SERAG: Self-Evolving RAG System for Query Optimization

Hanwen Liu*
University of Southern California
Los Angeles, USA
hanwen_liu@usc.edu

Qihan Zhang*
University of Southern California
Los Angeles, USA
qihanzha@usc.edu

Ryan Marcus
University of Pennsylvania
Philadelphia, USA
rcmarcus@seas.upenn.edu

Ibrahim Sabek
University of Southern California
Los Angeles, USA
sabek@usc.edu

ABSTRACT

We present SERAG, a **S**elf-**E**volving **RAG** System for Query Optimization. SERAG mitigates the cold start problem of learned query optimizers (LQOs) while continuously learning from execution feedback via a Retrieval-Augmented Generation (RAG) system. SERAG's self-evolution dynamically constructs prompts for LLMs using previous execution records, which in turn generates an optimized query plan. Our preliminary experimental results indicate that query plans generated by SERAG take, on average, 36% less execution time than PostgreSQL's. Additionally, it outperforms LQOs and demonstrates robust generalization capabilities across workloads.

ACM Reference Format:
Hanwen Liu, Qihan Zhang, Ryan Marcus, and Ibrahim Sabek. 2025. SERAG: Self-Evolving RAG System for Query Optimization. In *International Workshop on Exploiting Artificial Intelligence Techniques for Data Management (aiDM '25), June 22–27, 2025, Berlin, Germany*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3735403.3735997

1 INTRODUCTION

Query optimization is a fundamental and performance-critical problem in database systems that focuses on translating declarative user queries into efficient query plans. Human experts take decades to design and improve classical query optimizers (e.g., PostgreSQL [22]). To accelerate optimizer development and improve query performance, several studies have applied different techniques to query optimization, such as machine learning [1, 15, 16, 35, 37, 38, 40] and large language models [26, 36]. While these *learned* query optimizers (LQOs) represent exciting work and have even seen some industry adoption [39], we argue that existing LQOs capture at most two of three desirable properties:

(1) An LQO should *learn from its mistakes*, quickly incorporating feedback from the previously generated query plans.
(2) An LQO should *work immediately*, without requiring extensive training (i.e., the "cold start" problem).
(3) An LQO should *adapt* to changes in schema, workload, and data, without significant human effort and retraining.

For example, the Bao [16] system learns from its mistakes and adapts to changes, but requires several hours of training before becoming competitive. The Fastgres system [33] does not suffer from the cold start problem and can learn from its mistakes, but it cannot adapt to changes in query workload without significant retraining. Two recent systems [26, 36] use fine-tuned LLMs as an oracle for either plan generation or plan selection. However, due to the high cost of fine-tuning an LLM, these approaches cannot learn from their mistakes or continuously improve.

Why is there no existing system with all the desired properties? Intuitively, LQOs that do not use LLMs tend to have below-par reasoning skills: they must simplify the problem by either requiring large amounts of training data (and therefore having a "cold start") or by assuming the schema and workload are fixed (and therefore not being able to adapt). LQOs that do use LLMs can work immediately and adapt to changes, but correcting their mistakes using current techniques is too costly for the critical path of a DBMS.

Here, we propose SERAG, a RAG-based query optimizer that has all three of the desired properties. SERAG uses an LLM to avoid the cold start problem and adapt to changes. To learn from its mistakes, SERAG uses a continuously updated vector database (VecDB) of past query executions that serve as the query optimizer's "memory." This paper presents our current (and actively under development) design for SERAG, along with a preliminary experimental evaluation.

SERAG's key innovation is taking advantage of *few-shot learning*, a capability of LLMs to generalize to new problems by inserting several examples into the LLMs input (the prompt or context window). These additional examples serve to help LLMs to generalize to new problems [14, 17, 21]. Critically, few-shot learning works best when the provided examples are close to the problem-at-hand. RAG [13] frameworks combine the strengths of robust retrieval systems (e.g., VecDBs) [31]) with the creative capabilities of LLMs.

A critical part of any RAG system is ensuring that the underlying VecDB is filled with relevant and useful information. Although most RAG systems solve this issue by manually prepopulating the VecDB (e.g., with relevant documents), SERAG populates that dynamically at runtime. When SERAG sees a query plan that is working well, that plan is stored in the VecDB to be retrieved and used later. Similarly, when SERAG sees a query plan that performs poorly, the plan is stored in the database to serve as a negative example for future optimization. By continuously expanding and refining its VecDB, SERAG learns to repeat its successes and avoid its failures.

Experimental results show that SERAG outperforms PostgreSQL and an LQO (Bao [16]), and its self-evolving paradigm enables LLMs to better solve query optimization problems.

We summarize our contributions as follows:

(1) To the best of our knowledge, we show for the first time that RAG systems can be used for query optimization.
(2) SERAG mitigates LQO's cold start issue and generalizes well across different workloads without retraining, while still continuously learning from execution feedback.

*Both authors contributed equally to this work.

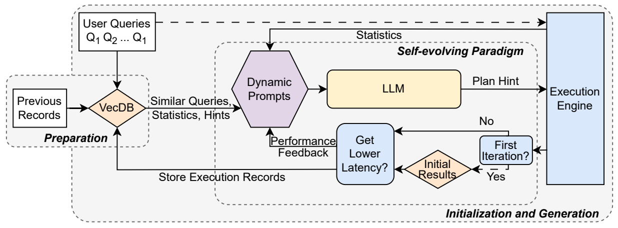


Figure 1: Overview of SERAG.

(3) Our preliminary experiments verify SERAG's promise, showing encouraging inference overhead and query performance.

2 PRELIMINARIES

Here, we briefly describe RAG, VecDBs, and LLMs, each of which is a critical component of SERAG. Due to space constraints, we are not able to present a complete discussion of these popular topics, so we refer the reader to various survey papers.

RAG. Retrieval-augmented generation [3] fetches external knowledge (e.g., from a VecDB) to augment an LLM's prompt without updating its weights. This method overcomes the static knowledge limits of standalone LLMs, although it requires careful text selection to fit within the context window of the model.

Vector Databases. VecDBs [20] store high-dimensional embeddings, which are numerical vectors representing the semantic content of raw data. They use specialized index structures to cluster similar vectors, enabling rapid similarity searches by comparing query vectors with stored ones using metrics like cosine similarity.

LLMs. Large language models are transformer-based architectures [6] with billions of parameters that excel at diverse NLP tasks [9, 10, 18]. Chain-of-thought training enhances their reasoning [32], enabling them to power question-answering systems.

3 OVERVIEW OF SERAG

The complete pipeline of SERAG consists of three stages: *Preparation*, *Initialization*, and *Generation*. Figure 1 shows the architecture of SERAG. During the *Preparation* stage, previous execution records are stored in a VecDB. For a user-submitted workload containing repeated queries (as in [30, 39]), the *Initialization* stage is the scenario when SERAG encounters a query Q for the first time (denoted by the dashed line): the query Q is sent to the database execution engine, and the initial results are stored in the VecDB. On subsequent encounters with query Q, it is first sent to the database engine to extract the necessary statistics S, including table cardinalities and filter cardinalities, without actual execution. In parallel, Q is sent to the VecDB to fetch k query examples by similarity search (see Section 3.1). Each query example contains the original query, the generated query plans (hints), and the corresponding database statistics (the same as the S we mention before). These query examples work as references while constructing the dynamic prompt (see Section 3.2). Then, the prompt is sent to an LLM to generate a query plan hint. The original query, along with the generated query hint, is executed by the database execution engine. The execution latency is compared with historical records, and this feedback is incorporated into the dynamic prompts to iteratively improve future plan generation (see Section 3.3).

3.1 Storage Design

For each workload, the VecDB stores 3 record collections [20] in an identical schema[1], denoted as C_p, C_i, and C_g, which correspond to the 3 stages in the pipeline: ***P**reparation*, ***I**nitialization*, and ***G**eneration*. During the preparation, C_p stores reference records that serve as query examples for LLM. This collection is static and remains unchanged during execution. In the initialization, C_i stores initial execution records as starting points. During pipeline running, C_g stores the execution records using the hint generated by SERAG.

Figure 2 shows the composition of collections. For each record, we maintain seven properties: `Id`, `Iteration`, `SQL_id`, `Vector`, `SQL`, `Hint`, and `Execution Time`. In C_p and C_i, the iteration is set to 0 to distinguish them from C_g, whose iterations start at 1. When new records come, our VecDB inserts them into the targeted collections using the layout described above. When extracting information, the given user query is first encoded as a fixed-length dense vector by *SentenceTransformer* [23]. Then, our VecDB fetches the k most related queries $Q_1, Q_2, \ldots, Q_k$ from collection C_p using cosine similarity [34]. Finally, these queries and plan hints are used to construct dynamic prompts.

3.2 Dynamic Prompt Design

Figure 3 shows the prompt structure, which comprises two components: the static system prompt SP and the dynamic user prompt UP. The SP specifies (1) the role of the LLMs (e.g., database experts), (2) the actions that the LLMs should perform, and (3) the constraints on these actions. The UP is dynamically constructed for each user query Q and consists of four parts. The first part of UP comprises k examples from k queries that are similar to Q (see Section 3.1); each example includes the query, database statistics, and the corresponding query plan. Notably, SERAG also supports not using any example as reference in the prompt, i.e., $k = 0$. The second part of UP provides information for the current Q: the SQL statement, database statistics, and the default query plan generated by the database engine. The third part of UP presents the best historical query plan generated by SERAG for Q and its corresponding performance gain relative to the default query plan. Note that this third part of UP serves as heuristics that enable the LLM to understand the current plan performance and generate an improved plan. Finally, the fourth part of UP adds expectations and regulations for LLMs, including (1) generating a better plan, (2) avoiding copying the current plans, and (3) ensuring output in the correct format.

3.3 Self-evolving Feedback Paradigm

SERAG's self-evolving feedback paradigm (depicted in the middle section of Figure 1) learns from historical records to enhance the LLM's current generation. This paradigm mainly focuses on the optimization of LLM's prompt *Prompt*. In Algorithm 1, for each query Q, we update the best historical execution plan $plan^*$ and its execution time t^* (lines 3–7). Then, its performance gain η over the initial execution time is computed to steer the LLM's behavior

[1]A collection in a VecDB is like a defined table in relational databases.

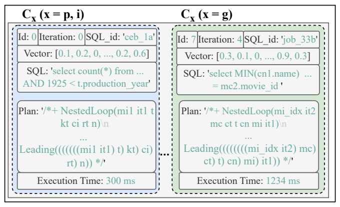
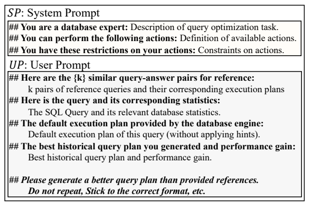

C_x (x = p, i)
C_x (x = g)
Id: 0 | Iteration: 0 | SQL_id: 'ceb_1a'
Vector: [0.1, 0.2, 0, ..., 0.2, 0.6]
SQL: 'select count(*) from ... AND 1925 < t.production_year'
Plan: '/*+ NestedLoop(mi1 it1 t kt ci rt n)\n ... Leading(((((((mi1 it1) t) kt) ci) rt) n)) */'
Execution Time: 300 ms
Id: 7 | Iteration: 4 | SQL_id: 'job_33b'
Vector: [0.3, 0.1, 0, ..., 0.9, 0.3]
SQL: 'select MIN(cn1.name) ... = mc2.movie_id '
Plan: '/*+ NestedLoop(mi_idx it2 mc ct t cn mi it1)\n ... Leading((((((((mi_idx it2) mc) ct) t) cn) mi) it1)) */'
Execution Time: 1234 ms

Figure 2: A Visual Example of Collection C_x $(x = p, i, g)$.


SP: System Prompt

## You are a database expert: Description of query optimization task.
## You can perform the following actions: Definition of available actions.
## You have these restrictions on your actions: Constraints on actions.

UP: User Prompt

## Here are the {k} similar query-answer pairs for reference:
k pairs of reference queries and their corresponding execution plans
## Here is the query and its corresponding statistics:
The SQL Query and its relevant database statistics.
## The default execution plan provided by the database engine:
Default execution plan of this query (without applying hints).
## The best historical query plan you generated and performance gain:
Best historical query plan and performance gain.

## Please generate a better query plan than provided references.
Do not repeat, Stick to the correct format, etc.

Figure 3: Dynamic Prompt in SERAG.

in the current generation. If the best historical execution time t^* exceeds the original execution time t_o, then $\eta = (t_o - t^*)/t_o < 0$, indicating that the best historical plan is worse than the original query plan, which serves as a negative example to offer feedback (line 9). Finally, SERAG merges this feedback information to obtain the evolved *Prompt* to prevent the LLM from falling into the same performance pitfall again (line 10). As the pipeline continuously runs, *Prompt* optimization drives the self-evolving process.

3.4 Supervised Fine-tuning for LLMs

To promote LLMs' understanding of hints formats, we apply supervised fine-tuning (SFT) [5] on local LLMs. In SFT, we first prepare a training set $\mathbb{D}_{\text{train}}$, which contains constructed pairs of prompts x and pre-generated hints $hint^*$ from the query optimizer. Then, for a given pre-trained LLM θ, we maximize the conditional predictive probability $p_\theta(hint^*|x)$ by minimizing the cross-entropy loss.

4 EVALUATION

We first introduce the experimental setup in Section 4.1. Then, we evaluate our SERAG using different benchmarks and multiple LLMs to address the following questions: (1) What is the performance gain over PostgreSQL and Bao [16] (Section 4.2)? (2) What is the performance of SERAG's variants (Section 4.3)? (3) What is the overhead of using RAG (Section 4.4)? (4) How effective is our self-evolving feedback paradigm? And What is the generalization ability of SERAG across workloads (Section 4.5)?

4.1 Experimental Setup

Environment. We run all our experiments on one server with an AMD Ryzen 9700X CPU, 128 GB of RAM, and NVIDIA A6000 GPU with 48GB memory. We set up PostgreSQL 16.4 with the corresponding *pg_hint_plan*. We use Milvus [31], three local LLMs (DeepSeek-Coder-Instruct-7B (DS-Coder), LLaMA3.1-8B and LLaMA3.2-3B's SFT version). We also use API calls to remotely access the GPT-o3-mini and DeepSeek-V3 Chat (DS-Chat) models. We warm up the machine before running the experiments to mitigate caching issues. The experiments are conducted iteratively. In each iteration, a complete query workload is used to process. Then, the same workload is used repeatedly across multiple iterations.

Evaluation Metrics. We focus on the following metrics: (1) *Performance Gain in Query Execution Latency:* To directly evaluate the quality of the generated hints, we measure the time t used by PostgreSQL, following generated hints, to construct a complete plan and execution. Then, we compute the performance gain η (same in Algorithm 1) over PostgreSQL default execution time t_o.

Algorithm 1 Prompt Evolving Paradigm in SERAG

```
1:  Init: User-submitted query $Q$, LLM's prompt $Prompt$
2:  $hint \leftarrow \text{LLM}(Prompt)$    ▷ Previous generated hint
3:  $plan, t \leftarrow \text{ExecEngine}(Q, hint)$    ▷ Previous execution record
4:  $plan^*, t^* \leftarrow GetBestHistoricalExecutionRecord(Q)$
5:  if $t < t^*$ then
6:      $plan^* \leftarrow plan$    ▷ Update the best historical execution plan
7:      $t^* \leftarrow t$    ▷ Update the best historical execution time
8:  $t_o \leftarrow \text{GetExecutionTimeFromInitializationStage}(Q)$
9:  Compute the performance gain of optimal record: $\eta \leftarrow \frac{t_o - t^*}{t_o}$
10: $Prompt \leftarrow \text{ConstrcutPrompt}(SP, UP, plan^*, \eta)$ ▷ From System
    prompt, User prompt, Current best plan, and Performance gain
```

(2) *Inference and RAG Retrieval Latency:* These metrics measure the latency overhead from LLM inference and extracting from VecDBs. (3) *Relative Execution Time (RET)*: We measure the overall execution time of all the queries in each iteration. For iteration i, we define $RET^{(i)} = \sum_j \hat{E}_j^{(i)} / \sum_j E_j^{(i)}$, where $\hat{E}_j^{(i)}$ and $E_j^{(i)}$ denote the execution time of query j on SERAG and PostgreSQL, respectively.

Benchmarks. Join-Order-Benchmark (JOB) [12] workload is used for generating query-answer examples stored in the vector database during the preparation stage. Then we evaluate SERAG using Cardinality Estimation Benchmark (CEB) [19] and Stack [16] workloads. We used 2 queries per template for CEB and 5 for Stack, with timeouts of 10s and 30s, respectively, to handle poor query plans.

SERAG's Variants and Generation Modes. We define three variants by how they incorporate examples (references) into the LLM's prompt. `PGRef` uses examples from PostgreSQL's execution plans. `LQORef` uses examples from a LQO (Balsa [35]), which provides higher-quality plans. `NoRef` does not use any examples. During the preparation stage, the execution records of the JOB are stored in the C_p collection of the vector database. Regarding the LLM outputs of SERAG, we have two modes: `Join Order` and `Complete Hint`. `Join Order` indicates that SERAG is asked to only output a join order format hint; in contrast, `Complete Hint` requires the LLM to generate a complete hint to reflect the details in the query plan.

4.2 Query Execution Latency Comparison

Table 2 shows the comparison among SERAG (with LLaMA3.1-3B SFT on JOB, `NoRef` variant), PostgreSQL, and an LQO (Bao [16], trained on JOB)[2]. SERAG outperforms both PostgreSQL and Bao on two benchmarks. Moreover, Bao shows a performance decrease when encountering a complex situation (transitioning from JOB to CEB); however, SERAG's performance on CEB is even better than on JOB (in `Join Order` mode). This comparison shows that SERAG performs well regardless of the benchmark used for fine-tuning.

4.3 Performance Gains of SERAG's Variants

In this experiment, we run all 3 variants of SERAG on the CEB workload for 50 iterations. We extract the optimal execution plan for each query to measure the performance gain relative to PostgreSQL. `Overall` gain denotes the improvement across all queries,

[2]The JOB training set used in SFT is different from the JOB test set used for evaluation.

Table 1: Overall and Filtered Performance Gains of SERAG ↑.

	SFT LLaMA3.2 3B				LLaMA3.1 8B				DS-Coder 7B			
Gen-Modes	Join Order		Complete Hint		Join Order		Complete Hint		Join Order		Complete Hint	
Variants	Overall	Filtered	Overall	Filtered	Overall	Filtered	Overall	Filtered	Overall	Filtered	Overall	Filtered
NoRef	63.62%	71.81%	45.31%	52.63%	12.84%	62.86%	9.31%	25.18%	0.63%	-17.29%	1.71%	18.79%
LQORef	20.09%	35.87%	36.80%	39.65%	10.93%	41.50%	39.26%	44.87%	-7.98%	-183.50%	1.36%	0.98%
PGRef	17.07%	62.21%	43.58%	48.04%	12.83%	52.93%	17.31%	38.57%	3.03%	42.59%	1.31%	4.96%

Table 2: Performance Gain over PostgreSQL ↑.

	JOB		CEB	
SERAG	Join Order	Comp. Hint	Join Order	Comp. Hint
(NoRef)	50.79%	62.32%	63.62%	45.31%
Bao	47.48%		22.60%	

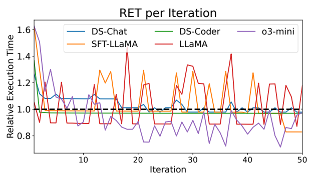


Figure 4: RET on CEB ↓.

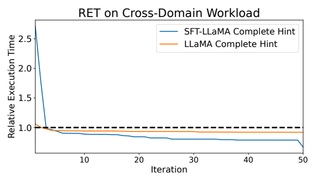


Figure 5: RET on Stack ↓.

whereas `Filtered` gain is computed only for queries in which SERAG generates query plans that differ from those of PostgreSQL.

Table 1 shows that the fine-tuned LLaMA3.2 outperforms the other two local LLMs, demonstrating that SFT is beneficial for LLMs to learn the hint format and thus perform better. Moreover, this model exhibits enhanced independent reasoning in the `NoRef` setting, even without requiring prepared records from PostgreSQL or LQO. For the complex task of generating a `complete hint` on vanilla models (Local LLaMA3 and DS-Coder), performance improves when additional references are provided (i.e., under the `LQORef` and `PGRef` configurations).

4.4 Inference and RAG Retrieval Time

We evaluate overheads incurred by the inference of LLM along with the RAG retrieval. We break down these overheads for the SFT LLaMA3.2-3B model in `NoRef` variant and `Join Order` generation mode, which performed best previously. The average retrieval time is 0.7 ms, and the inference time is approximately 236 ms on an NVIDIA A6000 and 160 ms on an NVIDIA A100. The dominant overhead arises from the inference time, which can be mitigated by using powerful GPUs. Furthermore, when considering execution time savings - where SERAG's query plans for the CEB workload average 520.67 ms compared to PostgreSQL's 922.86 ms) - SERAG's end-to-end time remains superior.

4.5 Self-evolution and Generalization

Self-evolving Feedback Evaluation. Figure 4 shows the *RET* metric when using five different LLMs with SERAG across 50 iterations. We use the `NoRef` variant of SERAG and set the generation mode as `Join Order`. GPT-o3-mini, SFT-LLaMA, and vanilla LLaMA models show a fluctuating performance over iterations. As observed, SFT-LLaMA ultimately surpasses the PostgreSQL baseline (the dashed line) at late iterations (around 50), benefiting from SFT. The performance gain is even more evident in GPT-o3-mini, which begins converging toward optimal performance as early as iteration 20. In contrast, the vanilla LLaMA's performance fails to converge (its best *RET* is 0.87). As for the DS-Chat and DS-Coder models, the performance gains (with $RET = 0.97$ and $RET = 0.96$, respectively) are quite minor, indicating that they are less sensitive to the multiple-iteration feedback. We derive additional insights: (1) Even when the LLM is closed-sourced (e.g., GPT-o3-mini), SERAG's feedback paradigm can still steer its performance and achieve self-evolving. (2) General LLMs (LLaMA3) and reasoning LLMs (GPT-o3-mini) exhibit superior performance compared to task-specific LLMs (DS-Coder and DS-Chat). They are also less dependent on high-quality examples in the given prompt.

Cross Workload Generalization. To evaluate SERAG's generalization across workloads and avoid scenarios where the test workload (CEB) is similar to JOB (used for LLM fine-tuning), we perform experiments on Stack, which is entirely different from JOB. We configure SERAG using the `NoRef` variant and `Complete Hint` generation mode. Figure 5 shows the self-evolving process of SERAG. With our self-evolving paradigm, all LLMs perform better on Stack. Notably, the fine-tuned LLaMA model exhibits decreasing execution times over iterations, demonstrating that fine-tuning enables the LLM to systematically understand the query optimization task and facilitate continuous learning on new workloads.

5 RELATED WORK

LLMs have demonstrated superior performance in numerous tasks within DBMS (e.g., knob tuning [4, 8, 11, 27], Text-to-SQL [2, 7], code generation [29], and schema understanding [28]). Recently, several systems have begun integrating RAG to perform SQL generation [24] and query rewrite [25]. While two recent works [26, 36] employ LLMs for query optimization, to the best of our knowledge, we are the first to integrate RAG for the query optimization.

6 CONCLUSION

This paper proposes SERAG, a self-evolving RAG system for query optimization. It leverages LLMs' pre-trained knowledge to avoid cold starts and enhance generalization across workloads. SERAG dynamically generates prompts and enables continuous evolution via execution feedback. Future work includes improving LLM generation efficiency and stability to better solve query optimization.

REFERENCES

[1] Lyric Doshi, Vincent Zhuang, Gaurav Jain, et al. 2023. Kepler: Robust Learning for Parametric Query Optimization. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–25.

[2] Dawei Gao, Haibin Wang, Yaliang Li, et al. 2024. Text-to-SQL Empowered by Large Language Models: A Benchmark Evaluation. *Proceedings of the VLDB Endowment* 17, 5 (Jan. 2024), 1132–1145.

[3] Yunfan Gao, Yun Xiong, Xinyu Gao, et al. 2023. Retrieval-augmented Generation for Large Language Models: A survey. *arXiv preprint arXiv:2312.10997* 2 (2023).

[4] Victor Giannakouris and Immanuel Trummer. 2024. Demonstrating λ-tune: Exploiting Large Language Models for Workload-adaptive Database System Tuning. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD.* 508–511.

[5] Beliz Gunel, Jingfei Du, Alexis Conneau, and Ves Stoyanov. 2020. Supervised Contrastive Learning for Pre-trained Language Model Fine-tuning. *arXiv preprint arXiv:2011.01403* (2020).

[6] Kai Han, An Xiao, Enhua Wu, et al. 2021. Transformer in transformer. *Advances in neural information processing systems* 34 (2021), 15908–15919.

[7] Zijin Hong, Zheng Yuan, Qinggang Zhang, et al. 2025. Next-Generation Database Interfaces: A Survey of LLM-based Text-to-SQL. arXiv:2406.08426

[8] Xinmei Huang, Haoyang Li, Jing Zhang, et al. 2024. Llmtune: Accelerate database knob tuning with large language models. *arXiv preprint arXiv:2404.11581* (2024).

[9] Juyong Jiang, Fan Wang, Jiasi Shen, et al. 2024. A Survey on Large Language Models for Code Generation. arXiv:2406.00515

[10] Ehsan Kamalloo, Nouha Dziri, Charles L. A. Clarke, and Davood Rafiei. 2023. Evaluating Open-Domain Question Answering in the Era of Large Language Models. arXiv:2305.06984

[11] Jiale Lao, Yibo Wang, Yufei Li, et al. 2023. Gptuner: A Manual-reading Database Tuning System via GPT-guided Bayesian Optimization. *arXiv preprint arXiv:2311.03157* (2023).

[12] Viktor Leis, Andrey Gubichev, Atanas Mirchev, et al. 2015. How good are query optimizers, really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215.

[13] Patrick Lewis, Ethan Perez, Aleksandra Piktus, et al. 2021. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. arXiv:2005.11401

[14] Ben Mann, N Ryder, M Subbiah, et al. 2020. Language Models are Few-shot Learners. *arXiv preprint arXiv:2005.14165* 1 (2020), 3.

[15] Ryan Marcus, Parimarjan Negi, Hongzi Mao, et al. 2019. Neo: A Learned Query Optimizer. *Proceedings of the VLDB Endowment* 12, 11 (jul 2019), 1705–1718.

[16] Ryan Marcus, Parimarjan Negi, Hongzi Mao, et al. 2021. Bao: Making Learned Query Optimization Practical. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD.* 1275–1288.

[17] Sewon Min, Mike Lewis, Luke Zettlemoyer, and Hannaneh Hajishirzi. 2022. MetaICL: Learning to Learn In Context. arXiv:2110.15943

[18] Humza Naveed, Asad Ullah Khan, Shi Qiu, et al. 2023. A Comprehensive Overview of Large Language Models. *arXiv preprint arXiv:2307.06435* (2023).

[19] Parimarjan Negi, Ryan Marcus, Andreas Kipf, et al. 2021. Flow-Loss: Learning Cardinality Estimates That Matter. *Proceedings of the VLDB Endowment* 14, 11 (jul 2021), 2019–2032.

[20] James Jie Pan, Jianguo Wang, and Guoliang Li. 2024. Survey of Vector Database Management Systems. *The VLDB Journal* 33, 5 (2024), 1591–1615.

[21] Ethan Perez, Douwe Kiela, and Kyunghyun Cho. 2021. True Few-Shot Learning with Language Models. In *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, et al. (Eds.), Vol. 34. 11054–11070.

[22] PostgreSQL. [n. d.]. https://www.postgresql.org/.

[23] Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence Embeddings Using Siamese Bert-networks. *arXiv preprint arXiv:1908.10084* (2019).

[24] Zhili Shen, Pavlos Vougiouklis, Chenxin Diao, et al. 2024. Improving Retrieval-augmented Text-to-SQL with AST-based Ranking and Schema Pruning. *arXiv preprint arXiv:2407.03227* (2024).

[25] Zhaoyan Sun, Xuanhe Zhou, and Guoliang Li. 2024. R-Bot: An LLM-based Query Rewrite System. *arXiv preprint arXiv:2412.01661* (2024).

[26] Jie Tan, Kangfei Zhao, Rui Li, et al. 2025. Can Large Language Models Be Query Optimizer for Relational Databases? *arXiv preprint arXiv:2502.05562* (2025).

[27] Immanuel Trummer. 2022. DB-BERT: a Database Tuning Tool that" Reads the Manual". In *Proceedings of the ACM International Conference on Management of Data, SIGMOD.* 190–203.

[28] Immanuel Trummer. 2024. Generating Succinct Descriptions of Database Schemata for Cost-Efficient Prompting of Large Language Models. *Proceedings of the VLDB Endowment* 17, 11 (July 2024), 3511–3523.

[29] Immanuel Trummer. 2025. Generating Highly Customizable Python Code for Data Processing with Large Language Models. *The VLDB Journal* 34, 2 (2025).

[30] Alexander van Renen, Dominik Horn, Pascal Pfeil, et al. 2024. Why TPC is Not Enough: An Analysis of the Amazon Redshift Fleet. *Proceedings of the VLDB Endowment* 17, 11 (July 2024), 3694–3706.

[31] Jianguo Wang, Xiaomeng Yi, Rentong Guo, et al. 2021. Milvus: A Purpose-built Vector Data Management System. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD.* 2614–2627.

[32] Jason Wei, Xuezhi Wang, Dale Schuurmans, et al. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. arXiv:2201.11903

[33] Lucas Woltmann, Jerome Thiessat, Claudio Hartmann, et al. 2023. FASTgres: Making Learned Query Optimizer Hinting Effective. *Proceedings of the VLDB Endowment* 16, 11 (July 2023), 3310–3322.

[34] Peipei Xia, Li Zhang, and Fanzhang Li. 2015. Learning Similarity with Cosine Similarity Ensemble. *Information sciences* 307 (2015), 39–52.

[35] Zongheng Yang, Wei-Lin Chiang, Sifei Luan, et al. 2022. Balsa: Learning a Query Optimizer without Expert Demonstrations. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD.* 931–944.

[36] Zhiming Yao, Haoyang Li, Jing Zhang, et al. 2025. A Query Optimization Method Utilizing Large Language Models. *arXiv preprint arXiv:2503.06902* (2025).

[37] Xiang Yu, Chengliang Chai, Guoliang Li, and Jiabin Liu. 2022. Cost-based or Learning-based? A Hybrid Query Optimizer for Query Plan Selection. *Proceedings of the VLDB Endowment* 15, 13 (2022), 3924–3936.

[38] Xiang Yu, Guoliang Li, Chengliang Chai, and Nan Tang. 2020. Reinforcement Learning with Tree-lstm for Join Order Selection. In *Proceedings of the International Conference on Data Engineering, ICDE.* IEEE, 1297–1308.

[39] Wangda Zhang, Matteo Interlandi, Paul Mineiro, et al. 2022. Deploying a Steered Query Optimizer in Production at Microsoft. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD.* 2299–2311.

[40] Rong Zhu, Wei Chen, Bolin Ding, et al. 2023. Lero: A Learning-to-rank Query Optimizer. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1466–1479.