DBG-PT: A Large Language Model Assisted Query Performance Regression Debugger

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

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

ABSTRACT

In this paper we explore the ability of Large Language Models (LLMs) in analyzing and comparing query plans, and resolving query performance regressions. We present DBG-PT, a query regression debugging framework powered by LLMs. DBG-PT keeps track of query execution instances, and detects slowdowns according to a user-defined regression factor. Once a regression is detected, DBG-PT leverages the capabilities of the underlying LLM in order to compare the regressed plan with a previously effective one, and comes up with tuning knob configurations in order to alleviate the regression. By exploiting textual information of the executed query plans, DBG-PT is able to integrate with close-to-zero implementation effort with any database system that supports the *EXPLAIN* clause. During the demonstration, we will showcase DBG-PT's ability to resolve query regressions using several real-world inspired scenarios, including plan changes because of index creations/deletions, or configuration changes. Furthermore, users will be able to experiment using ad-hoc, or predefined queries from the Join Order Benchmark (JOB) and TPC-H, and over MySQL and Postgres.

PVLDB Reference Format:
Victor Giannakouris and Immanuel Trummer. DBG-PT: A Large Language Model Assisted Query Performance Regression Debugger. PVLDB, 17(12): 4337 - 4340, 2024.
doi:10.14778/3685800.3685869

1 INTRODUCTION

In this demo, we explore the abilities of Large Language Models (LLMs) in comparing and reasoning about query plans, as well as addressing query performance regressions. We present an LLM-backed system, namely DBG-PT, that assists in debugging regressed queries in a database system. The key intuition behind DBG-PT is the fact that when a query regresses due to changes in its plan, we can leverage information about a previous, efficient query plan to understand why the latest plan leads to regression. Then, using this knowledge, we can take actions to alleviate the regression, such as providing appropriate hints to the optimizer. DBG-PT is built on the understanding that an LLM, with its capability to interpret and reason about arbitrary and unstructured textual data, represents a naturally efficient way to automate this diagnostic and optimization process.

The process of examining and debugging query regressions is notoriously hard, especially when the workload consists of lengthy queries with multiple joins, as they result in complex query plans. Executed query plans are insightful and contain a lot of valuable information that a Database Administrator (DBA) can utilize to understand and fix the root cause of a query regression, if possible. Usually, the first step towards debugging a query regression is to compare the two plans of the regressed query: a previously efficient query plan and the latest plan that was executed when the regression was detected. By comparing the two query plans, the user can make interesting observations about what caused the regression. For instance, by examining two query plans, the user can check whether the join order changed significantly, or when a different join strategy was selected for the same join, for instance, if a nested-loop join was replaced with a hash, or a sort-merge join. Another approach would be to compare the optimizer's estimated row counts for the intermediate results with their actual row counts, if this information is available. In case the error is high, that could imply that the optimizer made incorrect join order and strategy decisions due to cardinality estimation errors, a very common issue in queries that include multiple joins and index scans [4]. Such observations can be key in determining what caused a query regression and fixing it. For instance, if a newly added index is responsible for a chain of changes in a query plan that used to run faster, then it might make sense to use specific hints to instruct the optimizer not to use this index for that query. If for some reason a query used to be more efficient by using nested loops in previous executions, the DBA can explicitly disable all the other join strategies for that query, in order to force the optimizer to use nested loops only. However, the process of understanding these problems and taking the required actions to diagnose and fix query regressions require profound knowledge of database system internals, which DBAs have obtained over years of experience. Even for an advanced user, comparing query plans might become a very tedious process, as their complex structure makes it difficult for the human eye to identify potential plan differences. The database research community has been diligently working on automating the process of tuning database systems, while a lot of works have shown promising results on identifying query plan regressions due to index creations [1], or mistakes made by learned query optimizers [8]. However, all of these methods require substantial integration effort with the database system, and time consuming training phases, making it hard for a non-expert user to integrate them in their systems.

Large Language Models. The emergence of Large Language Models (LLMs) like ChatGPT since its first release in late 2022 has

Proceedings of the VLDB Endowment, Vol. 17, No. 12 ISSN 2150-8097.
doi:10.14778/3685800.3685869

opened new avenues in leveraging AI to address complex challenges across various domains, including database system tuning [6] and code generation [7]. LLMs are exceptionally good at parsing vast amounts of text, extracting relevant information, and providing insightful suggestions, making them valuable tools for automating and enhancing database management tasks. When used as agents, LLMs can eliminate the expensive training of the previously machine learning-based solutions, as they can quickly return an effective response after issuing a single API call, when the prompt is properly structured with the right context.

LLM-Assisted Query Regression Debugging. We believe that LLMs can be a natural solution when it comes to addressing query plan performance regressions for the following reasons. First, we observe that LLMs are very effective at identifying and reasoning about structural changes between different query plans. For instance, LLMs can easily catch differences in join order or strategy changes, or access methods (e.g. index scans vs full table scans). Second, having been trained on a vast amount of technical documents crawled from the internet, LLMs can provide very accurate answers when asked how to fix a query regression, provided they are given the right context, in our case, two query plans. The fact that an LLM has ingested data from sources that contain troubleshooting conversations about real systems and problems, like StackOverflow make them a great tool in resolving performance issues for a plethora of well-known database systems. As a result, a regression debugging tool built on top of an LLM can a a plug-and-play solution for most of the popular database systems.

In this context, we introduce DBG-PT, a framework that utilizes the power of LLMs to diagnose and fix query performance regressions. DBG-PT leverages LLMs' zero/few-shots abilities to analyze extensive textual data, including system specifications and query plan details, to diagnose the query regression root cause, and provide a set of required actions in order to fix the regression, for instance, by reverting the regressed plan to a previously effective one. In contrast to the majority of the automated tuning solutions, DBG-PT requires close-to-zero engineering effort to integrate with new systems, as it is simply connected with any database system compatible with the Python Database API Specification [1] and extracts the query plans by invoking the `EXPLAIN` clause, when it is supported by the target system. Furthermore, DBG-PT requires no retraining, or further adjustment in order to integrate with a new database system and workload. Instead, DBG-PT simply informs the LLM about the regression, the target database system and information about the underlying infrastructure.

DBG-PT's approach is simple, yet powerful; By keeping track of different query execution instances, it alerts the end user if a regression is detected. Upon the user's request, DBG-PT will automatically render a prompt which contains all the required context for the LLM to resolve the query regression, including the query plans, execution logs and hardware details. After sending the generated prompt to the LLM, it retrieves a response which highlights the differences between the efficient, and the regressed query plan, a set of recommended tuning knobs to fix the regression, and the LLM's reasoning for the recommended knobs. DBG-PT is inspired from the idea of our recent work on λ-Tune [2], which exploits information from an input set of queries, in order to tune an input database system in a workload-adaptive manner. Instead, DBG-PT optimizes regressed queries, by solely focusing on a single, regressed query and its query plan variations.

During demonstration, users will be able to experiment with DBG-PT to resolve query regressions for various scenarios, including regressions caused due to index creations, or configuration changes, and over two popular database management systems, including Postgres and MySQL. Furthermore, besides the uses cases that we will provide, users will be encouraged to come up with their own regression cases and explore DBG-PT's ability to resolve these regressions.

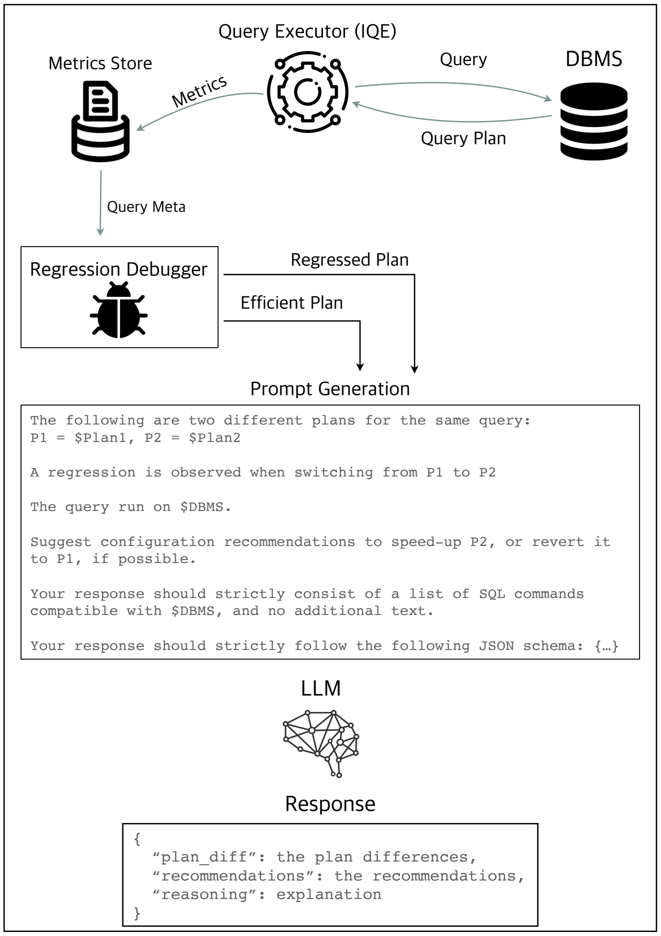
2 SYSTEM OVERVIEW

Figure 1 depicts DBG-PT's architecture. Our system consists of the following components.

Interactive Query Executor (IQE). This component is the first point of interaction between DBG-PT and the end user. Via IQE, the user is able to execute previously registered, or new ad-hoc queries to the target system, and database. At the same time, the user can also modify the tuning knobs for that query, and explore the different generated plans by the query optimizer of the target database. Once the user executes the query, the execution metrics are stored in DBG-PT's query metrics store, along with the query plan information. Furthermore, the performance history of every previously executed query can be investigated by the live performance plots on top of the page (see Figure 2). This component serves as a user-friendly point of interaction between the user and the database system. For this demonstration purposes, we will be using IQE in order to explore how the query optimizer behaves and its produced plans given a query, and different tuning knob sets.

Regression Debugger. The Regression Debugger is responsible for detecting query regressions, and generating the appropriate prompt (by invoking the Prompt Generator) in order to attempt to fix the regression. A regression is simply detected by comparing the latest execution of the query with previous execution instances. Let $T(q,t)$ the execution time of query q at time t. A query q is classified as regressed when $T(q,t') \geq \phi T(q,t)$, where $t' > t$ the latest execution timestamp of query q and ϕ a user-defined regression factor. In other words, a query is considered as regressed when it becomes ϕ *times slower*. In case of a detected regression, DBG-PT will obtain the two executed query plans at times t and t'. Next, the two query plans will be passed to the prompt generator, in order to *render* them into a prompt, which will be used to obtain the following three pieces of information, as depicted in Figure 3. First, we ask the LLM to provide a summary of the *plan differences*, to get a better understanding of what changed in the regressed query plan. Next, we retrieve a set of recommended configurations to fix the plan regression. In most of the cases, this will include knobs such as optimization hints that attempt to revert the plan in its initial state, before the regression. Finally, we ask the LLM to provide the reasoning about the recommended configurations, and how these are related to the two plans that we provide. This information provides a better understanding of the differences between the two plans, and how the recommended configurations can help in fixing the query regression.

[1] https://peps.python.org/pep-0249/



```
The following are two different plans for the same query:
P1 = $Plan1, P2 = $Plan2

A regression is observed when switching from P1 to P2

The query run on $DBMS.

Suggest configuration recommendations to speed-up P2, or revert it
to P1, if possible.

Your response should strictly consist of a list of SQL commands
compatible with $DBMS, and no additional text.

Your response should strictly follow the following JSON schema: {…}
```

```
{
  "plan_diff": the plan differences,
  "recommendations": the recommendations,
  "reasoning": explanation
}
```

Figure 1: DBG-PT

Prompt Generator. The prompt generator uses a base prompt template. It takes as input two query plans, and the hardware specification of the system, and renders them into the final prompt to be sent to the LLM to retrieve the three aforementioned pieces of information, along with the configuration recommendations to fix the query regressions.

3 DEMONSTRATION SCRIPT

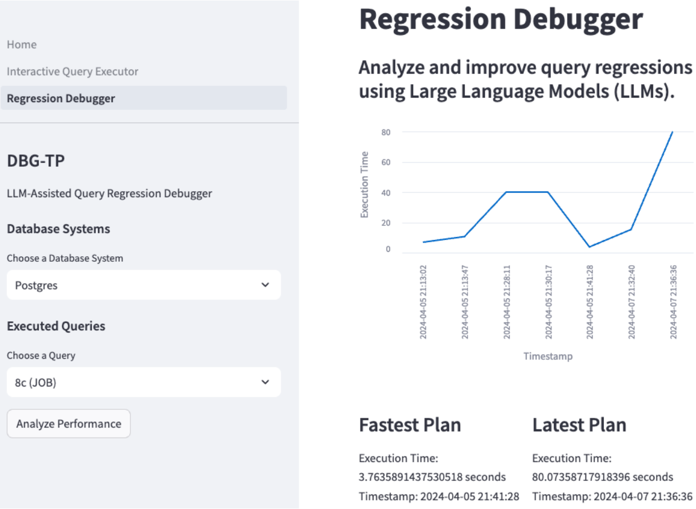
Our demonstration script consists of the following parts. First, users will be able to explore the available databases via the Interactive Query Executor (IQE) component of our user interface. Through this interface, users will be able to experiment with the following. First, they will be able to pick an input system (MySQL, Postgres, Spark SQL), database and workload, from our predefined library, which includes the Join Order Benchmark (JOB) and TPC-H. Users are welcome to experiment either using the predefined queries, or create and register their own ad-hoc ones to DBG-PT. Next, users will be able to experiment by modifying the tuning knobs for the specific query and see how the plan changes when they change the input knobs, for instance, by enabling or disabling specific join algorithms, modifying optimizer constants (e.g. scan costs), and enabling or disabling indexes. Next, they will be able to execute the query and measure its performance. Each query execution will be recorded in DBG-PT's database, along with its performance, and query execution plan. This way, users will be able next to compare the performance of the query under different execution contexts. This part of the demonstration will allow users to create their own query regression use-cases, in order to test the effectiveness of our LLM-assisted regression debugger in the latter demonstration phases.

In the next phase, users will be able to use DBG-PT in order to choose and debug pre-existing query regressions, that we will have created beforehand. Users will be prompted to pick and debug regressed queries from the following scenarios:

Scenario 1: Index-Related Regressions. In this scenario, users will be provided with a set of query regressions related to index availability. Two uses-cases will be provided in this scenario. In the first case, there will be a set of queries which performance eventually suffered because one or more *indexes where dropped.* This scenario will demonstrate how DBG-PT utilizes the LLM in order to compare the query plans, identify the missing index scans in the latest plan and finally recommend creating the appropriate indexes in order to restore the performance of the query. In the second use-case, we will demonstrate DBG-PT's ability to recover from a query regression which happened because new indexes were added to the database. Regressions of complex queries after index creations are a common scenario, as they are making the query optimizer's task even more complex, due to higher cardinality estimation errors [1, 4]. In this use-case, we will demonstrate how DBG-PT detects the occurrences of new index scans in the query plan, and the corresponding plan changes (e.g. join strategies) and its ability to recommend a set of configurations to revert to the initial efficient query plan.

Scenario 2: Knob-Related Regressions. In this scenario we well explore the use-case of query regressions due to the misconfiguration of a tuning knob. For instance, one tuning knob that is set globally because it benefits a subset of queries might cause another subset of queries to regress, e.g. by disabling the nested-loop joins. Users will be able to explore a set of regressed queries due to tuning knob changes and examine the differences between the execution times and structural differences among the different plans. Next, we will demonstrate how DBG-PT can use the underlying LLM in order to compare the two plans, identify the differences and recommend a set of configurations that will fix the regression, by utilizing past knowledge, i.e. the old, efficient query execution plan.

Scenario 3: Query Runtime Errors. In this scenario, we will showcase DBG-PT's ability in resolving runtime errors by incorporating query execution logs in the prompt. For instance, out-of-memory (OOM) errors are a very common issue in Spark SQL, especially in queries with multiple joins due to the exploding intermediate result sizes. In this scenario, we will demonstrate DBG-PT's effectiveness in leveraging execution logs during prompt construction, and preventing workers of crashing due to such errors by re-configuring the driver node with the appropriate knobs.



DBG-TP

LLM-Assisted Query Regression Debugger

Database Systems

Choose a Database System: Postgres

Executed Queries

Choose a Query: 8c (JOB)

Regression Debugger

Analyze and improve query regressions using Large Language Models (LLMs).

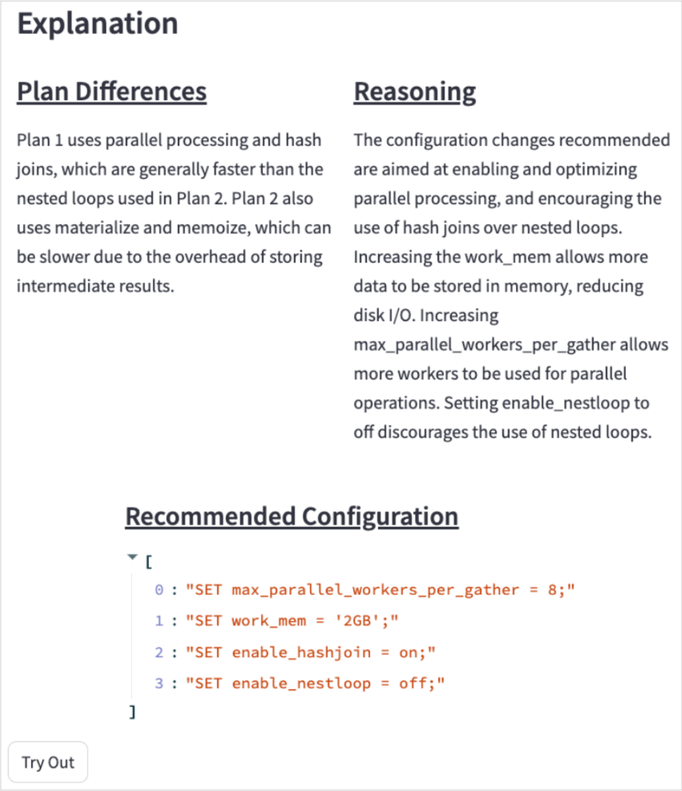
Fastest Plan

Execution Time:
3.7635891437530518 seconds
Timestamp: 2024-04-05 21:41:28

Latest Plan

Execution Time:
80.07358717918396 seconds
Timestamp: 2024-04-07 21:36:36

Figure 2: Regression Debugger: Performance Visualization



Explanation

Plan Differences

Plan 1 uses parallel processing and hash joins, which are generally faster than the nested loops used in Plan 2. Plan 2 also uses materialize and memoize, which can be slower due to the overhead of storing intermediate results.

Reasoning

The configuration changes recommended are aimed at enabling and optimizing parallel processing, and encouraging the use of hash joins over nested loops. Increasing the work_mem allows more data to be stored in memory, reducing disk I/O. Increasing max_parallel_workers_per_gather allows more workers to be used for parallel operations. Setting enable_nestloop to off discourages the use of nested loops.

Recommended Configuration

```
▾ [
    0 : "SET max_parallel_workers_per_gather = 8;"
    1 : "SET work_mem = '2GB';"
    2 : "SET enable_hashjoin = on;"
    3 : "SET enable_nestloop = off;"
]
```

Figure 3: DBG-PT: Regression Debugging

4 CONCLUSIONS AND RELATED WORK

Recently, there has been a significant interest in leveraging language models for database system tuning. DB-Bert [6] offers a reinforcement learning-based solution that extracts tuning recommendations from the database manual. GPTuner [3] introduces a Bayesian Optimization-based solution that extracts information both from the manual and a LLM. Additionally, D-Bot [9] employs an LLM-backed approach to detect and alleviate database anomalies, while Panda [5] focuses on performance debugging by providing troubleshooting recommendations based on execution metrics, rather than query plans. To the best of our knowledge, DBG-PT is the first approach to utilize LLMs for debugging regressed queries directly using information from executed plans. Unlike previous methods that primarily aim to optimize overall workload performance by adjusting tuning knobs without leveraging query-specific information, or at best use execution metrics as in Panda, DBG-PT pioneers in comparing executed plans of the same query to diagnose and fix performance regressions. Although DBG-PT is in its early stages, it has already demonstrated promising capabilities in automating query performance debugging. Through this demonstration, we will illustrate how our approach simplifies the complex task of debugging regressed query plans and showcases its effectiveness in leveraging the underlying LLM to resolve query regressions inspired by real-world scenarios.

REFERENCES

[1] 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.

[2] Victor Giannakouris and Immanuel Trummer. 2024. Demonstrating λ-Tune: Exploiting Large Language Models for Workload-Adaptive Database System Tuning. *Proceedings of the ACM SIGMOD (Accepted in the Demo-Track)* (2024).

[3] 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).

[4] 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.

[5] Vikramank Singh, Kapil Eknath Vaidya, Vinayshekhar Bannihatti Kumar, Sopan Khosla, Murali Narayanaswamy, Rashmi Gangadharaiah, and Tim Kraska. 2024. Panda: Performance debugging for databases using LLM agents. (2024).

[6] 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.

[7] Immanuel Trummer. 2023. From bert to gpt-3 codex: harnessing the potential of very large language models for data management. *arXiv preprint arXiv:2306.09339* (2023).

[8] Lianggui Weng, Rong Zhu, Bolin Ding Di Wu, Bolong Zheng, and Jingren Zhou. 2023. Eraser: Eliminating Performance Regression on Learned Query Optimizer.

[9] Xuanhe Zhou, Guoliang Li, and Zhiyuan Liu. 2023. Llm as dba. *arXiv preprint arXiv:2308.05481* (2023).