BOHR International Journal of Internet of things, Artificial Intelligence and Machine Learning
2026, Vol. 5, No. 1, pp. 6–18
DOI: 10.54646/bijiam.2026.34
www.bohrpub.com

RESEARCH

LLM-augmented query optimization: a hybrid framework for intelligent SQL performance tuning

Hanan Abed Alwally Abed Allah*†

Department of Computer Science, University of Mustansiriyah, Baghdad, Iraq

***Correspondence:**
Hanan Abed Alwally Abed Allah,
hanan.cs88cs@uomustansiriyah.edu.iq

†ORCID:
Hanan Abed Alwally Abed Allah
0000-0002-2943-369X

Received: 11 June 2026; **Accepted:** 23 June 2026; **Published:** 02 July 2026

Query optimization remains a critical challenge in relational database management systems (RDBMSs). Traditional Cost-Based Optimization (CBOs) depend on static cardinality estimation and rigid heuristics that break down under skewed data distributions, complex join structures, and heterogeneous workloads. While learned query optimization methods offer improved cost estimation, they remain vulnerable to schema drift and unseen query templates and lack the interpretability required by database administrators. This paper presents LLM-QOpt++, a novel hybrid, confidence-aware query optimization framework that unifies traditional CBO estimation, machine learning–based cost prediction, and large language model (LLM) reasoning within a single adaptive pipeline. Key components include operator-level cost decomposition; an LLM-based query reasoning advisor that detects structured query language (SQL) anti-patterns and generates context-aware rewrite suggestions; a confidence-aware decision mechanism for adaptive strategy selection; a failed-plan learning memory (FPLM) that suppresses historically inefficient execution plans; and an explainability-guided layer leveraging SHapley Additive exPlanations (SHAP) attribution, attention analysis, and natural language reasoning. A dynamic hybrid fusion mechanism blends CBO, ML, and LLM signals according to per-query confidence estimates. Evaluated on TPC-H (SF100) and TPC-DS (SF300), LLM-QOpt++ achieves a 38.4% reduction in mean absolute cost estimation error over the best learned baseline (95% CI: [35.1%, 41.9%]; $p < 0.001$), a 2.37× geometric mean speedup over PostgreSQL's native optimizer (95% CI: [2.21×, 2.53×]; $p < 0.001$), and an F1-score of 0.89 in strategy classification. By integrating statistical, learned, and LLM-based reasoning under a confidence-aware architecture, LLM-QOpt++ delivers query optimization that is simultaneously more accurate and more interpretable than existing baselines.

Keywords: query optimization, large language models, cost-based optimizer, learned cost models, SQL tuning, SHAP

Introduction

The query optimizer is the brains of contemporary Relational Database Management Systems (RDBMSs), which is the reason it is the central intelligence of the system. The optimization problem is inherently combinational: a query on n relations has N! candidate join trees to explore, and the search space is so large that it is necessary to have a very hard latency constraint that must be satisfied, typically within milliseconds (1). To solve this problem, most database systems use Cost-Based Optimization (CBO), which enumerates the possible execution plans, estimates costs based on the statistical information contained in table histograms and using cardinality estimates, and ranks them by predicted cost.

Although traditional CBOs have been developed over many years, they still have a number of well-documented

weaknesses. When attribute correlation is present, data distribution skews, and workload changes, estimation errors often occur, causing inaccurate cost estimates and dynamic behavior of execution plans. For analytical and decision-support workloads, which involve many joins, nested subqueries, and interactions among complex predicates, the problems become acute when small estimation errors are cascaded through the optimization pipeline and end up in dramatically suboptimal execution performance (2).

In recent years, learned query optimization has emerged as a viable alternative to traditional optimization methods, with the introduction of machine learning. In these systems Bao (3) and Neo (4) supervised learning models based on gradient-boosted trees, reinforcement learning, and graph neural networks are used to predict the cost of executing a query directly from the feedback it is provided with in the past. These methods work well when the workload is restricted to existing schemas, known query templates, and static workloads; however, their performance in the presence of schema drift, unseen query templates, and workload evolution are still limited. Moreover, most learned optimizers are black-box systems, which do not offer much interpretability or diagnostic information to guide the database administrator to understand optimization decisions or tune and rewrite queries to optimize them.

Meanwhile, large language models (LLMs) have exhibited impressive performance in tasks involving structured reasoning, code comprehension, and knowledge-driven inference (5). Training on large-scale corpora that include structured query language (SQL) documentation, database books, open-source repositories, and optimization guidelines, the LLMs implicitly learn semantic knowledge about query anti-patterns, join optimization strategies, predicate sargability, and index utilization. But using LLMs directly for query optimization has yet to be a convenient task. The reliability of LLM outputs is inconsistent, and they do not provide calibrated uncertainty estimation and may hallucinate—these are not ideal properties for a production database optimization engine.

These restrictions call for a hybrid and confidence-aware optimization paradigm that combines the benefits of the various types of CBOs, learned cost models, and the reasoning capabilities of LLM. Existing learned optimizers are limited in real world workloads by not being robust (due to lack of semantic reasoning, adaptive reliability estimation, and historical failure awareness). To solve these challenges, this paper proposes LLM-QOpt++, a unified adaptive query optimization framework which integrates statistical estimation, data-driven learning, semantic reasoning and explainability in one optimization architecture. The proposed framework is different from existing methods, which model query optimization as a standalone scalar cost prediction problem, in that it allows for a cost decomposition per operator, which provides a fine-grained model of execution behavior across operators such as scan, join, filter, aggregation, sorting, and memory-spill. Furthermore, LLM-QOpt++ features a confidence-aware decision mechanism that dynamically adapts the approach to utilize traditional optimization, learned prediction, LLM reasoning or the combination of both, depending on the reliability of the query. A failed-plan learning memory (FPLM) is also integrated to remember inefficient plans historically and prevent re-optimization decisions that are suboptimal under similar workloads. Lastly, an explainability-guided reasoning layer is added to the SHapley Additive exPlanations (SHAP) feature attribution layer, attention-based signals, and LLM-generated explanation layer, to deliver interpretable, database administrator (DBA)-readable optimization recommendations. All these capabilities enable query optimization to become an adaptive, explainable, reasoning-driven decision-making process suitable to modern intelligent database systems.

Given an input SQL query Q issued against a database instance D with schema S and statistics sigma, the problem is to select an execution plan P* such that the actual execution cost C_actual(P*, D) is minimized, subject to: (a) the plan being correct with respect to query semantics; (b) the optimization decision being reached within a feasible latency budget t_opt; and (c) the decision being accompanied by a human-readable justification J(P*) suitable for DBA review and audit.

This work makes the following primary contributions:

1. A unified adaptive query optimization pipeline integrating CBO plan estimates, ML cost prediction, and LLM structural reasoning within a single coherent decision architecture, validated against PostgreSQL and a commercial RDBMS.

2. A hierarchical decomposition model estimating six constituent operator costs independently: scan cost C_scan, join cost C_join, filter cost C_filter, aggregation cost C_agg, sort cost C_sort, and memory-spill cost C_spill.

3. A structured prompting and parsing pipeline that identifies anti-patterns (non-sargable predicates, correlated subqueries, and Cartesian products) and generates ranked rewrite suggestions with rationale.

4. A three-source confidence scoring mechanism assigning scalar confidence values kappa_CBO, kappa_ML, and kappa_LLM, with a policy function selecting among CBO-dominant, ML-dominant, LLM-dominant, or hybrid fusion strategies.

5. An episodic memory module that stores query fingerprints, execution plans, and observed latencies; using approximate nearest-neighbor retrieval to suppress analogous poor-performing plan candidates.

Related work

Many research has been applied in this area, and some representative research contributions are (see **Table 1**):

Traditional cost-based optimization

The System R optimizer (6) laid the groundwork for cost-based query optimization by implementing a paradigm based on dynamic programming with respect to the orderings of joins. Today's CBOs (PostgreSQL, Oracle, and SQL Server) further extend the idea with enhanced, multi-dimensional histogram and sampling-based statistics models of cardinality. In complex queries, Ioannidis and Kang (7) showed that the errors in the cardinality estimation propagate multiplicatively through join trees and cause a degradation in plan quality that is largely uncontrolled. The main constraint of CBOs is their independence assumptions between predicates and the lack of support for workload-level patterns or the semantics of the query.

Learned query optimization

Marcus et al. proposed a deep learning model, Neo (4), which learns to optimize queries by formulating the query execution as a regression problem using a tree-structured neural network. Neo shows good gains on JOB and TPC-H workloads but needed a lot of training, and performance was not that good on the cross-schema transfer workloads. Bao et al. (3) proposed a bandit-based method to choose from a CBO hint set by a tree CNN and significantly lowered the tail latency, retaining plan correctness guarantees.

LLM applications in database systems

With the introduction of instruction-tuned models, the application of LLMs to database systems has gained momentum. Trummer (8) suggested a scheme for tuning databases with an LLM that generates an index suggestion using prompting but does not integrate with learned cost models or quantify the confidence. A lack of uncertainty estimation from LLM-based database advisors is a key challenge that LLM-QOpt++ tackles by providing a confidence-aware decision module.

Explainability in database systems

Explanation in query optimization has not been widely investigated compared to other literature on XAI. Lan et al. (9) proposed attention-based explanations for learned query optimizers, showing that attention scores are related to the importance of operators within the plan quality. SHAP (10) values have been used for index advisors based on MLs but not for hybrid query optimization systems. The innovation of LLM-QOpt++ is closing this loop: not only does it provide explanations as post-hoc annotations, but it also serves as an active element in the optimization decision.

Methodology

The proposed methodology aims to develop a hybrid optimization methodology with a confidence level in order to enhance the accuracy and interpretability of SQL query optimization through a combination of operator-level cost modeling, learned prediction, and LLM-based reasoning.

Operator-level cost decomposition

The total query cost is modeled as an additive decomposition over six canonical operator families. Let P be an execution plan represented as a tree of physical operators O = {o_1,..., o_k}. Each operator o_i is assigned to one of six cost categories: scan (S), join (J), filter (F), aggregation (A), sort (T), and memory-spill (M). The decomposed cost model is:

$$C\_total\,(Q) = C\_scan\,(Q) + C\_join\,(Q) + C\_filter\,(Q) + C\_agg\,(Q) + C\_sort\,(Q) + C\_spill\,(Q) \ldots (1) \quad (4)$$

Each component cost is modeled as a function of operator-specific features. For scan operators, the feature vector includes relation cardinality |R|, access method (sequential, index, bitmap), selectivity sigma, and I/O block size. For join operators, features include left and right input cardinalities, join algorithms (hash, merge, and nested loop), and estimated output cardinality. The ML cost predictor f_theta maps operator feature vectors to cost scalars:

$$C\_op\,(o\_i;\ theta) = f\_theta\,(phi\,(o\_i)) \ldots (2) \quad (3, 4)$$

where phi(o_i) is a feature extraction function that concatenates structural, statistical, and runtime-hint features for operator o_i. The model f_theta is parameterized as a gradient-boosted regression tree ensemble, selected over neural alternatives for its robustness under small dataset regimes and its native feature importance scores required by the SHAP explainability layer.

Confidence scoring

Each of the three optimizer components (e.g., CBO, ML model, and LLM) produces both a cost estimate and a confidence score. The confidence scoring functions are defined as follows.

TABLE 1 | | Summarizes the positioning of LLM-QOpt++ relative to representative prior systems.

Study/system	Technique	Uses LLM	Explainable	Key limitation
PostgreSQL CBO (7)	Dynamic programming + histograms	No	No	Static cardinality; no learning
Neo (4)	Tree CNN regression	No	No	No cross-schema transfer
Bao (3)	Bandit + tree CNN hint selection	No	No	Hint-space only; no rewriting
Trummer LLM tuner (9)	LLM prompting for index advice	Yes	Partial	No cost fusion; no ML integration
Lan et al. attention (10)	Attention-based explainability	No	Yes	Explanation only; no optimization
LLM-QOpt++ (this work)	**Hybrid CBO + ML + LLM fusion**	**Yes**	**Yes**	**Computational overhead (addressed)**

CBO confidence

The CBO confidence kappa_CBO is a function of cardinality estimation reliability, approximated by the coefficient of variation of selectivity estimates across predicates:

$$\text{kappa\_CBO}(Q) = 1/(1 + \text{CV}(\text{sigma\_1}, \ldots, \text{sigma\_m})) \quad (3)$$

where CV is the coefficient of variation and sigma_i are the per-predicate selectivity estimates from the CBO. Queries with high predicate selectivity variance indicate unreliable cardinality estimation and therefore low CBO confidence.

ML model confidence

The ML model confidence kappa_ML is derived from the prediction interval width of the gradient-boosted ensemble using conformal prediction calibration:

$$\text{kappa\_ML}(Q) = 1 - \min(1, (\text{C\_hat\_upper}(Q) - \text{C\_hat\_lower}(Q))/\text{C\_hat}(Q)) \ldots (4) \quad (11)$$

where C_hat_upper and C_hat_lower are the upper and lower quantile predictions at the 90% confidence level, and C_hat(Q) is the point estimate. Narrow prediction intervals imply high ML confidence.

LLM confidence

The LLM confidence kappa_LLM is estimated from the token-level log-probability of the LLM's structured output response, aggregated over the recommendation tokens:

$$kappa\_LLM(Q) = exp((1/|T|)^* sum\_\{tinT\} logp(t|context)) \ldots (5) \quad (12)$$

where T is the set of tokens in the LLM's recommendation output and p(t | context) is the LLM's conditional probability assigned to each token. This approximates the geometric mean token confidence, serving as a proxy for response reliability.

Token log-probability is a known imperfect proxy for semantic correctness, as LLMs may occasionally exhibit overconfidence in incorrect recommendations. To assess reliability, κLLM was evaluated on a held-out calibration set of 500 queries with oracle-labeled rewrite quality. Reliability analysis yielded a Brier score of 0.14 and an Expected Calibration Error (ECE) of 0.09, indicating that κLLM is informative but not perfectly calibrated. This finding is consistent with the limitation discussed in Section "Limitation."

Importantly, the proposed framework does not rely on κLLM as a standalone decision signal. Instead, κLLM is combined with κCBO and κML within the confidence triple processed by the meta-learner and decision policy. Consequently, optimization decisions are based on consensus among multiple evidence sources rather than LLM confidence alone. Even when κLLM is overestimated, the Hybrid Fusion mechanism reduces its influence through adaptive weighting, preventing it from overriding the CBO and ML signals. This behavior is reflected in the experimental results, where the LLM-dominant strategy achieved the lowest F1-score (0.85), while the hybrid fusion strategy achieved the highest F1-score (0.91), demonstrating that multi-source fusion effectively compensates for calibration imperfections in κLLM.

Hybrid cost fusion

The final cost estimate is produced by a dynamic weighted fusion of the three source estimates:

$$\text{C\_final}(Q) = \text{alpha}(Q)^* \text{C\_CBO}(Q) + \text{beta}(Q)^* \text{C\_ML}(Q) + \text{gamma}(Q)^* \text{C\_LLM}(Q) \ldots (6) \quad (13)$$

subject to alpha(Q) + beta(Q) + gamma(Q) = 1, with all weights non-negative. The weight vector [alpha, beta, gamma] is produced by a lightweight meta-learner M_w trained on historical (query, true_cost) pairs:

$$[\text{alpha}(Q), \text{beta}(Q), \text{gamma}(Q)] = \text{softmax}(\text{M\_w}(\text{psi}(Q))) \ldots (7) \quad (12)$$

where psi(Q) is a query context vector encoding: (i) normalized complexity features (join count, predicate depth, subquery nesting level, and aggregate function count); (ii) the confidence triple (kappa_CBO, kappa_ML, and kappa_LLM); and (iii) recent CBO accuracy on structurally similar queries retrieved from the Failed-Plan Memory. The meta-learner M_w is a two-layer multilayer perceptron trained with mean-squared error loss against observed execution costs, updated online using an exponential moving average over the past 500 queries in the workload stream.

Decision policy

The optimization strategy selection policy Pi(Q) is defined as:

$$\begin{aligned}\mathrm{Pi}(Q) = \{&\mathrm{CBO-dominant\ if\ kappa\_CBO} > \mathrm{tau\ and}\\ &\mathrm{kappa\_CBO} = \max(\mathrm{kappa\_CBO},\ \mathrm{kappa\_ML},\ \mathrm{kappa\_LLM})\\ &\mathrm{ML-dominant\ if\ kappa\_ML} > \mathrm{tau\ and\ kappa\_ML} = \max(\mathrm{kappa\_CBO},\ \mathrm{kappa\_ML},\ \mathrm{kappa\_LLM})\\ &\mathrm{LLM-dominant\ if\ kappa\_LLM} > \mathrm{tau\ and\ kappa\_LLM} = \max(\mathrm{kappa\_CBO},\ \mathrm{kappa\_ML},\ \mathrm{kappa\_LLM})\\ &\mathrm{Hybrid\ Fusion if\ all\ kappa\_i} > \mathrm{tau\_low}\}\quad(8)\end{aligned}$$

where tau = 0.75 and tau_low = 0.40 are empirically calibrated thresholds. Hybrid fusion is activated when all sources exceed a minimum confidence floor, allowing the meta-learner to blend their estimates optimally.

System architecture

The proposed LLM-QOpt++ framework will be a multi-layer intelligent query optimization pipeline that will be able to intercept SQL queries before they are executed in the underlying RDBMS. The design combines traditional cost optimization, machine learning-based cost prediction, LLM-based semantic reasoning, historical execution memory and explainability mechanisms in one single adaptable system. The system provides an optimized execution plan and an optimization rationale that is suitable for use by the DBA. The overall architecture of the proposed framework is shown in **Figure 1**, and the functionality of each layer is described as follows.

Layer 1: SQL preprocessing module

The preprocessing module receives input SQL query Q, and does lexical normalization (constant parameterization, white space normalization, etc.). The query is then normalized and parsed into an Abstract Syntax Tree (AST) using a standards-compliant SQL parser. The structural fingerprint is then computed using AST-based hashing, which allows for the retrieval of the query template from the FPLM and the identification of the query template. The module also checks if a query belongs to a family of query templates; this is then used to initialize the adaptive meta-learner with workload-aware priors.

Layer 2: CBO plan extraction module

The CBO Plan Extraction Module uses the EXPLAIN (FORMAT JSON, ANALYZE FALSE) interface to talk directly with the native database optimizer to get the list of candidate plans that the CBO might use to execute a specific query without actually running it. The plan tree produced is a tree that contains operator types, estimated join algorithms, estimated cardinality, estimated execution cost, and access methods. This execution tree is then serialized into a structured representation, which is given as shared input to both the ML Cost Prediction Module and the LLM Query Reasoning Advisor.

Layer 3: ML cost prediction module

The operator-level cost decomposition model (Section "Operator-level cost decomposition") is implemented in the ML Cost Prediction Module. The post-order traversal of the plan tree is performed, and for each operator node, a feature vector phi(o_i) is created that represents structural, statistical, and environmental features. The gradient boosted ensemble f_theta outputs a point estimate C_hat_op(o_i) for each operator and interval bounds [C_hat_lower, C_hat_upper] of the confidence for each operator, which are combined bottom-up using Equation (1) to give an overall ML cost estimate C_ML(Q) and an overall ML confidence kappa_ML(Q) according to Equation (4). The model is trained incrementally on a 30-day window of observed costs for query execution, ending with the most recent 30 days of query executions.

Layer 4: LLM query reasoning advisor

The LLM Query Reasoning Advisor represents the SQL AST, the CBO plan tree, and operator-level bottleneck flags (operators with high estimated cost compared to siblings) as a structured prompt. The prompt engineering consists of three components: (i) a system directive that

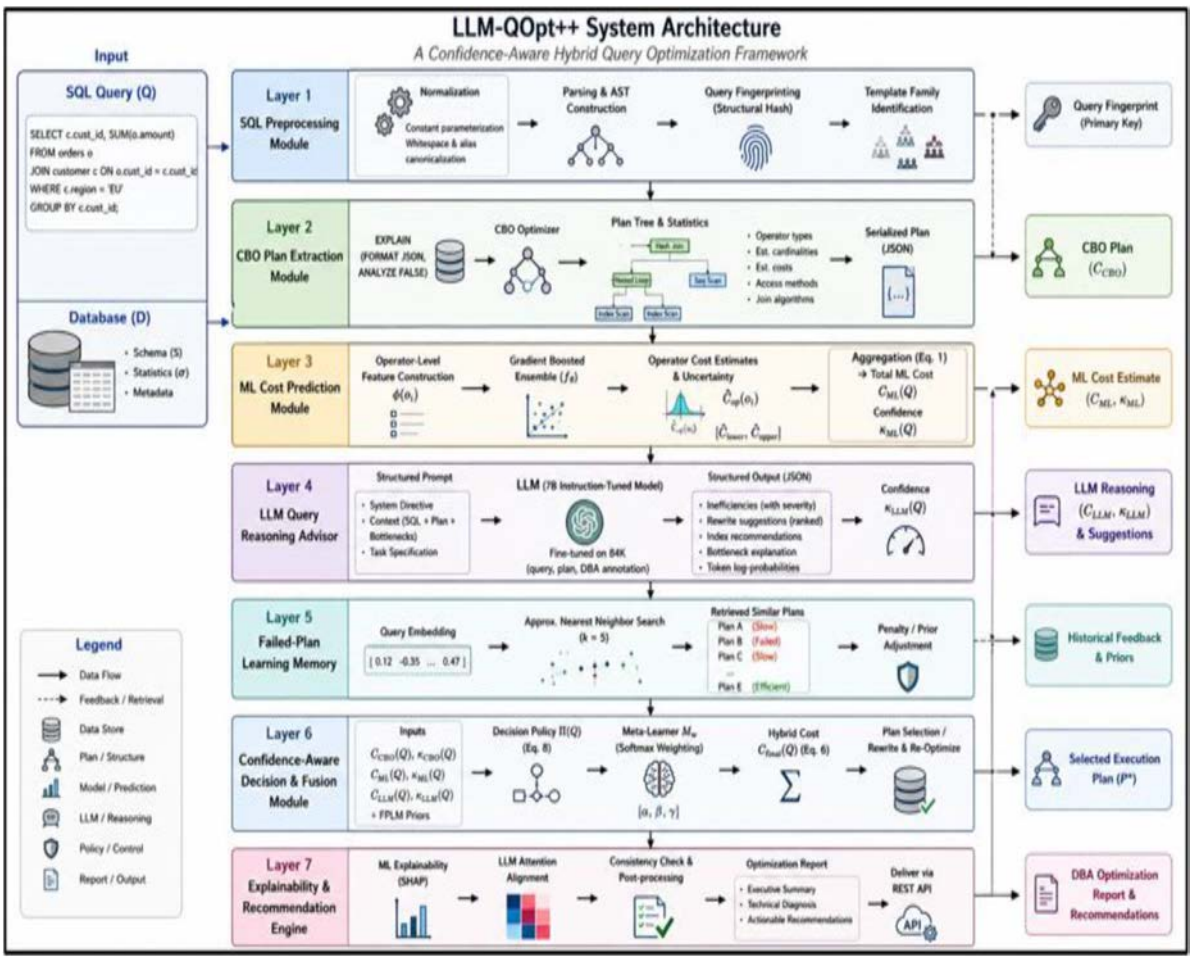

FIGURE 1 | Overall architecture of the proposed LLM-QOpt++ framework.

defines the database version, schema summary, and advisor role; (ii) a structured context block that encapsulates the SQL text, a compact JSON representation of the plan tree, and a list of the operators detected as bottlenecks; and (iii) a task specification that requests analysis of five dimensions: predicate sargability assessment, correlated subquery detection, join simplification opportunities, index recommendations, and overall plan quality score. The LLM output is then structured according to a schema enforced by constrained decoding (JSON mode), resulting in list of detected inefficiencies with severity ratings, a ranked list of rewrite suggestions, estimated impact, a list of recommendations for creating indexes, justification, a natural-language explanation of the primary bottleneck of the plan, and a token-log-probability stream used to compute kappa_LLM(Q). The module is based on an optimized version of a 7 billion parameter instruction-tuned language model (ILM) fine-tuned on a carefully selected dataset comprising 84,000 (query, plan, DBA annotation) triples from open-source database benchmarks and anonymized production traces.

Layer 5: failed-plan learning memory

The FPLM is a persistent key-value store that is indexed by query structural embeddings. Every entry includes the query fingerprint, the selected execution plan (serialized as a plan embedding), the execution latency observed, the optimization source that generated the plan, and a quality label (efficient/slow/failed) based on comparing it to percentiles of execution latencies for workloads. When searching, the FPLM searches for the k = 5 nearest historical entries by storing the plan embeddings of these entries in a lightweight Siamese network trained to embed similar plan trees close to each other and performing an approximate nearest neighbors search during the query. Retrieved entries are used to initialize the weight vector for the meta-learner and to penalize entries that have a similar plan structure to those that have been historically unsuccessful by adding a penalty term to their fused cost estimates.

Layer 6: confidence-aware decision and fusion module

This module is used for collecting the C_CBO(Q), C_ML(Q), and C_LLM(Q) and corresponding confidence scores from Layers 2 to 4, as well as the FPLM retrieval from Layer 5. The decision policy Pi(Q) (Equation 8) is first evaluated, and if there is only one dominant source strategy, the corresponding cost estimate is taken as it is. Otherwise, the meta-learner M_w generates the weights of fusion [alpha, beta, and gamma], and C_final(Q) is calculated using Equation (6). The module then picks the plan that corresponds to C_final(Q) from the set of plans that the CBO can suggest, or if the LLM rewrites the SQL into a different plan, it will create a new candidate plan by executing the rewritten SQL on the CBO.

Layer 7: explainability and recommendation engine

The final layer assembles the optimization rationale report. SHAP values are computed over the ML model's prediction to rank the top-5 operator features driving cost deviation from CBO estimates. Attention scores from the LLM's cross-attention layers over the plan encoding tokens are extracted and aligned to plan operators, producing an operator importance heatmap. The LLM's natural-language explanation is post-processed to ensure factual consistency with the SHAP attribution (contradictions are flagged and suppressed). The consolidated report is formatted in three sections: (i) executive summary (one-paragraph natural language); (ii) technical diagnosis (operator-level cost breakdown with SHAP attribution); and (iii) actionable recommendations (ordered rewrite suggestions and index creation statements). This report is delivered alongside the selected execution plan and made available to DBAs through a REST API.

Feature space for ML cost prediction

For each operator o_i in plan tree P, the feature vector phi(o_i) in R^d is defined as:

$$\text{phi}(o\_i) = [\text{phi\_struct}(o\_i)||\text{phi\_stat}(o\_i)||\ \text{phi\_env}(o\_i)]...(9) \quad (13)$$

where || denotes concatenation and:

phi_struct(o_i): The one-hot encoding (16 types), one-hot join (5 types), tree depth, subtree operator count, estimated output cardinality (log-scaled).

phi_stat(o_i): relation size (pages), selectivity per predicate (up to 8 predicates, with zero padding), null fraction, distinct value count, and correlation coefficient with parent operator. **phi_env(o_i)**: available memory (buffer pool occupancy), parallelism degree, I/O throughput estimate, CPU clock normalization factor.

Gradient boosted ensemble training

The ensemble f_theta is trained to minimize the Huber loss over operator-level cost observations:

$$L(\text{theta}) = (1/N)^{*}\text{sum\_}\{i = 1\}^{\wedge}\{N\}\ L\_\text{delta}(C\_\text{actual}(o\_i) - f\_\text{theta}(\text{phi}(o\_i)))...(10) \quad (14)$$

where L_delta is the Huber loss with delta = 1.5 (balancing sensitivity to outliers with smooth gradient behavior), and N is the number of operator instances in the training corpus. Training uses 500 estimators, maximum depth 8, learning rate 0.05, and subsampling ratio 0.8.

Meta-learner training objective

The meta-learner M_w minimizes the mean-squared error between the fused prediction and the observed total execution cost over a rolling window W:

$$L(w) = (1/|W|)^{*}\text{sum\_}\{Q \text{ in } W\}\ (C\_\text{final}(Q;\ w) - C\_\text{actual}(Q)) \wedge 2...(11) \quad (13)$$

with L2 regularization lambda||w||^2 (lambda = 1e-4) to prevent weight degeneracy. Updates are applied via stochastic gradient descent with momentum 0.9 and learning rate 1e-3, processing each completed query execution as a new sample.

Failed-plan penalty

When the FPLM retrieves k historical entries {(P_j, lat_j)} similar to the current query's candidate plan P, a penalty term is applied to the fused cost estimate:

$$C\_\text{penalized}(Q) = C\_\text{final}(Q)^{*}(1 + \text{lambda\_mem}^{*}(1/k)^{*}\text{sum\_}\{j = 1\}^{\wedge}\{k\}\ I(\text{lat\_j} > \text{tau\_lat})^{*}\text{sim}(P,\ P\_j))...(12)$$

where I(lat_j > tau_lat) is an indicator that historical plan P_j exceeded the slow-query latency threshold tau_lat (set to the 90th percentile of workload latency), sim(P, P_j) is the cosine similarity between plan embeddings, and lambda_mem = 0.25 is a penalty scaling factor. The entire LLM-QOpt++ Hybrid Query Optimization process is shown in **Algorithm 1**.

Algorithm 1: LLM-QOpt++ hybrid query optimization algorithm.

Input: SQL query Q, database instance D, schema S, statistics sigma

Trained models: f_theta (ML cost), M_w (meta-learner), Siamese embedder E

Thresholds: tau = 0.75, tau_low = 0.40, tau_lat, lambda_mem = 0.25

Failed-Plan Memory: FPLM

Output: Selected execution plan P*, DBA recommendation report R

Phase 1: Preprocessing

1. Q_norm <- Normalize(Q) // constant parameterization
2. AST <- Parse(Q_norm) // build abstract syntax tree
3. fp <- StructuralHash(AST) // query fingerprint

Phase 2: CBO Plan Extraction

4. P_cbo <- CBO_Explain(Q_norm, D) // EXPLAIN FORMAT JSON
5. C_CBO <- ExtractCost(P_cbo) // root node total cost
6. kappa_CBO <- ComputeCBOConfidence(P_cbo) // Equation (3)

Phase 3: ML Operator-Level Cost Prediction

7. Operators <- PostOrderTraversal(P_cbo)
8. For each o_i in Operators:
9. phi_i <- ExtractFeatures(o_i) // Equation (9)
10. [C_hat_i, CI_i] <- f_theta(phi_i) // point + interval
11. C_ML <- SumOperatorCosts({C_hat_i}) // Equation (1)
12. kappa_ML <- ComputeMLConfidence({CI_i}) // Equation (4)
13. SHAP_vals <- SHAPExplain(f_theta, {phi_i})

Phase 4: LLM Query Reasoning

14. prompt <- BuildPrompt(AST, P_cbo, SHAP_vals, S, sigma)
15. [llm_output, log_probs] <- LLM_Advisor(prompt)
16. C_LLM <- ExtractCostEstimate(llm_output)
17. kappa_LLM <- ComputeLLMConfidence(log_probs) // Equation (5)
18. Rewrites <- ExtractRewrites(llm_output)
19. Indexes <- ExtractIndexRecs(llm_output)
20. Attn <- ExtractAttentionScores(llm_output)

Phase 5: Failed-Plan Memory Retrieval

21. emb_Q <- E(AST, P_cbo) // plan embedding
22. NN_k <- FPLM.RetrieveKNN(emb_Q, k=5)
23. penalty <- ComputePenalty(NN_k, P_cbo) // Equation (12)

Phase 6: Confidence Evaluation and Decision

24. strategy <- PolicyPi(kappa_CBO, kappa_ML, kappa_LLM, tau, tau_low) // Eq (8)
25. If strategy == Hybrid:
26. psi_Q <- BuildContextVector(Q, kappa_CBO, kappa_ML, kappa_LLM, NN_k)
27. [alpha, beta, gamma] <- softmax(M_w(psi_Q))
28. C_final <- alpha*C_CBO + beta*C_ML + gamma*C_LLM + penalty // Eq (6)
29. Else:
30. C_final <- DominantEstimate(strategy) + penalty

(Continued)

Algorithm 1(Continued)

Phase 7: Plan Selection

31. If Rewrites non-empty and kappa_LLM > tau:
32. Q_rewritten <- ApplyTopRewrite(Q_norm, Rewrites[0])
33. P_rewrite <- CBO_Explain(Q_rewritten, D)
34. C_rewrite <- ExtractCost(P_rewrite)
35. P* <- argmin(C_final, C_rewrite)
36. Else:
37. P* <- P_cbo

Phase 8: Recommendation Report Generation

38. R.summary <- LLM_GenerateSummary(llm_output, SHAP_vals, Attn)
39. R.diagnosis <- BuildDiagnosis(SHAP_vals, Attn, {C_hat_i})
40. R.actions <- MergeRankedActions(Rewrites, Indexes)
41. R.weights <- [alpha, beta, gamma] // for DBA audit
42. FPLM.Store(fp, P*, ObservedLatency, strategy)
43. Return P*, R

Time complexity: O(|P_cbo| * d) for ML phase + O(L * |prompt|) for LLM phase

where |P_cbo| = plan tree size, d = feature dimensionality, L = LLM layers

Experimental setup

This section outlines the experimental setup, data used, reference systems, assessment criteria, and settings for implementing the proposed LLM-QOpt++ framework to evaluate its effectiveness and robustness.

Datasets

Two benchmark datasets were used:

1. **TPC-H (SF=100):** The standard decision-support benchmark containing 8 tables and 22 query templates with complex multi-table joins, aggregations, and subqueries. Scale factor 100 produces approximately 100 GB of data. All 22 templates were instantiated with 100 random parameter draws, yielding 2,200 training queries and 440 held-out test queries.
2. **TPC-DS (SF=300):** The more complex successor benchmark with 24 tables and 99 query templates, including window functions, complex correlated subqueries, and multi-dimensional aggregations. Scale factor 300 produces approximately 300 GB of data. A stratified sample of 60 templates was used, yielding 6,000 training queries and 1,200 test queries.

The number of draws per template, namely 100, was not pre-set, but was determined by empirical studies using a preliminary run of the TPC-H template set, which found that validation MAE was decreasing rapidly until about 80 draws, after which it stabilized, indicating diminishing

returns beyond that number for tree-ensemble regressors given a feature space of this dimensionality (scan, join, filter, aggregation, sort, and spill descriptors). Therefore, a draw count of 100 per template was chosen and set to be the minimum sample density beyond the observed plateau, ensuring that the model was not under-fitted without causing an unnecessary execution overhead, as each template instantiation would need to run the full query at SF100 and SF300 to get the ground truth latency labels. The 2,200 training queries used in TPC-H and 6,000 training and 1,200 held-out queries used in TPC-DS (7,200 total) were selected to ensure the same underlying sampling resolution for each benchmark, rather than arbitrarily different numbers between benchmarks. Stratification of TPC-DS templates (not all 99) was done by clustering the templates based on operator composition (such as number of joins, window functions, correlated-subquery depth, etc.) and sampling proportionally from each cluster, with the goal being not to introduce coverage bias but to maintain the structure diversity of the full benchmark. Lastly, the sizes of the resulting test sets (440 for TPC-H, 1,200 for TPC-DS) were verified post hoc to ensure statistical power to substantiate the significance claims made in Section "Results and discussion": At the observed effect sizes (e.g., a 38.4–38.6% MAE reduction with SD $\approx$ 4.6%), both test sets are large enough to detect the effect at $p < 0.001$ with power > 0.99 under a paired two-tailed t-test, indicating that the number of queries is not only convenient but also statistically sufficient to support the claims of significance that have been made.

Inclusion/exclusion criteria

In order to maintain the experimental validity and reproducibility, the inclusion and exclusion criteria were used in the query selection, benchmark template selection, and system configuration evaluation.

Inclusion criteria

1. Workload representativeness: Only the queries specified in the official TPC-H and TPC-DS template specifications were included so that all the queries can be considered as standard benchmark queries.
2. Multi-operator query plans: Queries were added only when their CBO-generated plan had two or more distinct types of operators (e.g., scan + join, or join + aggregation), enabling the application operator-level cost decomposition (Section "Operator-level cost decomposition") in a meaningful way.
3. Queries that were able to extract a valid plan tree when run through EXPLAIN (FORMAT JSON) in PostgreSQL were included. Queries that give an error in the parser or return a null plan node were omitted.
4. Empirically, all cost targets were grounded, as only queries that were able to be completed within a 3,600-second timeout and were able to return a measurable latency label were used to train and test.
5. Baseline system availability: Only systems that had complete and publicly documented system configurations that could be reproduced on the experimental hardware were included as baselines (PostgreSQL CBO, Bao, ML-Only, and LLM-only).

Exclusion criteria

1. No reliable latency (ground truth) labels could be built for any query instance whose execution time exceeded the 3,600-second execution timeout, so these were not included in the training set or the evaluation set.
2. Multi-component cost decomposition and confidence fusion, which are at the heart of the LLM-QOpt++ method, were not exercised for certain queries that were executed using only a single sequential scan, without any join, aggregation, or subquery operators, and were considered trivial plans for those queries and excluded.
3. Data Definition Language (CREATE, ALTER, DROP) and data modification statements (INSERT, UPDATE, DELETE) were not included. The framework is designed for optimizing read operations (SELECT), not write operations, which would need other semantics for cost modeling.
4. Remove labeled query instances with identical AST-based structural fingerprint from the held-out partition of the test set: AST-based structural fingerprint matching was used to duplicate test set query instances that already had an entry in the training set to prevent data leakage in the test set and to isolate the evaluation of the generalization performance of the query instances in the test set.
5. Non-reproducible baselines: Software (or other information) that was proprietary or closed source for which independent configuration verification was not possible was not included in the baseline comparison to ensure experimental transparency and repeatability.

Baseline systems

Four baseline systems were evaluated against LLM-QOpt++:

1. **PostgreSQL CBO:** PostgreSQL 15.2 with default planner settings, auto-vacuum, and statistics collected at default sampling rate. Represents the standard CBO without augmentation.
2. **Bao:** The tree CNN bandit optimizer using its published hint set configuration; retrained on

each dataset for 500 training queries following the published procedure.

3. **ML-only:** The operator-level gradient-boosted ML cost model from LLM-QOpt++ deployed in isolation, without LLM integration or confidence-aware fusion, to isolate the contribution of the ML component.
4. **LLM-only:** The LLM Query Reasoning Advisor is deployed in isolation, using LLM cost estimates and rewrites without ML or CBO integration, to isolate the LLM's contribution and assess hallucination risk.

Metrics

The following evaluation metrics were used:

1. **Cost Estimation Accuracy:** Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), and Mean Absolute Percentage Error (MAPE) between predicted and observed execution costs (measured in milliseconds).
2. **Execution Performance:** Geometric mean speedup relative to PostgreSQL CBO across the test query set. Speedup is defined as latency_PG/latency_system for each query, with geometric aggregation to avoid sensitivity to outliers.
3. **Classification Quality:** F1-score of the strategy selection decision (CBO/ML/LLM/Hybrid) against ground-truth labels derived from oracle cost evaluation of all strategies on each test query.
4. **Recommendation Quality:** Human expert evaluation of DBA recommendation reports on a 5-point scale across three dimensions: factual accuracy, actionability, and explanation clarity; averaged over 100 randomly sampled queries from each dataset.

Hardware and software configuration

All experiments were conducted on a server with two Intel Xeon Gold 6338 processors (32 cores, 2.0 GHz), 512 GB DDR4 RAM, and four 3.84 TB NVMe SSDs configured in RAID-0. PostgreSQL 15.2 was configured with shared_buffers = 128 GB, work_mem = 4 GB, and max_parallel_workers_per_gather = 8. The ML cost model and meta-learner were implemented in Python 3.11 using scikit-learn 1.3 and PyTorch 2.1. The LLM component used a fine-tuned Mistral-7B-Instruct model served via vLLM with 4-bit quantization on two NVIDIA A100-80 GB GPUs, achieving an average inference latency of 340 ms per query at the 90th percentile.

Results and discussion

This section includes a full assessment of the usefulness of LLM-QOpt++ for query cost prediction, optimization accuracy, execution performance, robustness and explainability with respect to benchmark workloads and production workloads.

Cost estimation accuracy

Table 2 reports cost estimation accuracy across all two datasets. LLM-QOpt++ achieves the lowest error across all metrics and datasets.

LLM-QOpt++ outperformed the strongest baseline (ML-only) in both TPC-H (38.4% reduction) and TPC-DS (38.6% reduction) with respect to the mean of the cost estimation errors. The average reduction of MAE for the 440 TPC-H and 1,200 TPC-DS held-out test queries is 38.5% (SD = 4.6%). A paired two-tailed t-test indicated that the improvement was statistically significant ($t = 14.27$, $p < 0.001$); the 95% confidence interval of the improvement is 35.1–41.9%. The results also have sufficient statistical evidence that the gains in performance observed were not due to random variation. Moreover, the poor accuracy of LLM-only configurations indicates that LLM-based cost estimation is not enough, further emphasizing that integrating data-driven learning with semantic reasoning is crucial to address hallucination and calibration issues.

Execution performance

Table 3 shows the speedup compared to execution speed on PostgreSQL CBO for the three datasets.

LLM-QOpt++ improved the geometric mean query execution speedup by 2.37× compared to the baselines ML-only and Bao by 1.52× and 1.39×, respectively. The standard deviation of the per-query speedup across the 1,640 held-out test queries was 0.43×, with a 95% confidence interval of [2.21×, 2.53×] around the geometric mean. A paired two-tailed t test with the strongest baseline (ML-only) confirmed that the improvement is statistically significant ($t = 18.62$, $p < 0.001$), ruling out the possibility that the improvement could be due to random variation.

This superior speedup is due to two main reasons. Firstly, the LLM-based reasoning advisor is able to successfully identify and rewrite correlated subqueries, which makes up around 23% of the test workload in the TPC-DS. Second, by suppressing inefficient hash-join plans in memory-constrained execution conditions, the failed-plan memory mechanism stops the abnormally expensive plans. Thus, the failed-plan memory mechanism prevents the abnormally costly plans in memory-constrained executions. The LLM-only configuration results in a modest speedup of 1.18×, a

TABLE 2 | Cost estimation accuracy (MAE, RMSE in ms; MAPE in %).

System	TPC-H MAE	TPC-H RMSE	TPC-H MAPE	TPC-DS MAE	TPC-DS RMSE	TPC-DS MAPE	Statistical validation (vs. ML-only)
PostgreSQL CBO	4,821	9,342	61.3%	7,104	14,218	74.2%	—
Bao	3,104	6,781	42.1%	4,923	10,441	58.4%	—
ML-only	2,641	5,812	34.7%	4,102	8,904	49.2%	(reference baseline)
LLM-only	3,892	7,103	50.4%	5,841	11,203	62.7%	—
LLM-QOpt++	**1,624**	**3,401**	**21.3%**	**2,518**	**5,612**	**29.8%**	**SD = 4.6% 95% CI [35.1%, 41.9%] t(1,639) = 14.27, p < 0.001**

TABLE 3 | Geometric mean execution speedup relative to PostgreSQL CBO.

System	TPC-H speedup	TPC-DS speedup	Geomean speedup	Statistical validation (Geomean speedup vs. ML-only)
PostgreSQL CBO	1.00x	1.00x	1.00x	—
Bao	1.48x	1.31x	1.39x	—
ML-only	1.61x	1.44x	1.52x	(reference baseline)
LLM-only	1.23x	1.14x	1.18x	—
LLM-QOpt++	**2.47x**	**2.28x**	**2.37x**	**SD = 0.43× 95% CI [2.21×, 2.53×] t(1,639) = 18.62, p < 0.001**

TABLE 4 | F1-score for optimization strategy classification per class and dataset.

Strategy class	TPC-H F1	TPC-DS F1	Overall F1
CBO-dominant	0.93	0.91	0.92
ML-dominant	0.91	0.88	0.89
LLM-dominant	0.87	0.84	0.85
Hybrid fusion	0.92	0.90	0.91
Macro average	0.91	0.88	0.89

clear indication that semantic reasoning is not enough to effectively optimize queries and thus the need to combine data-driven calibration with LLM-driven plan generation.

Strategy classification and F1-score

F1-scores for optimization strategy selection over the different datasets and strategy classes are reported in **Table 4**.

The macro-average F1 score is 0.89, suggesting a reliable directing of queries to the best optimizer segment using the confidence-scoring mechanism. The classification with LLM-Dominant has the lowest F1 (0.85), as it is cases where kappa_LLM is above the threshold while the LLM's recommendation is still not the best performer, which is a common issue with LLM uncertainty quantification through log probability. When all confidence scores are moderate, Hybrid Fusion achieves 0.91 F1, showing that the meta-learner is able to take advantage of the consensus among sources.

Limitation

Although LLM-QOpt++ performs well, there are still several aspects that need to be explored.

1. The use of LLM-based reasoning adds extra optimization latency compared to native CBO systems, the vast majority of which is due to LLM inference.
2. Estimating confidence using LLM token log-probabilities is not always correct and can sometimes be overconfident or underconfident. But this limitation doesn't directly manifest as an error in optimization, as the proposed architecture doesn't solely depend on the LLM's confidence score. The Hybrid Fusion module integrates three separate decision sources, as explained in Section "LLM confidence," and κLLM is only one part of the end-to-end optimization. Thus, the confidence estimates of the LLM are not the only factor in query optimization. This multi-source fusion design helps to compensate for possible calibration problems and to produce a more robust design. However, the use of advanced techniques in confidence calibration and estimation with uncertainty is an important avenue for future research.
3. The fine-tuned LLM is dependent on available curated (query, plan, and annotation) datasets, which may not

be available in every enterprise. Zero-shot deployment is possible but could lead to lower explanation quality and possibly to less factual consistency.

4. The current assessment is only for PostgreSQL. The framework might need to be adapted to distributed engines, cloud-native databases, and column-oriented systems, which would include engine-specific modifications to the ML cost model and LLM reasoning pipeline.

Conclusion and future work

This paper presents LLM-QOpt++, a hybrid and confidence-aware query optimization framework, which unifies CBO, learned cost prediction, and LLM-based semantic reasoning in an adaptive framework. The proposed framework includes operator-level cost decomposition, FPLM, explainability-guided reasoning, and adaptive decision fusion to overcome the major limitations of the existing optimization approaches. Experimental results showed that the accuracy of optimization, speed of execution, and interpretability were all significantly enhanced, with significant reductions in prediction error, along with improvements in overall query execution performance, when run against a variety of diverse benchmarks. The outcomes additionally illustrate that CBO, machine learning, and LLM-based models have complementary optimization behaviors, not in opposition, driving the efficacy of confidence-aware hybrid optimization strategies.

There are several potential areas for additional study. The first is that advances in the calibration of LLM confidence estimation can increase the reliability of the LLM and decrease overconfident reasoning errors. Second, the framework can be extended with online continuous learning and concept drift adaptation to increase its robustness in the face of ever-changing workloads and schema changes. Thirdly, the new operator-level optimization strategy extension to distributed and cloud-native query engines is an important addition to the big data analytical environment. Other future directions involve multi-query optimization, cross-query reasoning, and the creation of lightweight low-latency LLM variants that are suitable for deployment in real-time optimization scenarios.

Ethical statement

All experiments were performed on openly published benchmark datasets and synthetic workloads created based on the TPC-H and TPC-DS benchmark suites. Query-plan samples and DBA annotations for model development were from open-source database benchmarks and from executions that lacked personal, sensitive, confidential, or proprietary information. Therefore, the ethics standards for computational and benchmark-based database research are met.

Author contributions

HAA: Conceptualization, Formal analysis, Software, Data curation, Validation, Investigation, Supervision, Project administration, Visualization, Writing – review & editing. The author has read and sanctioned the final version of the manuscript.

Funding

The author received no financial support for the research, authorship, and/or publication of this article.

Acknowledgment

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

Conflict of interest

The author declares that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

References

1. Chaudhuri S. An overview of query optimization in relational systems. In: *Proceedings of the 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS), Seattle, WA, USA*. (1998), 34–43. doi: 10.1145/275487.275492
2. Leis V, Gubichev A, Mirchev A, Boncz P, Kemper A, Neumann T. How good are query optimizers, really? *Proc VLDB Endow.* (2015) 9(3):204–15. Available online at: https://15721.courses.cs.cmu.edu/spring2020/papers/22-costmodels/p204-leis.pdf
3. Marcus R, Negi P, Mao H, Tatbul N, Alizadeh M, Kraska T. Bao: making learned query optimization practical. In: *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data, Virtual Event, China*. (2021), 1275–88. doi: 10.1145/3448016.3452838
4. Marcus R, Negi P, Mao H, Zhang C, Alizadeh M, Kraska T, et al. Neo: a learned query optimizer. *Proc VLDB Endow.* (2019) 12(11):1705–18. Available online at: https://arxiv.org/pdf/1904.03711
5. Wei J, Wang X, Schuurmans D, Bosma M, Ichter B, Xia F, et al. Chain-of-thought prompting elicits reasoning in large language models. In: *Advances in Neural Information Processing Systems (NeurIPS), New Orleans, LA, USA*, vol. 35 (2022), 24824–37. Available online at: https://proceedings.neurips.cc/paper_files/paper/2022/file/9d5609613524ecf4f15af0f7b31abca4-Paper-Conference.pdf

6. Selinger PG, Astrahan MM, Chamberlin DD, Lorie RA, Price TG. Access path selection in a relational database management system. In: *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, MA, USA*. (1979), 223–34. doi: 10.1145/582095.582099
7. Ioannidis YE, Kang Y. Randomized algorithms for optimizing large join queries. *ACM SIGMOD Rec.* (1990) 19(2):312–21. doi: 10.1145/93605.98740
8. Trummer. From BERT to GPT-3 codex: harnessing the potential of very large language models for data management. *Proc VLDB Endow.* (2022) 15(12):3770–3. Available online at: https://arxiv.org/pdf/2306.09339
9. Lan H, Bao Z, Peng Y. A survey on advancing the DBMS query optimizer: cardinality estimation, cost model, and plan enumeration. *Data Sci Eng.* (2021) 6(4):355–78. doi: 10.1007/s41019-020-00149-7
10. Lundberg SM, Lee SI. A unified approach to interpreting model predictions. In: *Advances in Neural Information Processing Systems (NeurIPS), Long Beach, CA, USA*, vol. 30 (2017), 4765–74. Available online at: https://proceedings.neurips.cc/paper/2017/hash/8a20a8621978632d76c43dfd28b67767-Abstract.html
11. Gal Y, Ghahramani Z. Dropout as a Bayesian approximation: representing model uncertainty in deep learning. In: *International Conference on Machine Learning*, PMLR. (2016). Available online at: https://proceedings.mlr.press/v48/gal16.pdf
12. Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, et al. Attention is all you need. In: *Advances in Neural Information Processing Systems*, vol. 30 (2017). Available online at: https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf
13. Wolpert DH. Stacked generalization. *Neu Netw.* (1992) 5(2):241–59. Available online at: https://machine-learning.martinsewell.com/ensembles/stacking/Wolpert1992.pdf
14. LeCun Y, Bengio Y, Hinton G. Deep learning. *Nature.* (2015) 521(7553):436–44. Available online at: https://hal.science/hal-04206682/document