АЛГЕБРО-ЛОГИЧЕСКИЕ МЕТОДЫ В ИНФОРМАТИКЕ
И ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ

ALGEBRAIC AND LOGICAL METHODS IN COMPUTER
SCIENCE AND ARTIFICIAL INTELLIGENCE

Серия «Математика»
2026. Т. 56. С. 113—128

Онлайн-доступ к журналу:
http://mathizv.isu.ru

ИЗВЕСТИЯ
Иркутского
государственного
университета

Research article

УДК 004.657:004.8
MSC 68P15, 68T07
DOI https://doi.org/10.26516/1997-7670.2026.56.113

Training-Free Query Optimization via LLM-Based Plan Similarity

Nikita K. Vasilenko[1]✉, **Alexander V. Demin**[1]✉,
Vladimir S. Burlakov[2]✉

[1] Ershov Institute of Informatics Systems, Novosibirsk, Russian Federation
[2] Lomonosov Moscow State University, Moscow, Russian Federation
✉ vasilenko.nikita.research@gmail.com
✉ alexandredemin@yandex.ru
✉ vladimir.boorlakov@gmail.com

Abstract: Large language model (LLM) embeddings offer a promising new avenue for database query optimization. In this paper, we explore how pre-trained execution plan embeddings can guide SQL query execution without the need for additional model training. We introduce LLM-based Plan Mapping (LLM-PM), a framework that embeds the default execution plan of a query, finds its k nearest neighbors among previously executed plans, and recommends database hintsets based on neighborhood voting. A lightweight consistency check validates the selected hint, while a fallback mechanism searches the full hint space when needed. Evaluated on the JOB-CEB benchmark using openGauss, LLM-PM achieves an average 21% reduction in query latency. This work highlights the potential of LLM-powered embeddings to deliver practical improvements in query performance and opens new directions for training-free, embedding-based optimizer guidance systems.

Keywords: query optimization, LLM for databases, database hints

Acknowledgements: This work was supported by the The Ministry of Economic Development of the Russian Federation in accordance with the subsidy agreement (agreement identifier 000000C313925P4H0002; grant No 139-15-2025-012).

For citation: Vasilenko N. K., Demin A. V., Burlakov V. S. Training-Free Query Optimization via LLM-Based Plan Similarity. *The Bulletin of Irkutsk State University. Series Mathematics*, 2026, vol. 56, pp. 113–128.
https://doi.org/10.26516/1997-7670.2026.56.113

Научная статья

Оптимизация запросов через сходство планов выполнения на основе больших языковых моделей без обучения

Н. К. Василенко[1]✉, А. В. Демин[1]✉, В. С. Бурлаков[2]✉,

[1] Институт систем информатики им. А. П. Ершова, Новосибирск, Российская Федерация

[2] Московский государственный университет им. М. В. Ломоносова, Москва, Российская Федерация

✉ vasilenko.nikita.research@gmail.com

✉ alexandredemin@yandex.ru

✉ vladimir.boorlakov@gmail.com

Аннотация: Отмечается, что векторные представления, получаемые большими языковыми моделями, открывают многообещающее направление в оптимизации запросов к базам данных. Исследуется, как векторные представления планов выполнения, полученные предобученной моделью, могут направлять исполнение SQL-запросов без необходимости дополнительного обучения. Представляется LLM-based Plan Mapping (LLM-PM) — фреймворк, который кодирует стандартный план выполнения запроса, находит его k ближайших соседей среди ранее выполненных планов и на основе «голосования по окрестности» рекомендует набор подсказок оптимизатору. Облегчённая проверка согласованности валидирует выбранную подсказку, а при необходимости резервный механизм выполняет поиск по всему пространству подсказок. В экспериментах на бенчмарке JOB-CEB в системе openGauss LLM-PM в среднем сокращает задержку выполнения запросов на 21 %.

Ключевые слова: оптимизация запросов, LLM для баз данных, подсказки баз данных

Благодарности: Данная работа выполнена при поддержке Министерства экономического развития Российской Федерации в соответствии с соглашением о субсидировании (идентификатор соглашения 000000C313925P4H0002; грант № 139-15-2025-012).

Ссылка для цитирования: Vasilenko N. K., Demin A. V., Burlakov V. S. Training-Free Query Optimization via LLM-Based Plan Similarity // Известия Иркутского государственного университета. Серия Математика. 2026. Т. 56. С. 113–128.
https://doi.org/10.26516/1997-7670.2026.56.113

1. Introduction

Query optimization is central to relational databases, yet cost-based optimizers still stumble, mainly due to inaccurate cardinalities and simplified cost models, sometimes choosing slow joins (e.g., nested loops over hash). Many DBMSs expose optimizer hints (SQL directives for join order/algorithm or index use), but selecting them is brittle and risky. This has motivated ML-based approaches that treat the optimizer as a black box and steer it via hints or plan tweaks. Recent work shows LLMs can help [1; 4], but the latency and cost of generative pipelines hinder deployment in time-critical or resource-constrained settings.

In this paper, we introduce a lightweight method that leverages LLM embeddings of execution plans, rather than the queries themselves, to identify optimal sets of optimizer hints. Our approach combines the representational power of LLM embeddings with a simple, fast retrieval-based mechanism that requires no additional training. The goal is to bridge the expressive capabilities of LLMs with the need for efficient query optimization. Importantly, our method learns from past executions without an explicit training phase, as the pre-trained LLM provides semantic representations of execution plans.

Our objective is practical usability over maximal theoretical gains: a simple, robust system that can be realistically deployed. Despite its simplicity, plan embeddings alone match or surpass handcrafted encodings while demanding far less engineering. In experiments on standard benchmark, LLM-PM consistently nudges the optimizer toward faster plans without modifying the DBMS.

We view our contribution as evidence that "off-the-shelf" plan embeddings, combined with simple retrieval-based methods, offer a practical and low-overhead alternative to both traditional optimizers and complex learned pipelines, not as a bid for theoretical peak performance but as a step toward real-world usability.

In summary, this paper makes the following contributions: (1) A method that uses LLM-generated plan embeddings to guide optimizer hint selection via nearest-neighbor transfer. (2) A modular, training-free approach with low latency and cost that treats the optimizer as a black box and avoids generative overhead. (3) Empirical evidence that plan-level embeddings can outperform the default optimizer without complex learned pipelines. (4) We highlight the potential of LLM embeddings to improve query performance, offering a promising direction for efficient, LLM-driven query optimization systems.

2. Related Work

Recent research into improving the quality of query optimization can be broadly divided into two main directions. The first is aimed at completely replacing the traditional cost-based optimizer with learned models that generate query execution plans from scratch. The second, more incremental direction seeks to augment or guide the existing optimizer using machine learning techniques by refining cost estimates, improving cardinality predictions, or introducing optimizer hints.

2.1. Optimizers that Replace the Traditional Optimizer

One line of work aims to replace the cost-based optimizer with learned planners. NEO [5] predicts plan latency with a tree-convolution model and searches for the lowest-predicted plan-feasible, but label-hungry and slow to train. Deep Learning Cardinality Estimation models (MSCN [3] and its follow-up [9]) capture correlations missed by histograms and can substitute parts of the optimizer. Overall, such approaches trade mature heuristics for large training demands and raise open questions about generalization across schemas, data distributions, and workloads.

2.2. Optimizers that Guide the Traditional Optimizer

A second more pragmatic line keeps the native optimizer and **steers** it with learned signals (better cardinalities/costs or hints). AQO [2] swaps only cardinality estimation: after executions it stores observed cardinalities and trains a k-nearest-neighbor regressor; on repeats it feeds stored estimates to the stock optimizer. ACM [8] recalibrates cost constants online from buffer hits and per-operator CPU via smoothed linear regression, improving cost-time alignment without calibration workloads; it doesn't fix row-count errors. FASTgres [10] treats hinting as supervised classification, clustering queries and predicting join/index toggles, with a feedback loop that retrains after slowdowns. [11] frames hinting as learning-to-rank over candidate hinted plans. Lero [12] similarly learns pairwise plan preferences from runtimes, leveraging the optimizer for candidates and sidestepping absolute cost modeling. Overall, these methods preserve the DBMS search space while adding a lightweight learned advisor.

2.3. Large Language Models for Query-Plan Optimization

Recent research has begun to explore whether the rich prior knowledge encoded in foundation models, such as LLMs, can be leveraged to improve query optimization.

In [7], the authors fine-tune a generative LLM with serialized queries, metadata, and plans so that, given a query, the model directly outputs an

execution plan step by step. An alternative direction, closer to our work, is using LLMs for plan tuning rather than full plan generation. In [1], the authors propose an approach that keeps the existing optimizer but uses an LLM to assist in picking hints. In [4], query rewriting is used as an optimization method. This approach solves a similar query optimization task using a language model, but the method differs.

3. Design of LLM-PM

This chapter presents LLM-PM, a two-part framework for automatic hint selection based on LLM plan embeddings. It combines our proposed Plan-Mapping algorithm, which transfers hint sets from similar plans in embedding space, with an adaptive search procedure that exhaustively explores the hint space to identify optimal configurations. While the search algorithm itself is not novel, we include it as a fallback mechanism to ensure completeness and make the overall system self-contained.

3.1. Problem formulation

We formalize the task of automatic hint selection for query optimization as follows. Let Q be an SQL query submitted to the database. The database's query optimizer can produce an execution plan P for Q, which results in some cost $C(Q, P)$. In the default case (with no hints), the optimizer chooses a plan P_{default} based on its built-in heuristics and cost estimates, yielding execution time $T(Q, P_{\text{default}})$. Our goal is to find a hint set H for query Q such that when the optimizer is guided by H (and thus may choose a different plan P_H), the execution time is minimized. Formally, we seek:

$$H^*(Q) = \arg \min_{H \in \mathcal{H}(Q)} T\left(Q,\, P_H\right), \tag{3.1}$$

where $\mathcal{H}(Q)$ denotes the space of all valid hint configurations for query Q. In other words, $H^*(Q)$ is the optimal hint set that yields the lowest latency for Q. This is akin to the definition of steering an optimizer: selecting a hint or set of hints that leads to the fastest plan for the query. In practice, $\mathcal{H}(Q)$ can be very large.

3.2. Hint Sets Configuration and Search Strategies

Exhaustively testing every hint combination quickly becomes infeasible, so we accelerate the search with two techniques.

Timeout threshold. After each plan execution we update a variable $T_{\min} \leftarrow \min\big(T_{\min},\, T(Q,P)\big)$ (initially $T_{\min} = T(Q, P_{\text{default}})$). If any subsequent hinted plan exceeds the *current* $T_{\min}$, we abort its execution and

mark the associated hint set as *sub-optimal*. This prunes expensive plans early.

Plan caching. Each distinct physical plan is cached together with its measured latency. When the optimizer produces a plan already in the cache, we reuse the stored latency instead of executing the plan again. This is most effective when many hints have little or no impact on the chosen plan.

We control join and scan strategies through seven binary hints of the form *set(enable_X, v)*, where $v \in \{true, false\}$ and $X \in \{nestloop, hashjoin, mergejoin, seqscan, indexscan, indexonlyscan, bitmapscan\}$.

Tie-breaking Among Equivalent Hint Sets. Occasionally the same physical plan P can be generated by more than one hint configuration. For instance,

$$H_1 = (0, 1, 1, 1, 0, 1, 1), \quad H_2 = (0, 1, 1, 1, 0, 1, 0)$$

produce an identical plan even though they differ only in the final flag (*enable_bitmapscan*). Here, the i-th bit denotes the i-th Boolean knob listed earlier; a value of 1 disables the operator, whereas 0 keeps its default, enabled state. To select a unique "canonical" hint set we choose the configuration that disables the *fewest* operators:

$$\widehat{H}(P) = \arg \min_{H:P_H=P} \sum_{i=1}^{7} h_i. \tag{3.2}$$

In (3.2) H_2 is preferred because it disables one knob fewer than H_1. This rule is motivated by two considerations. *(i)* If a plan can be produced whether a knob is on or off, that knob appears to have no influence on the plan's efficiency, so leaving it enabled avoids unnecessary restrictions. *(ii)* Turning off additional operators compresses the optimizer's search space and may cause poor performance on queries or data distributions that differ from the training workload. Selecting the most permissive hint set therefore preserves as much of the optimizer's built-in expertise as possible while still steering it toward the desired plan.

Adaptive hint search algorithm. By terminating slow plans early and short circuiting repeats via the cache, the procedure focuses on a few promising configurations while still covering the entire hint space.

1) **Initialization.** Execute query Q with all operators enabled. Record its runtime as $T_{\min} = T(Q, P_{\text{default}})$ and initialize the cache $\mathcal{C} = \{(P_{\text{default}}, T_{\min})\}$.

2) **Parallel exploration of hint sets.** Distribute all non-default hint sets $\mathcal{H} = \{H_1, \ldots, H_{128}\} \setminus \{H_{\text{default}}\}$ across $w = 8$ worker threads.

3) **Plan evaluation for each $H \in \mathcal{H}$.**
   - Compute the physical plan P_H generated under H.
   - If P_H already appears in $\mathcal{C}$, retrieve its latency t directly; otherwise, execute Q under H with timeout $T_{\min}$, obtaining runtime t (or timeout).
   - If $t =$ Timeout, mark H as *sub-optimal*; otherwise, update the cache $\mathcal{C}[P_H] \leftarrow t$.

4) **Threshold update.** After each successful execution, update $T_{\min} \leftarrow \min(T_{\min}, t)$. This update is atomic across threads so that all workers share the tightest known threshold.

5) **Termination.** When all $H \in \mathcal{H}$ are processed, return the minimal runtime $T_{\min}$ and the plan–latency cache $\mathcal{C}$.

3.3. Plan-Mapping

This section presents plan mapping: using LLM plan embeddings to transfer hints from similar plans.

When planning a query with a new hint, a new, potentially faster plan emerges. It would be useful to also consider this new plan in the system. Following this idea, a second embedding for the potentially faster plan can be created and compared with the optimized plan resulting from applying the hint in the embedding store. In this way, the assignment of hints will undergo a two-step verification process: first, comparing the default plans and searching for a hint candidate, and second, evaluating the modified plan.

The goal is to decide, for a *new plan* P_0 obtained for the incoming query Q with no hints (all operators enabled), whether there exists a nearby plan in the embedding space whose associated hint set is likely to improve Q. We operate in the space of plan embeddings produced by a pre-trained LLM; the word "plan" below always refers to its embedding.

From the offline search we store triples $(\mathbf{d}_i, H_i^\star, \mathbf{o}_i)$, where $\mathbf{d}_i$ is the embedding of the i-th *default* plan, $H_i^\star$ is its optimal hint set, and $\mathbf{o}_i$ is the embedding of the corresponding *optimized* plan. If no hint set speeds up the query, then $H_i^\star$ is the all-enabled vector and $\mathbf{o}_i = \mathbf{d}_i$.

Key intuition. Plans close in embedding space tend to share physical structure, so the same hint set often remains effective within a local neighborhood. We use two steps:

1) **Neighborhood voting.** Among the N nearest default plans, choose the most frequent non-default hint set, assuming similar plans share inefficiencies.

2) **Consistency check.** Re-plan with the candidate hint set; for both the default and candidate plans, gather their K nearest neighbors with known runtimes, compare average runtimes, and keep the hint set whose neighborhood is faster.

This two-step test gives coverage through N-neighbor voting and safety through the K-neighbor runtime check.

Plan-Mapping algorithm. Let $\phi(\cdot)$ be the LLM encoder mapping a query plan to a d-dimensional vector. In Figure 1, the left circle shows the embedding of the default plan P_0 as point p_0, together with its N nearest neighbors (semi-transparent blue). We vote among those neighbors' hint sets to pick the candidate hint H_{cand}. On the right, we show the embedding of the re-planned query P_{cand} as $h(p_0)$ and its K nearest neighbors. By comparing the average runtimes in these two K-neighborhoods, one around p with default runtimes and the other around p_{new} with optimized runtimes, we decide whether H_{cand} genuinely improves performance. This two-stage check maximizes hint coverage using the first N-neighborhood vote while guarding against harmful suggestions using the second K-neighborhood consistency test.

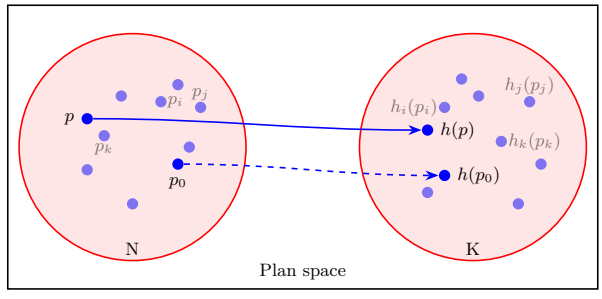


Figure 1. Two stage hint selection by N-vote and K-check

1) **Embed the default plan.** Compute

$$\mathbf{p}_0 = \phi(P_0),$$

where P_0 is the optimizer's default plan for query Q.

2) **Neighborhood voting.** Find the N nearest neighbors of $\mathbf{p}_0$:

$$\mathcal{N} = \{(\mathbf{d}_i, H_i^\star, t_i) \mid i \in \text{top-}N \text{ closest to } \mathbf{p}_0\},$$

where each neighbor carries its hint set $H_i^\star$ and known runtime t_i. Build a frequency table over non-default $\{H_i^\star\}$ and choose the most frequent as H_{cand}, breaking ties by smallest distance $\|\mathbf{p}_0 - \mathbf{d}_i\|$.

3) **Embed the candidate plan.** Apply H_{cand} to Q, let the resulting plan be P_{cand}, and compute

$$\mathbf{p}_{\text{cand}} = \phi(P_{\text{cand}}).$$

4) **Consistency check via two K-neighborhoods.**
   - Let
   $$\mathcal{K}_0 = \{\mathbf{d}_{0,1}, \ldots, \mathbf{d}_{0,K}\} \quad \text{and} \quad \mathcal{K}_{\text{cand}} = \{\mathbf{d}_{\text{cand},1}, \ldots, \mathbf{d}_{\text{cand},K}\}$$
   be the K nearest neighbors of $\mathbf{p}_0$ and $\mathbf{p}_{\text{cand}}$, respectively.
   - Retrieve $\{t_{0,1}, \ldots, t_{0,K}\}$ and $\{t_{\text{cand},1}, \ldots, t_{\text{cand},K}\}$ known runtimes.
   - Compute the average runtimes
   $$\bar{t}_0 = \frac{1}{K}\sum_{j=1}^{K} t_{0,j}, \qquad \bar{t}_{\text{cand}} = \frac{1}{K}\sum_{j=1}^{K} t_{\text{cand},j}.$$
   - *Accept* H_{cand} if $\bar{t}_{\text{cand}} < \bar{t}_0$; otherwise, *reject* it (i.e. keep the default plan).

Hyper-parameters. The algorithm has three knobs: (1) N – neighborhood size for the voting stage ($N = 16$ in our experiments); (2) K – consistency check via two K-neighborhoods; small K favours closest plans, large K increases coverage and stability ($K = 16$ in our experiments); (3) distance metric (Euclidean by default).

3.4. Overall system design

We integrate these techniques into a unified optimization pipeline: Plan-Mapping enables fast optimization when similar plans exist in the experience store; a thorough hint search handles novel queries; and accumulating execution experience improves performance over time. Figure 2 shows the two-stage pipeline.

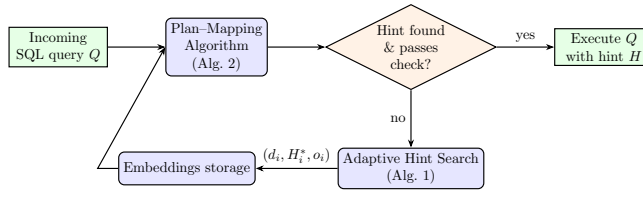


Figure 2. Overall-system control flow of LLM-PM.

1) **Plan-Mapping stage** The incoming SQL query Q is planned with all hints enabled, embedded, and compared to the reference workload using Plan-Mapping algorithm. If the selected candidate hint H_{cand} passes the two-neighborhood consistency test, the query proceeds directly to execution with H_{cand} ("fast path").

2) **Fallback to Adaptive Hint Search** When Plan-Mapping returns *no* hint (or the consistency test rejects the candidate), Q is forwarded to Adaptive hint search algorithm. This exhaustive yet timeout-pruned search inspects the entire space of 128 binary hint sets, finds the fastest configuration, and caches the $\langle$default plan, optimal hint, optimal plan$\rangle$ triplet as the optimal observed outcome for future use.

3) **Execution and continual learning.** Fallback search triplets continually grow the reference set that Plan-Mapping consults for each new query, creating a feedback loop that reduces reliance on exhaustive search and improves end-to-end latency over time.

This design yields low median latency-because most queries are solved by the lightweight, embedding-based mapper-while guaranteeing that even "difficult" queries eventually benefit from the thorough search routine.

4. Evaluation

In this chapter we first outline the experimental design, including datasets, baselines, and implementation details, before detailing the quantitative and qualitative metrics used to assess performance. We then present the results, analyze their statistical significance, and discuss the practical implications that emerge. Finally, we highlight the key insights learned from the evaluation, setting the stage for the conclusions that follow.

4.1. Setup and dataset

The experiments were conducted on a single-node openGauss server with 128 Kunpeng-920 CPU cores, 1 TB RAM, and a 2 TB NVMe SSD, configured with *shared_buffers* = 256 GB, *bulk_write_ring_size* = 10 GB, *work_mem* = 80 GB, *cstore_buffers* = 4 GB, and *query_dop* = 1 (single-threaded execution).

The experiments were carried out on the IMDB dataset, which contains 3 133 queries drawn from the JOB-CEB workload. The dataset relies on 16 templates from the original JOB benchmark, and each template contributes a different number of queries. Plan embeddings were generated with OpenAI's text-embedding-3-large model, so every query is paired with both its default and optimized execution plans, their embeddings, and the associated runtimes. Figure 3 illustrates how the default plans are arranged after reducing the high-dimensional embeddings to two dimensions with Principal Component Analysis (PCA). Each point denotes a single plan projected onto the first two principal components, and its color encodes the corresponding query-group label shown in the legend.

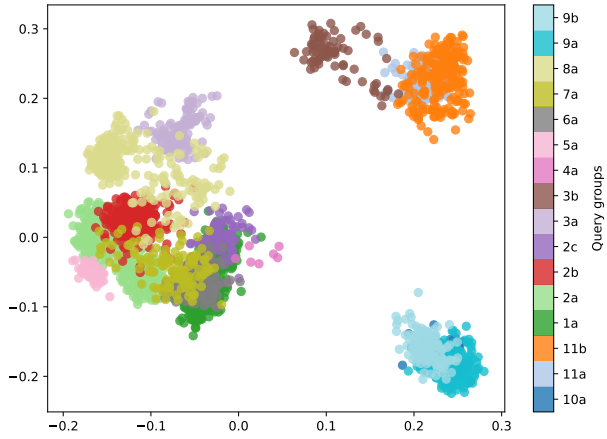


Figure 3. Default plans after PCA

In PostgreSQL, single hints, e.g., globally disabling nested-loop joins, can speed up JOB-CEB [1]; in openGauss the same change harms performance, complicating hint selection.

Table 4 (a) lists the ten most frequent *optimal* hint sets. For roughly half of the queries the optimal configuration coincides with the default (all hints enabled); in total, 86 of the 128 possible bit-vectors were selected as optimal at least once. Figure 4 (b) shows the global distribution of all observed hint sets. The leaders are diverse — disabling nested-loop joins (setting enable_nestloop = false) is *not* always the winning strategy.

(a)

Hintset	Count
0000000	1468
0001100	226
1000100	146
1100100	114
0000100	106
0100100	46
0100000	45
0000001	43
1000000	41
0011100	40

(b)

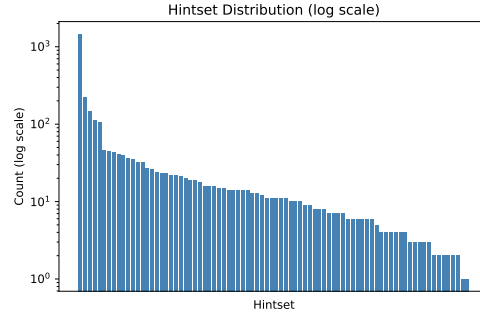


Figure 4. Hint-set statistics: (a) count table and (b) log-scale histogram.

4.2. Results

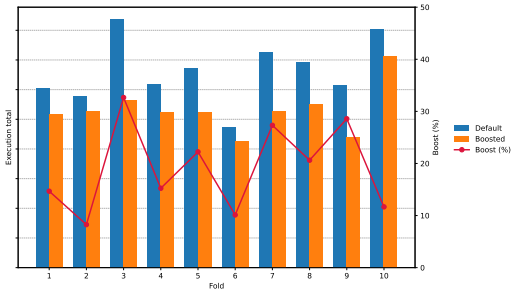
Figure 5 (a) summarizes the outcome of a ten-fold cross-validation: in each fold, 10% of the JOB-CEB queries were held out for testing, while the remaining 90% seeded the reference store for Plan-Mapping.

The aggregate runtime across the ten folds was, on average, reduced by **19.1%** (i.e. queries run on average 1.24× faster than the default optimizer). This gain is robust: even the slowest fold achieves an 8% speedup, and the best fold 32%.The total aggregate speedup achieved, computed by summing the runtimes of all test queries, is 21.1%. The maximum achievable speedup for the entire workload is 62.5%. On an *average fold*, 20% of queries accelerate, 20% decelerate, and the remaining 60% are unaffected. The 90th-percentile execution time falls by 24.7%, while the median time improves slightly by 2.1%.

Table 5 (b) breaks average query runtime into percentile bands. The "Upper bound for percentile" column gives the upper edge of each band; the lower edge is the previous percentile. For example, the 0.5 row covers the 0.4–0.5 quantile slice. "Default" and "Boosted" columns report the mean runtime inside each slice, while "Boost" shows the percentage change within that slice.

Plan-Mapping delivers its largest gains in the long-tail slices with the slowest queries, although a few low-latency slices show occasional slowdowns. A fallback search covers rare misses, capping downside when votes are inconclusive. The effect is consistent across folds and percentile bands, suggesting the method is not tailored to a specific split or subset.

(a)



(b)

Upper percentile bound	Mean default time	Mean boosted time	Boost
0.1	564	628	-11
0.2	1658	1651	0
0.3	2812	2740	3
0.4	4203	4096	3
0.5	6144	5954	3
0.6	8617	8293	4
0.7	12802	11681	9
0.8	19809	17059	14
0.9	35201	27923	21
1.0	121154	90734	25

Figure 5. (a) Per-fold latency improvement under Plan-Mapping. (b) Percentile-level breakdown of mean runtime and boost (%).

4.3. Ablation study

A natural question is whether our learned LLM embeddings provide any real benefit beyond acting as an expensive form of string matching. To answer it, we reran the evaluation under two ablated configurations in which the consistency check stage was disabled, in order to eliminate the influence of the additional verification step and assess whether LLM plan embeddings are truly useful for evaluating plan similarity compared to simple string-based matching metrics:

1) **Levenshtein distance** – candidates are retrieved purely by the Levenshtein edit distance;

2) **LLM embeddings (no consistency check)** – identical to our full system except that the consistency check is omitted.

In the Levenshtein variant we no longer embed plans or compare vectors with the Euclidean metric; instead we compute the edit distance

$$d_{\text{lev}}(p_1, p_2) = \min_{\pi} |\pi|,$$

where π is a sequence of single-character insertions, deletions, or substitutions that transforms plan string p_1 into plan string p_2.

Without the consistency check, the Levenshtein variant not only fails to accelerate the workload: it slows it down by **64.4%** and triggers **87** time-outs that exceeded the 450s. By contrast, LLM embeddings still deliver a **16.1%** speedup with only **5** time-outs, which is more than an order of magnitude fewer than with Levenshtein and close to the default optimizer. When the consistency check is re-enabled, the speedup rises to **21.1%** while time-outs drop to a single query.

These observations confirm that the learned embeddings capture information well beyond raw string similarity and, together with the consistency check, are essential for robust performance gains.

4.4. Discussion and future work

Our ablation study shows that learned LLM embeddings help with hint prediction, but even state-of-the-art OpenAI embeddings are trained on very little data linking query plans to explicit hint effects. The next step is to build or fine tune domain-specific embeddings whose corpora include rich SQL syntax, detailed optimization techniques, hint annotations, and cost feedback. Such embeddings could power stronger retrieval and improve an LLM's reasoning for database query optimization.

While the current Plan-Mapping pipeline delivers an average $\approx 20\%$ acceleration, roughly one-third of the theoretical maximum on this workload,

it also exposes clear opportunities for refinement. Half of the remaining gap arises during the candidate-hint selection stage: in an oracle configuration, where the consistency check is allowed to inspect the true runtimes of both default and candidate plans, we recoup almost 40% of the attainable speedup (two-thirds of the maximum). This suggests that a more sophisticated similarity metric or voting mechanism for the first stage could close much of the residual gap.

Hardware and engine heterogeneity further complicate direct comparisons. Prior work on PostgreSQL reports a substantial average speedup from a single "disable nested-loops" hint, whereas on our openGauss platform the same hint consistently degrades performance. This observation motivates our multi-neighborhood strategy and highlights that no single hint is universally optimal.

5. Conclusion

We have presented LLM-PM, a lightweight, training-free framework that leverages pre-trained LLM embeddings of execution plans to steer a traditional cost-based optimizer via hint recommendations. One of the primary goals of this study was to evaluate the utility of these embeddings, and our experiments provide clear evidence of their effectiveness. Importantly, this approach requires no changes to the underlying DBMS or extensive offline training: it simply reuses past plan outcomes encoded in the embedding space.

We evaluated the core component of the LLM-PM system, Plan-Mapping, on openGauss using a standard benchmark, demonstrating its consistent improvements in query performance. Looking forward, promising extensions include refining the embedding space with metric learning or hybrid plan-SQL features and dynamically adapting neighbor sizes based on local density. These enhancements could further narrow the gap toward optimal hint selection while preserving the simplicity and practicality that make Plan-Mapping suitable for real-world deployment.

References

1. Akioyamen P., Yi Z., Marcus R. The Unreasonable Effectiveness of LLMs for Query Optimization. *arXiv e-prints*, Nov. 2024, arXiv:2411.02862.
2. Ivanov O., Bartunov S. Adaptive query estimation. *arXiv e-prints*, Nov. 2017, arXiv:1711.08330.
3. Kipf A., Kipf T., Radke B., Leis V., Boncz P., Kemper A. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. *arXiv e-prints*, Sept. 2018, arXiv:1809.00677.
4. Li Z., Yuan H., Wang H., Cong G., Bing L. LLM-R2: A Large Language Model Enhanced Rule-based Rewrite System for Boosting Query Efficiency. *arXiv e-prints*, Apr. 2024, arXiv:2404.12872.

5. Marcus R., Negi P., Mao H., Zhang C., Alizadeh M., Kraska T., Papaemmanouil O., Tatbul N. Neo: a learned query optimizer. *Proc. VLDB Endowment*, 2019, vol. 12, pp. 1705–1718.
6. Negi P., Marcus R., Kipf A., Mao H., Tatbul N., Kraska T., Alizadeh M. Flow-loss: learning cardinality estimates that matter. *Proc. VLDB Endowment*, 2021, vol. 14, pp. 2019–2032.
7. Tan J., Zhao K., Li R., Yu J., Piao C., Cheng H., Meng H., Zhao D., Rong Y. Can Large Language Models Be Query Optimizer for Relational Databases? *arXiv e-prints*, Feb. 2025, arXiv:2502.05562.
8. Vasilenko N. K, Demin A. V, Ponomaryov D. K Adaptive Cost Model for Query Optimization. *The Bulletin of Irkutsk State University. Series Mathematics*, 2025, vol. 52, pp. 137–152. https://doi.org/10.26516/1997-7670.2025.52.137
9. Woltmann L., Hartmann C., Thiele M., Habich D., Lehner W. Cardinality estimation with local deep learning models. *Proceedings of the second international workshop on exploiting artificial intelligence techniques for data management*, 2019, pp. 1–8.
10. Woltmann L., Thiessat J., Hartmann C., Habich D., Lehner W. FASTgres: Making Learned Query Optimizer Hinting Effective. *Proc. VLDB Endowment*, 2023, vol. 16, pp. 3310–3322.
11. Xu X., Zhao Z., Zhang T., Kang R., Sun L., Chen J. Coool: A learning-to-rank approach for sql hint recommendations. *arXiv e-prints*, Apr. 2023, arXiv:2304.04407.
12. Zhu R., Chen W., Ding B., Chen X., Pfadler A., Wu Z., Zhou J. Lero: A learning-to-rank query optimizer. *Proc. VLDB Endowment*, 2023, vol. 16, pp. 1466–1479.

Об авторах

Василенко Никита Константинович, аспирант, Институт систем информатики им. А. П. Ершова СО РАН, Новосибирск, 630090, Российская Федерация, vasilenko.nikita.research@gmail.com, https://orcid.org/0009-0003-8727-3000

Демин Александр Викторович, канд. физ.-мат. наук, Институт систем информатики им. А. П. Ершова СО РАН, Новосибирск, 630090, Российская Федерация, alexandredemin@yandex.ru, https://orcid.org/0000-0002-2535-2016

About the authors

Nikita K. Vasilenko, Postgraduate, Ershov Institute of Informatics Systems SB RAS, Novosibirsk, 630090, Russian Federation, vasilenko.nikita.research@gmail.com, https://orcid.org/0009-0003-8727-3000

Alexander V. Demin, Cand. Sci. (Phys.-Math.), Ershov Institute of Informatics Systems SB RAS, Novosibirsk, 630090, Russian Federation, alexandredemin@yandex.ru, https://orcid.org/0000-0002-2535-2016

Бурлаков Владимир Сергеевич,мл. науч. сотр., Исследовательский Центр в сфере искусственного интеллекта МГУ, vladimir.boorlakov@gmail.com, https://orcid.org/0009-0009-1738-9239

Vladimir S. Burlakov, Research Scientist, MSU Research Center for Artificial Intelligence, 119991, Moscow, Russian Federation, vladimir.boorlakov@gmail.com, https://orcid.org/0009-0009-1738-9239

Поступила в редакцию / Received 23.09.2025
Поступила после рецензирования / Revised 24.11.2025
Принята к публикации / Accepted 08.12.2025