Demonstrating λ-Tune: Exploiting Large Language Models for Workload-Adaptive Database System Tuning

Victor Giannakouris
Cornell University
Ithaca, NY, USA
vg292@cornell.edu

Immanuel Trummer
Cornell University
Ithaca, NY, USA
it224@cornell.edu

ABSTRACT

We demonstrate λ-Tune, a tool that leverages Large Language Models (LLMs) for automated, workload-adaptive database system tuning. λ-Tune harnesses the ability of LLMs to process and comprehend arbitrary textual data in a zero-shot manner, employing a workload-adaptive optimization approach. Given a database system, the hardware specifications, and a set of queries, λ-Tune generates prompts to retrieve configuration recommendations for the tuning knobs and the physical design, tailored to the specific system and workload. Our framework utilizes a workload compression approach that extracts and includes in the prompt only the most insightful workload characteristics, while the prompt size can be adjusted by a user-defined token budget. Utilizing the zero-shot capabilities of LLMs, λ-Tune outperforms other LLM and machine learning-enhanced database tuning baselines, which rely on time-consuming tuning and training phases, as well as expensive hardware, such as GPUs. During demonstration, users will be able to experiment with λ-Tune to tune Postgres and MySQL, as well as explore and modify the configurations retrieved by the LLM in an interactive way through λ-Tune's user interface.

CCS CONCEPTS

• **Information systems** → **Query optimization**; **Autonomous database administration**; • **Human-centered computing** → **Natural language interfaces**.

KEYWORDS

Database, Tuning, Large Language Model

ACM Reference Format:

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 (SIGMOD-Companion '24), June 9–15, 2024, Santiago, AA, Chile.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3626246.3654751

1 INTRODUCTION

The database system research community has invested significant effort into developing automated configuration tuning solutions. Tuning a database system is hard; it demands extensive expertise and a profound understanding of the system's internals, and the workload. A DBA must determine which tuning knobs to adjust and the manner of their modification, as well as the physical design decisions. This complexity amplifies when managing multiple systems within a vast software stack, as even similar database systems can differ in their internals, tuning knobs, and APIs. Machine learning has been proven a useful tool to automate database systems tuning. Through the last years, numerous attempts have been made to utilize machine learning in query optimization [1, 3, 6], configuration tuning [7, 9], learned indexes [4], and index recommendations [2]. While these solutions have demonstrated promising results, many lack generality and automation. Most machine-learning-enhanced solutions require substantial integration effort and the development of different connectors for integration with the target database system. Even though some solutions offer a more generic API, they still demand time-consuming training periods, excessive training data, and expensive hardware (like GPUs) for training their models.

Since ChatGPT's first public release in late 2022, Large Language Models (LLMs) have gained immense attention from both industry and academia. Their capability to analyze arbitrary, unstructured text, extract insights, and produce meaningful responses to user queries has been a crucial factor in their fast-paced popularity. Consequently, LLMs have emerged as handy tools in addressing numerous challenges. Recent research highlights their utility across diverse domains in database systems, ranging from tuning [5] to code generation [8].

In this demo, we present λ-Tune, a framework that explores the ability of LLMs to automatically tune a database system. λ-Tune, harnesses LLMs' abilities in analyzing a wide spectrum of text in order to tune the target database system, in a workload-adaptive manner. λ-Tune composes prompts incorporating system-specific details, like hardware specifications, along with query-specific information. Impressively, the LLM can distill valuable insights from the incorporated plans and suggest configuration modifications to investigate potential enhancements in query execution, such as tuning knob configurations and physical design decisions. Our experimental evaluation illustrates λ-Tune's ability to significantly enhance query execution speeds, and outperform prior tools that use machine learning and language models for database system tuning, like GPTuner [5], DB-Bert [8] and UDO [9] in both optimization duration and overall workload execution across several use-cases.

2 SYSTEM OVERVIEW

As depicted in Figure 1, λ-Tune consists of three components: the Renderer (prompt generator), the Workload Compressor and the

SIGMOD-Companion '24, June 9–15, 2024, Santiago, AA, Chile

ACM ISBN 979-8-4007-0422-2/24/06…$15.00
https://doi.org/10.1145/3626246.3654751

Configuration Executor. We discuss the details of each individual component below.

Tuning Pipeline. λ-Tune takes as input three parameters: a workload $\mathcal{W}$, a hardware specification $\mathcal{H}$ and a database system $\mathcal{D}$. First, it passes these parameters to the Renderer, which transforms and embeds them into a prompt that describes the workload, the system and the hardware to the LLM. Next, it issues a query to the LLM n times, in order to retrieve n responses, each one including one configuration set $c \in C$. These configurations are usually almost identical, with some slight differences depending on the degree of randomization of the LLM. Each configuration set c contains a set of SQL commands, compliant with the target database $\mathcal{D}$. For instance, if the target system is Postgres, c will consist of a list of `"CREATE INDEX"` and `"ALTER SYSTEM SET $param_name = $value"` commands. Finally, the configuration superset C is passed to the Configuration Executor, which will incrementally try out all configurations and find the optimal one.

Workload Compressor. The motivation of developing a compression method is the desired API cost savings by reducing the prompt size. The key idea behind the compressor, is the intuition that LLMs can reason and understand the same input in multiple different formats, if they are provided with the right explanation about the given format. For instance, to describe two SQL queries that both include the join $A = B$, it is sufficient for the LLM to provide it with the information that $A = B$ appears in two queries. Then, we can safely discard the rest of the query text, like `SELECT`, `FROM`, `JOIN` and other clauses, and reduce the final prompt size. Using this approach, we can compress redundant information that appears in multiple queries into a much smaller text snippet. By describing the semantics of the compressed format comprehensively in our prompt, we provide the LLM with the same information, by minimizing the cost of the API call. The workload compressor takes as input a workload $\mathcal{W}$, consisting of a set of queries. It analyzes the queries by decomposing them into individual structural components, like join or filter conditions. As a condition might appear in multiple queries, the first step is to keep track of the condition frequencies using a frequency table. Next, using that frequency table, we can pick the most informative conditions to include in the prompt, according to a user-defined token budget $\mathcal{B}$. To achieve that, we model this process as a *Knapsack* problem, which we solve by using Integer Linear Programming (ILP). The cost of each condition to be added in the prompt, is modeled as its *number of tokens*, and the value as the *estimated cost* retrieved by the query optimizer of the target system. The cost is obtained by the output of the `EXPLAIN` clause, which contains the estimated cost of each operation (e.g. join or filter). We use the optimizer cost estimates because we want to prioritize and optimize for the most computationally expensive conditions, when the token budget is limited. If a condition appears in multiple queries with different costs in each query, we use the average cost retrieved as the value. The final output of the compressor is a set of selected conditions, which is passed to the Renderer for the final prompt generation.

Renderer. After compressing the workload, the Renderer takes the selected condition subset and embeds it to a prompt template, which also includes the target database system $\mathcal{D}$ and the hardware specifications $\mathcal{H}$. Finally, it outputs a prompt $\mathcal{P}$, which will be sent to the LLM API to retrieve the configurations. The prompt template instructs the LLM to return the configuration set using the SQL syntax of the target system, consisting of tuning knob changes and index creation commands.

Configuration Executor. λ-Tune can obtain multiple configuration sets, and try out all of them in order to find the optimal one. The naive approach would be to evaluate all configurations, keeping the one that achieves the minimum execution time. However, there might be configurations that run disproportionately slower than the more efficient ones. For instance, in the case of TPC-H on Postgres, an effective configuration will lead to a total execution time of ~16 seconds. On the other hand, there are some less efficient configurations that might lead to more than an hour of execution, for instance, if they miss one or more indexes. In such a case, the system will suffer from excessive execution overheads, trying out configurations that are far from optimal. λ-Tune's Configuration Executor component ensures that good configurations are found faster, without spending time in evaluating the whole workload using an inefficient one. To achieve that, we employ a configuration evaluation approach, which tries out different configurations in a round-robin fashion. Given an initial timeout t, λ-Tune will start evaluating each configuration for t seconds. After t seconds, the execution gets interrupted, and the next configuration will be evaluated. Once all configurations have been tested for t seconds, the second round will start, and t will become $2t$ in round 2, $4t$ in round 3, and so on, following a geometric progression. The last round of the pipeline will be the one in which at least one configuration completes. For all completed configurations in the final round, we keep the one that achieves the minimum execution time.

As mentioned before, the retrieved configurations might include recommended indexes, which need to be materialized in each configuration evaluation phase. In order to minimize index creation time in each round, we interleave index creation with query execution in each configuration evaluation stage. Before starting the evaluation, λ-Tune will analyze the workload and extract query/index associations by analyzing the join and filter conditions from the logical query plans. Thus, each index is created lazily, only before the execution of a query that might use that index. Using this approach, we make sure that when the timeout t is exceeded during some round, the time spent on index creation is minimized, creating only the indexes required by the queries we have executed in the current round.

3 EXTRACT OF EXPERIMENTAL EVALUATION

Due to space limitations, we present a small extract of our experimental evaluation.

Setup. All the experiments were ran on a EC2 p3.2xlarge instance, using the Deep Learning Base GPU AMI on Ubuntu 20.04. We use the TPC-H benchmark of the 1GB scale, and the Join Order Benchmark (JOB). We use λ-Tune to tune Postgres 12.0 and MySQL 8.0. In all experiments, we compare λ-Tune with DB-Bert, GPTuner, and UDO. We compare the optimization time needed by all three approaches, as well as the workload execution time. UDO is the only system among λ-Tune's competitors that supports index creation. Thus, we include it in both scenarios investigate whether additional index creation would lead to further performance gains.

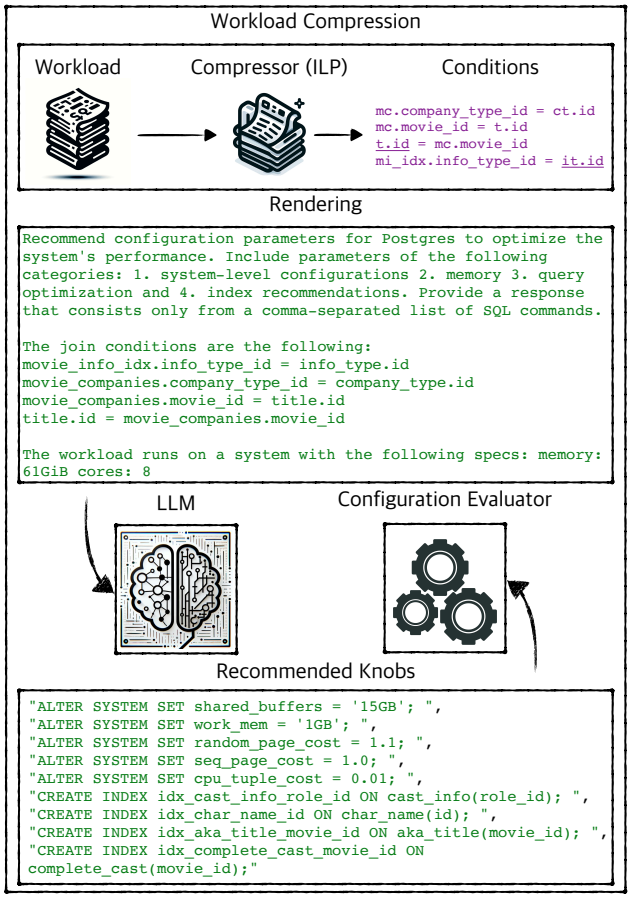
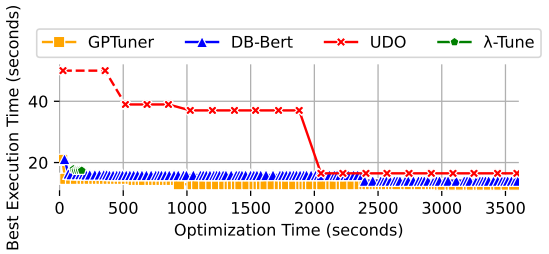
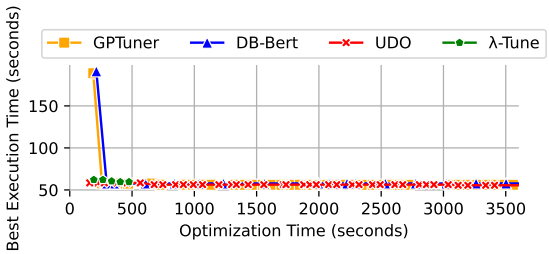


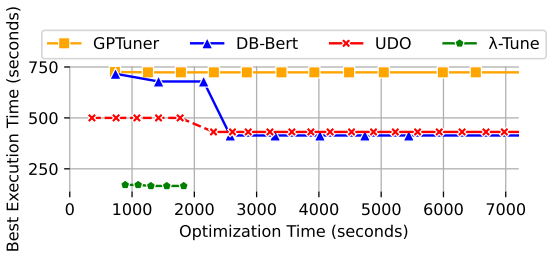
Figure 1: λ-Tune Architecture



(a) TPC-H (Postgres) with Original Indexes



(b) TPC-H (MySQL) with Original Indexes



(c) JOB (Postgres) with Index Creation

Figure 2: Comparison of λ-Tune to its Baselines

Experiments. Figure 2 depicts the results of our experimental evaluation. The X-axis depicts the optimization time, while the Y-axis the execution time of the best configuration found at each point of the plot. GPTuner and UDO take as input a timeout per trial (workload execution). For each experiment, we set this timeout to *three times the worst execution time found by λ-Tune*. The dashed lines in the plot indicate failed execution attempts because of the timeout.

We ran DB-Bert, GPTuner, and UDO for one hour in the first two experiments (original indexes) and for two hours in the third experiment (no initial indexes).

In the first experiment (Figures 2a and 2b), we tune Postgres and MySQL using the default TPC-H indexes. Thus, we do not perform any additional index creation. Next (Figure 2c), we tune Postgres for the Join Order Benchmark, without any indexes in the beginning. In the first two cases, we can see that λ-Tune is able to find an effective configuration within the same tuning time as its competitors, while substantial improvements are shown in the second scenario, in which λ-Tune recommends indexes. As we can see in both plots, λ-Tune's main advantage is the fact that it does not have any initial overhead due to training, in contrast to the baselines. Instead, it retrieves the configurations directly from the LLM and evaluates them immediately. By describing the configurations comprehensively to the LLM using our workload compression approach presented in Section 2, λ-Tune retrieves the most efficient configuration fast in the majority of our experiments. However, as shown in Figures 2a and 2b, λ-Tune achieves an almost identical performance with GPTuner and DB-Bert, both in optimization and workload execution time. In the scenarios in which there are no initial indexes, λ-Tune achieves significantly better performance than GPTuner and DB-Bert which do not create indexes, and similarly for UDO.

4 DEMONSTRATION

Our demonstration consists of two parts. First, visitors will be able to experiment with the prompt generator. Second, after generating one or more prompts, users will be able to tune the target database system using the configurations obtained from the previous step.

Prompt Generation. During this phase of the demonstration, users will be asked to experiment with the configuration generation component of λ-Tune. Using this component, users will be able to interact with λ-Tune, and generate different configuration

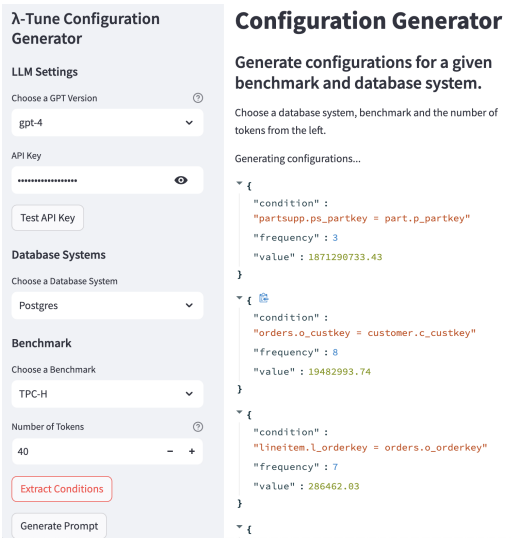


Figure 3: λ-Tune's Configuration Generator UI

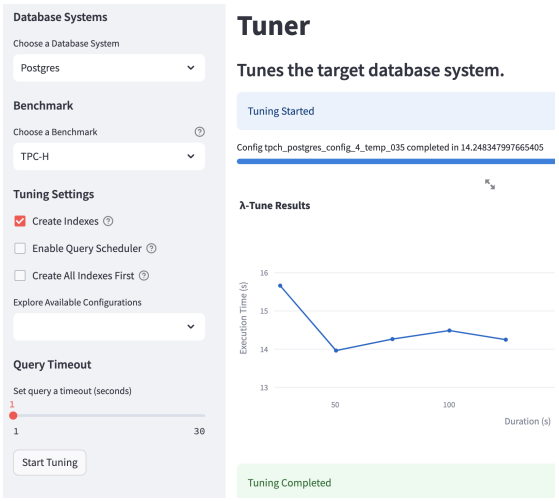


Figure 4: λ-Tune's Tuning UI

recommendations by modifying multiple parameters, including the target DBMS, the benchmark, and the number of tokens. By selecting different parameter combinations, users will be exploring the extracted conditions from the selected workload, and the generated prompts, as well as obtaining configurations for the target system from the LLM. Figure 3 depicts an instance of the condition extraction phase of the configuration generator, where users can explore each extracted condition, the frequency of the condition among the queries, as well as the average cost estimation of the query optimizer (which serves as the value for the ILP solver).

Tuning. In this phase, users will be allowed to tune the target system using the configurations created in the previous step. As depicted in Figure 4, users can interact with the UI by modifying multiple options. For instance, they can enable or disable physical design tuning (Create Indexes), and they can switch between creating all the indexes at the beginning of each configuration evaluation (Create All Indexes First), or, enabling the query scheduler which interleaves query execution and index creation (Enable Query Scheduler). Furthermore, during configuration evaluation, users will be able to see the current best configuration and its execution time through a live plot that is being updated online, during the evaluation phase.

5 CONCLUSIONS & RELATED WORK

The proposed demonstration presents λ-Tune, a tool that utilizes LLMs to provide universal, workload adaptive database system tuning. λ-Tune is not the first approach that utilizes NLP and LLMs to auto-tune a database system. DB-Bert [8] is a promising approach that tunes a database system by extracting information from its manual. On the other hand, GPTuner [5], provides an LLM-based tuning approach using Bayesian Optimization. Unfortunately, none of these systems support index recommendations, making it impossible to optimize a workload further, especially in the absence of initial indexes. UDO [9] is a universal tuning approach that supports index creation but requires excessive tuning and training time, making it particularly slow. λ-Tune tunes simultaneously system configurations, as well as index recommendations. Our experimental evaluation showcases, λ-Tune tunes the target database system fast using a few-shots approach, while it matches the performance of all of its competitors in the presence of indexes, and outperforms all of them when there are no indexes both in optimization and execution time.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Award No. 2239326.

REFERENCES

[1] Christoph Anneser, Nesime Tatbul, David Cohen, Zhenggang Xu, Prithviraj Pandian, Nikolay Laptev, and Ryan Marcus. 2023. AutoSteer: Learned Query Optimization for Any SQL Database. *Proceedings of the VLDB Endowment* 16, 12 (2023), 3515–3527.

[2] Bailu Ding, Sudipto Das, Ryan Marcus, Wentao Wu, Surajit Chaudhuri, and Vivek R Narasayya. 2019. Ai meets ai: Leveraging query executions to improve index recommendations. In *Proceedings of the 2019 International Conference on Management of Data*. 1241–1258.

[3] Victor Giannakouris and Immanuel Trummer. 2022. Building Learned Federated Query Optimizers. In *CEUR workshop proceedings*, Vol. 3186.

[4] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the 2018 international conference on management of data*. 489–504.

[5] 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).

[6] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2022. Bao: Making learned query optimization practical. *ACM SIGMOD Record* 51, 1 (2022), 6–13.

[7] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C Mowry, Matthew Perron, Ian Quah, et al. 2017. Self-Driving Database Management Systems.. In *CIDR*, Vol. 4. 1.

[8] 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.

[9] Junxiong Wang, Immanuel Trummer, and Debabrota Basu. 2021. UDO: universal database optimization using reinforcement learning. *arXiv preprint arXiv:2104.01744* (2021).