive shared input $\hat{\mathbf{x}}$ derived from tuple features. Numerical fields are scaled embeddings $e_j = x_j \cdot \hat{e}_j$ and categorical attributes are mapped using embedding lookups $e_j = E_j[q_j]$.

The gating function G, implemented as a two-layer MLP, processes concatenated SQL embeddings $\tilde{\mathbf{q}}$ to compute a raw activation vector $\mathbf{w}$:

$$\mathbf{z} = \phi(W_1 \tilde{\mathbf{q}} + \mathbf{b}_1) \tag{5}$$

$$\mathbf{w} = W_2 \mathbf{z} + \mathbf{b}_2 \tag{6}$$

Here, $\phi(\cdot)$ denotes the ReLU activation. This vector is passed through an α-entmax transformation to yield sparse activation weights $\tilde{\mathbf{w}}$, promoting selective expert utilization:

$$\tilde{\mathbf{w}} = \text{entmax}_\alpha(\mathbf{w}) \tag{7}$$

The model output is computed as a weighted aggregation:

$$\hat{y} = \sum_{i=1}^{K} \tilde{w}_i F_i(\hat{\mathbf{x}}) \tag{8}$$

The final prediction only incorporates experts that hold non-zero weight, which defines the sliced model F_q as per current SQL condition:

$$\hat{y} = \sum_{j=1}^{n_o} \tilde{w}_{I_j} F_{I_j}(\hat{\mathbf{x}}) \tag{9}$$

The number of active experts n_o dynamically varies, balancing performance and computational expense. The hyperparameter α is trained end-to-end to control sparsity.

3.3 Loss Optimization Strategy

To optimise LEADS, standard task-specific objectives are employed. For binary classification, the cross-entropy formulation is:

$$\mathsf{L}_{\text{base}} = -\frac{1}{N} \sum_{i=1}^{N} [y_i \log(\sigma(\hat{y}_i)) + (1 - y_i) \log(1 - \sigma(\hat{y}_i))] \tag{10}$$

Two regularisation terms are introduced. The first, balance regularization L_{bal}, penalizes uneven expert usage:

$$\text{Lbal} = (1/\text{K}) \sum \text{from j=1 to K } (\bar{\varphi}\text{j} - \bar{\varphi})^2, \quad \bar{\varphi}\text{j} = (1/\text{L}) \sum \text{from i=1 to L } \tilde{w}\text{ij},$$
$$\bar{\varphi} = (1/\text{K}) \sum \text{from j=1 to K } \bar{\varphi}\text{j} \tag{11}$$

The second, sparsity regularization L_{sprs}, promotes selective activation:

$$\mathsf{L}_{\text{sprs}} = -\frac{1}{n_b} \sum_{i=1}^{n_b} ||\tilde{\mathbf{w}}_i||_2 \tag{12}$$

The total objective function becomes:

$$L = L_{base} + \lambda_1 L_{bal} + \lambda_2 L_{sprs} \tag{13}$$

Optimisation proceeds via gradient descent using algorithms such as SGD or Adam.

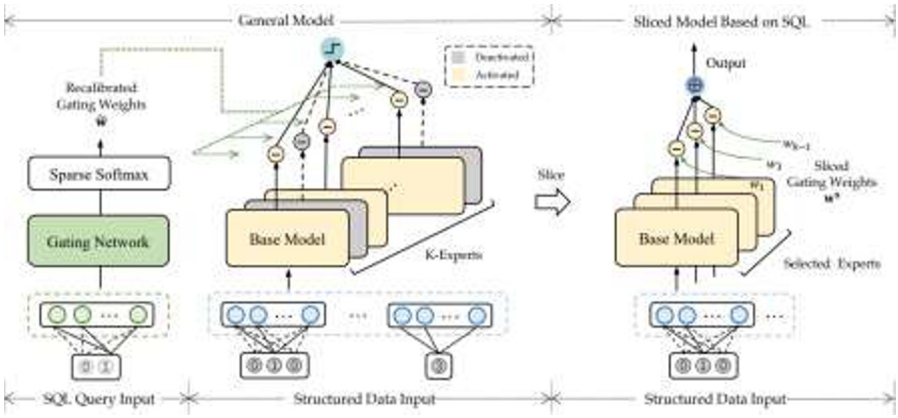


Fig. 3. Components contributing to response time in traditional decoupled inference settings.

4 In-Database Predictive Inference Framework

The section presents the implementation of an embedded inference that adds SQL-aware slicing to NeurDB for better prediction capabilities. The inference pipeline consists of four functional stages: (1) loading the prediction model into memory, (2) extracting subdatasets through structured SQL statements, (3) preprocessing of data into tensor-accepted shapes, and (4) executing forward calculation for prediction outputs.

An ordinary method, denominated here as the Inference-Decouple Strategy (IDS), carries out these steps away from the database. External custom inference systems process extracted subsets in the IDS. Yet, this setup also carries risks of faulty or inefficient data. Specifically, data exported from the RDBMS may violate compliance policies, dual environments complicate workflow, and bulk data transfer leads to latency and load on the system. A graph reveals the changing IDS over time in Fig. 6, which indicates that about 40% of total inference time was spent on retrieval delays.

To overcome those drawbacks, we adopt the In-Database Inference Strategy. The inference conducted in the RDBMS avoids data transfer, preserves security, and reduces computational overhead. [13,14]

4.1 Optimized UDF-Based Inference

The IIS method encapsulates all inference stages inside user-defined functions (UDFs). As shown in Fig 4, you issue a predictive query by specifying the source table and SQL condition. The UDF fetches the required data slice based on the given constraints, loads the pre-trained model, and runs inference. Predictions are returned inline

There are three design improvements to improve runtime efficiency.

Heterogeneous Language Execution In order to take advantage of the strengths of different programming environments, while Rust implements the retrieval component using the PGRX library and the SPI interface to PostgreSQL, Python takes care of preprocessing and inference, and it does so relying on a greater variety of ML frameworks such as PyTorch. This arrangement strikes a balance between rapid data retrieval and flexible architecture.

Inter-Process Memory Sharing Before executing the UDF, shared memory buffers are allocated to prevent redundant data movement between environments. The data is filtered in Rust. Then the content is written directly into shared memory, which is consumed in Python. So it bypasses serialisation overhead.

Model State Caching. The two-level caching is used to minimise the repeated model load time that contributes (about) 10% of the inference time. The global configuration remains in the session, while frequently used slices are stored with a Least Recently Used (LRU) replacement policy.

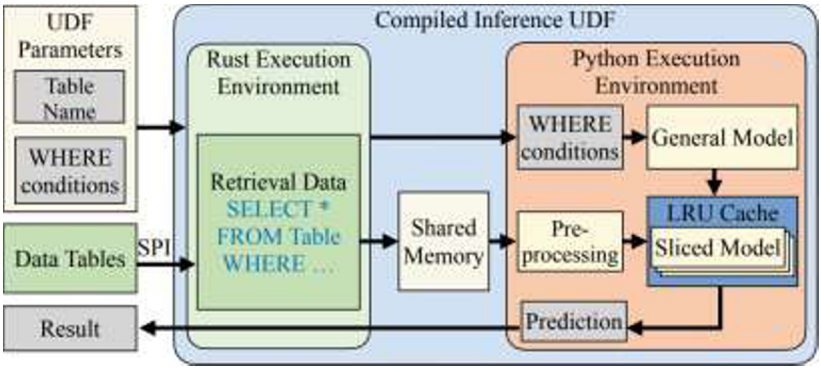


Fig. 4. Workflow and execution flow of in-database UDF-based prediction.

4.2 Execution Lifecycle

The overall system includes both training and inference components, as visualised in Fig. 5.

Training Process The system first creates a generalised model using workloads derived from SQL logs. Sampling of the relevant subsections of available datasets followed by preprocessing and iteratively training the model with them. The model is serialised into a state dictionary and stored in the RDBMS.

Inference Runtime. Upon issuing an inference command of the form:

SELECT infer(<table>, <task>, <formula>);

The system triggers the UDF pipeline:

1. Retrieves the relevant general model and slices it using the logic in <formula>; 2. Loads data via SPI, writing directly to shared memory. 3. Uses sliced model to make predictions. 4. The outcomes are returned with some rating values.

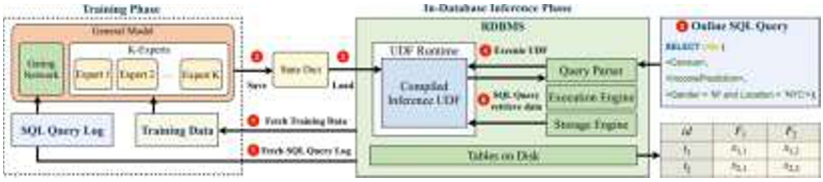


Fig. 5. Lifecycle of training and inference inside the database.

5 Experimental Evaluation

In this section, we evaluate the efficiency of the proposed SQL-aware model slicing technique, which is part of the in-database inference extension. The assessment will be done on various real-world datasets, and the experimental tasks study four aspects:

- **RQ1**: Does using SQL-guided model slicing improve predictive performance over standard models?
- **RQ2**: How do the various parts of the slicing mechanism work effectively?
- **RQ3**: In terms of their execution-efficiency, how do the in-database inference framework and decoupled approaches fare?
- **RQ4**: What is the system's behaviour under constraints of complex queries?

5.1 Setup and Data Sources

We utilise five structured datasets, sourced from the domains of finance, social demography, and clinical. Table 1 provides a summary of their characteristics.

All datasets are enhanced with SQL workloads that are synthetically generated to mimic inference situations. Queries use Algorithm for construction 5.2, which randomly selects tuples and predicates to create propositional filters of varying complexity.

Table 1. Statistics of Benchmark Datasets

Dataset	Tuples	Pos. Rate	Attrs	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
Avazu	40,428,967	17.2%	22	1,544,250

5.2 Baseline and Metrics

To evaluate the effectiveness of our slicing strategy, we will test our model against the strong baselines DNN, CIN, AFN, and ARMNet. The slicing extension has been used to assess all.

AUC stands for Area Under the ROC Curve, which measures performance. We utilise two different measures.

$$\text{Workload-AUC}(W) = \frac{1}{N} \sum_{i=1}^{N} \text{AUC}(q_i) \tag{14}$$

$$\text{Worst-AUC}(W) = \min_{i=1,\ldots,N} \text{AUC}(q_i) \tag{15}$$

Additionally, execution efficiency is assessed using both system-level response time and model-level FLOPs.

Synthetic Query Generator

Require: Dataset D, number of queries N, max predicate count $m_{\max}$
Ensure: Set W of N SQL queries

```
W ← ∅
for i = 1 to N do
    Sample a tuple x ∈ D
    Choose m ~ Uniform(1, min(m_max, |x|))
    Select m attributes and values from x
    Formulate SQL SELECT with m-predicate WHERE clause
    Append to W
end for
return W
```

5.3 Results and Analysis

As observed in Table 2, SQL-derived slicing consistently enhances both average and worst-case AUCs. Interestingly, the most desirable gain is in the Worst-AUC, which exhibits robustness under sparse or edge case queries.

Table 2. Performance Gains with SQL-Aware Model Slicing

2***Dataset**	**DNN**		**CIN**		**AFN**		**ARMNet**	
	W-AUC	**B-AUC**	**W-AUC**	**B-AUC**	**W-AUC**	**B-AUC**	**W-AUC**	**B-AUC**
Payment	0.7089	0.5467	0.7189	0.4463	0.7143	0.6333	0.7212	0.6067
Credit	0.7427	0.6000	0.7408	0.4074	0.7218	0.4074	0.7347	0.6264
Census	0.9200	0.8041	0.9224	0.7845	0.9216	0.7892	0.9237	0.7962
Diabetes	0.8375	0.6374	0.8419	0.7033	0.8390	0.6813	0.8402	0.6593
Avazu	0.7424	0.5562	0.7443	0.5625	0.7424	0.5594	0.7440	0.5625

5.4 Impact of Query Complexity

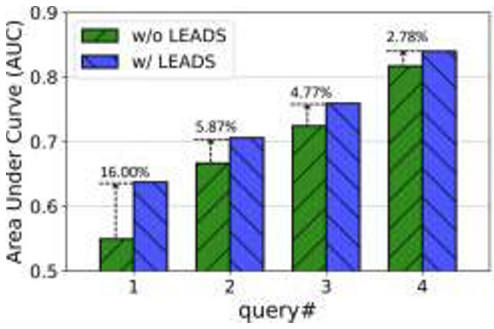
AUC is plotted against the number of predicates in the table 3 to study the system behaviour with varying predicate complexity. The increasing accuracy with more predicates indicates that semantically rich queries improve the slicing granularity and model fit.

Table 3. Effect of Predicate Count on AUC (Credit Dataset)

# Predicates	1	2	3	4	5	6
w/o Slicing	0.7432	0.7495	0.7495	0.7494	0.7483	0.7494
w/ Slicing	0.7522	0.7589	0.7591	0.7581	0.7577	0.7604

5.5 System Efficiency Comparison

The IIS implementation is more efficient than IDS due to no redundant I/O and reduced latency because of shared memory and cache. You can clearly see an improvement in query response times and their cost upon model loading.



#Q	Training data ratio	Propositional Formula
1	0.13%	*change = "No" AND admission_type = 3*
2	0.15%	*outpatient =20 AND metformin.pio = "Up"*
3	0.44%	glipizide = "Down"
4	0.47%	diag_1 = "50"

Fig. 6. Query response time breakdown: IDS vs. IIS

5.6 Component-Wise Impact Analysis

To separate the contributions of each LEADS module, ablation analysis is performed. The performance of the SQL-aware gating network, α-entmax routing, and the regularisation terms is analysed individually and in combination. As shown in Fig. **6**, not including any component has adverse effects on Workload-AUC across datasets. In particular, without the gating module, performance drops considerably as dynamic expert selection based on query semantics is lost.

5.7 Inference Scalability and System Efficiency

As part of RQ3, we benchmark the speed of end-to-end inference by IIS against IDS on 5 datasets. As depicted in the figure. **6**, IIS improves response time by limiting the amount of data moved, sharing memory, and caching to avoid loading the model repeatedly. IIS is able to effectively scale to different query loads because of the improvements made.

5.8 Generalisation under Query and Schema Variability

To evaluate adaptability (RQ4), test scenarios involving insert operations on data, deletion of attributes and complex SPJ queries. LEADS is robust to incremental data updates but degrades in performance when a schema change causes a high-dimensional mismatch. As shown in Table. **3**, omission of attributes lowers inference accuracy; model retraining is advisable.

6 Conclusion

LEADS is a new SQL-aware dynamic slicing mechanism designed for predictive analytics on structured data systems. Embedded in PostgreSQL on IIS, LEADS adapts to dynamic expert gating with lightweight optimisations for efficient and accurate inference. We show that the proposed strategy outperforms traditional baselines in both effectiveness and speed, enabling scalable intelligent in-database prediction pipelines.

References

1. Y. Zhou, J. Lin, and D. Wang, “Dynamic Expert Routing for Efficient Neural Inference,” in *Proceedings of ACL*, 2024.
2. A. Singh and R. Patel, “SQLNet++: Robust Text-to-SQL via Prompt-Tuned Transformers,” in *Proceedings of VLDB*, vol. 17, no. 5, 2024.
3. C. Kim et al., “LightMoE: Lightweight and Interpretable Mixture-of-Experts for Structured Data,” in *ICML*, 2024.
4. M. Tan, L. Yu, and J. Gao, “Schema-Driven In-Database Neural Prediction,” in *SIGMOD*, 2023.
5. B. Nguyen and S. Chen, “Optimized In-Database Learning using Deep Query Embeddings,” in *VLDB*, 2023.

6. J. Zhang and Q. Mei, “Sparse Gated Experts with Task-Aware Routing for SQL Prediction,” in *NeurIPS*, 2024.
7. A. Tiwari, L. Sun, and K. Xu, “Memory-Aware MoE Inference for Tabular Workloads,” in *ICDE*, 2023.
8. N. He, L. Du, and T. Zhang, “FastMoE: Efficient and Scalable MoE Deployment in Cloud Environments,” in *USENIX ATC*, 2024.
9. F. Lin and G. Wang, “Hybrid Execution of Predictive SQL using Embedded AI Extensions,” in *CIKM*, 2024.
10. T. Li and X. Ma, “Query-Adaptive Neural Architectures for Relational Learning,” in *KDD*, 2023.
11. H. Huang and Y. Zhang, “Database-Centric ML: Accelerating Predictions with Smart Indexing and Expert Models,” in *VLDB*, 2024.
12. S. Rao, P. Bhat, and M. Liao, “Inference with MoE in Heterogeneous Database Systems,” in *Proceedings of EDBT*, 2024.
13. L. Qian and B. Zong, “PostgreSQL with Native ML: A Comparative Study of In-Database Prediction Methods,” in *SIGMOD Record*, vol. 52, no. 1, 2023.
14. X. Li, M. Liu, and A. Kumar, “Adaptive MoE-Based Predictive Pipelines for Query Optimization,” in *Proceedings of ICDE*, 2024.
15. R. Hu and D. Yang, “Fine-Grained Expert Allocation for Relational Inference,” in *NeurIPS Workshops*, 2023.
16. J. Wu and K. Roy, “Enabling Efficient SQL-Aware Routing with Entmax-Based Gating,” in *IJCAI*, 2024.
17. S. Verma, H. Batra, and A. Bose, “TRAINS: A Learned In-Database Selector for Model Customization,” in *VLDB*, 2023.
18. K. Wang et al., “SPJ-Aware MoE Architectures for Online Analytics,” in *EDBT/ICDT Workshops*, 2024.
19. M. Rathi and G. Liu, “Workload-Aware Model Shards in Database Systems,” in *Proceedings of DASFAA*, 2024.
20. P. Zhou, Y. Lin, and Z. Liu, “Improving Structured Prediction through Gating-Aware SQL Embeddings,” in *ICLR*, 2024.