Data Science and Engineering
https://doi.org/10.1007/s41019-023-00235-6

RESEARCH PAPER

DB-GPT: Large Language Model Meets Database

Xuanhe Zhou[1] · **Zhaoyan Sun**[1] · **Guoliang Li**[1]

Received: 6 June 2023 / Revised: 27 June 2023 / Accepted: 26 July 2023

Abstract
Large language models (LLMs) have shown superior performance in various areas. And LLMs have the potential to revolutionize data management by serving as the "brain" of next-generation database systems. However, there are several challenges that utilize LLMs to optimize databases. First, it is challenging to provide appropriate prompts (e.g., instructions and demonstration examples) to enable LLMs to understand the database optimization problems. Second, LLMs only capture the logical database characters (e.g., SQL semantics) but are not aware of physical characters (e.g., data distributions), and it requires to fine-tune LLMs to capture both physical and logical information. Third, LLMs are not well trained for databases with strict constraints (e.g., query plan equivalence) and privacy-preserving requirements, and it is challenging to train database-specific LLMs while ensuring database privacy. To overcome these challenges, this vision paper proposes a LLM-based database framework (`DB-GPT`), including automatic prompt generation, DB-specific model fine-tuning, and DB-specific model design and pre-training. Preliminary experiments show that `DB-GPT` achieves relatively good performance in database tasks like query rewrite and index tuning. The source code and datasets are available at github.com/TsinghuaDatabaseGroup/DB-GPT.

Keywords Large language model · Database

1 Introduction

Large language models (LLMs) are pre-trained with a super large model capacity (e.g., over 170 billion network parameters in GPT-3 [1]) and a large data corpus (e.g., over 8 million website pages as training data), which is good at understanding human knowledge and instructions. Recently, LLMs have demonstrated superiority in various tasks like text generation [2], machine translation [3], and program synthesis [4]. Thus, a natural question is whether LLMs *can be used to accomplish database tasks*.

Task 1: Query Rewrite In Fig. 1, the query rewrite task is described in three parts. (*i*) *Instruction* includes the overall procedure and target of the task. In this case, we aim to write an equivalent query that can be executed on Postgres and achieves lower latency than origin query. Note it is critical to ensure the *LLM* captures the key points in the instruction, as these points may be overlooked or misunderstood. (*ii*) *Examples* are simplified demonstrations of query rewrite. These examples teach the *LLM* how to use rewrite rules, which cannot be well covered in the instruction. (*iii*) *Input* provides the necessary information to accomplish the task (e.g., the SQL query). We input the three parts in the form of (`[Instruction]`, `[Examples]`, `[Input]`) into LLMs, which asks LLMs to rewrite the input query (e.g., pulling up the nested subquery as table joins) and append the rewritten query after the input.

Task 2: Index Tuning Similarly, the index tuning task involves (*i*) the main procedure (e.g., creating a sequence of indexes) and task target (e.g., reducing the latency within limited space) and (*ii*) examples like inputting a two-table join query and outputting two indexes that use the columns in the queries and (*iii*) the input including some new queries, for which we need to create suitable indexes. In this case, the *LLM* recommends some indexes so as to actually optimize the bottleneck operators in the input queries (e.g., *orderby*) and avoid redundant indexes.

✉ Guoliang Li
liguoliang@tsinghua.edu.cn

Xuanhe Zhou
zhouxuan19@mails.tsinghua.edu.cn

Zhaoyan Sun
szy22@mails.tsinghua.edu.cn

[1] Department of Computer Science, Tsinghua University, Beijing, China

Published online: 19 January 2024

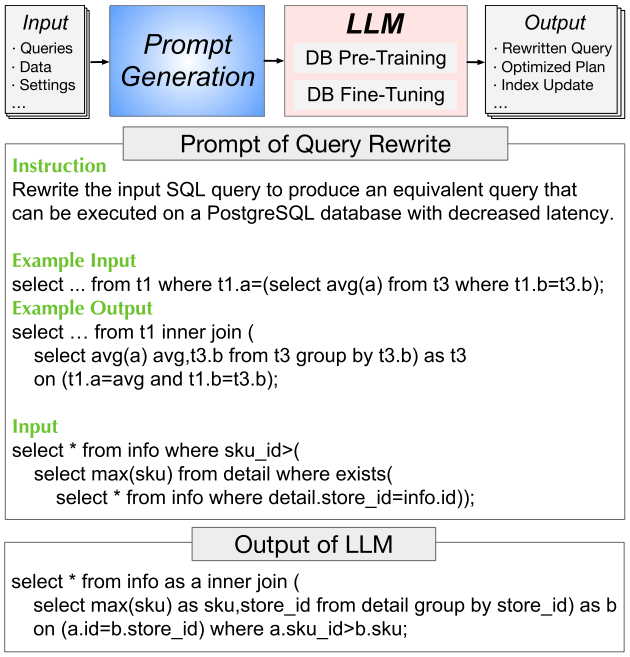


Fig. 1 Large language model for database

Compared with existing AI4DB works [5–7], LLMs for database (LLM4DB) have three advantages. (1) *Higher transfer capability*: Unlike existing instance-optimal works that can optimize an instance but cannot be extended to other instances, LLM4DB demonstrates exceptional transfer capability. By leveraging just a few fine-tuning samples, LLM4DB can achieve comparable performance on novel database tasks, making it adaptable to schema, workload, or even data and hardware changes; (2) *User-friendly interface*: LLM4DB offers an intuitive user experience by allowing users to provide some prompts as hints to guide the model's inference. Instead, AI4DB typically requires substantial amounts of training data (supervised models) or multiple iterations (reinforcement learning) to capture and incorporate user feedback. (3) *Prior Knowledge Learning*: LLM4DB is capable of extracting insights from existing database components, including documents or even code. By integrating the strengths of these components, LLM4DB can enhance its performance while mitigating their individual weaknesses.

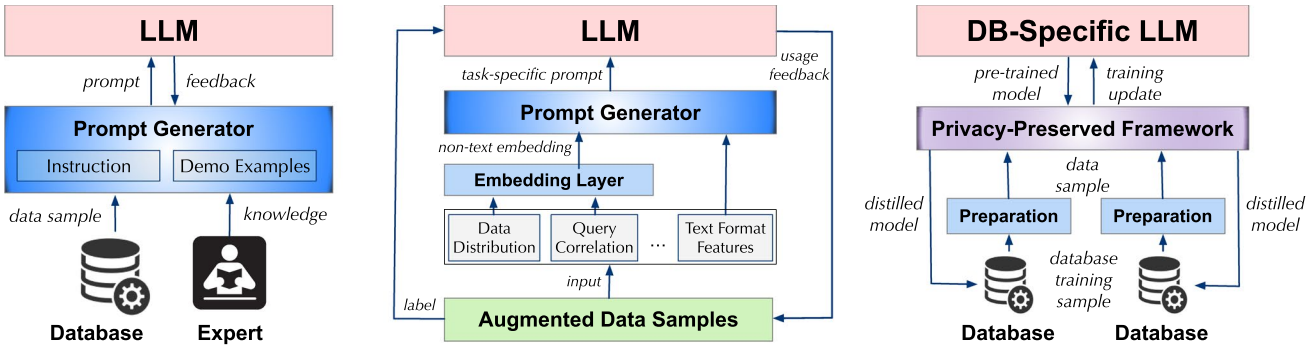
However, there remain some challenges to achieve comparable or even better performance.

C1 How to Generate Input Prompts for Database Tasks? First, the quality of the instructions provided to the *LLM* can affect its performance on a specific task. For instance, the writing style or complexity of the instructions may not be well suited for the model's comprehension, resulting in poor performance. Therefore, it is crucial to automatically select suitable task instructions (e.g., "*rewrite the SQL query to reduce complexity and improve performance...*") from a large pool of candidate instructions. Second, for the same task, it is important to provide some relevant examples for a given input (e.g., the rewrites of queries that are similar to the input query). These examples can provide insights on how to apply prior knowledge to handle complicated cases (e.g., rewriting queries that require to apply multiple rules) [8].

C2 How to Fine-Tune the LLMs for Database Tasks? First, data characters (e.g., data distributions, indexes) may significantly affect the optimization decisions of LLMs (e.g., building indexes for columns with a large number of distinct values). However, it is challenging for LLMs to capture the relations between data distribution and target tasks, e.g., describing the critical data characters in natural language or model-friendly embeddings. Second, since some database tasks only offer limited high-quality labeled samples (e.g., real queries with optimal rewritten strategy) for fine-tuning, we should explore how to better utilize the training samples.

C3 How to Design a Database-Specific LLM? First, different from NLP tasks, database tasks involve strict constraints (e.g., providing equivalent query plans for query rewrite) and structural information (e.g., the plan tree of a query), which are hard to support or learn by only using existing LLMs. Second, there are numerous public texts in NLP tasks, which can be taken as the training samples for LLMs. However, in databases, the data and queries are of high privacy, and it is vital to ensure the privacy while utilizing them to train the LLMs.

To tackle these challenges, we propose a database optimization framework by using LLMs (`DB-GPT`). We have three main contributions. ❶ We recommend several prompt generation methods that offer the valuable text information (e.g., task instructions [9–12], demonstration examples [2, 13–25]) to accomplish database tasks with high performance. The proposed methods include (*i*) automatically selecting the suitable task instruction, (*ii*) efficiently selecting demonstration examples with an RL model, and (*iii*) trading off between prompt length and *LLM* performance to reduce the model inference cost. ❷ We provide several methods to facilitate model fine-tuning using a small number of labeled samples for specific database tasks [21, 23, 26–29], including (*i*) non-text data embedding, (*ii*) annotations for low-quality data samples, (*iii*) contrastive learning for additional sample generation, and (*iv*) delta tuning that reduces the number of tunable network parameters while achieving similar performance. ❸ We propose to design and train a database-specific *LLM* [30, 31] with (*i*) validity checking for the output of LLMs, (*ii*) structural information learning from numerous database logs (composed of various workloads, optimization actions, and even result data) and (*iii*) federated learning that preserves data privacy during training the LLM.



(a) Input Prompt Generation. (b) LLM Fine-Tuning. (c) DB-Specific LLM Pre-Training.

Fig. 2 Three strategies of using *LLM* for database tasks

2 Opportunities of Using LLMs for DB

2.1 Overview

As shown in Fig. 2, this section presents three strategies that leverage LLMs to optimize database tasks, including *input prompt generation* (Sect. 2.2), *database-specific* LLM *fine-tuning* (Sect. 2.3), and *database-specific* LLM *design and pre-training* (Sect. 2.4). ***Input prompt generation*** aims to generate additional text information to guide LLMs in understanding the task requirements, which can directly use existing LLMs to optimize database tasks. Input prompt generation will not re-train *LLMs*, making it most efficient to use. However, it requires the LLMs to own the relevant database knowledge ahead of time, while only a few advanced LLMs like GPT−3.5 satisfy this requirement. *LLM **fine-tuning*** updates network parameters (a small part in delta tuning) so as to memorize task-specific knowledge, which can accept non-text features with additional embedding layers and achieve better performance than input prompt generation. ***DB-specific** LLM **design and pre-training*** require a large number of database-specific training samples to learn network parameters, which can serve as the foundation model for databases. That is, by providing essential system characters (e.g., equivalence verification) and training mechanisms (e.g., federated learning) for database tasks, it can enhance the effectiveness of both input prompt generation and LLM fine-tuning.

2.2 Input Prompt Generation

Motivation With the input x of a database task, we can add additional text information to the input x, called the input prompt x', which helps *LLM* to better understand the task requirements. However, different inputs may correspond to different optimal prompts (e.g., queries of different structures may require different rewrite examples) [2, 16, 22, 24, 25, 32, 33], and it is hard and tedious for users to give good prompts. We need to build a prompt generator to automatically derive the prompt for input x.

Methodology There are mainly two critical parts in prompts: (*i*) instruction to guide *LLM*, and (*ii*) demonstration examples for *LLM* to simulate. The generated prompts should be validated by the feedback of *LLM*, based on which we can optimize the generator for good *LLM* performance. Furthermore, when we use prompts to interact with *LLM*, we should reduce the interaction latency (e.g., reducing interaction rounds) and cost (e.g., the input token number of each interaction round) for real applications (Table 1).

Challenge 1 How to Automatically Generate Input Prompt? It is challenging to automatically generate appropriate instructions and demonstration examples to guide LLMs to optimize different database tasks with a limited number of prompt tokens (or interaction rounds with LLMs).

Vision 1 We can concatenate instruction and demonstration examples as additional text information in the prompt, which is organized as "*[Instruction] [Examples] Input: [x] Output:*." Note we place task instruction before demonstration examples for two reasons: (*i*) It follows the natural progression of teaching the model to finish a task, i.e., introducing the problem before showing how to solve it; (*ii*) Instruction provides the contextual information for the examples, making the model easier to build connection between the examples and task purpose (e.g., given the query rewrite instruction, the model can focus on the structural changes in the examples). Next we, respectively, explain how to automatically generate the instruction and demonstration examples (Fig. 3).

- *Instruction*

The quality of task instructions can impact the performance of *LLM* on different tasks. Thus, we present a method to

Table 1 LLM strategy comparison

	Input prompt	Fine-tuning	Pre-training
Tunable parameters	Thousands	Tens of billion	Hundred billion
Data samples	Dozens	Thousands	Millions
Input features	Text data	Text/non-text data	Text/non-text data

automatically generate instructions with a limited number of samples [10, 11]. First, we utilize *LLM* to suggest instruction candidates based on a small set of input–output pairs (e.g., five pairs for an instruction). Second, we rank these generated instructions based on a customized scoring function (e.g., the average performance on test workloads), and reserve the best instructions (e.g., top-10) as candidates. Third, we utilize search-based methods (e.g., Monte Carlo Search) to improve the candidates with *LLM* (e.g., outputting instruction variants with similar semantics as a candidate). Finally, we select the best instruction to serve as the input for the task.

- *Demonstration examples*

are selected from a candidate set $\{s_i\}$. Unlike instruction generation, example selection depends on the input x. If an example is more similar to the input, it provides more relevant information to the *LLM*. Specifically, we learn an input encoder $E_X(\cdot)$ and an example encoder $E_S(\cdot)$, and calculate the similarity between $E_X(x)$ and $E_S(s_i)$ for all candidate examples using L2 distance [2, 22, 24]. For instance, if x and s_i are both SQLs, they may share similar operators such as $COUNT(\cdot)$ or *JOIN*. We select the top k examples and place them before x in ascending order of similarity (i.e., more similar examples are closer to the input). Since adjacent tokens in an example s_i and x have similar position embeddings, this helps the *LLM* focus on the input–output mappings of the most similar examples [2]. Note that the candidate set of examples is typically collected from real-world applications, such as the 36 examples that cover typical rules for query rewrite. If there is no such candidate set for a new task, we can obtain a few hand-crafted examples from an expert, and use the *LLM* to derive more examples from them, (e.g., using prompts like "generate new queries with different structures but using the same rewrite rules").

Challenge 2: How to Efficiently Interact with LLM Using Prompts? Many database tasks require low latency. However, there are three factors that can increase the interaction latency and cost with *LLM*. (*i*) It can be time-consuming to actively generate prompt for input (e.g., retrieving suitable examples from candidate ones). (*ii*) Long prompts often include more useful information for *LLM*, but can take longer processing time for *LLM*. (*iii*) Some complex tasks can be better solved by calling *LLM* for several rounds and interactively adjusting the prompt. Thus, it is important to efficiently generate prompts and reduce the latency and cost of *LLM* interactions.

Vision 2 To address the issue of costly prompt generation, one possible solution is to train a reinforcement learning (RL) model, such as Q-learning, on a set of candidate examples. This model can be employed to identify the most suitable example for selection, thus eliminating the necessity to search through the entire collection of candidates [20]. By calling the RL model a fixed number of times, which is equal to the number of examples required in the prompt, we can generate the prompt more efficiently.

Secondly, it is important to strike a balance between prompt length and *LLM* performance. When selecting instructions and demonstration samples, we should not only consider their performance on the validation set but also prioritize shorter ones.

Finally, incorporating information from previous rounds into the prompt can improve the effectiveness of prompt-based interactions with *LLM*. For instance, when adjusting database knobs based on the outputs of the *LLM*, we can record the outputs (e.g., tuned knobs) and their actual performances (e.g., workload throughput under tuned knobs) from previous rounds, and incorporate them into the prompt. This can help the *LLM* make more accurate inferences during subsequent rounds of interaction [34].

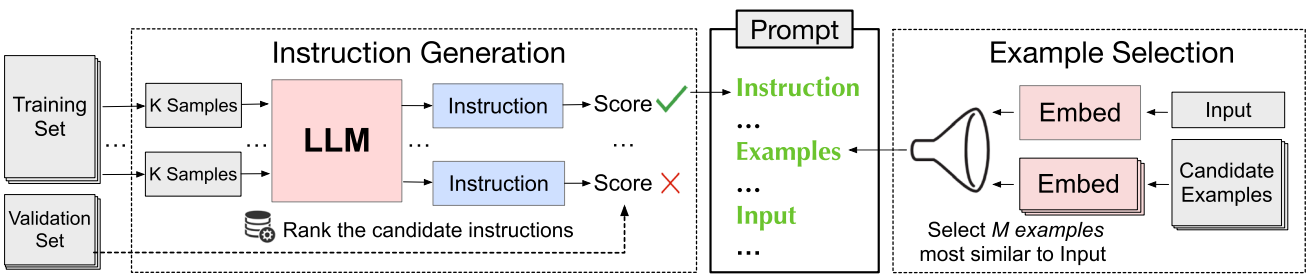


Fig. 3 Automatic prompt generator

2.3 LLM Fine-Tuning

Motivation Apart from text prompts, some database tasks (e.g., physical query plan generation) require non-textual information that is not readily expressible in natural language. Moreover, fine-tuning can enhance the task-specific performance of *LLM*. However, the effectiveness of the fine-tuned model is significantly influenced by the size and quality of the labeled data samples.

Methodology First, we can train non-text embeddings during fine-tuning and combine them with natural language embeddings. Second, we should make good utilization of existing data for model fine-tuning, and continuously collect new fine-tuning data from application feedbacks of *LLM*. Third, we explore new methods to enhance *LLM* fine-tuning.

Challenge 3 How to Embed Non-text Input Features? Some database features cannot be directly embedded as text due to its verbosity (e.g., data distributions), which easily exceeds the input length limit of *LLM* (e.g., ~3000 words for GPT−3.5 [35]). To address this issue, we need to explore how to embed non-text features and incorporate them with text input features during fine-tuning.

Vision 3 First, we provide two examples of non-text embeddings. (*i*) *Data distribution* is an important factor that affects various aspects of database tasks (e.g., operator costs, query results). We can use a model E_D to embed the distribution of table column data, and the embedding vector for column *t.c* is denoted as $E_D(t.c.data)$. For instance, we can first use quantiles such as (min, p01,..., p99, max) to approximate the distribution of the column data and then embed them with models like Transformer [30, 36]. Note that text embeddings represent the semantic and syntactic characteristics of text words. Thus, E_D should be trained to *convert these data characteristics into the same embedding space as text* during fine-tuning [37]. (*ii*) Query correlations reflect the execution state of workloads in the same database and form a graph model, which cannot be learned well by sequential models. Therefore, we can use a model E_R to embed the correlations between concurrent queries. The embedding vector for query q is denoted as $E_R(q, q.correlated\_queries)$, where $q.correlated\_queries$ are the queries accessing the same table columns as q. For instance, we can create a graph where each query is a node and the query correlations can be represented by the edge type and weight. With the graph, we can utilize models like graph neural network to embed the graph structural information into a vector $E_R(\cdot)$ [38, 39]. Note that the reason why *non-text features are not described in natural language* is that it may cause extremely long text input (e.g., a complex query may involve hundreds of columns and even more concurrent queries) and the relevant information may scatter sparsely among the long text, making it difficult for *LLM* to capture useful information.

Second, we can set the non-text embeddings to have the same dimensions as text embeddings, and train the non-text embedding models together with *LLM* in fine-tuning. During usage, we separately embed text features and non-text features, and then concatenate the embedding vectors to obtain an input sequence of *LLM*, which can be further processed.

Challenge 4 How to Provide Sufficient High-Quality Data for Fine-Tuning? Some database tasks may lack sufficient high-quality data (e.g., tens of thousands of samples) to fine-tune *LLM*. This may occur for two reasons. First, obtaining task-specific data is costly. For example, it takes weeks to collect fine-tuning data for knob tuning, which includes the target workload, knob settings, and the actual performance. Second, even with abundant data, there may be lack of high-quality annotations (e.g., reasoning processes for slow SQL query diagnosis) to facilitate the learning of task-specific knowledge by *LLM*. Therefore, it is essential to explore methods to make good use of existing data, and continuously collect data from *LLM* usage feedback.

Vision 4 We propose two potential solutions. First, we can use contrastive learning to generate additional fine-tuning samples from the dataset [27]. For instance, in knob tuning, we can obtain k knob settings along with their corresponding performance metrics. By using *LLM* to compare the performance of each pair of knob configurations, we can generate $\binom{k}{2}$ samples (which is significantly larger than k) for fine-tuning *LLM*.

Second, for low-quality data samples, we can leverage *LLM* to generate annotations, such as the reasoning process of data sample (chain of thought [23, 26, 40]), which can help to improve the quality of the data. For example, we can use LLMs to diagnose the root causes of slow database performance (e.g., the latency of a workload is over 50% higher than the normal latency), which can be used as annotations of monitoring metric data. Specifically, we first input the monitoring data (e.g., system views, query logs) and a set of potential annotations to *LLM*. Next, the *LLM* selects the annotation with the highest probability of causing the slow performance as the final annotation.

Third, we can monitor the performance of *LLM* and record scenarios where the LLM performs poorly by logging input features and its corresponding outputs. We add such data samples into fine-tuning data. We not only increase the size of our fine-tuning data, but also capture weaknesses in the *LLM*.

Challenge 5 How to efficiently fine-tune LLM? The large scale of LLM makes fine-tuning both time-consuming and costly. Therefore, we need to explore efficient fine-tuning methods.

Vision 5 Traditional fine-tuning allows to tune all the network parameters. Instead, tuning a small part of the mode parameters (delta tuning [28]) can achieve similar performance for the task and is much more efficient. Here, we introduce two delta-tuning methods. First, we can add extra tunable parameters to LLM. For example, we can inject adapter modules (e.g., small MLP) to each Transformer model [41]. We only tune these adapter modules during fine-tuning, and keep the original parameters of LLM frozen. Note that this method can also reduce the storage cost of the fine-tuned LLM. If we fine-tune LLM for many tasks, we can store the parameters of adapter modules and only store one copy of the original LLM. Second, the fine-tuning process optimizes parameters from Θ to Θ', and we denote the difference of parameters as $\Delta\Theta = \Theta' - \Theta$. Then, we can decompose $\Delta\Theta$ into low-dimensional representations, e.g., $W = BA$, where W denotes weight matrix of $\Delta\Theta$ and B, A denote the weight parameters we actually tune [42]. We assume $W \in R^{d\times k}$, $B \in R^{d\times r}$, and $A \in R^{r\times k}$ ($r \ll d$), where d, k denote the size of the weight matrix, and r denotes the decomposed rank. B and A have much fewer parameters than W and are more efficient to tune.

2.4 DB-Specific LLM Design and Pre-training

Motivation Unlike natural language tasks, database tasks have unique characteristics such as strict output constraints and a large amount of unique data features (e.g., data distributions, metadata, data statistics, and query logs). However, existing LLMs only use text data on surface web but do not capture the data in hidden database, and thus face difficulties in handling complicated database tasks.

Methodology First, we need a new model design to ensure the validity of the LLM output. Second, there are a large-scale diverse data in database, e.g., SQLs and their physical plans, slow SQLs, database metrics, database statistics, etc. We should collect and organize them into training samples for LLMs and teach LLMs how databases work. Third, data in database tasks is often business-sensitive and mission critical. We should avoid data leakage during both *LLM* pre-training and *LLM* inference.

Challenge 6 How to Ensure the Validity of LLM Output? Although *LLM* cannot guarantee 100% accuracy on the task results, certain database tasks require strict constraints (e.g., the output of query rewrite must be a semantically equivalent query and the query must be executable on the database). It is essential to ensure that all invalid outputs are detected or avoided.

Vision 6 We propose to adopt a hybrid method to ensure the validity of model output. First, we should design special training set so that *LLM* can maximize the possibility of generating valid outputs (e.g., queries that satisfy the SQL syntax). Second, for relatively simple cases, we adopt a non-learned checking layer to validate the output (e.g., using an SMT solver for simple SPJ queries [43]). For more complex cases, we adopt a learned checking layer (e.g., a binary classifier) to validate the output. Similar to a learned Bloom filter [44, 45], if the output is judged as "invalid", the learned checking layer feeds back the illegal result to the *LLM*, and the LLM regenerates the output. Otherwise, the output needs to be double-checked on simplified cases (e.g., comparing the execution results of the original and rewritten queries on sampled data) [28].

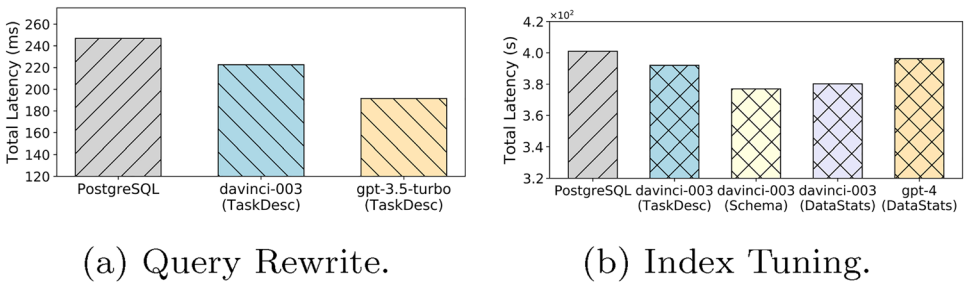
Challenge 7 How to Train LLM with Database Data? A significant amount of data is required to train a high-quality *LLM* [1, 46–49]. For a database-specific *LLM*, we expect it to learn general knowledge that applies to all database tasks. It can learn basic knowledge from database documents or blogs by human experts, but such natural language texts are of a limited size. Thus, it is extremely important to learn a database-specific *LLM*.

Vision 7 Training data in databases has different characteristics compared to natural language text corpus. First, the database training samples may have different formats, e.g., well-structured SQLs and query plans, semi-structured logs, and unstructured documents. Thus, we need to well represent different data samples and concatenate them for training *LLM* effectively.

For example, we can record query execution and obtain a sequence like "[Table Data][Query]→[Logical Plan]→[Physical Plan]→[Result][Execution Time]." By learning the relevance within such sequences, *LLM* can automatically learn how to conduct query optimization. Second, database data contains plenty of structural information (e.g., tree structure of query plan). Special designs are required to enable the DB-specific *LLM* to better utilize the structure information. For example, we can combine the design of graph neural networks to support complex graph topological structures [50].

Moreover, since some database data is very long and exceeds the length limit of Transformer-based LLMs, we can use long-range attention to learn correlations among distant words [51]. Specifically, we can store the encoded vectors (keys and values) of historical tokens in a large external memory. When training *LLM* for a particular token x_i, we can search for the k nearest vectors in the external memory and use the attention mechanism to encode x_i with them. This enables *LLM* to learn basic knowledge from long database training samples.

Challenge 8 How to pre-train and apply LLM while ensuring data privacy? User data in a database must be protected from leakage due to privacy concerns. This prevents user data from being integrated into the database provider for



(a) Query Rewrite. (b) Index Tuning.

Fig. 4 Performance comparison. **a** Query rewrite on 12 real queries of the *Shopmall* and *Goods* datasets. **b** Index tuning on 10 workloads of the *imdbload* dataset. The index space is limited within 500 M

centralized *LLM* pre-training. Moreover, some users cannot send their sensitive data as input to DB-GPT through inference APIs.

Vision 8 To address this challenge, we propose two solutions. First, we can use privacy-preserving federated learning to train *LLM* [30, 31]. In this approach, a server (e.g., database provider) collaborates with clients (e.g., users) to train *LLM* for several rounds. In each round, the client receives some server information (e.g., server network parameters) and updates their local network parameters. They then train their local model with their local data and send some local information (e.g., their local gradients) to the server. The server updates its network parameters by aggregating the local information from clients, and starts the next round by sending updated server information. This approach ensures data privacy since users always manipulate their own data.

Secondly, due to business reasons, we are unable to directly provide the network parameters in DB-GPT to users. Instead, we can distill the knowledge of DB-GPT into a smaller model that performs similarly to LLMs for desired tasks and provide this model to the user [26, 52]. Specifically, we require the user to provide a dataset of their database's application scenarios (e.g., workloads, data distributions). Using this dataset, we annotate with DB-GPT output (see details in Sect. 2.2) and train a simplified model with this new dataset. It is important to note that user applications are relatively deterministic, which makes a smaller model sufficient despite its lower generalizability.

3 Practices

We have conducted preliminary practices to show the effectiveness of automatically generated prompts, i.e., LLMs equipped with these prompts can achieve comparable performance as the state-of-the-art methods. To explore the upper limits of these LLMs, we test with two most advanced models (checked until June, 2023), including "text-davinci-003' that owns 175 billion parameters and supports up to 4,097 tokens per request, "gpt-4" that supports up to 8,192 tokens per request and generally can follow more complex instructions.

3.1 Implementation

There are three main steps in our initial implementation of DB-GPT. First, we generate the instruction from a small number of collected samples (splitting into training and evaluation sets), i.e., deriving several instructions using the *LLM* on training set and choosing the best instruction by evaluating on evaluation set (Sect. 2.2). Second, based on the task requirements, we collect other input features such as demonstration examples (e.g., query rewriting pairs) and data statistics (e.g., distinct value ratios of the columns). Finally, we concatenate the instruction, collected features, and the input into a prompt sequence, and rely on the *LLM* to output desired results based on the prompt sequence.

3.2 DB-GPT for Query Rewrite

Datasets We generate instructions on the training set and verify the performance on the evaluation set. (*i*) The training set contains 36 query rewrite pairs $\{(x, y)\}$, where y is rewritten from x with a unique rewrite rule. Thus, the training set covers some frequently used rewrite rules in real usage. (*ii*) The evaluation set contains 12 queries sampled from two real-word datasets: the dataset *Shopmall* contains 78 tables and 298,208 sampled tuples; and the dataset *Goods* contains 41 tables and 683,130 sampled tuples .[1]

Empirical Analysis As shown in Fig. 4a, DB-GPT (davinci-003) and DB-GPT (gpt−3.5-turbo) separately outperform the *PostgreSQL* rewriter by 9.8% and 22.4%. First, DB-GPT (davinci-003) effectively rewrites 6 out of the 12 queries, which results in latency reductions ranging from 0.6% to 82.5%. With the proper instruction, i.e., "*Rewrite the input SQL query to produce an equivalent query that can be executed on a PostgreSQL database with decreased latency*," DB-GPT (davinci-003) is inspired to apply more complex rewrite rules than the built-in rules in *PostgreSQL*, such as converting subqueries with aggregations into joins. Second, DB-GPT (gpt−3.5-turbo) outperforms DB-GPT (davinci-003) by optimizing some complex queries (e.g., queries that involve both join reordering and join predicate

[1] https://github.com/TsinghuaDatabaseGroup/DB-GPT/tree/main/prompt_template_scripts

pushdown), where the *davinci-003* model fails to address. Because *gpt-3.5-turbo* is trained on larger human language datasets, enabling it to undertake complex reasoning tasks (e.g., multi-step rewrites). Moreover, to further optimize the rewrite performance, we incorporate a demonstration example in DB-GPT models' prompt, which outlines the step-by-step process for the complex rewrites that DB-GPT without demonstration example fails to achieve (shown in Fig. 1).

We find the models used in DB-GPT learn to replace the subquery "sku_id > (...detail.store_id = info.id)" with INNER JOIN operator, and the subquery no longer depends on the external query. Note the demonstration example is not the same as the input query (e.g., no ">" in the example, and the names are abbreviated), which guides DB-GPT to rewrite queries involving similar structures. Besides, DB-GPT can also replace the subquery "exists(...)" with INNER JOIN operator, which is not included in the example. That means that DB-GPT can utilize the knowledge from the example without forgetting the rewrite rules learned from pre-training.

Limitations First, we should design a candidate example set for practical usage, which is large enough and covers all common rewrite skills (e.g., typical rules or even rule combinations). Then for an input query, similar examples can be selected from it, which can guide *LLM*. Second, query rewrite is a multi-step reasoning task, and we may interact with *LLM* for multiple rounds to obtain a better rewritten query [34]. For instance, we can input the temporarily rewritten query to *LLM*, and ask it to further rewrite it. This process can be repeated until the output query is optimal (e.g., evaluated by *LLM*) or reaching time limit.

DB-GPT for Index Tuning

Datasets. We first generate 60 instructions from the 180 pairs (workload, optimal indexes) on the *TPC-H* and *TPC-DS* datasets, from which we take the prompt that achieves the best performance on the 20 workloads of the *IMDB* dataset. Apart from instruction (*LLM(TaskeDesc)*), we test the input prompt with table schema (*LLM(Schema)*) and the input prompt with data statistics like distinct value ratios and tuple numbers (*LLM(DataStats)*).

Empirical Analysis The experimental results are shown in Fig. 4b. We have two observations. First, the three prompting methods for DB-GPT all achieve performance improvement over *PostgreSQL*, which does not recommend indexes for specific workloads.

Because DB-GPT finds operators that can be enhanced by indexes and builds indexes on the columns of these operators (e.g., create index on person_info(person_id, info_type_id) for the predicate "`person_info.person_id > 4158523 AND person_info.info_type_id <> 22`"). Second, for the three different prompts, *LLM(DataStats)* achieves lowest estimated costs, while *LLM(Schema)* works best in actual latency. Because *LLM(Schema)* selects many more single-column indexes than *LLM(DataStats)*, which may achieve higher estimated cost but are more flexibly used in actual execution. Thus, we need to judiciously select the data information (e.g., column types and data distributions) for prompts. Moreover, it is noteworthy that the performance of *gpt-4* is even inferior to that of the *text-davinci-003* model. This observation suggests that the relation between desired indexes and the physical constraints (e.g., storage budget, data characters) cannot be readily acquired by even extremely powerful language models, and causes misunderstanding. For example, almost 78% indexes selected by the *gpt-4* model are removed due to storage limit, because their space consumption cannot be well estimated by the language model without careful fine-tuning.

Limitations First, DB-GPT should be aware of the storage space for each index, which guides to achieve trade-off between performance and space consumption. Currently, we greedily limit the adopted indexes by DB-GPT (i.e., removing the remaining indexes after the index space is full). Second, different from query rewrite, index tuning involves many queries in a workload, which may exceed the input length limitation in the LLMs. Thus, it is vital to embed the workload queries into acceptable length or design new LLMs, especially for high-concurrency transactions. Third, index tuning also requires iteration mechanism that allows DB-GPT to update the index set for multiple times before fully optimizing the performance.

In summary, although LLMs demonstrate potential in handling database tasks, there are still many opportunities for further research. Firstly, there is a need for a well-designed LLM4DB architecture that supports common database functionalities. For instance, a comprehensive LLM can be designed to forecast and schedule workloads across different components, with each component or function is driven by a fine-tuned LLM. Additionally, it is crucial to improve the efficiency of model inference. This requires the model to generate reasonable predictions or decisions without excessively consuming system resources. Some potential approaches include distilling the model into smaller models and supporting concurrent model inferences.

4 Related Work

Large Language Models

Different from other ML models, large models are designed to *scale with increasing amounts of training data and computation resources* [53–56]. There are mainly two types of large model designs for natural language [57]. (*i*) ***Unidirectional language models*** (e.g., GPT-3 [1]) use a single direction of text (allowing for efficient parallelization on increased computation power), i.e., assigning a possibility

to a token sequence $x = (x_1, ..., x_n)$. For instance, if we break down the sequence from left to right, the model calculates $P(x) = P(x_1) \times ... \times P(x_n|x_1, ..., x_{n-1})$. Thus, such models are good at predicting tokens following the input text (e.g., text generation tasks). (*ii*) ***Masked language model***s (e.g., BERT [58], ERNIE [59]) mask some tokens in the text input, and aim to predict them based on the contextual information. For instance, *LLM* can calculate $P(x_i|x_1, ..., x_{i-1}, x_{i+1}, ..., x_n)$, where x_i is the masked token. Such models are more suitable to fill in the middle of the text (e.g., text classification tasks).

Large Language Models for Databases Some initial attempts have been made to use large language models for database [60, 61]. DB-BERT [60] relies on the BERT model to summarize the database tuning manuals into rules. These rules can be used to enhance the configuration exploration procedure, since the RL model only selects rules to tune the database rather explores the whole space. On the other hand, CodexDB [61] aims to utilize the GPT-3 model to translate SQL queries into other languages (e.g., Python), which achieves an accuracy of around 60%. These works show promise in promoting the use of large models for databases.

5 Conclusion

In this paper, we explored the use of large language models (LLMs) for accomplishing database tasks. We proposed a system `DB-GPT` that effectively uses LLMs for optimizing database tasks, including prompt generation, fine-tuning LLM, and database-specific LLM design. In the initial practice, we verified the relatively good performance of `DB-GPT` using prompt generation for database tasks like query rewrite and index tuning. We believe the use of LLMs will continue to benefit the field of database systems, including text2SQL, SQL2Plan, database diagnosis, and data tuning.

Acknowledgements This paper was supported by National Key R\&D Program of China (2023YFB4503600), NSF of China (61925205, 62232009, 62102215), Zhongguancun Lab, Huawei, TAL education, and Beijing National Research Center for Information Science and Technology (BNRist).

Authors' contributions GL makes contributions on the ideas. XZ contribution is fine-tuning LLM. ZS contribution is DB-specific LLM.

Data avilability https://github.com/TsinghuaDatabaseGroup/DB-GPT

Declarations

Competing interest Not applicable.

References

1. Brown Tom B et al (2020) Language models are few-shot learners. Adv Neural Inf Proc Syst 2020:1877–1901
2. Liu J, Shen D, Zhang Y, Dolan B, Carin L, Chen W (2022) What Makes Good In-Context Examples for GPT-3? DeeLIO 2022(3):100–114
3. Floridi L, Chiriatti M (2020) GPT-3: its nature, scope, limits, and consequences. Minds Mach 30(4):681–694
4. Svyatkovskiy A, Deng S K, Fu S, Sundaresan N (2020) Intellicode compose: code generation using transformer. In: FSE, pp 1433–1443
5. Zhou X, Chai C, Li G, Sun J (2022) Database meets artificial intelligence: a survey. IEEE Trans Knowl Data Eng 34(3):1096–1116
6. Li G, Zhou X, Cao L (2021) AI meets database: AI4DB and DB4AI. In: SIGMOD '21: International conference on management of data, virtual event, China, Jun 20-25. ACM, pp 2859–2866
7. Zhang X, Wu H, Chang Z, Jin S, Tan J, Li F, Zhang T, Cui B (2021) restune: resource oriented tuning boosted by meta-learning for cloud databases. In: SIGMOD '21: International conference on management of data, virtual event, China, Jun 20-25. ACM, pp 2102–2114
8. Dong Q, Li L, Dai D, Zheng C, Wu Z, Chang B, Sun X, Xu J, Li L, Sui Z (2022) A survey for in-context learning. arXiv preprint arXiv:2301.00234
9. Sorensen Taylor et al (2022) An information-theoretic approach to prompt engineering without ground truth labels. In: Proceedings of the 60th annual meeting of the association for computational linguistics, vol 1. Association for computational linguistics, Stroudsburg, pp 819–862
10. Zhou Y, Muresanu A I, Han Z, Paster K, Pitis S, Chan H, Ba J (2022) Large language models are human-level prompt engineers. In: The Eleventh International Conference on Learning Representations
11. Honovich O, Shaham S R B, Omer L (2022) instruction induction: from few examples to natural language task descriptions, pp 1–17
12. Razeghi Y, Logan R L, Wallace E, Singh S (2020) AUTOPROMPT : eliciting knowledge from language models with automatically generated prompts, pp 4222–4235
13. Yao L, Bartolo M, Moore A, Riedel S, Stenetorp P (2022) Fantastically ordered prompts and where to find them: overcoming few-shot prompt order sensitivity. 1:8086–8098
14. Zhao T Z, Wallace E, Feng S, Klein D, Singh S (2021) Calibrate before use: improving few-shot performance of language models. In International Conference on Machine Learning, pp 12697–12706
15. Fu Y, Peng H, Sabharwal A, Clark P, Khot T (2022) Complexity-based prompting for multi-step reasoning, pp 1–14
16. Kim H J, Cho H, Kim J, Kim T, Yoo K M, Lee S-G (2022) Self-generated in-context learning: leveraging auto-regressive language models as a demonstration generator
17. Wei J, Wang X, Schuurmans D, Bosma M, Ichter B, Xia F, Chi E, Le Q, Zhou D (2022) Chain-of-thought prompting elicits reasoning in large language models. Adv Neural Inf Proc Syst 35:24824–24837

18. Press O, Zhang M, Min S, Schmidt L, Smith N A, Lewis M (2022) Measuring and narrowing the compositionality gap in language models, pp 1–25
19. Zhou D, Schärli N, Hou L, Wei J, Scales N, Wang X, Schuurmans D, Cui C, Bousquet O, Le Q, Chi E (2022) Least-to-most prompting enables complex reasoning in large language models
20. Zhang Y, Feng S, Tan C (2022) Active example selection for in-context learning
21. Wang X, Zhu W, Wang William Y (2023) Large language models are implicitly topic models: explaining and finding good demonstrations for in-context learning. arXiv preprint arXiv:2301.11916
22. Wu Z, Wang Y, Ye J, Kong L Self-adaptive in-context learning. arXiv preprint arXiv:2212.10375
23. Shao Z, Gong Y, Shen Y, Huang M, Duan N, Chen W (2023) Synthetic prompting: generating chain-of-thought demonstrations for large language models
24. Rubin O, Herzig J, Berant J (2022) Learning to retrieve prompts for in-context learning. In: NAACL 2022 - 2022 Conference of the north American chapter of the association for computational linguistics: human language technologies, proceedings of the conference, pp 2655–2671
25. Levy I, Bogin B, Berant J (2022) Diverse demonstrations improve in-context compositional generalization
26. Magister L C, Mallinson J , Adamek J , Malmi E, Severyn A (2022) Teaching small language models to reason. arXiv preprint arXiv:2212.08410
27. Ouyang L, Wu J, Jiang X, Almeida D, Wainwright C L, Mishkin P, Zhang C, Agarwal S, Slama K, Ray A, Schulman J, Hilton J, Kelton F, Miller L, Simens M, Askell A, Welinder P, Christiano P, Leike J, Lowe R (2022) Training language models to follow instructions with human feedback. Advances in Neural Information Processing Systems, 35, 27730–27744
28. Ding N, Qin Y, Yang G, Wei F et al (2022) Delta tuning: a comprehensive study of parameter efficient methods for pre-trained language models. CoRR, arXiv:abs/2203.06904
29. Liu X, Zheng Y, Du Z, Ding M, Qian Y, Yang Z, Tang J (2021) GPT understands, too. AI Open
30. Yang Z, Liang E, Kamsetty A, Wu C, Duan Y, Chen X, Abbeel P, Hellerstein JM, Krishnan S, Stoica I, Berkeley UC (2019) Deep unsupervised cardinality estimation, vol 13
31. Yin X, Zhu Y, Hu J (2021) A comprehensive survey of privacy-preserving federated learning: a taxonomy, review, and future directions. ACM Comput Surv 54(6):1
32. Liu R, Wei J, Gu S S, Wu T-Y, Vosoughi S, Cui C, Zhou D, Dai A M (2022) Mind's eye: grounded language model reasoning through simulation, pp 1–18
33. Liu J, Liu A, Ximing L, Welleck S, West P, Le Bras R, Choi Y, Hajishirzi H (2022) Generated knowledge prompting for commonsense reasoning. In Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (volume 1: Long Papers), pp 3154–3169
34. Creswell A, Shanahan M (2022) Faithful reasoning using large language models. Number, pp 1–48
35. https://platform.openai.com/docs/models/gpt-3-5
36. Yang Z, Kamsetty A, Luan S, Liang E, Duan Y, Chen X, Stoica I (2020) Neurocard: one cardinality estimator for all tables. In: Proceedings of the VLDB endowment, vol 14, pp 61–73
37. Bin W, Angela W, Fenxiao C, Wang Yuncheng C (2019) Methods and experimental results. C. Jay Kuo, Evaluating word embedding models
38. Zhou J, Cui G, Shengding H, Zhang Z, Yang C, Liu Z, Wang L, Li C, Sun M (2020) Graph neural networks: a review of methods and applications. AI open 1:57–81
39. Zhou X, Sun J, Li G, Feng J (2020) Query performance prediction for concurrent queries using graph embedding. Proc VLDB Endow 13(9):1416–1428
40. Wiegreffe S, Hessel J, Swayamdipta S, Riedl M, Choi Y (2022) Reframing human-AI collaboration for generating free-text explanations. NAACL 2022:632–658
41. Houlsby N, Giurgiu A, Jastrzębski S, Morrone B, Laroussilhe Q de, Gesmundo A, Attariyan M, Gelly S (2019) Parameter-efficient transfer learning for NLP. In: 36th International conference on machine learning, ICML 2019, 2019 Jun, pp 4944–4953
42. Hu EJ, Shen Y, Wallis P, Allen-Zhu Z, Li Y, Shean W L, W, and Weizhu C (2021) LoRA: low-rank adaptation of large language models. 10:1–26
43. De Moura L, Bjørner N (2008) Z3: An efficient smt solver. ETAPS. Springer, Cham, pp 337–340
44. Kraska T, Beutel A, Chi E H, Dean J, Polyzotis N (2018) The case for learned index structures. In: Proceedings of the 2018 international conference on management of data, pp 489–504
45. Mitzenmacher M (2018) A model for learned bloom filters and related structures. arXiv preprint arXiv:1802.00884,
46. Sun Y, Wang S, Feng S, Ding S, Pang C, Shang J, Liu J, Chen X, Zhao Y, Yuxiang L, Liu W, Zhihua W, Gong W, Liang J, Shang Z, Sun P, Liu W, Ouyang X, Dianhai Y, Tian H, Hua W, Wang H (2021) Ernie 3.0: large-scale knowledge enhanced pre-training for language understanding and generation
47. Zhang S, Roller S , Goyal N, Artetxe M, Chen M, Chen S, Dewan C, Diab M, Li X, Lin XV, Mihaylov T, Ott M, Shleifer S, Shuster K, Simig D, Koura PS, Sridhar A, Wang T, Zettlemoyer L (2022) OPT: open pre-trained transformer language models. arXiv preprint arXiv:2205.01068
48. Zeng A, Liu X, Zhengxiao D, Wang Z, Lai H, Ding M, Yang Z, Yifan X, Zheng W, Xia X, Tam WL, Ma Z, Xue Y, Zhai J, Chen W, Zhang P, Dong Y, Tang J (2022) GLM-130B: an open bilingual pre-trained model. 06
49. Chowdhery A et al (2022) Palm: scaling language modeling with pathways
50. Han J, Rong Y, Xu T, Huang W (2022) Geometrically equivariant graph neural networks: a survey. arXiv preprint arXiv:2202.07230
51. Wu Y, Rabe M N, Hutchins D, Szegedy C (2022) Memorizing transformers, pp 1–19
52. Gou J, Baosheng Yu, Maybank SJ, Tao D (2021) Knowledge distillation: a survey. Int J Comput Vision 129(6):1789–1819
53. Qiu XP, Sun TX, YiGe X, Shao YF, Dai N, Huang XJ (2020) Pre-trained models for natural language processing: A survey. Sci Chin Technol Sci 63(10):1872–1897
54. De Mulder W, Bethard S, Moens M-F (2015) A survey on the application of recurrent neural networks to statistical language modeling. Comput Speech Language 30(1):61–98
55. Li J, Tang T, Zhao W X, Wen J-R (2021) Pretrained language models for text generation: A survey. arXiv preprint arXiv:2105.10311,
56. Jing K, Xu J (2019) A survey on neural network language models. arXiv preprint arXiv:1906.03591
57. Pengfei L, Weizhe Y, Jinlan F, Zhengbao J, Hiroaki H, Graham N (2023) Pre-train, prompt, and predict: a systematic survey of prompting methods in natural language processing 55:1–35
58. Devlin Jacob, Chang Ming-Wei, Lee Kenton, Toutanova Kristina (2019) Bert: Pre-training of deep bidirectional transformers for language understanding. In Proceedings of NAACL-HLT, pp 4171–4186
59. Zhang Z, Han X, Liu Z, Jiang X, Sun M, Liu Q (2020) ErniE: Enhanced language representation with informative entities. ACL, pp 1441–1451
60. Trummer I (2022) DB-BERT: a database tuning tool that "reads the manual". In: SIGMOD, pp 190–203
61. Trummer I (2022) Codexdb: synthesizing code for query processing from natural language instructions using GPT-3 codex. Proc VLDB Endow 15(11):2921–2928