SEFRQO: A Self-Evolving Fine-Tuned RAG-Based Query Optimizer

HANWEN LIU*, University of Southern California, USA
QIHAN ZHANG*, University of Southern California, USA
RYAN MARCUS, University of Pennsylvania, USA
IBRAHIM SABEK, University of Southern California, USA

Query optimization is a crucial problem in database systems that has been studied for decades. Learned query optimizers (LQOs) can improve performance over time by incorporating feedback; however, they suffer from cold-start issues and often require retraining when workloads shift or schemas change. Recent LLM-based query optimizers leverage pre-trained and fine-tuned LLMs to mitigate these challenges. Nevertheless, they neglect LLMs' in-context learning and execution records as feedback for continuous evolution.

In this paper, we present SEFRQO, a **S**elf-**E**volving **F**ine-tuned **R**AG-based **Q**uery **O**ptimizer. SEFRQO mitigates the cold-start problem of LQOs by continuously learning from execution feedback via a Retrieval-Augmented Generation (RAG) framework. We employ both supervised fine-tuning and reinforcement fine-tuning to prepare the LLM to produce syntactically correct and performance efficient query hints. Moreover, SEFRQO leverages the LLM's in-context learning capabilities by dynamically constructing prompts with references to similar queries and the historical execution record of the same query. This self-evolving paradigm iteratively optimizes the prompt to minimize query execution latency. Evaluations show that SEFRQO outperforms state-of-the-art LQOs, achieving up to 65.05% and 93.57% reductions in query latency on the CEB and Stack workloads, respectively, compared to PostgreSQL.

CCS Concepts: • **Information systems → Query optimization**.

Additional Key Words and Phrases: Query Optimization; Large Language Model; Database Management System; Retrieval Augmented Generation

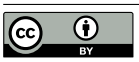
ACM Reference Format:
Hanwen Liu, Qihan Zhang, Ryan Marcus, and Ibrahim Sabek. 2025. SEFRQO: A Self-Evolving Fine-Tuned RAG-Based Query Optimizer. *Proc. ACM Manag. Data* 3, 6 (SIGMOD), Article 361 (December 2025), 27 pages. https://doi.org/10.1145/3769826

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 [37]). To try and accelerate optimizer development and improve query performance, several studies have applied different techniques to query optimization, such as machine learning [9, 28, 29, 58, 60, 62, 66] and large language models [47, 59].

*Both authors contributed equally to this work.

Authors' Contact Information: 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.



ACM 2836-6573/2025/12-ART361
https://doi.org/10.1145/3769826

While these *learned* query optimizers (LQOs) represent exciting work and have seen some industry adoption [65], we argue that existing LQOs capture at most two of three desirable properties:

(1) An LQO should *learn from its mistakes*, quickly and actively incorporating feedback from 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.

For example, the Bao [28] system learns from its mistakes and adapts to changes, but requires several hours of training data before becoming competitive. The Fastgres system [56] 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. Three recent systems applying LLMs (or their components) to query optimization [2, 47, 59] employ 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 immediately learn from their mistakes or continuously improve during runtime.

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.

In this paper, we propose SEFRQO[1], a RAG-based query optimizer that has all three of our desired properties. SEFRQO uses an LLM to avoid the cold start problem and adapt to changes. To learn from its mistakes, SEFRQO uses a continuously-updated vector database of past query executions that serve as the query optimizer's "memory". SEFRQO generates a hint-based query plan to guide the query optimizer in a classical database system (e.g., PostgreSQL [37]), where such a hint is optimized and generated using an LLM. SEFRQO relies on two complementary phases to optimize the hint generation process: *offline LLM fine-tuning* and *online LLM in-context learning*. Since these techniques have higher overhead than traditional query optimizers, SERFQO targets long-running OLAP queries where performance gains can outweigh increased optimization overhead.

To prepare a fine-tuned LLM capable of generating correctly formatted and efficient query plan hints, SEFRQO adopts a combined supervised fine-tuning (SFT) [12] and reinforcement fine-tuning (RFT) [54] technique. Using SFT, SEFRQO teaches the LLM the expected hint format. For RFT, SEFRQO introduces an efficient Query Group Relative Policy Optimization (Q_{GRPO}) approach, based on GRPO [43], to steer the LLM's output preferences toward hints that reduce query execution latency. Q_{GRPO} defines a latency-based reward model that interacts with the database execution engine to directly measure the corresponding latency of the generated query hint. Besides, Q_{GRPO} performs an in-group comparison among multiple hints generated for the same query; it obviates the need for additional labeled datasets as in other RFT algorithms, such as DPO (e.g. [47]). By not relying on a limited labeled dataset, Q_{GRPO}'s fine-tuning encourages LLMs' self-exploration, which further inspires their reasoning and generalization capabilities to handle previously unseen workloads or shifting database schemas.

Another key innovation of SEFRQO lies in leveraging online LLM RAG-based in-context learning, in which LLMs adapt to new tasks by conditioning on a few relevant RAG-based examples that are retrieved and provided directly within the input context (i.e., the prompt) without modifying model

[1]Source code available in https://github.com/ihanwen99/SEFRQO.

parameters. These examples allow the model to better understand tasks and generate more appropriate outputs [1, 3, 55, 57]. Importantly, in-context learning is most effective when the retrieved examples closely align with the target query. To facilitate this, SEFRQO builds and maintains a special vector database of the most relevant query execution records. Additionally, SEFRQO introduces a self-evolving feedback paradigm that enables the LLM to learn from historical execution records of the same query. The historical best plan generated by SEFRQO and its corresponding performance gain over the baseline provide additional contextual information. These components iteratively optimize the prompt, motivating the LLM to generate an optimized hint that minimizes the execution latency.

Our experimental results show that SEFRQO consistently outperforms both PostgreSQL and two state-of-the-art LQOs, namely Bao [28] and Balsa [58]. Our evaluations are based on three workloads: JOB [22], CEB [34], and Stack [28]. On CEB, SEFRQO achieves a 65.05% reduction in query execution time compared to PostgreSQL, and outperforms LQOs. On Stack, it can reduce up to 93.57% of the execution time for certain queries and 40.10% for overall performance. By leveraging a dynamic prompt with a self-evolving paradigm and successful fine-tuning, SEFRQO demonstrates surprising cross-domain capability on the previously unseen Stack workload, even surpassing LQOs directly trained on that workload in some metric. SEFRQO also supports flexible workload shifts that current LQOs find challenging. SEFRQO also supports the cross-DBMS scenario. On MySQL, it achieves a 26.28% reduction in execution time without requiring extra fine-tuning. Additionally, we observe that SEFRQO can learn from the provided query plan at the sub-plan level and generate an improved query plan through further reasoning and analysis. Furthermore, we conduct overhead analysis and ablation studies to evaluate several configurations. Finally, we conduct deeper investigations across various scenarios to test robustness and effects.

We summarize our contributions as follows:

(1) We are the first to formally define hint-based query optimization with LLMs as a two-phase process: offline LLM fine-tuning and online in-context learning.
(2) We propose an efficient supervised fine-tuning workflow to teach LLMs the hint format using a labeled dataset.
(3) We introduce Query Group Relative Policy Optimization that incorporates a latency-based reward model based on actual query execution and group relative advantage feedback, enhancing LLM reasoning in unseen scenarios.
(4) We present RAG-based prompt optimization, which leverages similar references and historical feedback to dynamically construct prompts and drive continuous self-evolution of SEFRQO. To our knowledge, this is the first application of RAG systems to query optimization.
(5) Our experiments demonstrate that SEFRQO outperforms both LQOs and PostgreSQL and generalizes effectively across multiple workloads and DBMSes without retraining.

2 Preliminaries

LLMs are transformer-based models [14] with billions of parameters that generalize well across a wide range of tasks (e.g., question-answering [19] and code generation [18]). During the pre-training stage, LLMs learn from a vast corpus of knowledge and capture general understanding capabilities. To address the query optimization task, we exploit the LLM to generate a hint-based query plan, and to adapt the LLM to this specific task, we rely on fine-tuning and RAG-based prompting procedures. First, we provide a brief background on the hint-based query optimization process (Section 2.1). Then, we describe how the LLM's generation (inference) process works (Section 2.2). Finally, we provide an overview of the two main LLM fine-tuning paradigms, namely supervised fine-tuning

(Section 2.3) and reinforcement fine-tuning (Section 2.4), as well as the RAG retrieval workflow (Section 2.5) we use in SEFRQO.

2.1 Hint-based Query Plan Generation

Hint-based plan generation is a commonly employed method in many learned query optimizers (LQOs) (e.g., [58, 66]). For example, LQOs (e.g., [58]) that are integrated with PostgreSQL [38] employ *pg_hint_plan* [35] to tweak the PostgreSQL [38] execution plans by incorporating "hint" in SQL comments. The hints support both a join-order mode (simple mode) and a full-plan mode (complete mode). In join-order mode, the hint is represented by a clause beginning with "`Leading`" to specify the join order of relations. For example, the hint: `Leading (a (b c))` indicates that $b \bowtie c$ is performed first, followed by $a \bowtie (b \bowtie c)$. Then, PostgreSQL will decide the detailed operators plan (e.g., operator type for each join). In full-plan mode, the hint conveys additional information, such as the scan types for tables (e.g., "`IndexScan`", "`SeqScan`") and the join types between tables (e.g., "`HashJoin`", "`NestedLoop`"). This hint format is interpreted by PostgreSQL to precisely control query planning and execution. An example of a full-plan mode hint is shown in the following section (see the third part in Figure 2). In our paper, we employ both modes as different query plan generation settings. Once a hint is accepted by the query optimizer, it is turned into a query plan and subsequently used for actual execution.

2.2 LLM Generation

Prompts are commonly used to interact with LLMs. A prompt is a piece of text that guides the model in generating more content. It may include a `System Prompt`, which defines the model's role or context (e.g., "You are a database expert performing query optimization task"), and a `User Prompt`, which specifies particular questions or tasks (e.g., "Generate a query plan for this query:" followed by detailed query information). When processing this natural-language prompt, the tokenizer separates the raw text into words and converts them into a sequence of discrete tokens. Each token is then mapped to an integer according to the model's vocabulary, after which the LLM can begin processing. The most fundamental input can be represented as a sequence of tokens $\mathbf{x} = [x_1, \ldots, x_n]$.

LLMs often adopt an auto-regressive approach to generate response sequences [14]. Their inference processes are typically based on next-token prediction. Specifically, the original input tokens and the previously generated output tokens from the previous steps are fed into the model to generate a new token at the current step. Assume an input sequence $\mathbf{x}$ is sent to an LLM parameterized by θ. The corresponding output token sequence can be expressed as $\mathbf{y} = [y_1, \ldots, y_T]$, containing T tokens. This token sequence $\mathbf{y}$ is then converted back to natural language by the tokenizer. In our context, $\mathbf{y}$ represents a query hint (See Section 2.1). Mathematically, the LLM's generation process can be expressed by the probability $p_\theta(\mathbf{y} \mid \mathbf{x})$, which factorizes into the product of conditional probabilities for each token y_t in the output sequence:

$$p_\theta(\mathbf{y} \mid \mathbf{x}) \;=\; p_\theta(y_1, y_2, \ldots, y_T \mid \mathbf{x}) \;=\; \prod_{t=1}^{T} p_\theta\big(y_t \,\big|\, \mathbf{x},\, y_{1:t-1}\big) \tag{1}$$

For clarity in the rest of the paper, we simplify the token-related notions and hide the tokenizer workflow in the LLM generation process by describing the whole process at the natural-language level. We use the notation x instead of $\mathbf{x}$ to represent the *prompt*, and use h instead of $\mathbf{y}$ to directly denote the *generated hint*.

2.3 Supervised Fine-tuning for LLMs

Supervised fine-tuning (SFT) [12] adjusts the parameters of LLMs by training them on high-quality, labeled data to enhance their ability to perform specific tasks. To improve LLM performance in our hint-based query plan generation task, a training set $\mathbb{D}_{\text{sft}}$ comprising pairs of prompt x and corresponding pre-generated hint output h^* needs to be constructed (We describe the details in Section 4). Here, h^* denotes the reference hint, distinguishing it from the LLM-generated hint h. During supervised fine-tuning, we maximize the conditional predictive probability $p_\theta(h^* \mid \mathrm{x})$ by minimizing the expectation $\mathbb{E}$ of the negative log-likelihood:

$$\mathcal{L}_{\text{sft}}(\theta) = \mathbb{E}_{(\mathrm{x},h^*)\sim\mathbb{D}_{\text{sft}}}\left[-\log p_\theta(h^* \mid \mathrm{x})\right] \tag{2}$$

Using SFT, the LLM is expected to become familiar with our task, i.e., learn the format of the query hint output and internal logic for generating an effective query plan.

2.4 Reinforcement Fine-tuning for LLMs

Reinforcement fine-tuning (RFT) provides an alternative method for enhancing the specific capabilities of LLMs by employing reinforcement learning to update their parameters. RFT has been proven effective in further improving the reasoning abilities of LLMs after the SFT stage (e.g., [54]). In our task, reasoning ability reflects an improved generalization capability when *handling previously unseen queries*. This implies that, in cross-domain scenarios (e.g., switching between completely different query workloads, such as Stack [28] and CEB [34]), LLMs with RFT are expected to generate hints indicating better query plans than those without RFT. Several RFT algorithms have been developed, such as Proximal Policy Optimization (PPO) [42], Direct Preference Optimization (DPO) [39], and the recently proposed Group Relative Policy Optimization (GRPO) [43]. During fine-tuning, PPO relies on a trained value model to objectively determine whether the generated output is `preferred` or `non_preferred`. This value model introduces additional memory requirements and computational overhead [43]. While DPO eliminates the need for a separate value model to reduce complexity, it requires paired preference data (consisting of `preferred` and `non_preferred` output) to guide the LLM's optimization process. Constructing such a preference-pair dataset before fine-tuning introduces an additional workload for data labeling, which is complex for our query optimization task. Moreover, although DPO increases the relative probability of preferred completions compared to dispreferred ones, it may reduce the model's absolute likelihood of generating preferred completions [36].

To avoid these limitations, we extend based on GRPO [43], a variant of PPO that enhances computational efficiency and robustness. GRPO incorporates group-based comparisons among generated responses to eliminate the need for an explicit value model, as required by PPO, and also removes the need for labeled data, as in DPO. GRPO generates multiple outputs from the same prompt, computes rewards for these outputs based on the user-defined reward function, and then evaluates their in-group advantages $\hat{A}$. Specifically, the GRPO algorithm first fixes the current policy model[2] as $\pi_{\theta_{\text{old}}}$. This naming distinguishes it from the policy model to be updated, π_θ. Then, for each prompt x in the GRPO training dataset $\mathbb{D}_{\text{GRPO}}$, an LLM generates a group of G hint plans $\{h_1, h_2, \ldots, h_G\}$ based on $\pi_{\theta_{\text{old}}}$. Then, these hints are executed with a database engine, and their corresponding execution latencies are used to optimize the policy model π_θ by maximizing the

[2]RFT treats the prompt as the environment state and each token generation as an action. So we use the reinforcement learning policy model π_θ notation rather than p_θ.

following objective:

$$\mathcal{J}_{\text{GRPO}}(\theta) \approx \mathbb{E}_{\text{x}\sim\mathbb{D}_{\text{GRPO}},\{h_i\}\sim\pi_{\theta_{\text{old}}}} \frac{1}{G}\sum_{i=1}^{G}\left\{ f\left(\frac{\pi_\theta(h_i \mid \text{x})}{\pi_{\theta_{\text{old}}}(h_i \mid \text{x})},\, \epsilon\right)\hat{A}_i - \beta\,\mathbb{D}_{\text{KL}}[\pi_\theta \| \pi_{\text{ref}}] \right\} \tag{3}$$

where function f and hyperparameter ϵ are employed to stabilize fine-tuning [42]. To prevent the updated policy π_θ from excessively deviating from the initial reference policy π_{ref}, a Kullback–Leibler (KL) divergence regularization [20, 43] term weighted by hyperparameter β is applied.

2.5 Retrieval via Embedding Similarity

Retrieval-augmented generation (RAG) [46] enhances large language models (LLMs) by incorporating external knowledge retrieved from a vector database [53], thereby supplementing the prompt with relevant information without altering the model's weights. This approach effectively mitigates the limitations imposed by the static knowledge of standalone LLMs. Consequently, the selection of the most relevant content from the vector database is critical to performance. To facilitate finer-grained control, the retrieved content can be segmented into smaller, semantically meaningful snippets when necessary [27, 40]. In our context, each snippet corresponds to a reference query d_i, and we define the full set of reference queries as follows: $\mathcal{D} = \{d_1, d_2, \ldots, d_N\}$. For each reference query d_i, we use an encoder $\phi(\cdot)$ to map it into a m-dimensional embedding space as: $\mathbf{e}_i = \phi(d_i) \in \mathbb{R}^m$. All embeddings can be organized into a matrix: $\mathbf{E} = [\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_N] \in \mathbb{R}^{m\times N}$. Given an query $\mathbf{u}$ for vector database, its embedding is computed as: $\mathbf{e}_\mathbf{u} = \phi(\mathbf{u}) \in \mathbb{R}^m$. The similarity between $\mathbf{u}$ and each d_i can be computed using cosine similarity as follows:

$$\text{sim}(\mathbf{u}, d_i) = \frac{\mathbf{e}_\mathbf{u}^\top \mathbf{e}_i}{\|\mathbf{e}_\mathbf{u}\|\,\|\mathbf{e}_i\|}. \tag{4}$$

Based on these similarity scores, we rank and pick the top k references to form the retrieved set $\mathcal{R}$, where: $\mathcal{R} \subset \mathcal{D}, |\mathcal{R}| = k$.

3 System Overview

In this section, we first formalize the definition of optimizing queries using SEFRQO (Section 3.1). Then, we give an overview of the SEFRQO's workflow and its key components (Section 3.2).

3.1 Problem Statement

Our ultimate goal is to reduce query execution latency using hints generated by LLMs. We achieve this through two complementary phases: *offline fine-tuning for the LLM abilities to generate hints*, and *online in-context learning for the LLM to generate optimal hints*.

Definition 1: Offline LLM Fine-tuning for Hints Generation. In this phase, we aim to fine-tune an LLM, parameterized by θ_{LLM}, such that, for any input SQL query $Q \in \boldsymbol{Q}$ and the relevant database statistics `stats` that help plan the execution of this query (e.g., table cardinalities), the LLM is able to generate a hint h that optimizes the execution of Q, with the objectives of (1) ensuring correct hint syntax (valid to become a plan) and (2) steering the LLM's preference towards generating hints that minimize query execution latency. These objectives can be formally defined as:

$$\arg\max_{\theta_{\text{LLM}}} \mathbb{E}_{\boldsymbol{Q},\text{stats}}\left[\log p_{\text{correct}}(h)\right] \qquad \text{(Syntax Correctness)} \tag{5}$$

$$\arg\min_{\theta_{\text{LLM}}} \mathbb{E}_{\boldsymbol{Q},\text{stats}}\left[\text{Latency}(Q, h)\right] \qquad \text{(Latency Preference)} \tag{6}$$

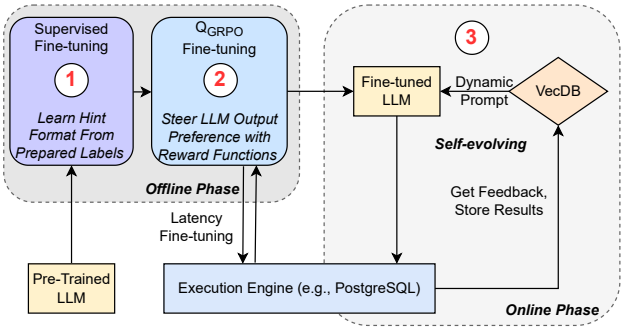


Fig. 1. System overview of SEFRQO.

where $\mathbb{E}_{Q,\text{stats}}$ denotes the expectation over the joint data distribution of input SQL queries Q and database statistics `stats`. This offline fine-tuning phase is achieved using SFT (see Section 2.3) and GRPO (see Section 2.4) as shown later.

Definition 2: Online LLM In-context Learning for Hints Generation. In this phase, for any input SQL query $Q \in \mathcal{Q}$ and its relevant database statistics `stats` mentioned in Definition 1, we aim to construct an optimized prompt x^* (noted as P in this section), during the runtime, that guides the fixed-parameter LLM to generate a hint h that minimizes the execution latency of this query (i.e., $h \leftarrow \text{LLM}_{\theta_{\text{LLM}}}(P)$). Formally, this corresponds to solving the following optimization problem:

$$\arg\min_{P} \text{Latency}(Q, h) \qquad \text{(Prompt Optimization)} \tag{7}$$

The optimized prompt P for any query is constructed based on the database statistics `stats` of this query and a few query references $\mathcal{K}$ that are dynamically retrieved from an up-to-date RAG vector database (see Section 2.5), as shown later.

3.2 Overview of SEFRQO

Figure 1 shows the overview of SEFRQO, which comprises three stages, each mapping to an optimization problem in Section 3.1. Corresponding to Equation 5, the first stage is **Supervised Fine-Tuning (SFT)** (see Section 4). SFT teaches the model the correct format of query hints by training LLMs on a pre-constructed dataset containing pairs of input prompts (user prompt as in Section 2.2) and output hints (training set $\mathbb{D}_{\text{sft}}$ in Section 2.3). SFT lays the foundation of our system because any further improvements depend on LLMs' ability to generate syntactically correct hints that a database execution engine (e.g., PostgreSQL) can accept. Corresponding to Equation 6, the second stage is **Query Group Relative Policy Optimization (Q_{GRPO}) fine-tuning** (see Section 5). In Q_{GRPO}, we incorporate latency feedback from actual query executions using the database engine (training dataset $\mathbb{D}_{\text{GRPO}}$ in Section 2.4). By designing a latency-based reward model and employing group relative advantage feedback, we steer LLMs' preferences toward generating hints with lower latency. As a reinforcement-learning process in a dynamic environment, Q_{GRPO} enhances reasoning ability for unseen workloads. These first two stages occur during the offline phase. Corresponding to Equation 7, the third stage is a **RAG-based Prompt Optimization** online phase, where the output prompt will be used to generate the final hint used for the query optimization. In this phase, we maintain a continuously updated RAG system incorporating live execution records (see Section 6). For each query Q, we dynamically construct a prompt containing execution records of similar queries as references and the historical execution analysis (see Section 6.4). Upon repeated

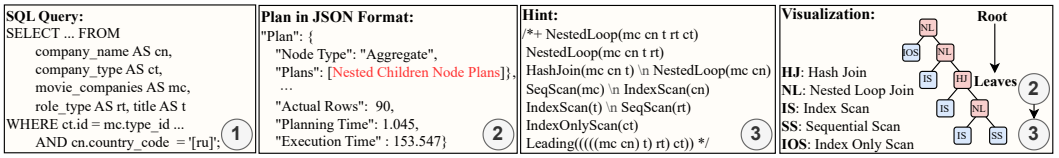


Fig. 2. Given the SQL text and other information (Step 1), we extract the query hint from the query plan in the JSON format (Step 2) and simplify the hint structure (Step 3). The rightmost sub-figure shows how we simplify the plan representation.

encounters with Q, the prompt is continuously optimized. This self-evolving procedure enables SEFRQO to further minimize the execution latency.

4 SFT for Learning Hint Format

In this section, we primarily focus on employing supervised fine-tuning to enable LLMs to understand the query optimization task and to learn the expected hint format.

As mentioned in Section 2.3, we need to construct a training set $\mathbb{D}_{\text{sft}}$ that consists of pairs of prompt x and corresponding generated hint output h^* (i.e., labeled data). We define the $\mathbb{D}_{\text{sft}}$ collection process in Algorithm 1. It shows the sequence of operations we perform for each query $Q_i \in \mathcal{Q}_{train}$ to build the training set. Also, Figure 2 shows a running example of the intermediate outputs of these different operations. First, we execute the query Q_i using the database execution engine to retrieve the detailed query execution plan and its intermediate statistics $\mathcal{E}_i$ (line 2) (using commands like EXPLAIN ANALYZE in PostgreSQL). We return the plan in a structured and nested JSON format to easily parse its details. Figure 2 (Part 2) shows an example of the JSON output of only some statistics of the "Aggregate" operator in the example query (Note that the remaining statistics for the Aggregate operator and other operators are summarized in red as "Nested Children Node Plans" for clarity). Then, we extract the tables $\mathcal{T}_i$ used in Q_i and get their cardinalities C_i from the JSON output (lines 3 and 4). Next, the original query Q_i is concatenated with the extracted cardinalities C_i to construct the prompt x_{Q_i} (line 5). Finally, we obtain the generated hint h^* (i.e., labeled output) for the query Q_i (line 6), pair it with the constructed prompt x_{Q_i}, and add them to the training set $\mathbb{D}_{\text{sft}}$. Figure 2 (Part 3) shows an example of the constructed hint for the plan tree structure highlighted in the rightmost part of the figure. After obtaining $\mathbb{D}_{\text{sft}} = \{(\text{x}_{Q_i}, h_i^*)\}$, we maximize the conditional predictive probability $p_\theta(h^*|\text{x})$ of the LLM as described in Section 2.3.

Algorithm 1 Supervised Fine-tuning Dataset Preparation

Input: Database engine $\mathcal{DB}$, Training query set $\mathcal{Q}_{train} = \{Q_1, Q_2, \ldots, Q_n\}$
Output: SFT training dataset $\mathbb{D}_{\text{sft}} = \{(\text{x}_{Q_i}, h_i^*)\}$

1: **for** each $Q_i \in \mathcal{Q}_{train}$ **do**
2: $\quad \mathcal{E}_i \leftarrow \mathcal{DB}(Q_i)$ ▷ *Generate execution record for Q_i*
3: $\quad \mathcal{T}_i \leftarrow \text{ExtractTables}(\mathcal{E}_i)$ ▷ *Extract used tables*
4: $\quad C_i \leftarrow \text{GetCardinalities}(\mathcal{T}_i, \mathcal{E}_i)$ ▷ *Retrieve table cardinalities*
5: $\quad \text{x}_{Q_i} \leftarrow \text{ConstructPrompt}_{\text{sft}}(Q_i, C_i)$ ▷ *Construct SFT prompt*
6: $\quad h_i^* \leftarrow \text{ExtractHint}(\mathcal{E}_i)$ ▷ *Extract hint as label*
7: $\quad$ Append (x_i, h_i^*) to $\mathbb{D}_{\text{sft}}$
8: **Return** $\mathbb{D}_{\text{sft}}$

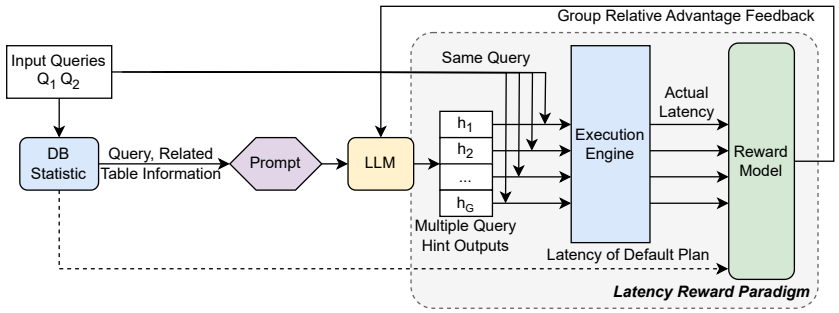


Fig. 3. Overview of the Q_{GRPO} workflow with Latency-based Reward Paradigm.

5 Query Group Relative Policy Optimization

This optimization stage further enhances the LLM's ability to perform query optimization tasks by steering the model's output toward a query hint with reduced latency. We propose Query Group Relative Policy Optimization (Q_{GRPO}) based on GRPO [43] to use reinforcement learning to continuously fine-tune θ_{sft}, which was obtained from the previous supervised fine-tuning stage. The key intuition is to have the LLM generate multiple hint outputs $\{h_1, h_2, \ldots, h_G\}$ for a given prompt x. Then, we use gradient descent to update the model's parameters by computing the in-group advantages of these hint outputs, which correspond to relative latency improvement rewards in our context. As a result, the policy model is optimized to an updated parameter set θ_{GRPO}. Figure 3 illustrates the Q_{GRPO} workflow and the latency-based reward paradigm.

5.1 Latency-based Reward Model

For a given query, the ultimate objective of query optimization is to reduce the latency of the query plan. To directly leverage feedback on query execution latency, we propose a latency-based reward model derived from real query execution and integrated with reinforcement fine-tuning for LLMs.

Algorithm 2 illustrates the latency-based reward model used in Q_{GRPO}. We compute the reward based on the actual latency speedup relative to the baseline latency obtained by executing the same query without any query hint. First, we execute the query without applying a query hint on the database execution engine to obtain the baseline latency t_d (line 1). Each LLM-generated hint output h_i is concatenated with the query Q and executed to record the actual latency t_i (line 3). We then calculate the latency improvement ratio $\triangle_{t_i}$ as the ratio between the baseline latency t_d and the actual latency t_i (line 4). A larger value of $\triangle_{t_i}$ indicates a better query hint. Finally, the latency

Algorithm 2 Latency-based Reward Model

Input: Query Q, corresponding multiple LLM outputs as hint plans$\{h_1, h_2, \ldots, h_G\}$, latency reward function R_{Latency}
Output: Rewards $\{r_1, r_2, \ldots, r_G\}$
1: Baseline latency: $t_d \leftarrow \text{ExecEngine}(Q)$ ▷ *without hint*
2: **for** each Hint Output $i \in \{1, 2, \ldots, G\}$ **do**
3: Actual latency: $t_i \leftarrow \text{ExecEngine}(Q, h_i)$ ▷ *with hint*
4: latency improvement ratio: $\triangle_{t_i} \leftarrow \frac{t_d}{t_i}$ ▷ *Improvement rate*
5: Latency reward: $r_i \leftarrow R_{\text{Latency}}(\triangle_{t_i})$
6: **Return:** $\{r_1, r_2, \ldots, r_G\}$

reward r_i is derived using the designed latency reward function R_{Latency} (line 5), which is defined as follows:

$$R_{\text{Latency}}(\triangle_{t_i}) = \tanh\left(\ln(\triangle_{t_i})\right) \tag{8}$$

where the logarithmic function ln is introduced to achieve symmetry reward. For example, a two-fold improvement ($\Delta_{t_i} = 2$) yields $\ln(2) \approx 0.693$, while a two-fold degradation ($\Delta_{t_i} = 0.5$) results in $\ln(0.5) \approx -0.693$. Although the logarithmic function dampens the effect of huge differences, since some excellent plans or poor plans can lead to order-of-magnitude speedups or slowdowns, a hyperbolic tangent function tanh is further employed to restrict the reward range to the interval $(-1, 1)$. Even when the subsequent procedure computes advantages based on in-group comparisons along with the normalization in Equation 9, reducing the sensitivity of the reward to extreme deviations can improve the overall stability of the fine-tuning procedure.

Suppose the current latency t_i is smaller than the baseline latency t_d. In that case, the latency improvement ratio is larger than 1 ($\triangle_{t_i} > 1$), leading to a positive reward in the end for this LLM-generated output, and vice versa. These rewards guide the LLM's preferences toward generating hints that reduce query latency.

5.2 Group Relative Advantage Feedback

Another key process in Q$_{GRPO}$ is the group relative advantage feedback. Based on the previously calculated rewards $\{r_1, r_2, \ldots, r_G\}$, we compute the advantage A_i for each hint response h_i, reflecting how much better or worse it is compared to others in the group:

$$A_i = \frac{r_i - \text{mean}(\{r_1, r_2, \ldots, r_G\})}{\text{std}(\{r_1, r_2, \ldots, r_G\})} \tag{9}$$

Here, $\text{mean}(\{r_1, r_2, \ldots, r_G\})$ denotes the average reward of the group, and $\text{std}(\{r_1, r_2, \ldots, r_G\})$ denotes the standard deviation of rewards within the group. Subsequently, we compute the ratio of the probability of generating the response h_i under the new policy π_θ to that under the old policy $\pi_{\theta_{\text{old}}}$, denoted as $\frac{\pi_\theta(h_i|\text{x})}{\pi_{\theta_{\text{old}}}(h_i|\text{x})}$. This ratio indicates the degree to which the new policy diverges from the old policy for a given response, thereby guiding the direction of policy updates. Q$_{GRPO}$ employs gradient descent to iteratively maximize the optimization objective defined in Equation 3. The simplified gradient[3] used for updating model can be expressed as:

$$GC_{\text{GRPO}}(\text{x}, h, \pi_\theta, \pi_{\text{ref}}) = \hat{A}_i + \beta\left(\frac{\pi_{\text{ref}}(h_i \mid \text{x})}{\pi_\theta(h_i \mid \text{x})} - 1\right) \tag{10}$$

Algorithm 3 presents the complete fine-tuning process of Q$_{GRPO}$. This algorithm first initializes the policy model π_θ and the reference policy model π_{ref} as π_{sft} (line 1), and iterates over batches to update π_θ (lines 2–9). Specifically, we first save the current policy model as $\pi_{\theta_{\text{old}}}$ to distinguish it from an updating model π_θ (line 3). Then, the LLM generates G query hints as outputs $\{h_i\}_{i=1}^{G} \sim \pi_{\theta_{\text{old}}}$ sampling from the saved policy model $\pi_{\theta_{\text{old}}}$ (line 6). Next, we compute the rewards $\{r_i\}_{i=1}^{G}$ for each generated hint using the latency-based reward model described in Section 5.1 (line 7). Then, we calculate the group relative advantage $\{A_i\}_{i=1}^{G}$ for each generation according to Equation 9 (line 8). After computing these values for the batch, we update the LLM parameters via gradient descent with π_{ref} and π_θ by using Equation 10 (line 9).

[3]This simplified gradient assumes $\pi_{\theta_{\text{old}}} = \pi_\theta$, as in [43].

Algorithm 3 Query Group Relative Policy Optimization

Input: Initial policy model $\pi_{\theta_{\text{sft}}}$, reward function r_ϕ, prompts $\mathcal{P}$
Output: Optimized policy model $\pi_{\theta_{\text{GRPO}}}$

$\pi_\theta \leftarrow \pi_{\theta_{\text{sft}}}$, $\pi_{\text{ref}} \leftarrow \pi_{\theta_{\text{sft}}}$ ▷ *Initialization*
for step = $1, \ldots, M$ **do**
  Save current policy model by $\pi_{\theta_{\text{old}}} \leftarrow \pi_\theta$
  Sample a batch $\mathcal{P}_b$ from $\mathcal{P}$
  for (query, table information) pair x = $(Q, \mathcal{T}) \in \mathcal{P}_b$ **do**
    Sample G generated hints $\{h_i\}_{i=1}^{G} \sim \pi_{\theta_{\text{old}}}(\cdot \mid \text{x})$
    Compute rewards $\{r_i\}_{i=1}^{G}$ through $r_i = r_\phi(h_i)$
    Compute relative advantage $\{A_i\}_{i=1}^{G}$ via Equation 9
  Update π_θ by maximizing the GRPO objective ▷ *via gradient descent in Equation 10, using π_{ref}*
Return: π_θ as $\pi_{\theta_{\text{GRPO}}}$

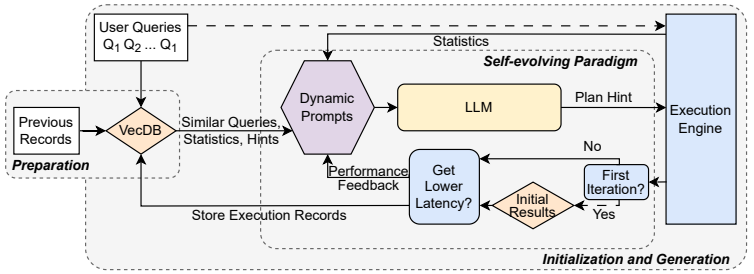


Fig. 4. Overview of SEFRQO's RAG component.

6 RAG-based Prompt Optimization

After the previous offline fine-tuning procedures, we expect our system to continuously evolve during the online stage to generate query hints, which lead to lower query execution latency. For a fixed fine-tuned LLM, different input prompts for the same query Q can result in varying qualities of hint outputs, which in turn influence the execution latency of the resulting query plans. As formalized in Equation 7, by improving the prompt $\mathbf{x}$ to an optimized prompt $\mathbf{x}^*$, the LLM generates a query hint h that minimizes the execution latency for Q. In this section, we present a RAG-based prompt optimization method. Our RAG workflow dynamically constructs the prompt $\mathbf{x}$ for the LLM based on historical execution records of previously generated query plans. This self-evolving paradigm enables online in-context learning for hint generation.

6.1 Architecture of Self-evolving RAG

Figure 4 illustrates the full pipeline of SEFRQO's RAG-based prompt optimization (the self-evolving phase in Section 3.2). It comprises three stages: *Preparation*, *Initialization*, and *Generation*.

During the *Preparation* stage, we collect previous query execution records (e.g., execution times and corresponding query plans generated by PostgreSQL's default optimizer). Notably, SEFRQO does not require a predefined workload for this stage. All collected records are loaded into the vector database (VecDB) to bootstrap the system. These execution records serve as reference examples that can be incorporated into the prompt x to enhance LLM generation.

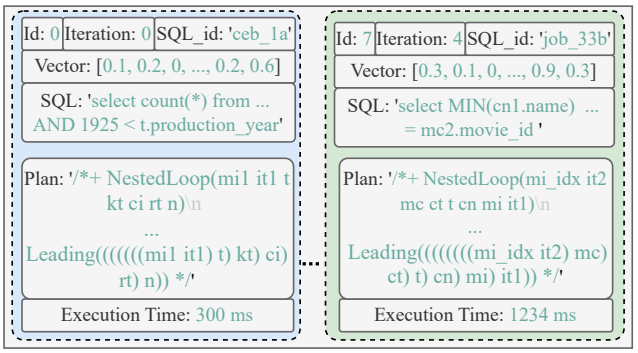


Fig. 5. A visual example of storing execution records.

After the *Preparation* stage, when a query Q is encountered for the first time by our RAG system, it triggers the *Initialization* stage (denoted by the dashed line). In this stage, Q is executed directly on the database engine (e.g., PostgreSQL without applying any query hint h), and the corresponding execution record is stored in VecDB.

For subsequent encounters with Q, our RAG system enters the *Generation* stage. Here, Q is sent to the database engine to extract necessary statistics *stats* (e.g., table cardinalities and filter selectivities) without actual execution. These statistics are then incorporated into the prompt $\mathbf{x}$ to provide additional context for better planning. In parallel, Q is sent to VecDB to retrieve k reference queries via similarity search (as in Section 2.5), along with their execution records. These reference records are also used to dynamically construct the prompt $\mathbf{x}$ (see Section 6.3). Next, the LLM generates a query hint h based on this prompt. The query Q, together with the generated hint h, is executed on the database engine. If the new execution record achieves lower latency than the historical record for Q, the latest record replaces the old one in VecDB. If our RAG system encounters the same query Q again, the stored current best history record for Q is retrieved from VecDB and used to compute feedback information (e.g., performance gain η over PostgreSQL's default plan for Q), which is then incorporated into the dynamic prompt. This feedback paradigm drives continuous self-evolution, iteratively improving future plan generation (see Section 6.4).

6.2 Vector Database Storage Design

Figure 5 shows how we store execution records in VecDB. For each record, we maintain seven properties: `Id`, `Iteration`, `Vector`, `SQL_id`, `SQL`, `Plan`, and `Execution Time`. `Id` serves as the primary key. `Iteration` denotes the current iteration of the online pipeline; for the *Preparation* and *Initialization* stages, it is set to 0 to distinguish them from the *Generation* stage, whose iterations begin at 1. `Vector` contains the embedding of the original `SQL` used for similarity search. `SQL_id` is a unique identifier for each query, comprising the benchmark name and the internal query name (e.g., `job_33b`). It is used to distinguish the current query from its reference queries during similarity search, ensuring that only similar yet distinct queries are retrieved from the vector database. `SQL` denotes the raw query text. `Plan` represents the query plan extracted from the execution record using the workflow in Figure 2. Finally, `Execution Time` stores the actual execution latency.

When a new record arrives, our VecDB inserts it according to the layout described above. To extract relevant information, the input query Q is first encoded as a fixed-length vector using an encoder model (e.g., *SentenceTransformer* [15]). Then, VecDB retrieves the k most related queries

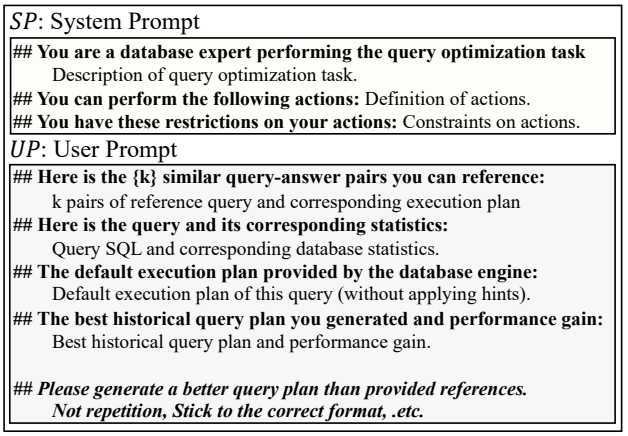


Fig. 6. Dynamic prompt in SEFRQO.

$\{Q_1, Q_2, \ldots, Q_k\}$ (or fewer if only a few similar queries exist) based on cosine similarity of their embeddings. Finally, these queries and their associated execution records are used to construct the dynamic prompt $\mathbf{x}$.

6.3 Dynamic Prompt Construction

Figure 6 illustrates the complete structure of the dynamic prompt $\mathbf{x} = (\text{SP} \,\|\, \text{UP})$. It begins with a static system prompt, SP, which will not change. UP specifies the LLM's role (e.g., database expert), the actions the LLM should perform, and the regulations of these actions in the context of our query optimization task. For a query Q, SEFRQO generates a dynamic user prompt, UP, which initially includes the k most similar queries and their execution plans as references. These records provide the LLM with strongly correlated contextual information, as they represent the queries most similar to Q in the embedding space derived from the SQL text. The LLM can learn from these existing plans, even those for other queries, to better construct a query hint h for the input query Q. Finally, the query statistics $stats$, containing table cardinalities and filter selectivities, are appended to the prompt as additional context.

The following part of the user prompt contains the currently generated best query plan for query Q and the corresponding performance gain η relative to the baseline execution record (i.e., Q executed without any hint h during the *Initialization* stage). These two items serve as heuristics for the LLM to understand the status (i.e., relative performance gain) of the current hint h and to target an optimized hint h^*. At the end of the user prompt, we add expectations and regulations for the LLM, including (1) generating a better hint, (2) avoiding copying the plan provided by the execution engine or its own historical hints, and (3) ensuring output in the correct format so that the hint constitutes a valid plan.

6.4 Self-evolving Feedback Paradigm

The middle part of Figure 4 depicts the self-evolving feedback paradigm of SEFRQO. This paradigm enables the LLM to learn from historical execution records of the same query Q, thereby enhancing the current iteration's generated hint h. For example, when we run the JOB 20a query [22] with SEFRQO, the partial default query plan obtained from PostgreSQL during the *Preparation* stage is `NestedLoop(k mk)` → `NestedLoop(k mk cc)`. Then, in the *Generation* stage (first iteration), SEFRQO generates a hint for the plan `NestedLoop(mk k)` → `NestedLoop(mk k t)`. In later

Algorithm 4 Prompt Optimization in our RAG System

1: **Input:** Input query Q, Previous prompt $Prompt$
2: **Output:** Optimized prompt for current iteration $Prompt^*$
3: $h \leftarrow \text{LLM}(Prompt)$ ▷ *Previous generated hint*
4: $plan, t \leftarrow \text{ExecEngine}(Q, h)$ ▷ *Previous execution record*
5: $plan^*, t^* \leftarrow GetHistoricalBestExecutionTime(Q)$
6: **if** $t < t^*$ **then**
7: $\quad plan^* \leftarrow plan$ ▷ *Update historical best plan*
8: $\quad t^* \leftarrow t$ ▷ *Update historical best execution time*
9: $t_o \leftarrow \text{GetBaselineExecutionTime}(Q)$ ▷ *Retrieve execution time from record stored in the initialization stage from VecDB*
10: Compute the performance gain: $\eta \leftarrow \frac{t_o - t^*}{t_o}$
11: $\text{UP} \leftarrow \text{UP} \cup \{(plan^*, \eta)\}$ ▷ *Insert historical best plan and its performance gain into user prompt*
12: $Prompt^* \leftarrow \text{ConstrcutPrompt}(SP, UP)$
13: **Return** $Prompt^*$

iterations during the feedback process, the hint evolves to `HashJoin(mk k)` → `NestedLoop(mk k cc)`. In this example, we observe that SEFRQO identifies two key insights: (1) joining `mk` and `k` first is beneficial, and performing this join with a HashJoin while adjusting the join order yields better efficiency than a NestedLoop; and (2) the nested loop join NestedLoop(mk k t) is efficient and should be retained.

Algorithm 4 illustrates the cross-iteration prompt optimization process (the straight line below line 8 indicates the separation between previous and current iterations). Here, we use $Prompt$ and $Prompt^*$ in place of the original notation $\mathbf{x}$ to denote the prompt from the previous iteration and the optimized prompt used in the current iteration, respectively. In the previous iteration part, we obtain the generated hint h by sending $Prompt$ to LLM. Then, we execute query Q with h on the database engine to retrieve the resulting query plan and its execution time (lines 3–4). We then retrieve the historical best plan $plan^*$ and its execution time t^* from VecDB (line 5). Finally, we compare the current iteration's result with the historical record, updating $plan^*$ and t^* if the current execution is superior (lines 6–8).

After that, we retrieve the baseline execution time t_o for query Q from VecDB (line 9). This time is stored during the *Initialization* stage, representing the query executed without applying any hint. We then compute the performance gain η of the historical best execution time t^* over the baseline, $\eta = (t_o - t^*)/t_o$ (line 10). Note that if $t^* > t_o$, then $\eta < 0$, indicating that the current best plan performs worse than the baseline. We merge the best plan $plan^*$ and η into the user prompt UP (as in Section 6.3) to provide useful information guiding LLM generation (line 11). Finally, we construct the optimized prompt $Prompt^*$ by concatenating the static system prompt SP and the dynamically updated user prompt UP (line 12).

This optimized prompt, $Prompt^*$, prevents the LLM from repeating the performance pitfall encountered with the previously generated hint h and continuously inspires the LLM to produce an optimal hint h^*. As the pipeline runs iteratively, feedback from hints generated in earlier iterations drives the self-evolving process. Meanwhile, the reference queries stored in VecDB are continuously updated, further enhancing the effectiveness of the references. These components iteratively optimize the prompt $Prompt^*$, motivating the LLM to generate an optimized hint h^* that minimizes the execution latency for query Q.

7 Evaluation

We first introduce our experimental setup in Section 7.1. Then, we evaluate SEFRQO using different benchmarks and multiple LLMs to address the following questions: (1) What is the performance gain over PostgreSQL and current LQOs in static and dynamic scenarios (Section 7.2)? (2) How effective is SEFRQO's self-evolving RAG feedback paradigm (Section 7.3)? (3) What is the effect of SFT and Q_{GRPO} on the quality of generated plans (Sections 7.4)? (4) What is the overhead of SEFRQO and which factors influence it (Section 7.5)? (5) How do different system settings affect the performance of SEFRQO (Section 7.6)? (6) How does SEFRQO perform under various scenarios (Section 7.7)?

7.1 Experiment Setup

Environment. Unless mentioned otherwise, we run all our experiments on one server with an AMD Ryzen 9700X CPU, 128 GB of RAM, and an NVIDIA A6000 GPU with 48GB of memory. We conducted the majority of our experiments on PostgreSQL (version 16.4) with the corresponding *pg_hint_plan* version, and employed MySQL (version 8.4) [31] to assess SEFRQO's cross-DBMS capability. We use the Milvus [53] vector database for our RAG process. We evaluated SEFRQO using 6 local LLMs: LLaMA3.1-3B (**L3B**), LLaMA3.1-8B (**L8B**), their two SFT versions (**LS3B**, **LS8B**), and their two SFT-plus-Q_{GRPO} versions (**LSG3B**, **LSG8B**). We warm up the machine before running the experiments to mitigate caching issues. We use *SentenceTransformer*[4] as the embedding model to embed the plain SQL to a vector representation in our retrieval process.

SEFRQO Variants. We evaluate three variants of SEFRQO: (1) the default configuration, which leverages the full-plan hint in combination with reference queries retrieved from the vector database; (2) a variant that utilizes only the join order component of the hint (the "Leading" clause as detailed in Section 2.1), referred to as **JO**; and (3) a variant that relies exclusively on the historical execution records of the current query Q itself, without incorporating any reference queries from the RAG, referred to as **NR**. For clarity, we denote models using these configurations with suffixes, e.g., an LSG3B model employing the NR variant is denoted as **LSG3B-NR**. Unless mentioned, the number of retrieved reference queries is fixed at $k = 1$. The effect of varying k is investigated in Section 7.6.

LQO Baselines. We compare SEFRQO with two state-of-the-art LQOs: Bao [28] and Balsa [58]. Bao is a steering LQO that tunes query plans generated by traditional optimizers through a learned model that selects different hint sets to enable/disable some operators like Hash Join, Index Scan, etc. Balsa is a generative LQO that learns a model to construct the query plan from scratch. For LLM-based LQOs, due to the lack of available complete code for LLM-QO [47] and LLMOpt [59], we were unable to successfully reproduce their experiments. The only available LLM-based LQO implementation is LLMSteer [2], which we use as our primary baseline for evaluation in this category. LLMSteer uses a text-embedding LLM to generate an embedding vector for the input SQL query and performs predictions using simple neural networks. To have a fair comparison, we follow the best configurations of all baselines as mentioned in their papers[5].

Evaluation Metrics. We focus on these metrics: (1) *Performance Gain in Query Execution Latency:* In this metric, we measure the performance gain over PostgreSQL. We compute this performance gain in two modes. The first mode is *Overall Performance Gain*, where we calculate the weighted sum of the performance gains in "all" the workload queries using SEFRQO (or any LQO) over using PostgreSQL. The weight of each query is the ratio between its corresponding execution time and the total time of all workload queries when running with PostgreSQL. The second mode is *Filtered Performance gain*, where the gain is calculated the same way, yet over the queries that have been

[4]Paraphrase-MiniLM-L6-v2 Model.

[5]Balsa disables Bitmap Scan, Tid Scan, and Genetic Query Optimizer.

Table 1. Performance gain (PG) in static scenarios ↑.

Workload	PG	Balsa	Bao	L3B	L8B	LS3B	LS8B	LSG3B	LSG8B
CEB	Overall	**37.57%**	22.60%	7.46%	21.78%	53.93%	**60.04%**	50.13%	48.36%
	Filtered	**59.22%**	30.13%	17.93%	47.56%	56.76%	**65.05%**	55.34%	61.28%
Stack	Overall	67.17%	**76.90%**	5.54%	15.21%	**40.10%**	26.77%	39.62%	32.58%
	Filtered	**89.83%**	82.85%	9.16%	37.38%	43.29%	87.68%	42.77%	**93.57%**

accelerated "only". (2) *Relative Execution Time (RET):* In this metric, we compute the ratio between the overall execution time of all the queries using SEFRQO (or any LQO) and using PostgreSQL. (3) *Inference and RAG Retrieval Latency:* These metrics measure the latency overhead of the LLM inference and the RAG retrieval from the vector database. (4) *Homogeneous Rate (HR):* We assess the diversity of LLM-generated hints by calculating the ratio between (i) the number of invalid hints or hints identical to those plans generated by the PostgreSQL and (ii) the total number of generated hints. A higher *HR* indicates that the model is either generating invalid hints (rejected by the database as plans) or producing repetitive hints that copy the execution engine's own plans. Note that we do not distinguish between invalid hints and the execution engine's default plans because invalid hints result in the same default plans, making them indistinguishable in practice. Evaluating the diversity of hints generated by LLMs is essential because only distinct hints can effectively influence the database's execution behavior.

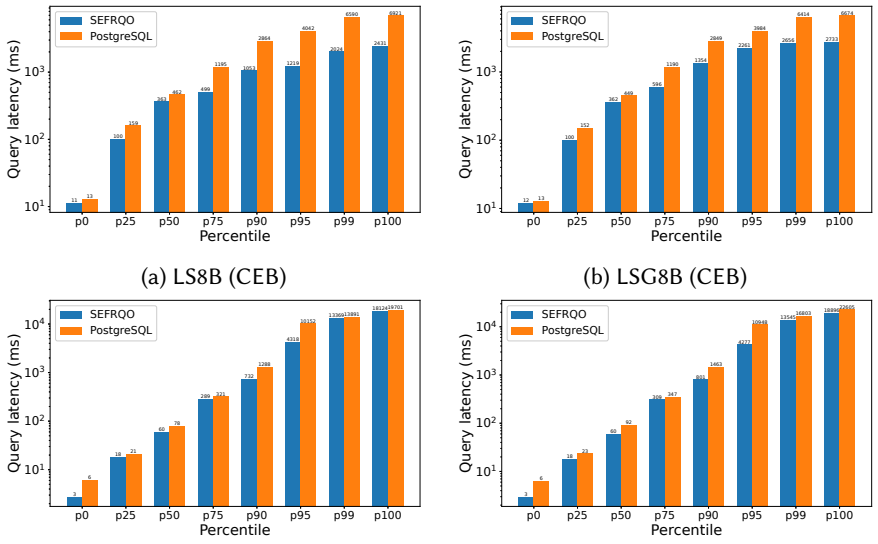
For all evaluation metrics, we use ↑ to denote "the higher, the better," and ↓ to denote "the lower, the better." These symbols will be consistently used across all figures and tables.

Benchmarks. We use the *Join Order Benchmark (JOB)* [22] workload to (1) fine-tune the LS3B, LS8B, LSG3B, and LSG8B models and (2) pre-populate the retrieval corpus in the RAG system with query-answer examples during the preparation phase. Subsequently, we evaluate SEFRQO on two workloads: the *Cardinality Estimation Benchmark (CEB)* [34] and *Stack* [28]. Both JOB and CEB are constructed on the IMDB movie database schema and data, with CEB containing more complex query patterns and a wider range of cardinality estimation challenges than JOB. The Stack workload is derived from the Stack Overflow dataset; while its query style bears similarity to the IMDB-based workloads, it features a greater number of slow queries. For CEB, we use two queries per template; for Stack, five queries per template. To prevent poorly optimized plans from excessively blocking the benchmarking, we enforce the following timeout thresholds for different benchmark: 10s for the JOB and CEB workloads, and 30s for Stack.

7.2 Query Execution Latency Evaluation

In this section, we focus on the *performance gain in query execution latency* for each query during a 50-iteration run of SEFRQO. We evaluate on both static and dynamic scenarios. In static scenarios, we keep using the same workload for all iterations during the online stage. The first static scenario employs the CEB workload, which is used to test SEFRQO under a different setup while retaining the same schema and data distribution. The second static scenario employs the Stack workload, which is entirely different from the JOB workload that is used for fine-tuning, and is used to verify SEFRQO's cross-domain transfer capability (i.e., workload switch). The dynamic scenario comprises a mixture of the CEB, JOB, and Stack workloads, alternating across different episodes. For each episode, one workload is selected randomly and executed for a number of iterations. We set 10 episodes and randomly allocate iterations to each episode, enforcing a total of 50 iterations.

Performance on Static Workloads. Table 1 presents the performance gains achieved on static workloads. The comparison baseline is PostgreSQL's default query plan. For LQOs, both Balsa and



(a) LS8B (CEB) (b) LSG8B (CEB)

(c) LS8B (Stack) (d) LSG8B (Stack)

Fig. 7. Query latency distributions on CEB and Stack.

Bao are trained and evaluated on the same workload (either CEB or Stack). On the CEB workload, SEFRQO with the LS8B model exhibits superior performance relative to the other models, and it outperforms LQOs. This result also suggests that fine-tuning is crucial, as the vanilla models (L3B and L8B) show significantly less improvement compared with models that have undergone supervised fine-tuning (LS3B and LS8B). Interestingly, a slight performance degradation is observed when applying Q_{GRPO} after the SFT process. This is because SFT-based LLMs benefit the most (i.e., leverage the maximum of the fine-tuned knowledge) when they get fine-tuned and tested on similar workloads, such as JOB and CEB. However, when SEFRQO confronts an unseen workload that differs significantly from the fine-tuning workload (as shown right below with the Stack workload) or when the RAG support is disabled (as shown in Section 7.4), SFT-plus-Q_{GRPO} models demonstrate their advantages that stem from the exploratory nature of reinforcement fine-tuning, which enhance the reasoning capabilities of larger LLMs to handle such challenging situations. On Stack workload, both the SFT and SFT-plus-Q_{GRPO} models exhibit good performance. Even though the LLMs have never seen this workload before, the filtered performance gains for LS8B and LSG8B are comparable to those of Balsa or outperform it. Notably, these LQOs are trained on Stack for 50 iterations. This scenario also demonstrates the remarkable knowledge-transfer capabilities of SEFRQO with the SFT and SFT-plus-Q_{GRPO} models, as evidenced by the substantial performance gains they achieve over their vanilla counterparts.

Figure 7 compares the query latency distributions on both CEB and Stack workloads. SEFRQO consistently outperforms PostgreSQL across all percentiles, indicating that it benefits the majority of queries and steadily reduces their latency. In particular, on the CEB workload, SEFRQO is especially effective at optimizing long-tail (high-percentile) queries, achieving approximately 2.5× speedup.

Comparison against LLM-based LQOs. As mentioned in Section 7.1, we experiment with LLM-Steer [2] as a representative for LLM-based LQOs. However, it is not fully integrated into a database engine and requires workloads preprocessed into a specific format (CSV files containing the SQL file name, raw SQL text, a mapping between hint lists and runtime lists, and a special representation

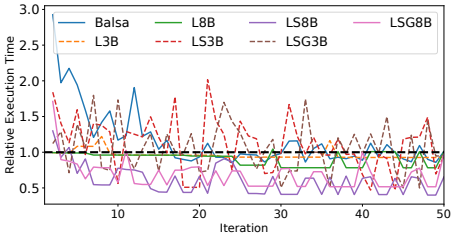


Fig. 8. RET on CEB workload across iterations ↓.

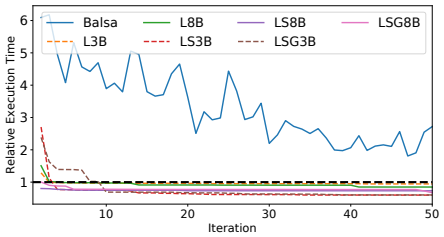


Fig. 9. RET on Stack workload across iterations ↓.

of the query plan), without publicly available preprocessing source code. The authors only provided preprocessed files for the JOB and CEB workloads. Therefore, we only reproduced LLMSteer under the same setup as the first static scenario: training on the JOB workload and evaluating on the CEB workload. LLMSteer achieves an Overall Performance Gain (i.e., improvement across all queries) of 45.14%, while SEFRQO achieves 60.04%. Although LLMSteer's embedding-based generation and specialized architecture offer faster inference, they result in lower overall performance.

Performance on Dynamic Scenario. Table 2 shows the capability of SEFRQO to handle changing workloads. Notably, most existing LQOs do not support dynamic workloads, as their model architectures are typically tied to the schema of the training workload. However, since Bao can accommodate various schemas, we include it in this experiment. In general, larger LLMs yield better results, and SFT is crucial for enhancing performance. Since approximately two-thirds of the workloads originate from the IMDB schema (i.e., JOB and CEB workloads), the dominance of familiar patterns may diminish the effectiveness of SFT-plus-Q_{GRPO} models, which tend to perform better on substantially different workloads such as Stack. Additionally, we can see that Bao underperforms compared to SEFRQO, and exhibits a significant performance drop relative to its results in static settings. These results demonstrate the superiority of SEFRQO in handling dynamic workloads, effectively eliminating reliance and constraints associated with specific workloads.

Table 2. Performance gain in dynamic scenario ↑.

Model	Performance Gain	
	Overall	Filtered
L3B	2.64%	20.11%
L8B	15.92%	54.83%
LS3B	50.25%	58.65%
LS8B	**53.25%**	**69.98%**
LSG3B	33.88%	46.38%
LSG8B	38.83%	51.38%
Bao	14.82%	31.36%

7.3 Self-evolving Feedback Paradigm

In this section, we shift our focus to the overall execution latency for all queries in each iteration. We use the *RET* metric to measure the execution time ratio of SEFRQO compared to PostgreSQL.

Figure 8 shows the self-evolving process under the CEB workload. Among all evaluated models, SEFRQO with LS8B exhibits the best performance, achieving the lowest *RET* of 0.41. It also shows a trend and potential for continuous decrease with additional iterations. In contrast, SEFRQO with

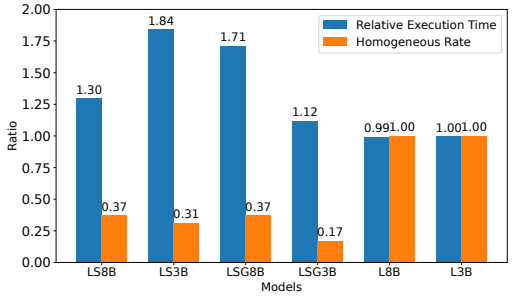


Fig. 10. RET and HR for investigating fine-tuning effects (on the first iteration of SEFRQO) ↓.

vanilla models (L3B, L8B) shows only marginal performance gains throughout the process. Notably, SEFRQO with L8B exhibits minor fluctuations, SEFRQO with L3B remains largely unchanged. This behavior likely results from L3B consistently generating the same query hint, thereby failing to effectively utilize information from the dynamic prompt we construct. However, this phenomenon is mitigated with larger LLMs, which actively absorb input and generate more varied hints.

This experiment further proves that domain-specific fine-tuning plays a crucial role in SEFRQO. Both LS3B and LS8B outperform their vanilla counterparts; however, LS3B demonstrates notable instability compared to LS8B, with performance fluctuation observed several times across all iterations. This phenomenon may indicate that smaller LLMs with supervised fine-tuning exhibit higher instability than larger models in our self-evolving paradigm.

Now, let us switch the perspective to compare SEFRQO with LQOs. Balsa achieves only limited performance improvements over the baseline (denoted by the black dashed line) within each iteration. Although Balsa achieves promising results in the previous evaluations (i.e., by gathering the best queries among all iterations, as shown in Table 1), these improved queries are scattered across different iterations. This indicates that within 50 iterations, Balsa does not consistently generate high-performance query plans in the same iteration, which means Balsa requires more iterations to reach peak performance. This is more obvious in Figure 9 when testing on the Stack workload: Balsa fails to outperform the baseline within our iteration range despite exhibiting a trend towards performance improvement over iterations.

As Figure 9 shows, LLMs exhibit more stable evolving progress (i.e., lower variance and fewer fluctuations across iterations) on the Stack workload compared to their behavior on the CEB workload. This may originate from the fact that the LLMs are not fine-tuned on the Stack workload, causing them to be more cautious and reluctant to substantially change their behaviors in generating hints. Consequently, it enables a smaller model to outperform a larger model. As evidence, LS3B and LSG3B are unable to surpass LS8B on the CEB workload, but they ultimately achieve better performance than LS8B on the Stack workload. In this scenario, LSG3B with Q_{GRPO} fine-tuning achieves the best performance.

7.4 SFT and Q_{GRPO} Effects

Previous experiments verify SEFRQO's collaborative capability by combining LLM fine-tuning with the self-evolving paradigm. In this section, we focus on the plan generation ability based on LLM's internal reasoning, without involving the self-evolving paradigm introduced by RAG. We follow the initial setup, where dynamic prompts include only PostgreSQL execution records, excluding the best plan or performance gain, and evaluate using *RET* and *HR*.

Figure 10 shows the results for different LLMs on the CEB static workload (as in Section 7.2). Both vanilla LLMs (i.e., L8B and L3B) primarily replicate PostgreSQL's default execution plans or repeatedly generate invalid hints and fall back to the PostgreSQL plan, yielding performance similar to PostgreSQL's plan (i.e., RET close to 1) and a high homogeneity rate (i.e., HR close to 1). In contrast, the fine-tuned LLMs produce more diverse plans (i.e., lower HR), favoring exploration over simply copying the input query plan. This indicates an improvement in the reasoning abilities. Even when the initial plan is suboptimal (i.e., higher RET), our self-evolving paradigm reduces the final RET (see Figures 7 and 8).

We observe the pronounced Q_{GRPO} effect in the 3B models: from LS3B to LSG3B, both the *HR* and *RET* decrease further. This shows that Q_{GRPO} enhances the models' reasoning abilities, enabling 3B-level models to generate more diverse and superior query hints. Notably, LSG3B outperforms the larger LS8B model on both metrics. This demonstrates that a successful Q_{GRPO} fine-tuning process can steer the model's output preferences to achieve better task-specific performance compared to larger LLMs without Q_{GRPO}. Given that Q_{GRPO} is based on reinforcement learning, it introduces inherent uncertainty into LLM fine-tuning processes, which may require multiple fine-tuning runs to obtain a reliable model. For example, we show that the LSG8B model does not show improvement over LS8B. This may be attributed to the more complex reasoning nature of 8B models, which prevents them from demonstrating superiority without the self-evolving RAG paradigm.

7.5 Inference and RAG Retrieval Time

This experiment evaluates the overhead incurred by LLM inference and RAG retrieval. We analyze these overheads for SEFRQO using the LS3B model. The LLM inference time is approximately 236 ms on an NVIDIA A6000 and 160 ms on an NVIDIA A100. The average RAG retrieval latency is 0.7 ms. We find that the dominant overhead arises from LLM inference, which can be mitigated by using powerful GPUs. Furthermore, considering execution time savings (with SEFRQO's query plans for the CEB workload averaging 520.67 ms compared to PostgreSQL's 922.86 ms), SEFRQO's end-to-end latency still remains lower for OLAP.

The inference time of LLMs is also positively correlated with output length. In our experiments, we observed two types of erroneous hint generation. Although PostgreSQL will reject such hints, they still incur a significantly longer hint generation process. Figure 11 presents the two most classic errors. The first is **Repeated Text Chunks**, where the generated hint repeatedly includes the same table names multiple times. The second is **Bracket Mismatching**, in which excessive and unbalanced parentheses are generated. One direct implication is to incorporate a validation procedure during inference: when an unusable hint is generated, the process can be aborted promptly to reduce inference time.

Repeated Text Chunks:	**Bracket Mismatching:**
/*+ ... HashJoin(n mi mi ... mi mi mi) ... */	/*+ Leading((((((((((((s t) tq) q) */

Fig. 11. Two classical errors during LLMs' generations, unneeded output characters increase inference time.

7.6 Ablation Studies

To better understand how different configurations affect the SEFRQO's overall performance and reveal the most promising setup, we conduct a series of ablation studies on 3B-scale models.

Generation Mode. Figure 12 depicts the *HR* for each iteration. We observe that LSG3B-JO achieves a higher *HR* (approximately 70%) than the other configurations. These results indicate that when LLM in SEFRQO is constrained to generate only a join-order hint, it mainly reproduces or approximates

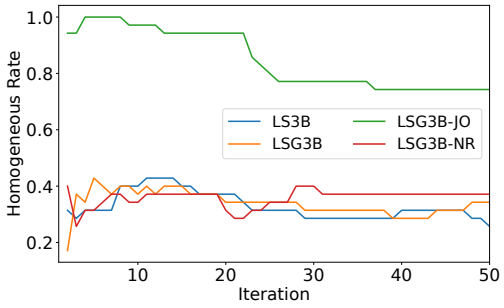


Fig. 12. HR ↓.

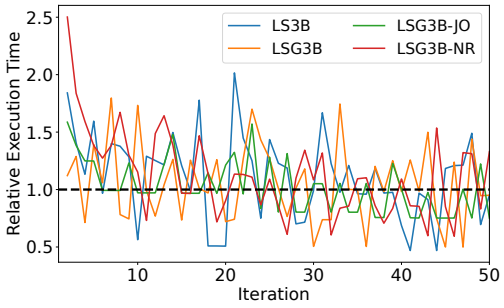


Fig. 13. RET ↓.

the default plan provided by the execution engine and is reluctant to propose new hints to improve performance. Table 3 presents the performance gains over the baseline, comparing LSG3B with LSG3B-JO. The results demonstrate that the full-plan hint generation mode is substantially more effective than the **JO** mode.

Similar Queries as Reference. A key feature of our system is extracting reference queries by similarity search and integrating them into the prompt for in-context learning. This experiment evaluates its impact. Comparing LSG3B with LSG3B-NR, Table 3 reports the performance gain and Figure 13 presents the *RET*. These results indicate that LSG3B consistently outperforms LSG3B-NR, demonstrating the effectiveness of our design.

Table 3. Performance gain on static CEB workload ↑.

Performance Gain	LS3B	LSG3B	LSG3B-JO	LSG3B-NR
Overall	**53.93%**	**50.13%**	24.81%	41.26%
Filtered	**56.76%**	**55.34%**	48.65%	52.61%

How Many References are the Best Setup? We evaluate SEFRQO with LS3B on the CEB static workload by varying k (the number of reference queries in the prompt) from 1 to 3 and assess the impact on performance. Table 4 presents the performance gains and the lowest *HR* for each configuration. Observing the *HR* metric, we find that as k increases, more enriched prompts lead the LLM to generate more dynamic hints that deviate from the default reference query plan. However, the performance gain does not improve monotonically with k, indicating that too few or too many reference queries may lead to a suboptimal solution. Too few references leave SEFRQO lacking key information that could benefit performance, while too many references may confuse the LLM and cause performance degradation.

Table 4. Performance gain ↑ and HR ↓ for different k.

k	Performance Gain		Optimal HR
	Overall	Filtered	
1	53.93%	56.76%	0.26
2	57.75%	**62.16%**	0.20
3	**57.99%**	58.87%	**0.14**

The above ablation studies confirm the crucial role of reference queries in the prompt. They help identify the optimal hyperparameter k and generation mode for SEFRQO, showing that a moderate prompt length balances inference latency and performance.

7.7 Extra Investigations

Beyond the main evaluation, we further conduct a set of additional investigations to examine the generality, stability, and adaptability of SEFRQO. Specifically, we explore its performance on resilience to inaccurate cardinality estimations, its robustness under data distribution shifts, and its portability across different DBMS environments.

Resilience to Inaccurate Cardinality. Cardinality estimation is important for query optimization; inaccurate cardinality could lead to sub-optimal query plans. Therefore, it's necessary to investigate its effect on SEFRQO. We evaluate the static CEB workload by perturbing each filter and table's cardinality using a random scaling factor n. When the original estimate is ≥ 10, n is sampled uniformly from $[-1, 1]$; otherwise, from $[0, 1]$. Note that the database itself remains unchanged; only manual estimation errors are introduced in the prompt. When using LSG8B, SEFRQO achieves a 51.50% Overall Performance Gain and a 57.11% Filtered Performance Gain; when using LS8B, it achieves 50.26% and 54.48%, respectively. A slight degradation can be obeserverd, however, SEFRQO is still comparable to the static experiment in Section 7.2.

Robustness under Data Drift. Here, we study how well SEFRQO can handle drifts in the underlying data over time. Besides the original IMDB database, we construct a downsized version containing only records dated after the year 2000. We evaluate SEFRQO on the CEB workload by alternating between these two databases every 5 iterations, and other conditions are just the same as Section 7.2. Figure 14 presents the execution times over 50, showing only the 5 iterations that use the original database in each alternation cycle, to allow comparison with the corresponding static environment. We observe more frequent spikes, indicating that feedback from the downsized distribution interferes with SEFRQO's self-evolution process. Nevertheless, for the LS8B model, this variance occasionally leads to the least execution times in some iterations.

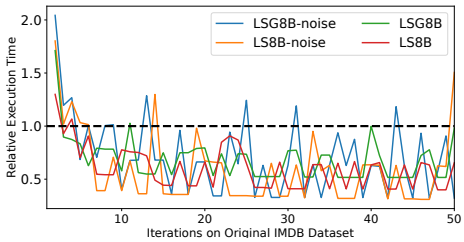


Fig. 14. RET for data distribution shifting situations ↓.

Effect of Different Similarity Calculation Methods. In SEFRQO, we employ cosine similarity to retrieve similar queries as in-context learning examples. Besides, we also evaluate inner product and L_2 distance similarity. On the static CEB workload, SEFRQO with LSG8B using default cosine similarity yields approximately an average of 3% greater performance gain than the other methods. Interestingly, inner product introduces the highest variance in generated query plans, producing 150 more distinct plans over 50 iterations compared to the other methods.

Can SEFRQO Operate with Different DBMSes? We evaluate SEFRQO with LSG8B on MySQL using the CEB workload to test whether the knowledge can be transferred between different DBMSes. Although these LLMs are supervised fine-tuned on PostgreSQL's hint format, we apply

a simple hint translator (We translate hints from PostgreSQL to MySQL by picking each table in a sequence from the "Leading" clause to generate a join order to change MySQL's behavior) and achieve an Overall Performance Gain of 26.28% and a Filtered Performance Gain of 30.98%. These results demonstrate that SEFRQO can perform effectively in a cross-DBMS environment.

8 Related Work

Non-LLM Learned Query Optimizers. In recent years, Learned Query Optimizers (LQOs) have experienced significant advancements, with a variety of neural architectures—such as Tree-CNNs [28, 29, 58], Tree-LSTMs [61, 63], and GNNs [4, 5] which are employed to serve as cost predictors or evaluators within these systems. Broadly speaking, LQOs can be categorized based on their generation paradigm into two types: *steering* and *generative* categories. In the *steering* category, the LQO refines query plans generated by traditional optimizers using neural networks. Bao [28] adjusts the traditional query optimizer behavior by enabling/disabling the different hint sets that this optimizer can choose from. LEON [7] employs a ranking-based approach combined with uncertainty-driven exploration to enhance the query execution performance. LOGER [5] leverages a graph transformer along with a beam search strategy to achieve improved execution speedups. Lero [66] uses a relative-ranking-based method to choose the optimal plan. To sum up, this type has the advantage of being minimally invasive, preserving the core logic of the database engines and reducing the risk of producing poorly performing plans. However, these LQOs are inherently limited by the capabilities of the traditional optimizers they rely on. In the *generative* category, the LQO directly controls the plan generation process. Balsa [58] and Neo [29] use the same neural network structure to generate the query plan step by step using reinforcement learning. Neo uses the typical Tree-CNN, and Balsa adds a simulation strategy to boost the performance. Lemo [30] uses a shared buffer manager component and a plan search algorithm to generate a better plan. These strategies allow for exploring a wider range of execution plans, which can lead to the discovery of more efficient execution. Nevertheless, this greater flexibility also introduces higher variability and potential instability in performance.

LLMs and RAG for Databases. LLMs demonstrate strong performance across various NLP tasks [18, 19, 33], and are increasingly applied to database tasks such as knob tuning [11, 17, 21, 48], Text-to-SQL [10, 16], code generation [51], and schema understanding [50]. DBBert [48] and GPTuner [21] use LLMs to interpret database documents for knob tuning. There is potential to integrate query-level planning and optimization with system-level knob tuning in SEFRQO, a combination rarely explored in previous work because these two approaches are typically considered orthogonal. CARD [41] and GPT-DB [49] leverage LLMs to generate relational data analytics code and SQL pipelines for data analysis. PICARD [41] further enforces SQL syntax through incremental constraint decoding, while CAESURA [52] translates natural language into hybrid query plans combining SQL and Python UDFs. Similarly, RAG has been widely applied to tasks such as text generation [13, 32], visual question answering [6, 23], code synthesis [8, 18, 25], and recently, it has been used to effectively address many challenges in the database domain. For example, in SQL generation, systems like [44] improve accuracy by retrieving structurally similar queries and schema context. Query rewriting also benefits from RAG; approaches like Rafe [26] retrieve prior rewrite examples to transform natural language queries into efficient, semantically equivalent forms, often surpassing traditional optimizer rules [45]. In text-to-SQL, ReFSQL [64] reduces hallucinations by retrieving relevant question–SQL pairs to guide generation.

Recently, three early trials to explore the LLM in the query optimization context, LLMSteer [2], LLM-QO [47], and LLMOpt [59], have been proposed. LLMSteer uses only embedding models for hint set selection, which is light, but the improvement is limited. LLM-QO focuses primarily on

applying fine-tuning to improve the model's performance. LLM-QO uses a DPO-based reinforcement fine-tuning method, which requires constructing a dataset of "preferred" and "non_preferred" pairs beforehand. In contrast, our GRPO-based approach eliminates this overhead and potentially enhances the reasoning abilities of LLMs by training in actual execution environments. On the other hand, LLMOpt [59] employs LLMs to replace components in the Bao [28] workflow, fine-tuning them for plan candidate generation and selection. These works represent initial explorations of applying LLMs to replace LQO pipeline. However, SEFRQO adopts an LLM-native perspective to define optimization problems via model fine-tuning and prompt optimization. Our self-evolving RAG design also shows continuous learning capabilities that previous methods cannot easily achieve, and SERAG [24] is our initial work.

9 Conclusion and Future Work

This paper proposes SEFRQO, a self-evolving RAG system for query optimization. It leverages LLMs' fine-tuning strategies to avoid cold starts and enhance generalization across workloads. It introduces a two-phase framework that combines offline SFT, Q_{GRPO} and online in-context learning. SEFRQO dynamically generates prompts and continuously evolves through execution feedback. We explore the knowledge transfer capabilities of LLM-based query optimization and demonstrate its potential for practical deployment. Various experiments show that SEFRQO's superiority over previous LQOs and the baseline execution engine. To the best of our knowledge, this work is the first to apply RAG to query optimization in databases, thereby extending the frontier of this research area. Currently, SEFRQO targets OLAP queries, as its overhead remains a challenge for OLTP. Overcoming this limitation is left for future work.

References

[1] Rishabh Agarwal, Avi Singh, Lei Zhang, Bernd Bohnet, Luis Rosias, Stephanie Chan, Biao Zhang, Ankesh Anand, Zaheer Abbas, Azade Nova, et al. 2024. Many-shot in-context learning. *Advances in Neural Information Processing Systems* 37 (2024), 76930–76966.

[2] Peter Akioyamen, Zixuan Yi, and Ryan Marcus. 2024. The unreasonable effectiveness of LLMs for query optimization. In *Machine Learning for Systems Workshop at the 38th Conference on Neural Information Processing Systems (NeurIPS 2024)*.

[3] Ekin Akyürek, Dale Schuurmans, Jacob Andreas, Tengyu Ma, and Denny Zhou. 2022. What learning algorithm is in-context learning? investigations with linear models. *arXiv preprint arXiv:2211.15661* (2022).

[4] Baoming Chang. 2024. *A Robust and Explainable Query Optimization Cost Model Based on Bidirectional Graph Neural Networks.* Ph. D. Dissertation. University of Ottawa.

[5] Tianyi Chen, Jun Gao, Hedui Chen, and Yaofeng Tu. 2023. Loger: A learned optimizer towards generating efficient and robust query execution plans. *Proceedings of the VLDB Endowment* 16, 7 (2023), 1777–1789.

[6] Wenhu Chen, Hexiang Hu, Xi Chen, Pat Verga, and William W Cohen. 2022. Murag: Multimodal retrieval-augmented generator for open question answering over images and text. *arXiv preprint arXiv:2210.02928* (2022).

[7] Xu Chen, Haitian Chen, Zibo Liang, Shuncheng Liu, Jinghong Wang, Kai Zeng, Han Su, and Kai Zheng. 2023. Leon: A new framework for ml-aided query optimization. *Proceedings of the VLDB Endowment* 16, 9 (2023), 2261–2273.

[8] Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2022. Cocomic: Code completion by jointly modeling in-file and cross-file context. *arXiv preprint arXiv:2212.10007* (2022).

[9] Lyric Doshi, Vincent Zhuang, Gaurav Jain, Ryan Marcus, Haoyu Huang, Deniz Altinbüken, Eugene Brevdo, and Campbell Fraser. 2023. Kepler: robust learning for parametric query optimization. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–25.

[10] Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2024. Text-to-SQL Empowered by Large Language Models: A Benchmark Evaluation. *Proc. VLDB Endow.* 17, 5 (Jan. 2024), 1132–1145. doi:10.14778/3641204.3641221

[11] Victor Giannakouris and Immanuel Trummer. 2024. Demonstrating λ-tune: Exploiting large language models for workload-adaptive database system tuning. In *Companion of the 2024 International Conference on Management of Data*. 508–511.

[12] 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).
[13] Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Mingwei Chang. 2020. Retrieval augmented language model pre-training. In *International conference on machine learning*. PMLR, 3929–3938.
[14] Kai Han, An Xiao, Enhua Wu, Jianyuan Guo, Chunjing Xu, and Yunhe Wang. 2021. Transformer in transformer. *Advances in neural information processing systems* 34 (2021), 15908–15919.
[15] Kai Han, An Xiao, Enhua Wu, Jianyuan Guo, Chunjing Xu, and Yunhe Wang. 2021. Transformer in transformer. *Advances in neural information processing systems* 34 (2021), 15908–15919.
[16] Zijin Hong, Zheng Yuan, Qinggang Zhang, Hao Chen, Junnan Dong, Feiran Huang, and Xiao Huang. 2025. Next-Generation Database Interfaces: A Survey of LLM-based Text-to-SQL. arXiv:2406.08426 [cs.CL] https://arxiv.org/abs/2406.08426
[17] Xinmei Huang, Haoyang Li, Jing Zhang, Xinxin Zhao, Zhiming Yao, Yiyan Li, Zhuohao Yu, Tieying Zhang, Hong Chen, and Cuiping Li. 2024. Llmtune: Accelerate database knob tuning with large language models. *arXiv preprint arXiv:2404.11581* (2024).
[18] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A Survey on Large Language Models for Code Generation. arXiv:2406.00515 [cs.CL] https://arxiv.org/abs/2406.00515
[19] 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 [cs.CL] https://arxiv.org/abs/2305.06984
[20] Solomon Kullback and Richard A Leibler. 1951. On information and sufficiency. *The annals of mathematical statistics* 22, 1 (1951), 79–86.
[21] Jiale Lao, Yibo Wang, Yufei Li, Jianping Wang, Yunjia Zhang, Zhiyuan Cheng, Wanghu Chen, Mingjie Tang, and Jianguo Wang. 2023. Gptuner: A manual-reading database tuning system via gpt-guided bayesian optimization. *arXiv preprint arXiv:2311.03157* (2023).
[22] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215.
[23] Weizhe Lin, Zhilin Wang, and Bill Byrne. 2023. FVQA 2.0: Introducing adversarial samples into fact-based visual question answering. *arXiv preprint arXiv:2303.10699* (2023).
[24] Hanwen Liu, Qihan Zhang, Ryan Marcus, and Ibrahim Sabek. 2025. SERAG: Self-Evolving RAG System for Query Optimization. (2025).
[25] Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. 2022. Reacc: A retrieval-augmented code completion framework. *arXiv preprint arXiv:2203.07722* (2022).
[26] Shengyu Mao, Yong Jiang, Boli Chen, Xiao Li, Peng Wang, Xinyu Wang, Pengjun Xie, Fei Huang, Huajun Chen, and Ningyu Zhang. 2024. Rafe: ranking feedback improves query rewriting for rag. *arXiv preprint arXiv:2405.14431* (2024).
[27] Yuren Mao, Xuemei Dong, Wenyi Xu, Yunjun Gao, Bin Wei, and Ying Zhang. 2024. Fit-rag: black-box rag with factual information and token reduction. *arXiv preprint arXiv:2403.14374* (2024).
[28] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making learned query optimization practical. In *Proceedings of the 2021 International Conference on Management of Data*. 1275–1288.
[29] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *Proc. VLDB Endow.* 12, 11 (jul 2019), 1705–1718. doi:10.14778/3342263.3342644
[30] Songsong Mo, Yile Chen, Hao Wang, Gao Cong, and Zhifeng Bao. 2023. Lemo: A Cache-Enhanced Learned Optimizer for Concurrent Queries. *Proceedings of the ACM on Management of Data* 1, 4 (2023), 1–26.
[31] MySQL. 2025. MySQL 8.4. https://dev.mysql.com/downloads/mysql/.
[32] Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, et al. 2021. Webgpt: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332* (2021).
[33] Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Naveed Akhtar, Nick Barnes, and Ajmal Mian. 2023. A comprehensive overview of large language models. *arXiv preprint arXiv:2307.06435* (2023).
[34] Parimarjan Negi, Ryan Marcus, Andreas Kipf, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. 2021. Flow-Loss: Learning Cardinality Estimates That Matter. *Proc. VLDB Endow.* 14, 11 (jul 2021), 2019–2032. doi:10.14778/3476249.3476259
[35] OSSC-DB. [n. d.]. pg_hint_plan. https://github.com/ossc-db/pg_hint_plan.
[36] Arka Pal, Deep Karkhanis, Samuel Dooley, Manley Roberts, Siddartha Naidu, and Colin White. 2024. Smaug: Fixing Failure Modes of Preference Optimisation with DPO-Positive. arXiv:2402.13228 [cs.CL] https://arxiv.org/abs/2402.13228
[37] PostgreSQL. [n. d.]. https://www.postgresql.org/.

[38] PostgreSQL Global Development Group. [n. d.]. https://www.postgresql.org/.
[39] Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. 2023. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems* 36 (2023), 53728–53741.
[40] Tolga Şakar and Hakan Emekci. 2025. Maximizing RAG efficiency: A comparative analysis of RAG methods. *Natural Language Processing* 31, 1 (2025), 1–25.
[41] Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. PICARD: Parsing incrementally for constrained auto-regressive decoding from language models. *arXiv preprint arXiv:2109.05093* (2021).
[42] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).
[43] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. 2024. DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models. arXiv:2402.03300 [cs.CL] https://arxiv.org/abs/2402.03300
[44] Zhili Shen, Pavlos Vougiouklis, Chenxin Diao, Kaustubh Vyas, Yuanyi Ji, and Jeff Z Pan. 2024. Improving Retrieval-augmented Text-to-SQL with AST-based Ranking and Schema Pruning. *arXiv preprint arXiv:2407.03227* (2024).
[45] Zhaoyan Sun, Xuanhe Zhou, and Guoliang Li. 2024. R-Bot: An LLM-based Query Rewrite System. *arXiv preprint arXiv:2412.01661* (2024).
[46] Toni Taipalus. 2024. Vector database management systems: Fundamental concepts, use-cases, and current challenges. *Cognitive Systems Research* 85 (2024), 101216.
[47] Jie Tan, Kangfei Zhao, Rui Li, Jeffrey Xu Yu, Chengzhi Piao, Hong Cheng, Helen Meng, Deli Zhao, and Yu Rong. 2025. Can Large Language Models Be Query Optimizer for Relational Databases? *arXiv preprint arXiv:2502.05562* (2025).
[48] Immanuel Trummer. 2022. DB-BERT: a Database Tuning Tool that" Reads the Manual". In *Proceedings of the 2022 international conference on management of data*. 190–203.
[49] Immanuel Trummer. 2023. Demonstrating gpt-db: Generating query-specific and customizable code for sql processing with gpt-4. *Proceedings of the VLDB Endowment* 16, 12 (2023), 4098–4101.
[50] Immanuel Trummer. 2024. Generating Succinct Descriptions of Database Schemata for Cost-Efficient Prompting of Large Language Models. *Proc. VLDB Endow.* 17, 11 (July 2024), 3511–3523. doi:10.14778/3681954.3682017
[51] Immanuel Trummer. 2025. Generating highly customizable python code for data processing with large language models. *The VLDB Journal* 34, 2 (2025), 21.
[52] Matthias Urban and Carsten Binnig. 2023. CAESURA: Language Models as Multi-Modal Query Planners. *arXiv preprint arXiv:2308.03424* (2023).
[53] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, et al. 2021. Milvus: A purpose-built vector data management system. In *Proceedings of the 2021 International Conference on Management of Data*. 2614–2627.
[54] Peiyi Wang, Lei Li, Zhihong Shao, R. X. Xu, Damai Dai, Yifei Li, Deli Chen, Y. Wu, and Zhifang Sui. 2024. Math-Shepherd: Verify and Reinforce LLMs Step-by-step without Human Annotations. arXiv:2312.08935 [cs.AI] https://arxiv.org/abs/2312.08935
[55] Noam Wies, Yoav Levine, and Amnon Shashua. 2023. The learnability of in-context learning. *Advances in Neural Information Processing Systems* 36 (2023), 36637–36651.
[56] Lucas Woltmann, Jerome Thiessat, Claudio Hartmann, Dirk Habich, and Wolfgang Lehner. 2023. FASTgres: Making Learned Query Optimizer Hinting Effective. *Proc. VLDB Endow.* 16, 11 (July 2023), 3310–3322. doi:10.14778/3611479.3611528
[57] Sang Michael Xie, Aditi Raghunathan, Percy Liang, and Tengyu Ma. 2021. An explanation of in-context learning as implicit bayesian inference. *arXiv preprint arXiv:2111.02080* (2021).
[58] Zongheng Yang, Wei-Lin Chiang, Sifei Luan, Gautam Mittal, Michael Luo, and Ion Stoica. 2022. Balsa: Learning a query optimizer without expert demonstrations. In *Proceedings of the 2022 International Conference on Management of Data*. 931–944.
[59] Zhiming Yao, Haoyang Li, Jing Zhang, Cuiping Li, and Hong Chen. 2025. A Query Optimization Method Utilizing Large Language Models. *arXiv preprint arXiv:2503.06902* (2025).
[60] 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.
[61] 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.
[62] Xiang Yu, Guoliang Li, Chengliang Chai, and Nan Tang. 2020. Reinforcement learning with tree-lstm for join order selection. In *2020 IEEE 36th international conference on data engineering (ICDE)*. IEEE, 1297–1308.
[63] Xiang Yu, Guoliang Li, Chengliang Chai, and Nan Tang. 2020. Reinforcement learning with tree-lstm for join order selection. In *2020 IEEE 36th international conference on data engineering (ICDE)*. IEEE, 1297–1308.

[64] Kun Zhang, Xiexiong Lin, Yuanzhuo Wang, Xin Zhang, Fei Sun, Cen Jianhe, Hexiang Tan, Xuhui Jiang, and Huawei Shen. 2023. Refsql: A retrieval-augmentation framework for text-to-sql generation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*. 664–673.

[65] Wangda Zhang, Matteo Interlandi, Paul Mineiro, Shi Qiao, Nasim Ghazanfari, Karlen Lie, Marc Friedman, Rafah Hosn, Hiren Patel, and Alekh Jindal. 2022. Deploying a Steered Query Optimizer in Production at Microsoft. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 2299–2311. doi:10.1145/3514221.3526052

[66] Rong Zhu, Wei Chen, Bolin Ding, Xingguang Chen, Andreas Pfadler, Ziniu Wu, and Jingren Zhou. 2023. Lero: A learning-to-rank query optimizer. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1466–1479.

Received April 2025; revised July 2025; accepted August 2025