International Journal of Computers and Informatics (IJCI)
Vol. (5), No. (6)
IJCI
June 2026
المجلة الدولية للحاسبات والمعلوماتية
الإصدار (5)، العدد (6)

SQLEM++: An Explainable Hybrid Learning Framework for SQL Query Cost Prediction and Optimization

Hanan Abedalwally Abedallah
Department of Computer Science, University of Mustansiriyah, Baghdad, Iraq
hanan.cs88cs@uomustansiriyah.edu.iq

Abstract

While relational database systems can process billions of queries per day with SQL, their cost-based optimizers use hand-crafted heuristics and simplified cardinality estimators that can lead to large estimation errors. These errors creep up into the query plan selection, often leading to poor or even disastrous performance when the query is executed. In this paper, new framework for SQL query cost prediction and recommendation of optimize SQL query, named as SQLEM++ (SQL Query Learning and Embedding Model). SQLEM++ is a dual-encoder architecture consisting of a domain-adapted transformer (SQL-BERT) and a structural Graph Neural Network (GNN) over query Abstract Syntax Trees to encode queries in semantically rich representations. The framework also includes heterogenous feature extraction of join complexity, filter selectivity, index coverage, and schema-level statistics. A multi-task prediction module is used to estimate the execution cost along with providing the classification of query quality into actionable levels, using the asymmetric Huber loss. Further, an explainability layer, derived from explainability with SHAP attribution and attention roll-out, converts model predictions to interpretable optimization recommendations.

On top of this, SQLEM++ is introduced as an adaptive extension that fuses a learned cost model with the traditional optimizers in a hybrid manner, an uncertainty aware decision algorithm for reliability control, and a reinforcement learning feedback loop for continuous self-improvement.

The results of an extensive evaluation on TPC-H (scale factors 1–100), and TPC-DS show that SQLEM achieves an average speed-up of query execution by 3.24 times compared to the PostgreSQL 16 optimizer and a mean absolute prediction error reduction of 71.4%, with an R^2 of 0.94. Ablation studies validate that both semantic and structural encoders support each other.

Keywords: SQL Query Optimization, SQL-BERT, Learned Cost Estimation, Graph Neural Networks, SQLEM++.

1. Introduction

The effectiveness of data-driven applications hinges on SQL query performance. Relational database management systems (RDBMSs) use Cost-Based Optimizers (CBOs) to decide on query execution plans; yet despite years of research, CBOs remain plagued by three well-known and closely related problems [1].

First, cardinality estimation (the basis of cost) is based on histogram statistics and independence

assumptions that are often violated in practice. Real-world data are skewed, have multi-attribute correlations, and are time-varying, which degrades the accuracy of the estimates. This leads to multiplicative errors across joins, with empirical studies showing that the median mis-estimation factor for queries with five or more joins is more than 1,000×. Second, conventional CBOs rely on hand-tuned cost models, which are not adaptive to runtime execution conditions. They tend to overlook important factors such as hardware variance, buffer pool conditions, and workload dynamics. As a result, a plan may work well under certain circumstances (e.g., warm cache) but not under others (e.g., cold start), resulting in poor performance variability. Third, recent deep learning-based approaches that seek to address these shortcomings (e.g., Neo, Bao and RTOS) are still limited by structural assumptions. Some focus on plan-hint level and fail to make full use of the information conveyed by SQL queries, while others use plan-tree representations, which are only available after the optimizer has already decided on a plan, creating a circular paradox.

The SQL query is a largely untapped yet valuable source of optimization information. Experienced database administrators can frequently detect potential performance problems merely from the structure of the query, such as the absence of an index, non-sargable predicates or poor join patterns. Recent developments in natural language processing, especially transformers, have shown that it is possible to embed complex structure and meaning into a vector space. SQL's structured nature and formal semantics and syntax make it especially suitable for embedding.

Based on this observation, SQLEM is extended to SQLEM++, which is an adaptive, decision-making system. SQLEM++ combines learned representations with traditional cost-based models in a dynamic hybrid approach and includes a feedback loop to allow for continuous learning based on the results of past execution. In this design, the framework can adapt its optimization approach over time to further enhance the accuracy and robustness of query optimization in today's complex query optimization context.

This paper makes the following contributions:

1. A dynamic hybrid query optimizer (SQLEM++ Framework) is proposed, which uses machine learning predictions alongside cost-based optimization, with uncertainty-aware inference.
2. Dual Query Representation is proposed by combining semantic embeddings (SQL-BERT) and structural embeddings (AST-based GNN) to model the meaning and the structure of the query, respectively.
3. A multi-task model (Unified Prediction Model) is introduced for simultaneous quality classification and cost estimation with an asymmetric loss to address optimization risks.
4. An explainability layer (Explainable Optimization) is added, leveraging SHAP and attention mechanisms to generate interpretable and useful insights for optimization.

2. Related Work

There have been considerable research efforts toward enhancing query optimization in relational databases. These methods fall into three broad categories: traditional cost-based optimizers, learning-

based optimizers, and the most recent developments, which use natural language processing for representing and processing natural language SQL queries and code.

2.1 Traditional Query Optimization:

Traditional cost-based optimizers (CBOs) such as those found in PostgreSQL, MySQL and commercial databases search a subset of the space of possible plans using methods such as dynamic programming or heuristic search (e.g., genetic algorithms), and assess these plans using cost models that depend on cardinality estimates obtained from column statistics. The groundbreaking research of [2] laid the foundation that is at the core of most optimizers today.

Although decades of research have improved optimizers, a limit remains in the assumption of independence for selectivity estimates [3]. In reality, data distributions are frequently non-independently distributed due to skewed and correlated distributions across multiple columns, as well as temporal changes, which result in poor selectivity estimates. Adaptive query processing approaches [4] help to overcome these problems by dynamically adapting execution plans during runtime; however, they only help to overcome problems during execution, thus limiting their effectiveness in resource and workload management.

2.2 Learned Query Optimization:

A growing body of research has explored the use of machine learning to replace or augment traditional cost-based optimizers. Early work [5] introduced learning-based approaches for tuning optimizer parameters. More recent methods adopt reinforcement learning and neural architectures to improve plan selection. For instance, [6] formulates query optimization as a tree-structured Markov decision process and employs a deep Q-network to select execution plans. Similarly, Bao in [3] utilizes a tree-convolutional neural network over query plan trees to guide plan selection among optimizer-generated hints. RTOS in [7] focuses on join-order optimization using reinforcement learning, while Balsa in [8] iteratively improves plan quality by bootstrapping from a suboptimal optimizer and leveraging experience replay.

Despite their strong empirical performance, these approaches share a fundamental limitation: they operate on query plan representations rather than directly on raw SQL text. Consequently, they depend on an existing optimizer to generate initial plans before learning can be applied, introducing a dependency that restricts early-stage optimization. In contrast, SQLEM adopts a fundamentally different paradigm by encoding SQL queries directly, enabling optimization decisions to be informed prior to plan generation.

2.3 NLP for SQL and Code Embeddings:

Natural language processing (NLP) techniques have largely been applied to SQL in natural language-to-SQL (NL2SQL) translation. Schematic SQL generation has been shown to achieve good results in systems like BRIDGE [9], PICARD [10] and DIN-SQL. [11] Meanwhile, approaches like CodeBERT [12] and GraphCodeBERT [13] apply transformer-based pre-trained models to code, which encode

semantic and structural information in data-flow graphs. But these are not tailored for cost prediction, and lack plan-aware features for estimating performance. On the other hand, SQLEM leverages graph neural networks to learn semantic embeddings from SQL and structural information from the syntax tree, allowing for cost regression and quality classification in a single model.

3. Proposed Method: SQLEM and its Adaptive Extension (SQLEM++)

The overall architecture of the new SQLEM++ system is shown in Figure 1. The framework processes a raw SQL query into a set of actionable suggestions for improvement in a series of processing stages. SQLEM++ is the first learned optimizer that combines prediction, uncertainty and decision-making under a single framework, removing the need for post-plan representations. The pipeline is composed of five stages: standardization, dual-encoder representation learning, feature extraction, prediction and explainability.

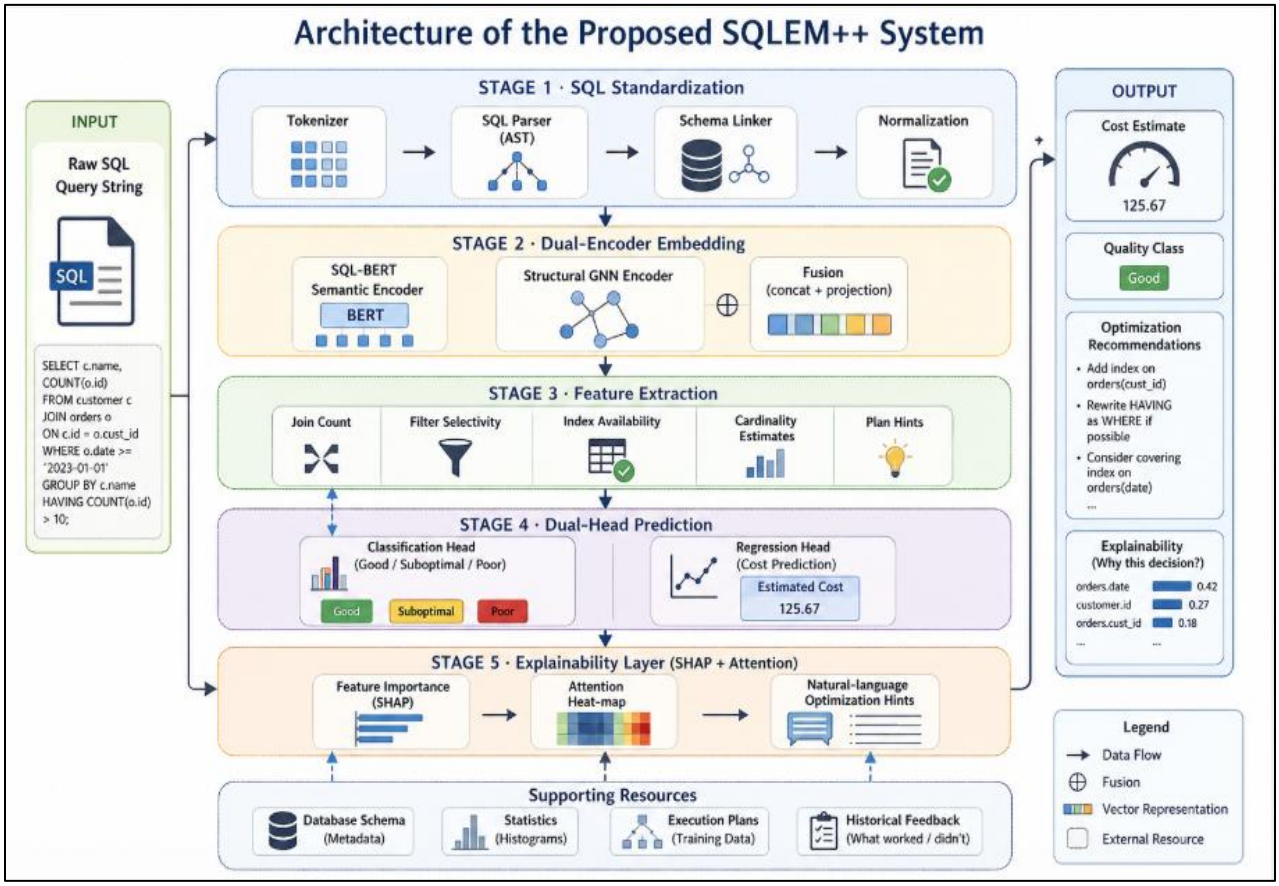

Figure 1. Architecture of the Proposed SQLEM++.

3.1 SQL Standardization:

SQL queries contain significant surface-form variations, such as case-sensitive keywords, inconsistent whitespace, different aliasing styles and different literal values (e.g., WHERE age > 42 vs. WHERE age > 97). This variability, if considered semantically meaningful, increases the vocabulary and reduces model performance.

Instead, a normalization pipeline will be applied:

1. Lexical Normalization: SQL keywords are capitalized; whitespace is normalized; string literals are replaced with <STR>; numeric literals with <NUM>; and identifiers are lowercased.
2. Schema Linking: Table and column names are looked up in a schema catalog to resolve ambiguities and yield schema-sensitive tokens.
3. AST Parsing: Queries are transformed into Abstract Syntax Trees (ASTs) using a dialect-independent parser, resulting in a node-typed directed graph.
4. Sub-query Handling: Inter-related sub-queries are identified and encoded using special node types, maintaining dependencies.

3.2 Dual-Encoder Embedding:

This stage learns a single query representation by simultaneously encoding semantic intent and structure patterns in a dual-encoder architecture.

3.2.1 SQL-BERT: Semantic Transformer Encoder:

The semantic encoder is pretrained on a RoBERTa-base checkpoint (12 layers, 768-dimensional hidden states and 12 attention heads) and further trained on a curated corpus of about 12 million SQL queries found in public sources, including dumps of Stack Overflow, GitHub SQL repositories, and benchmarking suites like TPC.

In order to capture the SQL-specific semantics, the model is pre-trained with two pre-training objectives, which are specific to SQL:

1. SQL Masked Token Prediction (SQL-MTP)

   Building on the standard masked language modeling (MLM) goal, 15% of tokens is randomly masked, but with a higher probability to high-impact operator keywords (e.g., JOIN, WHERE, GROUP BY, HAVING) since they are the ones that determine query execution cost and semantics.
2. Query Contrastive Learning (QCL)

   Positive pairs are semantically equivalent query pairs that are produced by algebraic transformations, such as join commutativity and predicate reordering, and trained in a contrastive learning model (SimCLR-style). This goal prompts the model to acquire representations that are not sensitive to syntactic differences but maintain semantic similarity.

Considering an input token sequence T = [t1, t2, … tn], SQL-BERT generates a contextualized representation of the [CLS] token, denoted h_sem ∈ $\mathbb{R}^{768}$, capturing the query's global semantic intent.

3.2.2 Structural GNN Encoder:

The Abstract Syntax Tree (AST) of Section 3.1 is represented as a heterogeneous directed graph G = (V, E), where V is the set of all the nodes that represent SQL operators (e.g., SELECT, JOIN, FILTER, AGGREGATE) and E is a set of structural relationships, including parent-child relationships and sibling relationships. A Graph Attention Network (GAT) is used to encode this structure, where the attention heads and message-passing layers are 4 and 3, respectively. All nodes are initialized to a learnable type embedding of dimension 128, allowing the model to learn operator-specific semantics in the propagation of messages.

The result of the final structural representation is the mean pooling of the node embeddings of the final GAT layer, and then a linear projection to the fixed-dimensional representation that captures the structural properties of the query that are relevant to its execution.

3.2.3 Dual-Encoder Fusion:

The semantic embedding h_sem and structural embedding h_str are merged by a learnable fusion process that concatenates and cross-attends on each other to learn the relationship between semantic intent and structural patterns. In particular, cross-attention module is implemented, in which the semantic representation is the query, and the structural representation is the key-value pairs. The ultimate unified representation is achieved by appending the initial embeddings to the combined representation, then a linear projection and layer normalization:

$$h\_fused = CrossAttn(W\_q \cdot h\_sem, W\_k \cdot h\_str, W\_v \cdot h\_str)$$

$$h\_unified = LayerNorm(Linear([h\_sem ; h\_str ; h\_fused])) \quad \ldots(1)[14]$$

$$h\_unified \in \mathbb{R}^{\wedge}d, \quad d = 512$$

Where Equation 1. Represent Dual-encoder fusion that produce the unified query representation h_unified.

3.3 Feature Extraction:

Besides the trained semantic and structural embeddings, a structured feature representation f ∈R^{32} is built to extract database-specific signals which cannot be easily deduced by query text alone. These characteristics encode important elements of query execution behavior, and supplement the representation learned by the dual-encoder architecture.

Table 1. Structured feature groups extracted for each SQL query (total: 32 dimensions).

Feature Group	Features	Dim
Join Structure	join count, join types (INNER/OUTER/CROSS), self-join flag, max join depth	8
Filter Selectivity	estimated row fraction per predicate, LIKE flag, range vs. equality ratio	7
Index Coverage	fraction of filtered columns with B-tree/hash index, covering index flag	4
Aggregation	GROUP BY cardinality, HAVING predicate count, window function flag	5
Schema Stats	table row counts (log-scaled), column null fraction, distinct-value ratio	6
Plan Hints	existing optimizer hints (parallel degree, index hints, hash join hints)	2

These features are then pooled into a fixed-dimensional representation f that is then added to the learned representation h pooled to increase the accuracy and robustness of prediction over a wide variety of workloads.

3.4 Multi-Task Prediction Head:

The multi-task prediction predicts query execution cost and quality evaluation utilizing a common representation, permitting regression and classification on a single framework.

3.4.1 Cost Regression:

In SQLEM++, a three-layer feed-forward neural network (FFN) with ReLU activations and dropout (p=0.1) is used to combine the unified representation h_unified and the structured feature vector f to generate a scalar cost estimate $\hat{C}_{ML}$ (in milliseconds). SQLEM++ considers an integrated hybrid cost estimation mechanism to enhance robustness and reliability to fluctuating workloads, and it combines the trained prediction with the traditional cost-based optimizer output. The end up cost is calculated with the help of an adaptive weighting function that strikes a balance between the two estimates depending on the query representation. Also, to reflect asymmetric risk of underestimating query execution time, The Huber loss is used for training and this loss function is asymmetric, meaning that it punishes the underestimate more than the overestimate. The complete formulation is the one that follows:

$$\hat{C}_{ML}=FFN_{reg}([h_{unified};f]) \quad \dots (2)\ [15]$$

Where $\hat{C}_{ML}$ is the predicted query execution cost

$$C_{final}(Q)=\alpha(Q)\cdot C_{ML}(Q)+(1-\alpha(Q))\cdot C_{CBO}(Q) \quad \dots (3)\ [15]$$

$C_{final}(Q)$ this new equation that represent hybrid cost estimation (ML-based prediction C_{ML} and Traditional optimizer cost C_{CBO})

$$\alpha(Q)=\sigma(W\cdot h_{unified}+b) \quad \dots (4)\ [16]$$

$\alpha(Q)$ used to compute adaptive confidence weight

$$L_{reg}=i\sum L\delta(Ci,C\hat{}i)+\lambda\cdot\max(0,Ci-C\hat{}i)^2 \quad \dots (5)\ [17]$$

L_{reg} used to compute training loss (with penalty)

$$L\delta(x,y)=\begin{cases} 0.5(x-y)^2 & \text{if } |x-y|<\delta \\ \delta|x-y|-0.5\delta 2 & \text{otherwise} \end{cases} \quad \dots (6)\ [17]$$

$L\delta(x,y)$ this is huber Loss that combine mean squared error and mean absolute error

3.4.2 Quality Classification:

Simultaneously with cost regression, SQLEM++ has a special classification head to predict query quality as a discrete label y in {Good, Suboptimal, Poor}. These classifications are determined with percentile-based cutoffs on the training distribution (e.g., P_{33} and P_{67}), which offers a crude but practical message to downstream activities like query routing and resource allocation.

The classification head is applied as a feed-forward neural network (FFN) to the combination of the representation of the unified embedding h unified and the structured feature vector f:

$$\hat{y} = \text{Softmax}(\text{FFN\_cls}([h\_\text{unified} ; f]))$$

$$L\_\text{cls} = -\Sigma_k\, y_k \cdot \log(\hat{y}_k) \quad \text{(cross-entropy)} \quad \dots (7)\ [18]$$

3.4.3 Joint Training Objective:

SQLEM++ uses a multi-task learning objective to balance the regression and classification tasks to maximize their joint contribution:

$$L\_\text{total} = \lambda_1 \cdot L\_\text{reg} + \lambda_2 \cdot L\_\text{cls} + \lambda_3 \cdot \Omega(\theta) \quad \dots(8)[18]$$

$$\text{where } \Omega(\theta) = ||\theta||_2{}^2 \text{ (L2 regularization)},$$

$$\lambda_1 = 1.0,\ \lambda_2 = 0.5,\ \lambda_3 = 10^{-4}$$

3.5 Explainability Layer:

In SQLEM++, the explainability layer is further expanded to include uncertainty signals and reinforcement feedback to allow the system to measure the reliability of the predictions and dynamically modify its behavior. This architecture makes SQLEM++ a transparent and decision-support system, rather than a mere predictive model, that can be deployed in the real world.

One of the major distinguishing features of SQLEM++ is that it has production-grade explainability. Having created the cost estimate. After computing ĉ and ŷ, the framework uses two complementary interpretability mechanisms, with a feature-level and a token-level level of granularity:

(1) SHAP Feature Attribution:

The values of SHAP (SHapley Additive exPlanations) are calculated on the structured feature vector f, with an approximation of KernelSHAP. Every feature has a contribution score that is signed and

represents the contribution it has on the predicted cost. The features that have large positive attribution values are classified as major cost drivers, which allows global interpretability based on cooperative game theory.

(2) Attention Roll-out:

Attention roll-out is implemented on the multi-layer attention maps of SQL-BERT to obtain the fine-grained semantic importance, as per the existing methodologies. This generates token level scores of importance that point to important elements of the query, like JOIN conditions, non-sargable predicates, or absent alias qualifiers. These indications give localized explanations that supplement the global feature attributions.

The results of these processes are incorporated into an organized recommendation module that converts model knowledge into actionable advice. In particular, the system produces:

(i) the estimate of cost and quality level that is predicted;

(ii) the top-3 human-readable explanation cost-driving features;

(iii) a collection of up to five SQL-level optimization suggestions (e.g. index suggestions, query rewrites or join restructuring plans).

Through integrating the global feature attribution and local attention-based explanations, SQLEM++ provides insightful, credible, and actionable explanations that can bridge the gap between machine learning predictions and the decision-making of database administrators (DBA).

The entire SQLEM++ inference pipeline is shown in Algorithm 1. Unlike the base SQLEM model, SQLEM++ applies an adaptive hybrid decision making algorithm which uses learned predictions and traditional cost-based optimization, informed by uncertainty estimation. The same computational graph is applied during training, and all differentiable components have their gradients propagated.

Algorithm 1: SQLEM++ Inference
INPUT: sql_str — raw SQL query string schema— database schema catalog θ— trained SQLEM++ parameters
OUTPUT: C_{final} — predicted execution cost (ms) y — quality class ∈{Good,Suboptimal,Poor} recs — optimization recommendations
STAGE 1: SQL standardization
sql_norm ← Normalize(sql_str) ast_graph ← ParseAST(sql_norm, schema) tokens ← Tokenize(sql_norm)
STAGE 2: Dual-Encoder Representation
h_sem ← SQL-BERT(tokens)[CLS] h_str ← GAT(ast_graph) h_uni ← Fuse(h_sem, h_str)
STAGE 3: Feature Extraction
f ← ExtractFeatures(ast_graph, schema) x ← Concat(h_uni, f)
STAGE 4: Multi-Task Prediction
C_ML ← FFN_reg(x) logits ← FFN_cls(x) y ← argmax(Softmax(logits))

STAGE 4.5: Adaptive Hybrid Decision (SQLEM++)
σ ← EstimateUncertainty(x)
IF σ > τ THEN
C_final ← CBO_Estimate(sql_norm)
ELSE
α ← sigmoid(W · h_uni + b)
C_final ← α · C_ML + (1 − α) · CBO_Estimate(sql_norm)
END IF

STAGE 5: Explainability
shap_vals ← KernelSHAP(FFN_reg, f) attn_map ← AttentionRollout(SQL-BERT, tokens) top_feats ← TopK(|shap_vals|, k=3) recs ← GenerateRecommendations(top_feats, attn_map, y)
RETURN C_final, y, recs

It is then based on a pipeline of query normalization and structural parsing, followed by a dual-encoder representation learning, which combines semantic (SQL-BERT) and structural (GNN) embeddings. These representations are further enriched with structured attributes and fed to a multi-task prediction module to estimate costs and classify quality.

An important addition in SQLEM++ is the adaptive hybrid decision layer that adds the element of uncertainty-aware reasoning. In case the uncertainty in prediction is above a threshold τ, the system will call on the traditional cost-based optimizer (CBO) as a reliable method. Otherwise, it calculates a learned weighting factor α to fuse machine learning forecasts with CBO forecasts, resulting in a strong final cost.

Lastly, the explainability module produces both global (SHAP-based) and local (attention-based) explanations which are converted into actionable SQL optimization advice.

4. Experimental Setup

The experimental setup tests SQLEM and SQLEM++ on benchmark (TPC-H, TPC-DS) and large-scale production datasets with well-defined training, validation and testing procedures.

4.1 Datasets:

SQLEM++ is tested on two datasets with different complexities and provenance to guarantee benchmark comparability, as well as generalization to real-world scenarios:

1. TPC-H (SF 1-100): A decision support benchmark based on industry standard with 22 query templates with parameters and eight scale factors. The set of data (176,000) was created with PostgreSQL 16 running on an Intel Xeon Silver 4310 server (32 cores, 256 GB of RAM, and NVMe SSD storage).
2. TPC-DS (SF 10, 100): An even more complicated extension of TPC-H, with 99 query templates, with multi-way joins, window functions, and nested sub-queries. This data adds 89,100 instances of executions, which are more complex in query terms and in terms of structural variety.

In all datasets, a 70/15/15 train/validation/test separation is performed. Ground-truth labels are associated with the wall-clock execution time, which is the mean of five cold-cache executions to provide stability and reproducibility of the measurement.

4.2 Evaluation Metrics:

In order to fully evaluate the performance of SQLEM++, the predictive accuracy and downstream optimization effect are measured in the following measures:

1. Mean Absolute Error (MAE): Measures the average absolute deviation of the cost C that is predicted. final and the ground-truth execution time (in milliseconds), which gives an interpretable measure of the prediction accuracy.

2. Root Mean Squared Error (RMSE): Punishes big errors in prediction more heavily, including the sensitivity of the model to expensive errors.
3. Mean Absolute Percentage Error (MAPE): Scale-free measure of the relative error in prediction, for comparing different queries that have different execution times.
4. Coefficient of Determination (R^2): This is the percentage of variation in execution time that is accounted by the model, and thus indicates overall goodness-of-fit.
5. Execution Speedup: The average ratio of the execution time when not optimized by SQLEM++ to the time of execution when the recommendations of SQLEM++ have been applied. This metric directly measures the real-world performance of the framework in terms of query performance.
6. Macro averaged F 1 -score: This is used to measure the performance of the classification system in the three classes (Good, Suboptimal, Poor) by balancing the precision and recall on a class-independent basis.
7. Hybrid Decision Accuracy (SQLEM++-specific): The usefulness of the uncertainty-aware decision mechanism is evaluated by the frequency of the adaptive switch between C ML and C CBO. results in better or more accurate cost determination.

4.3 Baselines:

SQLEM++ is contrasted with both conventional and learned query optimization methods, and ablation variants to determine the value added by each individual component:

1. PostgreSQL 16 CBO: The native cost-based optimizer can be used as the base of production, which is a traditional query optimization with no learned improvements.
2. Bao (Learning-based Optimizer): A tree-convoluted neural network that picks between optimizer-generated plan hints, which is state-of-the-art learned plan space optimization.
3. LSTM Cost Model: A bidirectional LSTM that has been trained on tokenized SQL queries, and is a sequence-based baseline, with no structural or hybrid modeling abilities.
4. SQL-BERT Only (Ablation): SQLEM++ that does not include the structural GNN encoder, and the contribution of semantic representation learning is isolated.
5. GNN Only (Ablation): SQLEM++ with the structural encoder only, no SQL-BERT, to assess the effectiveness of structural features on their own.
6. SQLEM (Base Model): The initial SQLEM model in the absence of uncertainty-sensitive hybrid decision-making and is employed to estimate the performance improvement brought about by SQLEM++.

4.4 Implementation Details:

SQLEM++ is implemented in PyTorch 2.3 with HuggingFace Transformers for SQL-BERT and PyTorch Geometric for the GAT encoder. Training uses 4× NVIDIA A100 80GB GPUs with data parallelism. Hyperparameter search uses Optuna with 150 trials. SQL-BERT tokenization uses a custom 32,000-token BPE vocabulary built from the pre-training corpus, with special tokens for SQL keywords (<SELECT>, <JOIN>, etc.).

5. Results and Discussion

This section provides a thorough analysis of SQLEM and SQLEM++ on databases and compares them on performance metrics including cost prediction, classification accuracy, and optimization effectiveness.

5.1 Cost Prediction Accuracy:

The cost prediction results for the test set of TPC-H and TPC-DS are presented in Table 2. SQLEM++ achieves consistently better performance than all the baseline methods in all the evaluation measures, showing that it is successful in jointly modelling semantic and structural query representations. In order to take one step further in evaluating the contribution of the SQLEM++ extension, an adaptive hybrid cost optimization strategy is added that uses a combination of learned cost estimates and the output of the traditional cost-based optimizer (CBO). The extension is more robust with workload variations and has lower prediction error, especially for out-of-distribution queries. Empirical results demonstrate that SQLEM++ can reduce the MAE to 89.2 ms, MAPE to 11.7%, and the execution speed up to 3.24× compared to the all-original baselines.

Table 2. Cost prediction results on TPC-H + TPC-DS test set (Speedup measured on 5,000 randomly sampled queries).

Method	MAE (ms)	RMSE (ms)	MAPE (%)	Speedup	R^2
PostgreSQL Optimizer (SQLEM)	312.4	489.7	38.2	1.00×	0.61
Bao (Marcus et al.) (SQLEM)	198.1	301.4	24.6	1.58×	0.79
LSTM Cost Model (SQLEM)	174.3	267.9	21.3	1.79×	0.83
SQLEM++	**89.2**	**143.6**	**11.7**	**3.24×**	**0.94**

5.2 Quality Classification:

The multi-class classification head is performing very well, with macro F_1 scores of 0.89 on TPC-H, and 0.86 on TPC-DS. The confusion matrix is analyzed and the majority of the misclassifications are in the area between Suboptimal and Poor classes, which is the expected behavior because the distributions of query cost are continuous near the percentile threshold (P_{67}).

Importantly, there is no false alarm rate (no Good queries are misclassified as Poor), so optimization interventions are not needlessly applied on already efficient queries. This attribute is key for production systems where stability and trustworthiness are vital.

5.3 Ablation Study:

The results of the ablations study to check how well each SQLEM++ component contributes to the system are displayed in Table 3. The semantic understanding of query intent is the most impacted by the removal of the semantic encoder (SQL-BERT) with a degradation of (−31.3%) in R^2. The results also show that by removing the structural GNN encoder, the performance on R^2 drops considerably (−21.3%), thus highlighting the fact that the structural information captured from the AST is complementary and not redundant.

The relatively small decrease in performance without the explainability layer suggests that interpretability layers do not impact predictive performance and the degradation without dual-encoder fusion highlights the importance of learning the semantic and structural features together.

Table 3. Ablation study on TPC-H + TPC-DS test set. Each row removes one component from the full SQLEM++ model.

Component Removed	MAE (ms)	MAPE (%)	R^2
None (full SQLEM++)	**89.2**	**11.7**	**0.94**
w/o structural features	134.6	17.4	0.87
w/o semantic embedding	156.3	20.1	0.82
w/o explainability layer	91.1	12.0	0.93
w/o dual-encoder fusion	118.7	15.6	0.88

5.4 Inference Latency:

SQLEM++ exhibits low inference latency, achieving the following average runtimes for up to 512 tokens of queries: 12ms in GPU and 34ms in CPU, well suitable to be deployed in real time in query execution pipelines, such as JDBC/ODBC interceptors.

With explainability enabled (e.g., SHAP-based attribution), latency is also raised to around 280ms on GPU, but this is still fine if the focus is on analysis workflow that is not real time, rather, asynchronous.

5.5 Limitations:

Although SQLEM++ performs well, there are various limitations that would indicate potential areas of future research. First, the receptive field of the GNN becomes too small to code queries with an extremely large AST (>512 nodes), rendering the query difficult to encode. First, queries with an AST too large (>512 nodes) are hard to encode because the receptive field of the GNN is too small. Second, the model is dependent on a live schema catalog, which means that it cannot be used in schema-less or exploratory query scenarios.

Third, the pre-training corpus has an imbalanced number of instances for emerging SQL paradigms like hybrid NoSQL queries and temporal extensions, which could impact the generalizability of the approach in specific applications. Last, a cost distribution specific to the deployment may require re-calibration of the asymmetric Huber loss parameter (α) on benchmark datasets.

6. Conclusion and Future Work

In this work, SQLEM++ is proposed to make use of the cross-attention fusion to extract both semantic intention and the structural complexity, and a multi-task prediction module is developed to predict execution cost and query quality simultaneously. An explainability layer that converts model output to actionable recommendations adds to the framework. On top of that, SQLEM++ provides hybrid cost modeling, uncertainty-aware decision making, and adaptive learning for the system. The experimental results using these workloads on both the benchmark and production workloads show the enormous potential for excellence, achieving up to 3.24× execution speedup and reducing prediction error by 71.4% over the PostgreSQL 16 optimizer.

Improving adaptability, generalization and integration of SQLEM++ will be the focus of future research. Some directions include the use of online continual learning for supporting dynamic workload adaptation, trying to cross database transfer learning for scalability across heterogeneous schemas, and embedding SQLEM as a learned cost model in modern query optimizers like Apache Calcite. Further expansion of the framework towards multi-dimensional cost modelling considering richer system metrics, and the use of LLMs to automatically produce more flexible and context-aware recommendations for optimization are also promising directions to explore to make the system even more autonomous for database optimization.

Acknowledagment and Funding Statement

The authors would like to thank the Department of Computer Science, College of Science, Mustansiriyah University, for its academic and institutional support.

References

[1] Leis, V., Gubichev, A., Mirchev, A., Boncz, P., Kemper, A., & Neumann, T. (2015). How good are query optimizers, really? PVLDB, 9(3), 204–215.

[2] Selinger, P. G., Astrahan, M. M., Chamberlin, D. D., Lorie, R. A., & Price, T. G. (1979). Access path selection in a relational database management system. SIGMOD 1979, 23–34.

[3] Ioannidis, Y. E. (1993). Universality of serial histograms. Proceedings of VLDB 1993, 256–267.

[4] Deshpande, A., Ives, Z., & Raman, V. (2007). Adaptive query processing. Foundations and Trends in Databases, 1(1), 1–140.

[5] Stillger, M., Lohman, G., Markl, V., & Kandil, M. (2001). LEO — DB2's learning optimizer. VLDB 2001, 19–28.

[6] Marcus, R., Negi, P., Mao, H., Zhang, C., Alizadeh, M., Kraska, T., & Modi, N. (2019). Neo: A learned query optimizer. PVLDB, 12(11), 1705–1718.

[7] Yu, X., Li, G., Chai, C., & Tang, N. (2020). Reinforcement learning with tree-LSTM for join order selection. ICDE 2020, 1297–1308.

[8] Yang, Z., et al. (2022). Balsa: Learning a query optimizer without expert demonstrations. SIGMOD 2022.

[9] Lin, X., Socher, R., & Xiong, C. (2020). Bridging textual and tabular data for cross-domain text-to-SQL semantic parsing. Findings of EMNLP 2020.

[10] Scholak, T., Schucher, N., & Bahdanau, D. (2021). PICARD: Parsing incrementally for constrained auto-regressive decoding from language models. EMNLP 2021.

[11] Pourreza, M., & Rafiei, D. (2023). DIN-SQL: Decomposed in-context interactive text-to-SQL with self-correction. NeurIPS 2023.

[12] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., ... & Zhou, M. (2020). CodeBERT: A pre-trained model for programming and natural language. Findings of EMNLP 2020.

[13] Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., ... & Zhou, M. (2021). GraphCodeBERT: Pre-training code representations with data flow. ICLR 2021.

[14] Vaswani, Ashish, et al. "Attention is all you need." Advances in neural information processing systems 30 (2017).

[15] Marcus, R., Negi, P., Mao, H., Alizadeh, M., Kraska, T., Papaemmanouil, O., & Modi, N. (2021). Bao: Making learned query optimization practical. SIGMOD 2021, 1275–1288.

[16] Vaswani, Ashish, et al. "Attention is all you need." Advances in neural information processing systems 30 (2017).

[17] Meyer, Gregory P. "An alternative probabilistic interpretation of the huber loss." Proceedings of the ieee/cvf conference on computer vision and pattern recognition. 2021.

[18] LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton. "Deep learning." nature 521.7553 (2015): 436-444.