The Unreasonable Effectiveness of LLMs for Query Optimization

Peter Akioyamen
University of Pennsylvania
peterai@seas.upenn.edu

Zixuan Yi
University of Pennsylvania
zixy@seas.upenn.edu

Ryan Marcus
University of Pennsylvania
rcmarcus@seas.upenn.edu

Abstract

Recent work in database query optimization has used complex machine learning strategies, such as customized reinforcement learning schemes. Surprisingly, we show that LLM embeddings of query text contain useful semantic information for query optimization. Specifically, we show that a simple binary classifier deciding between alternative query plans, trained only on a small number of labeled embedded query vectors, can outperform existing heuristic systems. Although we only present some preliminary results, an LLM-powered query optimizer could provide significant benefits, both in terms of performance and simplicity.

1 Introduction

Query optimization is the task of transforming complex SQL queries into efficient programs (Selinger et al. [1979]), referred to as query plans. Optimizers represent substantial engineering efforts (Giakoumakis and Galindo-Legaria [2008]), often spanning hundreds of thousands of lines of code (Graefe and McKenna [1993]). Most query optimizers today are driven by complex, manually-written heuristics. Despite significant advancements, query optimizers (QOs) are far from perfect, frequently making costly mistakes (Leis et al. [2015]).

Recent work has shown that machine learning techniques can be used to *steer* query optimizers in the right direction, helping the optimizer determine which plan to select for query execution. Researchers have used supervised learning (e.g., Woltmann et al. [2023]), reinforcement learning (e.g., Marcus et al. [2021]), and hybrid approaches (e.g., Anneser et al. [2023]) to effectively steer optimizers. However, each approach performs sophisticated feature engineering on statistics kept internal by the database, and, as a result, requires complex and deep integration with the underlying query optimizer. This has a number of downsides that have hindered practical adoption (Zhu et al. [2024]).

In this extended abstract, we present initial results for LLMSteer, a simpler approach to steering QOs. Instead of manually engineering complex features from plans or data statistics, we use a large language model (LLM) to embed raw SQL submitted by the database user. We then train a supervised learning model on a small labeled set of queries to predict the optimal direction in which to steer the QO. This places the entire "steering" component outside of the database, simplifying integration.

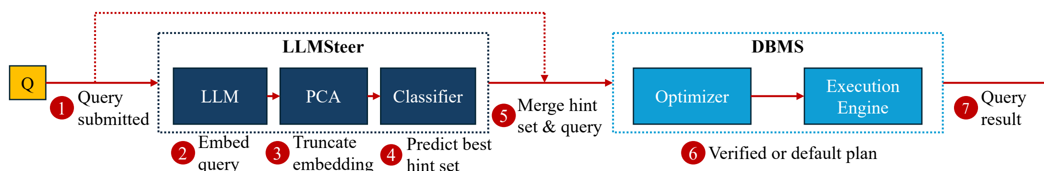


Figure 1: LLMSteer system model.

Machine Learning for Systems Workshop at NeurIPS 2024.

As database experts, we did not expect this simple approach to work. Common wisdom within the database community is that complex features — such as cardinality estimates (Kipf et al. [2018]) or operator models (Heinrich et al. [2022]) — are required for the task. Experimentally, we show that LLMs are capable of making these decisions without any such information. We found no simple explanation for LLMs' apparent success; the LLM-based approach was insensitive to subtle syntax changes and worked across two different workloads. In Section 2, we describe LLMSteer and its simple yet powerful design. In Section 3, we present results from initial experiments. We conclude in Section 4 with a discussion of future directions and questions left unanswered.

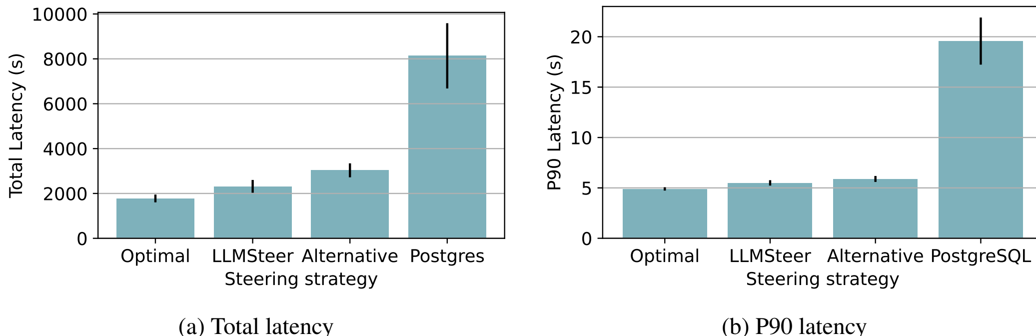
2 LLMSteer: A Surprisingly Simple Approach to Query Steering

Query hints Given a SQL query, an optimizer can generate several plan variants, each of which may use different operators or data access patterns. *Hints* are optional keywords or clauses that can be inserted into a query to guide the optimizer into generating plans with specific characteristics, providing a coarse-grained way to influence a query's execution plan. For example, a hint may indicate to the optimizer that it should only consider plans with hash joins, make use of a helpful index, or limit parallelism.

Optimizer steering Of course, determining the correct hint for a query requires a priori knowledge of the data and workload. *Steering* an optimizer is the task of selecting a hint or set of hints ("hint set") for a particular query such that the plan selected results in reduced or minimal latency. Though hints can be effective in fine-tuning database performance, selecting hints can be extremely complicated, and providing the optimizer with incorrect hints can severely degrade query latency. As a result, the practice of manually issuing a hint is used sparingly and commonly restricted to experts most familiar with the underlying data (Marcus et al. [2021]).

Problem definition LLMSteer attempts to automatically determine an appropriate hint for a query once that query is submitted to the system. Given a small set of labeled embedded query vectors, LLMSteer trains a classification model to map unseen queries to an appropriate hint. We evaluate the quality of LLMSteer's decisions using two common metrics (van Renen et al. [2024]): the change in total and P90 tail latency. When executing a query workload, *total latency* is the cumulative execution time of all queries. *P90 tail latency* is the 90th percentile query latency.

LLMSteer An overview of LLMSteer is depicted in Figure 1. When a query is first submitted to LLMSteer (1), the raw SQL is embedded using a large language model, producing an embedding vector (2). Since our goal is to train a supervised learning model using a small number of examples k, and since LLMs often use high d-dimensional embeddings (i.e., $d > k$), we next apply dimensionality reduction (3). The final feature vector is passed through a classifier to determine the choice of optimal hint for the given query (4). The hint is then combined with the original SQL query (5) and submitted to the database management system (DBMS) where a query plan is generated and executed (6).



(a) Total latency (b) P90 latency

Figure 2: Mean LLMSteer performance on 10-fold cross-validation testing workloads.

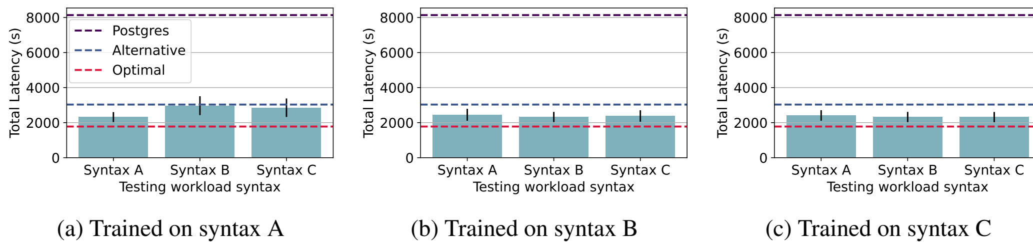
3 Initial Results

We set out to answer two preliminary questions: first, can LLMSTEER find hints for queries that outperform existing query optimizers? Second, since LLMSTEER operates on SQL syntax directly, how robust or sensitive is LLMSTEER to non-semantic syntactic changes in SQL queries (e.g., changes in indentation or whitespace)?

Experimental setup Data used in this work contains 3246 SQL queries, 113 originating from Leis et al. [2015]'s Join Order Benchmark (JOB) and the remaining 3133 are the "core" subset of Negi et al. [2021]'s Cardinality Estimation Benchmark (CEB).[1] To obtain reliable estimates of latencies, each query is executed under a given hint set 5 times and latencies are averaged over the runs to yield a final latency that is used (Yi et al. [2024]). All queries are executed using PostgreSQL version 16.1. A collection of 48 hint sets is considered, the same as those used by Marcus et al. [2021] and Heinrich et al. [2022]. To generate embeddings, we use OpenAI's `text-embedding-3-large` model. Given the latencies of queries under each hint a priori, we determine the hint with the highest potential for improvement if applied perfectly and use this hint as our alternative plan. Afterward, a binary label is generated for each query by determining which of the two possibilities (the default plan or the alternative plan) produces the query plan with lower latency. Queries that perform better under the default plan are given a label of 0, while queries that perform better under the selected hint are given a label of 1. Approximately 30% of queries in the data preform better with the selected hint.

We considered a number of models and found support vector machines with the RBF kernel trained on 120 principal components most performant; models trained on 5, 50, and 120 principal components were evaluated, preserving approximately 50%, 80%, and 90% of variance in the original embeddings. Class weights were used, defined as the ratio of class frequencies, otherwise no hyperparameter tuning was performed and default values were used for all models — the final SVM model used regularization strength $C = 1.0$ and a kernel coefficient of $\gamma = 1/120\sigma_X^2$. To train models we employed a 10-fold cross-validation procedure with stratified random sampling. Our code, along with the data used in the analysis and produced embeddings, are available on GitHub.[2]

Experiment 1: LLMSTEER vs. Postgres Optimizer We evaluate the performance of LLMSTEER against the native Postgres optimizer on P90 and total latency (Figure 2). LLMSTEER represents a significant improvement on the Postgres default, reducing total and P90 latency by 72% on average across testing cross-validation folds. LLMSTEER performs near optimal relative to the steering strategy that selects the correct hint set for every query, achieving a total and P90 latency that is only 30% and 12% higher. Additionally, LLMSTEER shows stability, with performance gains consistent across testing workloads with minimal deviation.



(a) Trained on syntax A (b) Trained on syntax B (c) Trained on syntax C

Figure 3: Mean total latency of LLMSTEER trained on augmented syntaxes across 10-fold cross-validation testing workloads. Syntax A represents original queries, Syntax B represents formatted queries with spaced indention, Syntax C represents formatted queries with tabbed indentation.

Experiment 2: Robustness to Syntactic Changes SQL queries in original training and testing workloads are structured as single-line declarative statements. In practice, database users will rarely structure queries like this, as it impedes the ability to create complex queries and debug SQL statements. There are many ways to alter a query without changing its semantic meaning (Listings 1 & 2), and LLMs are likely to produce different embeddings for queries based on their syntax. To

[1] The "core" subset was specially selected to minimize similarity and overlap in query structure.

[2] https://github.com/peter-ai/LLMSteer.

assess robustness to such syntactic changes, we modified each query in various ways. We refer to "Syntax A" as the original phrasing of each query, and introduce "Syntax B" and "Syntax C," which use newline characters at the end of keyword blocks (i.e., SELECT, FROM, WHERE) and use either spaces or tabs respectively for indentation. Figure 3 shows that LLMSTEER exhibits robustness to at least these classes of syntax changes. Notably, when LLMSTEER trained on original queries (Syntax A), it was still effective on workloads with Syntax B and C; despite a 28% increase in total latency when tested on syntax B and 27% when tested on Syntax C, this still represented a reduction of 64% relative to PostgreSQL. LLMSTEER performed best when tested on a workload with the same syntax as it was trained on, but when trained on queries with Syntax B and C in particular, we observed minimal decrease in performance regardless of the syntax used in the testing workload.

```
SELECT t.title AS movie, cn.
    country_code AS country
FROM title AS t INNER JOIN
    movie_companies AS mc ON t.id
     = mc.movie_id AND
    company_name AS cn INNER JOIN
     mc ON cn.id = mc.company_id
WHERE t.production_year > 2005;
```

Listing 1: Example SQL query performing joins using the INNER JOIN keyword.

```
SELECT t.title AS movie, cn.
    country_code AS country
FROM company_name AS cn,
     movie_companies AS mc,
     title AS t
WHERE t.production_year > 2005
     AND t.id = mc.movie_id
     AND cn.id = mc.company_id;
```

Listing 2: Example SQL query performing joins in the WHERE clause.

Unfortunately, our simplified approach did not scale. Considering PostgreSQL's 48 hint sets, there are too few queries associated with each class, making it challenging for a classifier to learn the complex relationship between queries and hints. The distribution of queries across the collection of hints is also skewed, with the most frequently optimal hint set occurring 525x more often than the least frequently optimal. Despite this, even in the absence of a more complex strategy, the ability to steer the optimizer between just two alternatives leads to significantly improved performance.

4 Conclusion and Future Work

In this extended abstract, we present LLMSTEER, demonstrating its usage in effectively steering query optimizers. Benchmarked against PostgreSQL's default query optimizer, results from initial experimentation show that LLMSTEER is capable of reducing total and tail latency by 72% on average. We were surprised to discover that LLMSTEER worked, since established wisdom of the database community indicates that the system should *not* have been successful. With this, we have far more questions than answers. Here we limit ourselves to highlighting some key considerations.[3]

Does the LLM matter? The quality of embeddings are often highly dependent on the downstream task and the LLM used. At the time of this work, OpenAI's `text-embedding-3-large` did not rank within the top 30 models on the overall massive text embedding benchmark (MTEB) (Muennighoff et al. [2022]). There may be models which can create richer representations of SQL queries, containing additional semantic information that may be helpful in steering optimizers. Moreover, can cheaper or smaller models be just as effective?

How robust is LLMSTEER to syntax changes? Further investigation into how syntax impacts performance is necessary. For example, assessing the effects of comma-first notation, inclusion of comments, formatting of keywords and identifiers (e.g., lowercase, uppercase, title case), and any combination of these modifications on the ability for models to learn and generalize is of practical importance. Obviously, being robust to simple semantic-preserving reformulations of queries is critical for any real-world deployment.

LLM Fine-Tuning Can an LLM be fine-tuned on the task of steering query optimizers? That is, can we teach an LLM to select the optimal hint given a query in a few-shot setting, or by fine-tuning an LLM on SQL directly, and would this prove to be more effective than LLMSTEER?

We are actively seeking collaborators for this work.

[3] An extended discussion can be found in the preprint of this work on arXiv.

References

C. Anneser, N. Tatbul, D. Cohen, Z. Xu, P. Pandian, N. Laptev, and R. Marcus. Autosteer: Learned query optimization for any sql database. *Proceedings of the VLDB Endowment*, 16(12):3515–3527, 2023.

L. Giakoumakis and C. A. Galindo-Legaria. Testing SQL Server's Query Optimizer: Challenges, Techniques and Experiences. *IEEE Data Eng. Bull.*, 31:36–43, 2008.

G. Graefe and W. J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Proceedings of the Ninth International Conference on Data Engineering*, ICDE '93, pages 209–218, Washington, DC, USA, 1993. IEEE Computer Society. ISBN 978-0-8186-3570-0. URL http://dl.acm.org/citation.cfm?id=645478.757691.

R. Heinrich, M. Luthra, H. Kornmayer, and C. Binnig. Zero-shot cost models for distributed stream processing. In *Proceedings of the 16th ACM International Conference on Distributed and Event-Based Systems*, DEBS '22. ACM, June 2022. doi: 10.1145/3524860.3539639. URL http://dx.doi.org/10.1145/3524860.3539639.

A. Kipf, T. Kipf, B. Radke, V. Leis, P. Boncz, and A. Kemper. Learned cardinalities: Estimating correlated joins with deep learning. *arXiv preprint arXiv:1809.00677*, 2018.

V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *Proceedings of the VLDB Endowment*, 9(3):204–215, 2015.

R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska. Bao: Making learned query optimization practical. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 1275–1288, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383431. doi: 10.1145/3448016.3452838. URL https://doi.org/10.1145/3448016.3452838.

N. Muennighoff, N. Tazi, L. Magne, and N. Reimers. Mteb: Massive text embedding benchmark. *arXiv preprint arXiv:2210.07316*, 2022. doi: 10.48550/ARXIV.2210.07316. URL https://arxiv.org/abs/2210.07316.

P. Negi, R. Marcus, A. Kipf, H. Mao, N. Tatbul, T. Kraska, and M. Alizadeh. Flow-loss: Learning cardinality estimates that matter. *arXiv preprint arXiv:2101.04964*, 2021.

P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In J. Mylopolous and M. Brodie, editors, *SIGMOD '79*, SIGMOD '79, pages 511–522, San Francisco (CA), 1979. Morgan Kaufmann. ISBN 978-0-934613-53-8. doi: 10.1016/B978-0-934613-53-8.50038-8. URL https://www.sciencedirect.com/science/article/pii/B9780934613538500388.

A. van Renen, D. Horn, P. Pfeil, K. E. Vaidya, W. Dong, M. Narayanaswamy, Z. Liu, G. Saxena, A. Kipf, and T. Kraska. Why TPC is not enough: An analysis of the Amazon Redshift fleet. *Proceedings of the VLDB Endowment*, 2024. URL https://www.amazon.science/publications/why-tpc-is-not-enough-an-analysis-of-the-amazon-redshift-fleet.

L. Woltmann, J. Thiessat, C. Hartmann, D. Habich, and W. Lehner. FASTgres: Making Learned Query Optimizer Hinting Effective. *Proceedings of the VLDB Endowment*, 16(11):3310–3322, Aug. 2023. ISSN 2150-8097. doi: 10.14778/3611479.3611528. URL https://dl.acm.org/doi/10.14778/3611479.3611528.

Z. Yi, Y. Tian, Z. G. Ives, and R. Marcus. Low rank approximation for learned query optimization. In *Proceedings of the Seventh International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, aiDM '24, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400706806. doi: 10.1145/3663742.3663974. URL https://doi.org/10.1145/3663742.3663974.

R. Zhu, L. Weng, W. Wei, D. Wu, J. Peng, Y. Wang, B. Ding, D. Lian, B. Zheng, and J. Zhou. PilotScope: Steering Databases with Machine Learning Drivers. *PVLDB*, 17(5):980–993, 2024. doi: 10.14778/3641204.3641209.