OmniTune: A Universal Framework for Query Refinement via LLMs

Eldar Hacohen
Bar-Ilan University
Ramat Gan, Israel
eldar.hacohen@live.biu.ac.il

Yuval Moskovitch
Ben-Gurion University of the Negev
Be'er-Sheva, Israel
yuvalmos@bgu.ac.il

Amit Somech
Bar-Ilan University
Ramat Gan, Israel
somecha@cs.biu.ac.il

Abstract

Numerous studies have proposed solutions for SQL query refinement, where the goal is to make minimal adjustments to an input query to satisfy a given set of constraints. While effective, these approaches typically address specific query types and constraints, whereas, in practice, users may need to refine a diverse range of queries based on their requirements. To address this, we present OmniTune, a universal framework for query refinement. OmniTune features a Refinement Problem Wizard for defining refinement tasks in natural language and a flexible Refinement Engine, which employs an LLM-based multi-agent architecture to support any query refinement problem. We demonstrate OmniTune across various query refinement scenarios using real-world datasets.

CCS Concepts

• **Information systems** → **Data management systems**; **Language models**.

Keywords

Database Query Refinement, Large Language Models (LLMs)

ACM Reference Format:
Eldar Hacohen, Yuval Moskovitch, and Amit Somech. 2025. OmniTune: A Universal Framework for Query Refinement via LLMs. In *Companion of the 2025 International Conference on Management of Data (SIGMOD-Companion '25), June 22–27, 2025, Berlin, Germany*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3722212.3725121

1 Introduction

The problem of SQL query refinement has been widely studied [1, 4–6], with the goal of minimally adjusting a given query to ensure its results satisfy some predefined constraints. For example, prior work [5] focuses on refining the `WHERE` clause predicates in conjunctive Select-Project-Join (SPJ) queries to meet *cardinality* constraints, adjusting queries when too few or too many tuples are returned. More recent studies [1, 4, 6] considered constraints over different data subgroups in the output. While existing works have proven useful for various applications, they address specific definitions of constraints and refinement, often tailored to (simple) types of queries. In contrast, users may seek to refine a wide range of queries, including those with group-by clauses, subqueries, or window functions, and my require distinct, ad hoc definition of constraints and refinement criteria. For illustration, consider the following example for a group-by query refinement problem.

Example 1.1. Consider Clarice, a data analyst working for a national excellence foundation that aims to award scholarships to law students based on their performance. The foundation seeks to ensure a balanced number of qualifying students from each region while maintaining comparable performance levels among all students. Since each region receives equal scholarship funding from the government, the number of qualifying students should be closely aligned across regions. The foundation aims at awarding 500-2000 students from each region. To determine the required bars for the scholarship awarding, Clarice uses the Law Students dataset[1], which contains demographic information and the performance of law students from different regions. For each student, it includes the Law SAT (LSAT) examination scores and their undergrad GPA (UGPA). She executes the following query Q.

```
SELECT region, avg(UGPA), avg(LSAT), COUNT(*) FROM Students
WHERE UGPA > 3.5 AND LSAT > 40
GROUP BY region
```

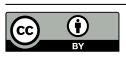
Q provides information on the number of students from each region with UGPA above 3.5 and LSAT of over 40. Clarice observed that fewer than 500 students met the specified criteria in all regions, and there was significant variation in the average LSAT and UGPA across regions, concluding the specified bar values should be adjusted to meet the foundation requirements. □

Unfortunately, as the simple group-by query exemplified above is not trivially supported by previous work, addressing it would require substantial research efforts. Specifically, one would need to formally define the refinement problem, design a constraint satisfaction framework, and explore adaptations of existing solutions to support these query settings. Alternatively, refining the query manually can be a tedious and repetitive task.

To this end, we present OmniTune, a universal framework for SQL query refinement. Given an input database and query, OmniTune first assists users in *formulating* the query refinement problem by enabling them to express constraints in natural language and select or compose an appropriate notion of query refinement tailored to their needs. OmniTune then employs a novel, multi-agent architecture that leverages large language models (LLMs) to first formulate an explicit query refinement problem based on the descriptions provided by the user, then perform an iterative optimization process to ensure the input query is minimally modified while satisfying the user-defined constraints.

More specifically, using an in-context learning approach [7], we first construct executable constraint satisfaction and refinement distance functions based on the user's input specifications, defined

[1] https://www.kaggle.com/datasets/danofer/law-school-admissions-bar-passage



SIGMOD-Companion '25, Berlin, Germany

ACM ISBN 979-8-4007-1564-8/2025/06
https://doi.org/10.1145/3722212.3725121

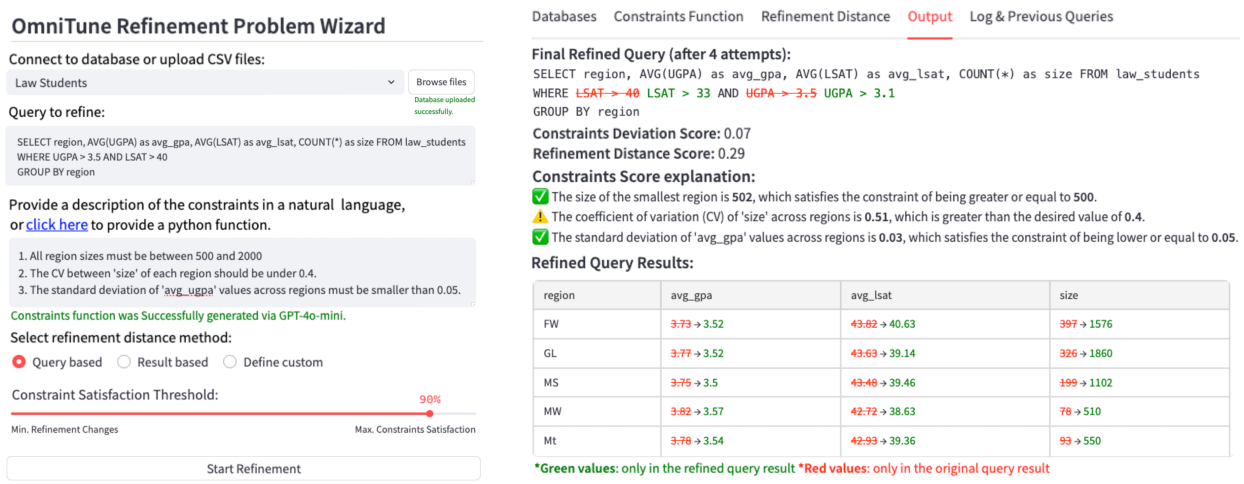


Figure 1: OmniTune Refinement Wizard (Left-hand side) and the refined query generated by the OmniTune refinement engine.

through the OmniTune Refinement Problem Wizard (illustrated on the left-hand side of Figure 1). Next, we employ the OmniTune Refinement Engine, a flexible and generic LLM-based optimizer we devise particularly for *universal* query refinement problems – in which the objective functions are derived from the user's specifications and are unknown in advance. Our solution builds upon the well-known *actor-critic* paradigm [3], adapted from classical reinforcement learning to LLM-based agents. In our context, an LLM-based *actor* agent iteratively generates a refined query attempt based on a series of *database inquiries* (detailed below) and feedback from the *critic* agent. The critic evaluates each refinement attempt by analyzing the refinement distance, constraint satisfaction scores, and past iterations. After a predefined number of iterations, OmniTune returns the minimally modified query that best satisfies the given constraints, as illustrated on the right-hand side of Figure 1.

Interactive Demonstration Overview. We invite the audience to define custom refinement problems and explore the refined queries generated by OmniTune. To showcase its capabilities, we present diverse refinement tasks on three real-world datasets, including ones from [1, 6], enabling comparisons with prior solutions.

2 Model & Problem Definition

The problem of query refinement was studied in a line of work [1, 4–6]. Intuitively, given a dataset D, a query Q, and some constraints on the output, the goal is to apply minimal changes to Q to obtain Q' such that the output of Q' over D satisfies the constraint. While these studies share similar overarching goals, they differ in their specifics. Particularly, the class of supported queries and constraints vary, as well as the definition of distance between queries used to asses the minimality of modifications applied to obtain Q'. Some solutions guarantee to satisfy the constraint, while others allow for approximate satisfaction. In our proposed solution, implemented in OmniTune, we generalize the concept of query refinement, allowing OmniTune to address a wide range of query refinement problems, including those from previous works. This is achieved by abstracting the main component of the problem, as detailed below.

Constraints satisfaction objective function. To allow for approximate satisfaction of the result of the output query over the database D with respect to the given constraints, we define the constraints satisfaction objective function $\Psi(Q, D)$. It takes as input a query Q and a dataset D and returns a value in the range of $[0, 1]$, indicating the (normalized) deviation of $Q(D)$ from the constraints satisfaction. A lower value indicates greater satisfaction of the constraints, with 0 representing full satisfaction.

Example 2.1. Continuing Example 1.1, recall that the foundation seeks a balanced number of qualifying students per region while maintaining comparable performance levels. Each region should award between 500 and 5000 students. These goals can be formalized using the following three constraints: (1) The number of awarded students from each region must be between 500 and 2000, (2) the standard deviation of the average UGPA across regions must be less than 0.02 and (3) The coefficient of variation (the ratio of the standard deviation to the mean) for the number of students across regions must be less than 0.4. A constraints satisfaction objective function should return a value in the range of $[0, 1]$, representing the degree of (un)satisfaction. This function can be defined as the fraction of unsatisfied constraints. For example, if the first two constraints are satisfied but the last is not, it would return $\frac{1}{3}$. Alternatively, we could measure the deviation from each constraint's satisfaction and return an average satisfaction value, similar to the deviation function defined in [1]. Figure 2 shows a Python code implementation of the latter, generated by OmniTune. □

Refinement distance objective function. The query refinement problem aims to make *minimal* modifications to the query to satisfy the constraints. Note that the notion of refinement as defined in [5] and used in previous work refers to modifications of constants in the WHERE clause. Namely, Q' is a refinement of Q if Q' can be obtained from Q by modifying the constant in the predicate of Q. Namely, not every query Q' is considered a *valid* refinement of Q. Moreover, minimality may be defined in various ways, such as based on the set of output tuples of the queries [1, 6], known as outcome-based distance [1], or by the semantic distance in query predicates, referred to as predicate-based distance [1]. It may also involve a combination of both approaches.

To this end, we define the refinement distance objective function Δ, which, when given two queries and a dataset a positive real

```python
def constraints_satisfaction_objective(result_df: pd.DataFrame) -> float:
    # Constraint 1a: Size of smallest region must be larger than 500
    min_size_penalty = (500 - result_df['size'].min()) / 500
    min_size_penalty = max(0, min_size_penalty)

    # Constraint 1b: Size of largest region must be smaller than 2000
    max_size_penalty = (result_df['size'].max() - 2000) / 2000
    max_size_penalty = max(0, max_size_penalty)

    # Constraint 1: Normalized size constraint
    size_penalty = 0.5 * (min_size_penalty + max_size_penalty)

    # Constraint 2: CV between 'size' of each region should be under 0.4
    cv_size = ((result_df['size'].std() / result_df ['size'].mean()) - 0.4) / 0.4
    cv_size = max(0, cv_size)

    # Constraint 3: SD of 'avg_gpa' across regions should be under 0.05
    std_dev = (result_df['avg_gpa'].std() - 0.05) / 0.05
    std_dev = max(0, std_dev)

    # Aggregate constraints and normalize
    return (size_penalty + cv_size + std_dev) / 3
```

Figure 2: LLM-generated Constraints Objective Function

number indicating the distance between the queries. If the query Q' is not a valid refinement of Q, $\Delta(Q, Q', D) = \infty$. We note that while the previous work focused on SPJ queries and thus defined the validity of refinement with respect to that class, OmniTune is not limited to SPJ queries, and the concept of valid refinement may be defined for other classes of queries. For instance, consider a GROUP BY with HAVING close query Q. We can define a valid refinement of Q to be a query with similar structure, allowing modification, to both the constant in the WHERE clause and the HAVING clause.

Universal query refinement. We are now ready to define the Universal Query Refinement. We follow previous works and frame the problem with respect to a threshold ε over the constraint satisfaction divination. Intuitively, this threshold helps eliminate queries whose output "deviates too much" from the desired constraints.

Problem 1 (Universal Query Refinement)**.** Given a SQL query Q, an input database D, a constraint satisfaction objective function Ψ, a refinement distance objective function Δ, and threshold ε over the constraint satisfaction deviation, find refinement query Q' such that $\Psi(Q', D) < \varepsilon$ and $\Delta(Q, Q', D)$ is minimal.

When D is clear from the context, we use $\Psi(Q')$ and $\Delta(Q, Q')$.

3 OmniTune Architecture

The architecture of OmniTune consists of two main components: (1) the Refinement Problem Wizard, which helps users formulate a valid, universal refinement problem, and (2) the OmniTune Refinement Engine, based on an LLM multi-agent architecture used to effectively refine the input query.

3.1 Refinement Problem Wizard

To facilitate the formulation of a universal query refinement problem (as defined in Problem 1), the OmniTune Wizard assists users in defining a constraint objective function Ψ, a corresponding threshold ε, and a refinement distance function Δ. Users can either select from existing functions or specify their own using natural language.

Defining constraints. We use an LLM-based approach to generate a valid constraint function Ψ, using an in-context learning approach [7]. We formulate a constraints function generation prompt by populating a prompt template (see [2]) with the following elements: (1) The schema of the input dataset D, and the query Q;

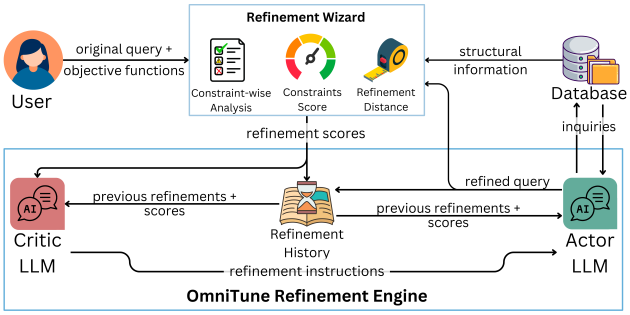


Figure 3: OmniTune Refinement Engine Architecture

(2) Few-shot example pairs of constraints descriptions and corresponding valid objective functions (implemented in Python), obtained from [1, 5, 6]; (3) Implementation guidelines for generating a valid constraints function, as defined in Section 2. Specifically, the function must have an output range within $[0, 1]$, and if multiple constraints are provided, their scores should be normalized to prevent bias toward any particular constraint. For instance, in our running example (see Figure 1), given the three input constraint descriptions, the OmniTune Wizard generates the constraint function depicted in Figure 2. Each constraint is implemented separately and normalized before computing the final score as the average of all constraint components. Users can further edit the generated function manually or through additional LLM interactions.

Defining refinement distance. Users can select an existing query distance function Δ or define a custom one. The available functions are adapted from previous works: (1) *Query (syntax)-based distance* [4], compares the predicates of the modified query Q' to those of the original query Q; and (2) *Results-based distance* [6], which compares the queries by measuring the dissimilarity between the *returned tuples* using Jaccard distance. Users can compose their own distance functions by providing the code or a natural language description. For the latter, we use an LLM prompting approach similar to that of the constraint objective function, incorporating few-shot examples and output specifications (see [2]).

3.2 LLM-Based Refinement Process

Once the query refinement problem is defined, the dataset D, query Q, and generated objective functions Ψ, Δ, and the threshold ε are passed to the OmniTune Refinement Engine. Since the objective functions are derived from the user's specifications, existing solvers for predefined refinement distances and constraints [1, 4, 6] are not applicable. Inspired by [8], we design a flexible LLM-based optimization engine using an iterative, multi-agent approach, adapting the *actor-critic* method from reinforcement learning [3]. OmniTune iteratively employs two LLM-based agents, as depicted in Figure 3: the *actor*, formulates query refinements, while the *critic* evaluates their refinement distance and constraint satisfaction, providing feedback. The agents can perform inquiries on the database D and the refined query results, and access a summary of previous refinement attempts to inform their decisions. The OmniTune Refinement Engine performs an *iterative* refinement process. At each step $1 \leq i \leq N$ (We use N=10), the engine generates a new refinement attempt Q_i based on an analysis of the scores of Q_{i-1}. In the first step ($i = 1$), the engine analyzes the input query Q. A refinement attempt Q_i is performed using the following process:

1.A. Explain the constraints score $\Psi(Q_{i-1})$ (Critic). The critic's first task is to *explain* the score $\Psi(Q_{i-1})$ of the last refinement attempt Q_{i-1}, in order to minimize it in the next refinement step. We formulate a prompt asking the critic agent to dissect the code of the constraints function Ψ, alongside Q_{i_1}, its results $Q_{i-1}(D)$, and score $\Psi(Q_{i-1})$, and provide constraints score *explanation*, each inspecting a single facet of the constraints function.

Example 3.1. Assume the engine is at Step 4 of the refinement process for our example query Q. Table 1 summarizes previous attempts, listing the WHERE predicate values for *UGPA*, and *LSAT* and the corresponding scores (ignore the last column). For Q_3, the `WHERE` clause is `'UGPA' > 3.0 AND 'LSAT' > 36`, yielding $\Delta(Q_3, Q) = 0.24$ (using query based distance [1]) and $\Psi(Q_3) = 0.12$. The critic decides to first examine the absolute deviation from the minimum region size constraint using the Pandas query `df['size'].min() - 500`. The smallest resulting region has a size of 404, meaning it must be increased by 96 to meet the minimum requirement. The remaining explanations, generated similarly, are as follows : *(1) Max size:* `1762` (meets $\leq$ 2000 constraint); *(2) CV(size):* `52%` (exceeds max 40%); and *(3) SD(avg_ugpa):* `2.4%` (meets $\leq$ 5% constraint). □

1.B. Generate score summary & instructions (critic). Next, we prompt the critic to generate concrete suggestions for the actor agent. The feedback is derived from the score explanations in Stage 1.A and an analysis of the refinement history. To prevent exceeding the LLM's context window, we provide a summarized view of previous refinements, as shown in rows 0–2 of Table 1, including scores and explanations produced by the critic.

Example 3.2. Given the score explanations from Example 3.1, the critic deduces that the main factor affecting the result is the fact that in $Q_3(D)$ the smallest region is below the requirement of *'Size'>=500*. Examining the effect on the size constraints in the provided refinement history, it speculates that relaxing the LSAT value will be effective for increasing the minimal region size, while retaining a better refinement distance than relaxing the UGPA. It then outputs the following instructions for the actor: *"decrease the LSAT value in order to increase the minimal region size, while balancing with a slight increase in UGPA value"* □

2.A, 2.B. Inquire the database, Compose Q_i (Actor). The actor agent obtains the feedback for Q_i generated by the critic. It is then (2.A) instructed to issue up to k database inquiries, limited to one-line, simple Pandas queries (we use k=3 in our implementation), and (2.B) formulates the next refinement query Q_i based on the inquiries issued in Step 2.A and the given feedback.

Example 3.3. Based on the critic's feedback the agent issues a *percentile count* command for columns UGPA and LSAT, showing the 25%, 50% and 75% percentiles, which results as follows:

	UGPA	LSAT
25%	3.00	33.00
50%	3.30	37.00
75%	3.50	41.00

Inspecting the results, the actor then composes Q_4, deciding to relax the LSAT value to 33 (the 25% percentile value), while increasing the UGPA threshold to 3.1. Q_4 and its results are depicted in the right-hand side of Figure 1, alongside the scores and explanations.

Q_i	UGPA	LSAT	$\Delta(Q, Q_i)$	$\Psi(Q_i)$	Score Explanation
0	3.5	40	0.00	0.36	High SD & CV, low size
1	3.1	36	0.21	0.15	Low CV, high size
2	3.1	35	0.24	0.13	High SD, low CV, high size
3	**3.0**	**36**	**0.24**	**0.12**	————

Table 1: Refinement History Example (abbreviated)

See that Q_4 meets the required constraints deviation limit $\varepsilon \leq 0.1$, and the only constraint that is not fully satisfied is that CV of the regions' size – 0.51 compared to the desired value of 0.4 □

4 Interactive Demonstration

We demonstrate OmniTune using multiple query refinement scenarios on real-life datasets:

A Walk-through of OmniTune. We will start the demonstration by showing an example query refinement scenario, as depicted in Figure 1. Participants will be invited to interact with the OmniTune refinement wizard and formulate the problem's components. We will let the audience explore the solutions generated by the system and the refinement process. OmniTune annotates and explains each refinement step simultaneously, showing the current query, highlighting differences from the previous query results, and presenting a brief summary of the current status of constraint satisfaction.

Interactive composition of new refinement tasks. Next, users are encouraged to compose a new query refinement task using the OmniTune Refinement Problem Wizard. Specifically, it would be interesting to define *new*, custom refinement problems for complex queries that were not tackled ad-hoc in previous works, and varying the number of constraints and their complexity. We have prepared several such cases, e.g., for refining queries with a HAVING clause, multiple WHERE clauses, and window functions. Using the OmniTune Refinement Problem Wizard, users can examine the resulted refinement distance and constraints objective functions, edit and modify them as necessary, and then observe the refinement process done by the OmniTune LLM-based refinement engine.

A look under the hood. Last, we will compare the full OmniTune engine to simpler versions (e.g., which utilize a single prompt, or more advanced language models) as well as to ad-hoc optimization solutions to specific refinement problems such as [1, 4–6].

Acknowledgments. This work was partially supported by the U.S.-Israel Binational Science Foundation (BSF) under grant number 2022279 and the Israel Science Foundation grant 2121/22.

References

[1] Felix S. Campbell, Alon Silberstein, Julia Stoyanovich, and Yuval Moskovitch. 2024. Query Refinement for Diverse Top-k Selection. *PACMMOD* 2, 3 (2024).
[2] OmniTune Github. 2024. https://github.com/analysis-bots/OmniTune. (2024).
[3] Vijay Konda and John Tsitsiklis. 1999. Actor-critic algorithms. *NeurIPS* 12 (1999).
[4] Jinyang Li, Yuval Moskovitch, Julia Stoyanovich, and HV Jagadish. 2023. Query Refinement for Diversity Constraint Satisfaction. *PVLDB* 17, 2 (2023).
[5] Chaitanya Mishra and Nick Koudas. 2009. Interactive query refinement. In *EDBT*.
[6] Suraj Shetiya, Ian P. Swift, Abolfazl Asudeh, and Gautam Das. 2022. Fairness-Aware Range Queries for Selecting Unbiased Data. In *ICDE*.
[7] Jason Wei, Xuezhi Wang, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *NeurIPS* (2022).
[8] Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. 2024. Large Language Models as Optimizers. (2024). arXiv:cs.LG/2309.03409 https://arxiv.org/abs/2309.03409