Demonstrating GPT-DB: Generating Query-Specific and Customizable Code for SQL Processing with GPT-4

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

ABSTRACT

GPT-DB generates code for SQL processing in general-purpose programming languages such as Python. Generated code can be freely customized using user-provided natural language instructions. This enables users, for instance, to try out specific libraries for SQL processing or to generate non-standard output while processing.

GPT-DB is based on OpenAI's GPT model series, neural networks capable of translating natural language instructions into code. By default, GPT-DB exploits the most recently released GPT-4 model whereas visitors may also select prior versions for comparison. GPT-DB automatically generates query-specific prompts, instructing GPT on code generation. These prompts include a description of the target database, as well as logical query plans described as natural language text, and instructions for customization. GPT-DB automatically verifies, and possibly re-generates, code using a reference database system for result comparisons. It enables users to select code samples for training, thereby increasing accuracy for future queries. The proposed demonstration showcases code generation for various queries and with varying instructions for code customization.

PVLDB Reference Format:
Immanuel Trummer. Demonstrating GPT-DB: Generating Query-Specific and Customizable Code for SQL Processing with GPT-4. PVLDB, 16(12): 4098 - 4101, 2023.
doi:10.14778/3611540.3611630

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/itrummer/CodexDB.

1 INTRODUCTION

Database management systems typically process queries using a fixed set of standard operator implementations. Their behavior can be influenced via system parameters but this allows only a limited degree of customization. Using a general-purpose programming language to write query-specific code for SQL processing gives developers maximal flexibility. This is useful, for instance, to achieve optimal performance tradeoffs for specific, frequently occurring queries. It can also be useful to test ideas for novel processing mechanisms on a small set of benchmark queries, before implementing a full-blown processing engine based on those ideas. Also, writing custom code enables non-standard output, allowing users, for instance, to gain detailed information describing intermediate results. The flexibility of query-specific code comes at a price: even for experienced developers, writing such code is time-consuming and error-prone. GPT-DB [10] (formerly CodexDB) helps by partially automating the code generation process, specifically for SQL queries, taking into account natural language instructions for customization.

Similar to GitHub's Copilot [4], GPT-DB is based on the GPT model [7]. This model can translate natural language instructions into code. Different from GitHub Copilot, GPT-DB is specialized for generating code that processes SQL queries. Users select a database (characterized by a schema and by files storing base table data) and submit SQL queries, along with natural language instructions on how to generate associated code. Those instructions may, for instance, instruct GPT-DB to use specific libraries or to generate specific, non-standard output. Then, GPT-DB generates one or multiple code variants, processing the query on disk files and writing the query result to disk when executed. Furthermore, GPT-DB offers users several options for automated verification, including processing queries on a reference system to verify correctness of generated results. If satisfied with generated code, users can add code to a library of code samples. These samples are used to train GPT-DB, thereby improving result quality.

One of the primary contributions of GPT-DB is the automated generation of query-specific "prompts". Prompts are small text documents that serve as input to the GPT model. These documents describe the code to generate in natural language. Prompts generated by GPT-DB integrate a description of the database schema and paths to the associated files (both generated using GPT-DB's catalog). Furthermore, GPT-DB generates a query plan that describes processing steps at a high level of abstraction (enabling customization via user instructions). Different from traditional query plans, this plan is described in natural language to enable integration into the prompt. Finally, GPT-DB integrates user-provided natural language instructions into the prompt for customization. The resulting prompt is submitted as input to GPT, triggering code generation.

The proposed demonstration enables visitor to use GPT-DB live. Visitors may select between several hundred example databases, enter queries, and specify natural language instructions influencing the code generated for query processing. Beyond that, the demo enables users to try different prompt structures, compare GPT model variants, and to experiment with various automated code verification strategies. The demonstration interface shows generated code for query processing, query results obtained by executing that code, as well as logs detailing GPT-DB's internal processing steps.

Proceedings of the VLDB Endowment, Vol. 16, No. 12 ISSN 2150-8097.
doi:10.14778/3611540.3611630

2 BACKGROUND AND PRIOR WORK

The proposed demonstration focuses on GPT-DB, described in more detail in a recent paper [10] but never demonstrated before[1]. GPT-DB is enabled by recent advances in the domain of language models (including models for natural language as well as models for code). These advances are based on two key ideas: a new neural network architecture, the Transformer [13], and a particularly successful application of the general idea of transfer learning.

Among other advantages, Transformer models enable highly efficient parallelization of the training process. This has allowed creating Transformer models with parameter counts in the hundreds of millions to hundreds of billions [2, 3]. While such models can benefit from very large amounts of training data, it is often prohibitively expensive to gather such data for highly specialized tasks. Instead, Transformer models are typically trained on large amounts of unlabeled text or code corpora, using tasks such as predicting the next token. Given sufficient amounts of generic training data, it turns out that such models can solve a variety of tasks, described simply as part of the input text. In particular, it has been shown that such models are often able to write code in genera-purpose programming languages, based on a natural language description of the task to solve [7].

GPT-DB is a framework on top of GPT models, similar to GitHub's Copilot [4]. GPT-DB differs by its focus on SQL query processing, motivating specialized components for query parsing, query optimization, and query verification. Whereas GitHub's Copilot offers generic code completion functionality, it does not decompose queries into processing steps nor does it generate prompts describing database schemata. More broadly, GPT-DB relates to prior work exploiting language models in the context of data management [12]. Language models have been used in the context of natural language query interfaces [6, 15], as well as for tasks such as data cleaning and data preparation [9], data integration [8], fact checking [5], or database tuning [11]. GPT-DB differs by its focus on code generation for SQL processing.

3 SYSTEM OVERVIEW

Figure 1 shows an overview of the GPT-DB system. GPT-DB generates query-specific code for processing SQL queries, using any version of OpenAI's GPT model series [1]. This code is specified in general-purpose programming languages such as Python (i.e., it does not rely on an SQL backend). It executes directly on files containing table data and writes the query result to disk.

Along with queries, users may provide natural language instructions customizing generated code. These instructions may, for instance, express preferences with regards to libraries used for processing. As another example, they can describe customized logging output to generate during query execution. By submitting the same query with different instructions for customization, GPT-DB can generate a large number of code variants quickly.

GPT-DB parses incoming queries into a tree representation. Next, it uses a query planner to generate a query processing plan. Traditional query planers represent query plans as trees. However, GPT-DB generates code using any of the GPT model versions, in particular GPT-4. This model accepts text as input. Therefore, GPT-DB represents a query plan as a short text, summarizing processing steps. Processing steps are described at a high level of abstraction (i.e., the text plan corresponds to a logical, rather than a physical query plan). This opens up the possibility to customize operator implementations using user instructions, as described in more detail later. To generate plan text, the query planner instantiates simple, operator-specific text templates.

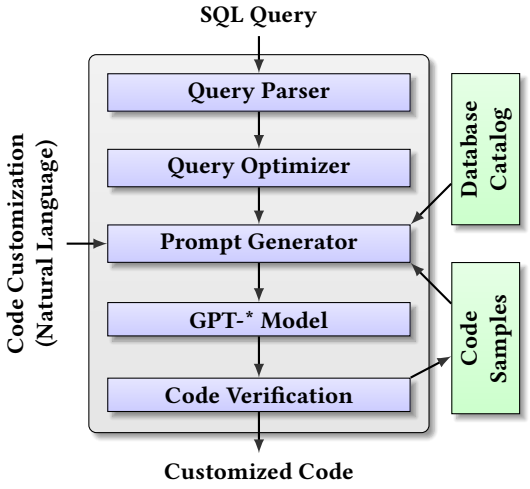

Figure 1: Overview of GPT-DB.

The Prompt Generator component combines the plan text with other information into a short text document, a "prompt", instructing GPT how to generate code. Beyond the plan, the prompt contains a description of the database schema (i.e., table and column names) and paths to the associated files. Finally, the prompt integrates (natural language) instructions for code customization, provided as input by the user. GPT-DB uses GPT to generate code, using the prompt as input.

GPT-DB offers options to verify generated code automatically. For instance, it can verify whether the generated code produces the correct query result upon execution, using a traditional SQL engine as a reference. GPT-DB can be configured to re-generate code (with some degree of randomization) until the generated code produces accurate query results. Beyond automated verification, users can confirm that generated code follows their instructions via manual inspection. If the code is satisfactory, users can choose to add it to a library of code samples. By integrating such samples into prompts submitted to GPT, GPT-DB increases the chances to generate valid code significantly.

Example 3.1. Figure 2 shows a prompt for the query "SELECT year FROM concert GROUP BY year ORDER BY count(*) DESC LIMIT 1". Triple quotes in the first and in the last line hint at the programming language to use (Python). This is important as GPT generates code in various languages. The upper part of the prompt describes the database schema and file locations. The lower part describes processing steps. Steps two to seven are generated automatically, based on the input query. Step one contains instructions provided by the user, instructing GPT to generate code using a

[1]Compared to the prior paper, the name has been changed from CodexDB to GPT-DB to indicate that visitors can choose between multiple GPT model versions.

```
"""
Table stadium with columns 'stadium_id','location','name','capacity','highest','lowest','average', stored in 'stadium.csv'.
Table singer with columns 'singer_id','name','country','song_name','song_release_year','age','is_male', stored in 'singer.csv'.
Table concert with columns 'concert_id','concert_name','theme','stadium_id','year', stored in 'concert.csv'.
Table singer_in_concert with columns 'concert_id','singer_id', stored in 'singer_in_concert.csv'.
Processing steps:
1. Use pandas library.
2. Load data for table 'concert'.
3. Group rows from results of Step 2 using 'year'.
4. Order rows from results of Step 3 using number of rows (descending).
5. Keep only 1 rows from results of Step 4.
6. Create table with columns 'year' from results of Step 5.
7. Write results of Step 6 to file 'result.csv' (with header).
"""
```

Figure 2: Prompt for code generation integrating language hints (green, Lines 1 and 14), a description of the database (blue, Lines 2 to 5), processing steps (black, Lines 8 to 13), and natural language instructions for code customization (red, Line 7).

```
import pandas as pd

df_concert = pd.read_csv('/tmp/concert.csv')
df_concert_grouped = df_concert.groupby('year')
df_concert_grouped = df_concert_grouped.size().\
    sort_values(ascending=False)
df_concert_grouped = df_concert_grouped.head(1)
df_year = df_concert_grouped.\
    reset_index()[['year']]
df_year.to_csv(
    'result.csv', index=False, header=True)
```

Figure 3: Code generated by GPT for query "SELECT year FROM concert GROUP BY year ORDER BY count(*) DESC LIMIT 1" with instructions "use pandas library".

specific library (pandas). Figure 3 shows corresponding code, generated by GPT (newlines and comments are omitted due to space restrictions). In contrast, Figure 4 shows code generated for the same query without any customization instructions.

4 DEMONSTRATION PLAN

The demonstration setup encompasses a table with laptop, connected to a portable projector. Visitors can use GPT-DB via the laptop which must be connected to the Internet to access OpenAI's GPT service. The requirements on the quality of the Internet connection are very modest (sending one prompt and receiving one short piece of code for each query). If the conference Internet fails, a mobile hotspot (established via cell phone) would suffice.

Each visitor receives a short introduction, explaining the system and the interface. Then, visitors may proceed to use the GPT-DB system themselves. Figure 5 shows the user interface (with annotations). Users must first select a relational database to query (Area A in Figure 5). For the demonstration, all databases of the SPIDER benchmark [14] are pre-loaded and can be queried by GPT-DB. Next, users customize code generated for query processing via natural language instructions (Area B). For instance, in Figure 5, GPT-DB is instructed to use the pandas library for query processing and to

```
import csv
from collections import Counter

with open('/tmp/concert.csv', 'r') as f:
    reader = csv.DictReader(f)
    rows = list(reader)

grouped_rows = {}
for row in rows:
    year = row['year']
    if year in grouped_rows:
        grouped_rows[year].append(row)
    else:
        grouped_rows[year] = [row]

ordered_rows = sorted(grouped_rows.items(),
    key=lambda x: len(x[1]), reverse=True)
kept_rows = ordered_rows[:1]
table = [{'year': row[0]} for row in kept_rows]

with open('result.csv', 'w') as f:
    writer = csv.DictWriter(
        f, fieldnames=['year'])
    writer.writeheader()
    writer.writerows(table)
```

Figure 4: Code generated by GPT for query "SELECT year FROM concert GROUP BY year ORDER BY count(*) DESC LIMIT 1" without customization.

generate progress updates after each processing step (see natural language instructions in Area B).

Then, users enter SQL queries (Area C), select the number of code generation retries (Area D), and start code generation and query processing (Area E). While processing, the interface generates various types of output (Area F shows a small part). These outputs include, for instance, the prompt generated by GPT-DB as well as the completion returned by GPT, statistics on generation time, as well as the results of code verification steps. The latter consist of the results generated when executing generated code on a data sample, as well as a comparison of this result to the result generated by a

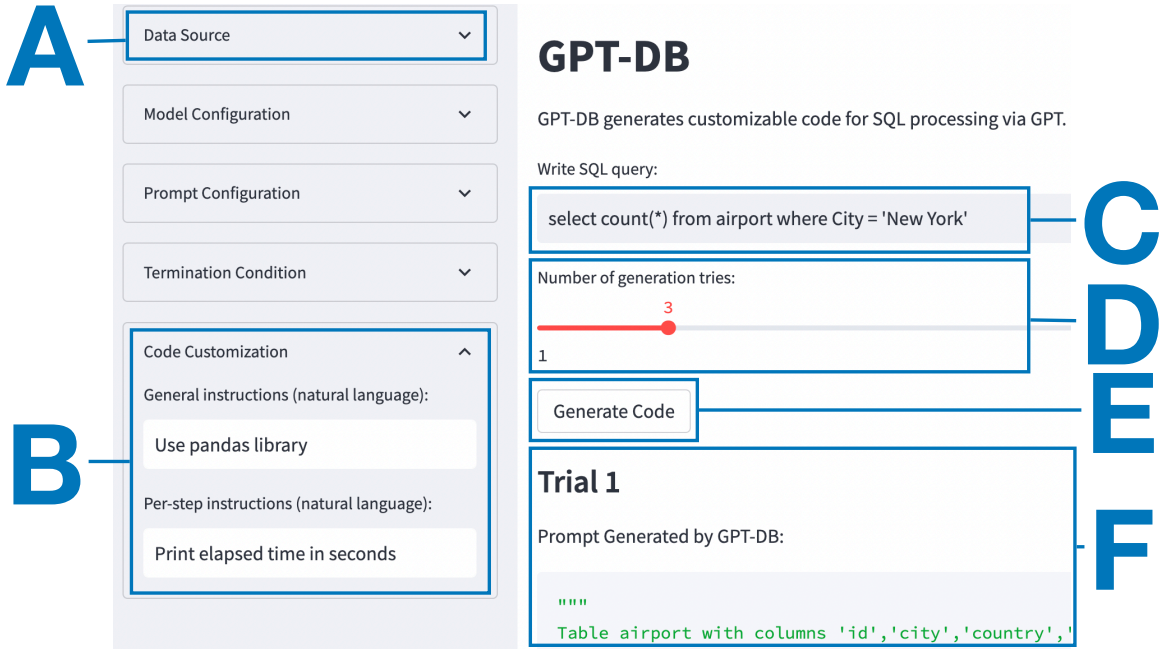

Figure 5: User interface of GPT-DB: users select data sources (A) and customization instructions (B), then enter an SQL query (C), choose the number of retries (D), and start processing (E), resulting in output (F) for each try.

reference engine. Ultimately, if code generation succeeds, the code as well as the query result (obtained by executing the code on the full data) are returned. Note that code generation only takes a few seconds, making the system suitable for a live demonstration.

Beyond the aforementioned options (data source, natural language instructions, and query), users can configure the code generation process in various ways (bar on the left in Figure 5). For instance, users can choose between several prompt types. While the primary prompt variant used by GPT-DB features an auto-generated query plan, described in natural language, alternative variants use the SQL query alone or vary the description of the database schema. Visitors may try different prompt versions for a fixed query and database, thereby gaining insights into the impact of prompt structure on success ratio. Also, users can choose between different variants of the GPT model. Whereas GPT-DB uses the GPT-4 Davinci model by default (the largest and most powerful model), visitors may try out smaller variants. This sheds light on tradeoffs between generation time and quality of the generated code. Finally, visitors can change the verification mechanism used as a stopping criterion. Verification options range from code execution without errors to result consistency, compared to a reference SQL engine.

REFERENCES

[1] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, Harsha Nori, Hamid Palangi, Marco Tulio Ribeiro, and Yi Zhang. 2023. *Sparks of Artificial General Intelligence: Experiments with an early version of GPT-4.* Technical Report. arXiv:arXiv:2303.12712v3

[2] Jacob Devlin, Ming Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *NAACL*. 4171–4186. arXiv:1810.04805

[3] Luciano Floridi and Massimo Chiriatti. 2020. GPT-3: Its Nature, Scope, Limits, and Consequences. *Minds and Machines* 30, 4 (2020), 681–694. https://doi.org/10.1007/s11023-020-09548-1

[4] Nat Friedman. 2021. Introducing GitHub Copilot: your AI pair programmer. *https://github.blog/2021-06-29-introducing-github-copilot-ai-pair-programmer/* (2021).

[5] Georgios Karagiannis, Mohammed Saeed, Paolo Papotti, and Immanuel Trummer. 2020. Scrutinizer: A mixed-initiative approach to large-scale, data-driven claim verification. *VLDB* 13, 12 (2020), 2508–2521.

[6] Fei Li and HV Jagadish. 2014. NaLIR: an interactive natural language interface for querying relational databases. In *SIGMOD*. 709–712.

[7] OpenAI. 2021. https://openai.com/blog/openai-codex/.

[8] Sahaana Suri, Ihab Ilyas, Christopher Re, and Theodoros Rekatsinas. 2021. Ember: No-Code Context Enrichment via similarity-based keyless joins. *PVLDB* 15, 3 (2021), 699–712. arXiv:arXiv:2106.01501v1

[9] Nan Tang, Ju Fan, Fangyi Li, Jianhong Tu, Xiaoyong Du, Guoliang Li, Sam Madden, and Mourad Ouzzani. 2021. Rpt: Relational pre-trained transformer is almost all you need towards democratizing data preparation. *PVLDB* 14, 8 (2021), 1254–1261. https://doi.org/10.14778/3457390.3457391 arXiv:2012.02469

[10] Immanuel Trummer. 2022. CodexDB: Synthesizing code for query processing from natural language instructions using GPT-3 Codex. *PVLDB* 15, 11 (2022), 2921 – 2928.

[11] Immanuel Trummer. 2022. DB-BERT: a database tuning tool that "reads the manual". In *SIGMOD*. 190–203.

[12] Immanuel Trummer. 2022. From BERT to GPT-3 Codex: Harnessing the Potential of Very Large Language Models for Data Management. *PVLDB* 15, 12 (2022), 3770 – 3773.

[13] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*. 5999–6009. arXiv:1706.03762

[14] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir R. Radev. 2020. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, EMNLP 2018*. 3911–3921. https://doi.org/10.18653/v1/d18-1425 arXiv:1809.08887

[15] Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning. *CoRR* abs/1709.0, 1 (2017), 1–12. arXiv:1709.00103 http://arxiv.org/abs/1709.00103