CodexDB: Synthesizing Code for Query Processing from Natural Language Instructions using GPT-3 Codex

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

ABSTRACT

CodexDB enables users to customize SQL query processing via natural language instructions. CodexDB is based on OpenAI's GPT-3 Codex model which translates text into code. It is a framework on top of GPT-3 Codex that decomposes complex SQL queries into a series of simple processing steps, described in natural language. Processing steps are enriched with user-provided instructions and descriptions of database properties. Codex translates the resulting text into query processing code. An early prototype of CodexDB is able to generate correct code for up to 81% of queries for the WikiSQL benchmark and for up to 62% on the SPIDER benchmark.

PVLDB Reference Format:
Immanuel Trummer. CodexDB: Synthesizing Code for Query Processing from Natural Language Instructions using GPT-3 Codex. PVLDB, 15(11): 2921 - 2928, 2022.
doi:10.14778/3551793.3551841

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

1 INTRODUCTION

CodexDB processes SQL queries while allowing far-ranging customization without expert developer skills. Users specify natural language instructions, along with their queries, which influences code generated for query processing. The enabling technology for this system is OpenAI's GPT-3 Codex model. Codex is a large neural network, currently available via a private beta test, that translates natural language instructions into code. This paper presents first experimental results and an outlook on future steps.

The range of applications is vast. To name just a few, consider the following use cases.

Example 1.1. A developer wants to benchmark different data processing frameworks (e.g., Pandas and Vaex in Python or Tablesaw and Morpheus in Java) on a specific SQL workload and hardware platform. Traditionally, doing so requires either modifying an existing database management system or writing query-specific code from scratch. With CodexDB, that developer specifies queries, together with natural language instruction such as "Use pandas library". While CodexDB may not succeed at generating code for each workload query, obtaining performance results for a subset can guide future development efforts. Also, generated code can be manually validated and reused in case of recurrent queries.

Example 1.2. A user needs help "debugging" a complex SQL query. To that purpose, the user wants to print intermediate results during query processing. CodexDB allows users formulating natural language instructions that are executed after each processing step. Instructing CodexDB to "Print intermediate results" has the desired effect and helps with query debugging.

CodexDB accepts queries, together with natural language instructions, as input. These instructions customize the way in which queries are executed. CodexDB generates code to process queries while complying with additional instructions. A first option is to submit queries and instructions directly to GPT-3 for code generation. We will see in Section 4 that this approach does not work.

Instead, CodexDB adapts techniques from classical query planning. It decomposes complex SQL queries into sequences of simple processing steps. In contrast to prior work, those steps are formulated in natural language using corresponding text templates. Finally, automatically generated plan steps are interleaved with user-provided instructions. The resulting text is enriched with information about the database schema and physical layout. The final text is submitted to GPT-3 Codex (as a so-called "prompt"). Using this approach as a starting point, CodexDB generates code for sample queries in a training step. The resulting code samples can be integrated into prompts generated at run time to increase the chances of success. An early prototype of CodexDB generates correct code in many cases for two popular text-to-SQL benchmarks. Also, it is able to customize generated code using simple instructions, inspired by the use cases outlined before.

The original scientific contributions are the following:

- The paper presents the vision behind CodexDB, a system that processes SQL queries while allowing customization via natural language instructions.
- The paper discusses first experimental results, based on an early prototype of CodexDB.
- The paper outlines next steps and future research.

The remainder of this paper is organized as follows. Section 2 discusses recent progress in natural language processing and compares CodexDB to prior work. Section 3 describes the architecture of the first prototype. Section 4 reports first experimental results in multiple scenarios. Section 5 discusses future research plans.

2 BACKGROUND AND RELATED WORK

CodexDB is enabled by recent advances in the domain of natural language processing. Those advances have been fuelled by two key

Proceedings of the VLDB Endowment, Vol. 15, No. 11 ISSN 2150-8097.
doi:10.14778/3551793.3551841

ideas: a novel neural network architecture, the Transformer [33], and new training paradigms, implementing the idea of transfer learning [23]. The Transformer is nowadays the dominant architecture in the domain of language processing [36]. Among other advantages, it lends itself better to parallelization than prior methods. This has, in part, enabled the creation of very large, pre-trained language models. Such models are pre-trained on tasks for which large amounts of training data are easily available, e.g. predicting the next word in text snippets. While pre-training is very expensive, the resulting models can be easily specialized for new tasks via different methods. *Fine-tuning* describes a process in which pre-trained models are used as a starting point for further training on more specialized tasks (reducing the amount of training samples and computational overheads by orders of magnitude via pre-training [8]). Until recently, fine-tuning has been the primary method of exploiting pre-trained language models. The latest generation of pre-trained models, most notably OpenAI's Generative Pre-Trained Transformer (GPT) version 3, unlocks new possibilities. It turns out that sufficiently large models can oftentimes solve new tasks without specialized training ("zero-shot learning"), based on inputs describing the task in natural language alone [3]. Precision increases if the input integrates few (i.e., typically less than ten) examples pairing tasks of the same type with solutions ("few-shot learning"). This is the method currently used by CodexDB. The final development that enabled this paper is the emergence of the Codex variant of GPT-3 [5, 22]. The primary difference between GPT-3 Codex and the original GPT-3 model lies in the data used for pre-training. GPT-3 Codex is trained on code and technical documentation. This results in a model whose primary use case is the translation of natural language commands into code.

CodexDB relates to interfaces such as GitHub's Copilot [6] as both provide layers on top of GPT-3 Codex. CodexDB differs by its focus on SQL query processing. Copilot uses partial code as input and suggests code completions. CodexDB shields users from writing code themselves. Instead, it enables users to submit queries with additional natural language instructions. The current prototype supports users by automatically handling training, prompt generation, code generation, and query evaluation. It generates complex prompts that integrate database schema information as well as a decomposition of queries into simple plan steps. Future versions of CodexDB will provide more features to specifically support SQL query processing, as outlined in Section 5 in detail.

CodexDB connects to prior work on natural language interfaces in the database community [15, 24, 35]. So far, the focus was on "democratizing access to data", i.e. enabling lay users to work with database systems. CodexDB enables lay users to customize code for query processing via natural language instructions (as well as making such changes easier for more advanced users).

CodexDB relates to prior work exploiting machine learning [12, 13, 21] and specifically Transformers [30, 31] in the context of database systems. It connects broadly to prior work using GPT-3 for program synthesis [11, 17, 18]. It differs by its focus on customizable SQL query processing. Prior work on code generation for query processing [14, 34] cannot integrate natural language instructions.

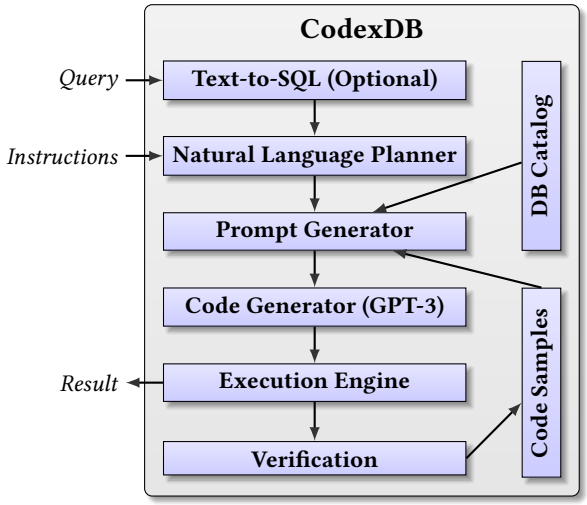

Figure 1: Overview of CodexDB prototype.

3 SYSTEM OVERVIEW

Figure 1 shows an overview of CodexDB. Users enter a query as well as natural language instructions, influencing the code generated for query processing. The query is formulated either in SQL or in natural language. In the latter case, the query is first translated into an SQL query via text-to-SQL methods [16, 27, 39].

The SQL query and natural language instructions form the input to the query planner. This planner differs from prior query planners by its output format. As the plan is translated into code by GPT-3 in the following steps, the plan is formulated as a sequence of natural language steps. User-provided natural language instructions are included as steps in such plans.

More precisely, the planner treats the nodes in the query tree in post-order. Each node is translated into a processing step, formulated in natural language. To do so, the planner uses text templates that are associated with specific node types. As plan steps are numbered, intermediate results are referred to via the number of the step generating them. The last step in the plan instructs GPT-3 to write the query result into a file at a specified location.

Currently, CodexDB allows users to specify two types of instructions: instructions that refer to the plan execution as a whole (e.g., instructions on which libraries to use for processing) as well as instructions that are executed after each step (e.g., instructions determining customized logging output). Instructions of the former type are pre-pended to the template-based processing steps (i.e., they become the first plan step) while instructions of the latter type are inserted after each processing step (i.e., the number of plan steps doubles). Note that, currently, the planner does not perform any cost-based or heuristic optimization.

Code generation is initiated by submitting a prompt to GPT-3 for completion. This prompt represents the start of a program that GPT-3 Codex tries to finish. The prompt integrates details about the data in the database, extracted from the database catalog, including the names of tables and their columns, as well as a path to the corresponding files. This description is generated using a simple text template with placeholders for column and table names. Also,

```
"""
Table Data with columns 'Player','No_','Nationality','Position',
'Years_in_Toronto','School_Club_Team', stored in 'Data.csv'.
Processing steps:
1. Load data for table Data.
2. Print progress updates.
3. Check if 'Player' equals 'dell curry'.
4. Print progress updates.
5. Filter results of Step 1 using results of Step 3.
6. Print progress updates.
7. Create table with columns 'Years_in_Toronto'
   (aka. result ) from results of Step 5.
8. Print progress updates.
9. Write results of Step 7 to file 'result.csv' (with header).
10. Print progress updates.
"""
```

Figure 2: Example prompt for code generation integrating a description of the database (blue), processing steps (black), and natural language instructions (red).

the prompt integrates the aforementioned plan steps. The prompt is passed on to GPT-3 which answers with a piece of code. CodexDB tries to execute the code, and to read the generated query result. If the code does not execute (or if it does not generate a result file), CodexDB executes up to a configurable number of retries. With each retry, the "temperature" (a Codex parameter determining the degree of randomness in code generation) is increased to enable new solutions. If successful, the result is returned to the user.

CodexDB can be used in a zero-shot setting (i.e., it generates code with instructions not seen before). Alternatively, it executes a training phase before run time with fixed instructions. The purpose of training is to generate a library of code samples, generated using the target instructions. During training, CodexDB uses sample queries for which the query result is known. It retries code generation until the query result matches the known one (or until it reaches the maximal number of retries). At run time, a specified number of samples is randomly selected from that library and included into the prompt. Having examples in the prompt ("few-shot learning") increases the success probability, as shown in Section 4.

Example 3.1. Consider the query `Select "Years_in_Toronto" as Result from Data where "Player" = 'dell curry'` from the WikiSQL benchmark. Assume a user enters this query, together with the per-step instructions "Print progress updates". Figure 2 shows the prompt for this query, interleaving automatically generated processing steps with user instructions and providing context on the database schema.

4 EXPERIMENTS

The goal of the experiments is threefold. First, to verify that CodexDB generates correct code in many cases. Second, to evaluate the degree to which code can be customized via natural language instructions. Third, to compare CodexDB to other baselines. Section 4.1 discusses the experimental setup while Section 4.2 reports results.

4.1 Setup

All experiments are executed on an AWS EC2 instance of type t2.xlarge with 16 GB of RAM, four virtual CPUs, and 800 GB of EBS storage. The instance uses Amazon's Deep Learning AMI (Version 53) and runs Ubuntu 18.04. CodexDB is implemented in Python 3 and accesses OpenAI's GPT-3 Codex model via OpenAI's Python API. The experiments use the "Cushman" and "Davinci" versions of Codex with an estimated parameter count of 6.7 billion and 175 billion parameters respectively [3, 28]. The generated code is in Python, the language both models are most capable in [22].

The following experiments compare CodexDB to baselines that try translating queries directly to code. This is the most direct method of using GPT-3, making the comparison interesting. The experiments use the WikiSQL [39] and SPIDER [38] benchmarks, two popular benchmarks for text-to-SQL translation. Text-to-SQL translation relates to the problem solved by CodexDB, as it involves natural language commands and generation of (SQL) code. The experiments only consider up to the first hundred queries from both benchmarks as treating all queries is prohibitively expensive[1]. The data on which queries operate is stored in the .csv format.

The experiments evaluating CodexDB focus on the key step of translating an SQL query into code, possibly with additional natural language instructions. Translating natural language questions into SQL queries is a well studied problem. Corresponding results for the WikiSQL and SPIDER benchmarks are available with recent methods achieving a precision of over 90 % [37] for WikiSQL and over 70 % for SPIDER [19]. We consider a test case (characterized by a natural language query with associated data) as "solved" if the generated program is executable and generates the correct result. This proxy for correctness is often used to evaluate natural language query interfaces [25, 39]. A subset of generated programs was manually validated as well. Unless noted otherwise, CodexDB retries generating a program once if the first generated program is not executable. If the first program executes but generates an incorrect result, the corresponding test case is not solved. CodexDB uses a temperature of zero for the first try and increases the temperature (determining the degree of randomization during code generation) by an amount determined by the formula $0.5/N$ where N is the maximal number of allowed tries (typically two).

To test customization, we consider six natural language instructions. Three of them focus on processing methods by instructing CodexDB to use specific libraries: "Use pandas library", "Use vaex library", and "Use datatable library". The other three instruct CodexDB to generate specific logging output after each processing step: "Print 'Done.' ", "Print intermediate results", and "Print progress updates". The first three instructions are added once as first plan step. The last three are added after each step of the initial plan. Note that the following figures and tables abbreviate those instructions slightly (e.g., in the figure legends).

4.2 Results

The performance varies across different prompts and models. In a zero-shot scenario (i.e., the prompt does not contain training samples) and using prompts generated by CodexDB, GPT-3 Codex

[1] At the time of writing, OpenAI Codex is only available to beta testers and access is subject to a rate limit of 20 requests per minute.

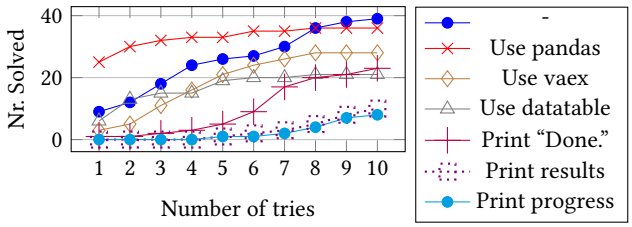

Figure 3: WikiSQL queries out of 50 solved during training for different instructions as function of the number of tries.

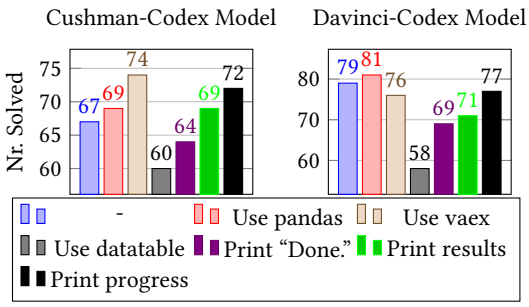

Figure 4: Number of WikiSQL queries solved out of 100 for different natural language instructions.

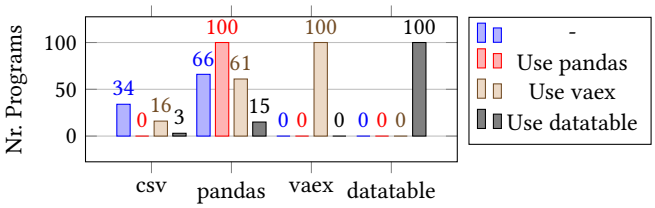

Figure 5: Number of WikiSQL programs out of 100 importing specific libraries for library-related instructions.

solves 22 out of 100 WikiSQL queries while GPT-3 Cushman only solves 11. No queries are solved when using prompts that contain queries alone. Clearly, the prompts of CodexDB, enriched by query plans, are necessary to generate correct code. The Davinci model (which features most parameters) solves significantly more test cases than the Cushman version. On the other side, average generation times (seven seconds versus two seconds) are higher for Davinci.

Language models are often fine-tuned to increase performance for specific tasks. This option is not yet available for the Codex series of GPT-3. Instead, we consider few-shot scenarios [3] in the following. Here, examples with solutions are integrated as part of the prompt.

Figure 3 reports the results of a preparation run, using 50 queries from the WikiSQL training set and the Davinci model. As training is executed before run time, up to ten tries are allowed. Furthermore, it is assumed that solutions for training samples are available, allowing to stop code generation only if the execution result is correct (as opposed to using the first executable code). Figure 3 reports solved test cases as a function of the (maximal) number of tries. Different lines are associated with additional natural language instructions ("-" designates no additional instructions). Training took between 1510 seconds (when instructed to use the pandas library) and 8,300 seconds (with instructions "print intermediate results"). Given enough tries and results to compare to, CodexDB solves 80% of test cases without additional instructions.

After training, successfully solved queries can be integrated as examples into the prompt. Using two examples per prompt (rather than zero or four) and applying the Davinci model (rather than the Cushman version) maximizes the number of queries solved out of 100 test queries from WikiSQL without additional instructions. Unless noted otherwise, this configuration is used next.

We test customization by adding the instructions described in Section 4.1. Figure 4 reports the number of test cases solved with different instructions. In most cases, adding more instructions tends to decrease success ratio for the largest model. Interestingly, the impact varies across instructions. In particular, asking CodexDB to use the pandas library slightly *increases* performance. This seems reasonable as the pandas library is popular (i.e., the training set of GPT-3 Codex likely includes various example codes) and supports operations similar to SQL operators. Manual analysis of the first 20 programs generating the correct result shows that they are indeed correct. Generated programs have a median length between 437 (with instructions "Use datatable library") and 875 (with instructions "Print intermediate results") characters. The corresponding SQL queries have a median length of only 77 characters, illustrating the difficulty of the code generation task.

So far, we discussed correctness. Next, we examine whether additional instructions are reflected in the generated programs. Figure 5 reports the number of generated programs (out of 100) that import certain libraries. Without specific instructions, 34% of generated programs import the "csv" library while 66% import pandas. Incorporating instructions to use pandas, vaex, or datatable into the prompt ensures that each generated programs imports the associated library. In some cases, in particular for vaex, programs import multiple libraries (both, csv and pandas). Manual inspection of the generated code reveals that some of those programs contain redundancy (e.g., by importing data using vaex, then transforming into pandas data frames). While this subset of programs formally satisfies the instructions (they import, i.e. "use", the corresponding library), they do not entirely reflect its spirit.

The data sets of the WikiSQL benchmark are too small for meaningful execution performance measurements. For measuring execution performance, data were scaled by factor 1,000,000 via row duplication. Considering the first ten WikiSQL queries for which correct programs were generated with all instructions, CodexDB has a total execution time between 51 seconds (with instructions "Use datatable library") and 240 seconds (with instructions "Use vaex library"). A widely used traditional database management system required 368 seconds for the same queries (counting time for loading data from disk and writing out the query result to ensure a fair comparison). This shows that CodexDB is reasonably efficient.

Figure 6 refers to logging-related instructions. The figure shows how many out of 100 generated programs contain certain types of

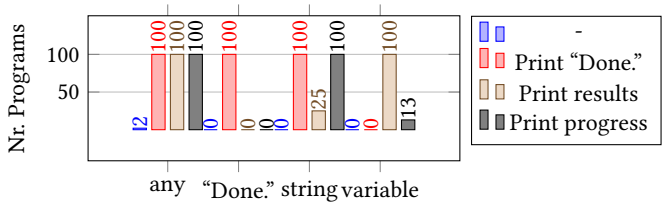

Figure 6: Number of WikiSQL programs containing specific types of print statements for output-related instructions.

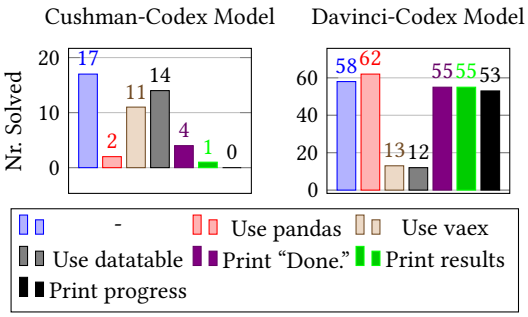

Figure 7: Number of SPIDER queries solved out of 100 for different natural language instructions.

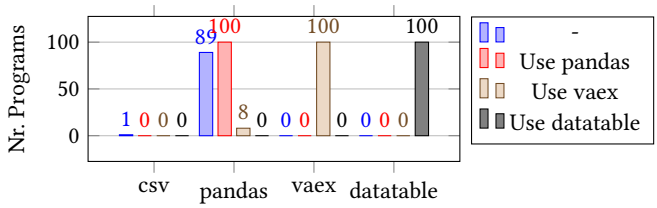

Figure 8: Number of SPIDER programs out of 100 importing specific libraries for library-related instructions.

print commands, distinguished by the operand. The figure considers presence of any print commands, commands printing out "Done.", commands printing hard-coded strings, and commands printing out variables. Without further instructions, only 2% of generated programs contain any print statements. This ratio increases to 100% for any of the logging-related instructions. Instructing CodexDB to print "Done." after each step is reflected by the presence of corresponding print commands in each program. Instructing CodexDB to print intermediate results ensures that each generated program prints out variables. Requiring progress updates leads to programs printing out hard-coded strings in all (100%) and printing out variables in some (13%) cases. Note that this instruction leaves room for interpretation (as the form of progress updates is not specified). Manual inspection reveals that most generated code includes print commands after each step, outlining the action performed at a high level of abstraction.

Figures 7, 8, and 9 show success ratio and impact of additional instructions on the first 100 queries of the SPIDER benchmark. Samples were generated during a training phase, using the same

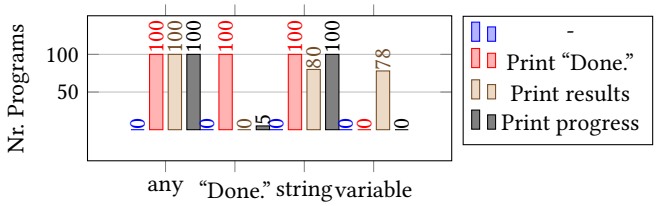

Figure 9: Number of SPIDER programs containing specific types of print statements for output-related instructions.

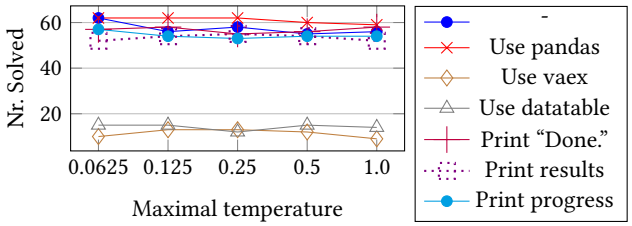

Figure 10: Impact of temperature on success ratio for SPIDER.

training process as for WikiSQL on the training partition of SPIDER. Between 20% (with instructions "Use vaex library") and 68% (with instructions "Print progress updates") of training queries were solved. All figures report few-shot results with two samples per prompt for Cushman and four samples for Davinci (this maximizes the number of test queries solved without added instructions). Results reported in figures refer to the dev partition of the SPIDER benchmark (which is frequently used for evaluation and publicly available). The settings for retries and temperature are the same as before. Text-to-SQL methods typically perform significantly worse on SPIDER, compared to WikiSQL [19]. SPIDER features multi-table queries and separates databases used for training and testing. As shown in Figure 7, CodexDB solves up to 62% of queries (compared to up to 81% for WikiSQL). The gap between Davinci and Cushman is larger on SPIDER, showing that complex benchmarks require larger models. Generated programs for SPIDER queries range from a median length of 578 characters (with instructions "Use vaex library") to a median length of 1395 characters (for instructions "Print progress updates"). Figure 5 shows that library-related instructions always lead to code importing the referenced library. Figure 9 shows similar results to Figure 6, except for a slight decrease in the ratio of programs that print intermediate results, given the corresponding instructions.

Figure 10 tests if a different temperature threshold can improve performance on SPIDER (using the Davinci model). The x-axis shows the maximal temperature (i.e., the temperature used during the retry, if any). No single setting works best for all scenarios and performance is relatively stable with regards to temperature.

5 RESEARCH AGENDA

The CodexDB project aims at creating a system that is highly customizable via natural language instructions, while achieving comparable accuracy and performance to traditional database systems used via text-to-SQL query interfaces. At the same time, the system

must prevent ethical and legal issue due to the use of large language models [1]. The current prototype does not yet achieve those goals. The following paragraphs discuss steps towards the CodexDB vision, open research questions, and associated challenges.

Improve Success Ratio. Increasing the ratio of queries for which CodexDB produces correct results is a primary research goal. The current version adds randomly selected samples to the prompt in order to solve more queries. In this context, a first research question is whether we can increase success ratio by selecting samples more carefully. Adding samples that are similar to input problems can increase the performance of GPT-3 on natural language benchmarks significantly [20]. When adapting this approach for query code generation, a first challenge is the definition of a similarity function on SQL queries that maximizes performance gains.

A second research question is whether performance can be improved further by fine-tuning GPT-3. Given a dynamic input workload, a first challenge is to choose when to fine-tune (using previous queries as training data). Fine-tuning too frequently is undesirable as the process is very expensive. On the other side, fine-tuning may increase accuracy and reduce generation cost as it decreases prompt size (due to the omission of samples) and the number of retries (after unsuccessful generation attempts). CodexDB should decide automatically if and when to fine-tune, based e.g. on workload properties and user-defined precision or cost constraints.

Reduce Code Generation Cost. Using smaller models for code generation (e.g., GPT-3 Ada or Babbage or their open-source equivalents such as GPT-Neo) decreases overheads. However, they may not produce correct code for complex queries. CodexDB should automatically select the most appropriate model for each incoming query. Doing so is challenging: simple metrics such as the character length of the input question do not necessarily correlate with query complexity. Other options include the use of a (relatively cheap) classifier to assess the required model size, and a staged approach that starts with small models, switching to larger ones if necessary.

Reduce Data Processing Cost. Achieving acceptable performance for multi-table queries on large data sets requires join ordering. Classical query optimization methods use a cost model for join ordering [26], based on properties of physical operators. CodexDB, however, generates code implementing relational operators on the fly. As no physical operators are known at optimization time, it is challenging to estimate processing cost of a given join order. Whether classical query optimization methods (e.g., cost-based optimizers with a generic cost model [7] or simpler heuristics [4]) can be adapted to this new scenario is an open question.

Generating code for query processing in lower-level languages such as C likely decreases execution costs. This leads to new challenges: code in lower-level languages is often more verbose than its Python equivalent. Transformer models generally place hard limits on the size of the code window (including code read and written). Switching to lower-level languages may therefore require multiple model invocations to generate all query code. This, however, makes it challenging to ensure that the generated code pieces are consistent and can be composed.

Improve Options for Customization. During the training phase, CodexDB verifies correctness of generated code by comparing the execution result to the reference. Currently, CodexDB cannot verify whether generated code complies with additional user instructions. Doing so requires integrating feedback from users (already since instructions may be ambiguous). How to integrate feedback from users is an open question. A challenge here is to balance the need for verification with the need to minimize overheads for users. CodexDB must decide for how many and for which samples to collect feedback and how to collect it. E.g., if instructions refer to desired logging output, users can validate output generated during execution. Otherwise, users can validate a natural language description by GPT-3 summarizing the generated code.

Currently, users can influence prompts generated by CodexDB only by inserting natural language instructions before or after plan steps. Future versions of CodexDB will offer users the possibility for prompt-tuning in order to specialize CodexDB for specific scenarios. In the simplest version, users specify templates for prompt elements (e.g. determining whether questions or queries are integrated into the prompt, as well as the precise representation of specific types of plan steps such as joins). Later version will allow users to specify a search space over prompt templates (e.g., by specifying alternative templates for specific plan step types). CodexDB will integrate approaches for automated prompt tuning [29] to select the best version within that search space for a given scenario. Doing so is challenging as the prompt search space grows exponentially in the number of choice points. Furthermore, evaluating the quality of a prompt requires generating and evaluating code for a sufficiently large number of training queries.

Prevent Ethical and Legal Issues. CodexDB leverages large language models which have been criticized for their environmental footprint, as well as ethical and legal issues that stem from the use of large amounts of uncurated training data [1, 2]. Language models for code generation have been criticized specifically for their ability to memorize training data, some of which may be subject to copyright constraints [9, 32, 40]. Using smaller models whenever possible, one of the ideas for reducing code generation costs discussed before, limits the environmental footprint. As an additional measure, CodexDB will consider expected data processing costs during model selection, ensuring that code generation overheads are not disproportional to data processing overheads.

GitHub plans to add mechanisms to detect cases where code generated by GitHub Copilot matches samples in the training data [40]. Via similar methods, CodexDB could regenerate code with a higher degree of randomization if such matches are found. Using carefully selected training data has been put forward as a more reliable solution to this problem [1]. In case of CodexDB, this training data could include code of open-source database management systems that is available under permissive licenses (or code generated by such systems for query processing). Based on such a training corpus, a Transformer model specialized for query code generation can be trained from scratch. Training medium-sized Transformer models from scratch takes hours [10]. The resulting model may however realize attractive size-accuracy tradeoffs due to specialization.

6 CONCLUSION

CodexDB enables far-ranging customization via natural language commands. Experiments with a first prototype are promising but also hint at significant potential for improvements.

REFERENCES

[1] Emily M. Bender, Timnit Gebru, Angelina McMillan-Major, and Shmargaret Shmitchell. 2021. *On the dangers of stochastic parrots: Can language models be too big?* Vol. 1. Association for Computing Machinery. 610–623 pages. https://doi.org/10.1145/3442188.3445922

[2] Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, Erik Brynjolfsson, Shyamal Buch, Dallas Card, Rodrigo Castellon, Niladri Chatterji, Annie Chen, Kathleen Creel, Jared Quincy Davis, Dora Demszky, Chris Donahue, Moussa Doumbouya, Esin Durmus, Stefano Ermon, John Etchemendy, Kawin Ethayarajh, Li Fei-Fei, Chelsea Finn, Trevor Gale, Lauren Gillespie, Karan Goel, Noah Goodman, Shelby Grossman, Neel Guha, Tatsunori Hashimoto, Peter Henderson, John Hewitt, Daniel E. Ho, Jenny Hong, Kyle Hsu, Jing Huang, Thomas Icard, Saahil Jain, Dan Jurafsky, Pratyusha Kalluri, Siddharth Karamcheti, Geoff Keeling, Fereshte Khani, Omar Khattab, Pang Wei Kohd, Mark Krass, Ranjay Krishna, Rohith Kuditipudi, Ananya Kumar, Faisal Ladhak, Mina Lee, Tony Lee, Jure Leskovec, Isabelle Levent, Xiang Lisa Li, Xuechen Li, Tengyu Ma, Ali Malik, Christopher D. Manning, Suvir Mirchandani, Eric Mitchell, Zanele Munyikwa, Suraj Nair, Avanika Narayan, Deepak Narayanan, Ben Newman, Allen Nie, Juan Carlos Niebles, Hamed Nilforoshan, Julian Nyarko, Giray Ogut, Laurel Orr, Isabel Papadimitriou, Joon Sung Park, Chris Piech, Eva Portelance, Christopher Potts, Aditi Raghunathan, Rob Reich, Hongyu Ren, Frieda Rong, Yusuf Roohani, Camilo Ruiz, Jack Ryan, Christopher Ré, Dorsa Sadigh, Shiori Sagawa, Keshav Santhanam, Andy Shih, Krishnan Srinivasan, Alex Tamkin, Rohan Taori, Armin W. Thomas, Florian Tramèr, Rose E. Wang, William Wang, Bohan Wu, Jiajun Wu, Yuhuai Wu, Sang Michael Xie, Michihiro Yasunaga, Jiaxuan You, Matei Zaharia, Michael Zhang, Tianyi Zhang, Xikun Zhang, Yuhui Zhang, Lucia Zheng, Kaitlyn Zhou, and Percy Liang. 2021. On the Opportunities and Risks of Foundation Models. (2021), 1–212. arXiv:2108.07258 http://arxiv.org/abs/2108.07258

[3] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. *Advances in Neural Information Processing Systems* 2020-Decem (2020). arXiv:2005.14165

[4] N Bruno. 2010. Polynomial heuristics for query optimization. In *ICDE*. 589–600. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5447916

[5] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *http://arxiv.org/abs/2107.03374* (2021). arXiv:2107.03374 http://arxiv.org/abs/2107.03374

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

[7] Andrey Gubichev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *PVLDB* 9, 3 (2015), 204–215.

[8] Neil Houlsby, Andrei Giurgiu, Stanisraw Jastrzębski, Bruna Morrone, Quentin de Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019. Parameter-efficient transfer learning for NLP. *36th International Conference on Machine Learning, ICML 2019* 2019-June (2019), 4944–4953. arXiv:1902.00751

[9] Gavin D Howard. 2021. GitHub Copilot: Copyright, Fair Use, Creativity, Transformativity, and Algorithms *. (2021), 1–13.

[10] Huggingface. 2022. Training a causal language model from scratch. *https://huggingface.co/course/chapter7/6?fw=pt* (2022).

[11] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2021. *Jigsaw: Large Language Models meet Program Synthesis*. Vol. 1. Association for Computing Machinery. 1–12 pages. arXiv:2112.02969 http://arxiv.org/abs/2112.02969

[12] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2018. Learned cardinalities: estimating correlated joins with deep learning. In *CIDR*. arXiv:1809.00677 http://arxiv.org/abs/1809.00677

[13] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2017. The Case for Learned Index Structures. 1 (2017), 1–30. https://doi.org/10.1145/2348283.2348367 arXiv:1712.01208

[14] Konstantinos Krikellas, Stratis D. Viglas, and Marcelo Cintra. 2010. Generating code for holistic query evaluation. In *ICDE*. IEEE, 613–624. https://doi.org/10.1109/ICDE.2010.5447892

[15] Fei Li and HV Jagadish. 2014. NaLIR: an interactive natural language interface for querying relational databases. *SIGMOD* (2014), 709–712. https://doi.org/10.1145/2588555.2594519

[16] Fei Li and HV Jagadish. 2016. Understanding natural language queries over relational databases. *SIGMOD Record* 45, 1 (2016), 6–13.

[17] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, and Rémi Leblond. 2022. Competition-Level Code Generation with AlphaCode. *DeepMind Technical Report* (2022), 1–73.

[18] Pietro Liguori, Erfan Al-Hossami, Domenico Cotroneo, Roberto Natella, Bojan Cukic, and Samira Shaikh. 2022. Can We Generate Shellcodes via Natural Language? An Empirical Study. *arXiv:2202.03755v1* (2022). arXiv:2202.03755 http://arxiv.org/abs/2202.03755

[19] Xi Victoria Lin, Richard Socher, and Caiming Xiong. 2020. Bridging textual and tabular data for cross-domain Text-to-SQL semantic parsing. *Findings of the Association for Computational Linguistics Findings of ACL: EMNLP 2020* (2020), 4870–4888. https://doi.org/10.18653/v1/2020.findings-emnlp.438 arXiv:2012.12627

[20] Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. 2021. What Makes Good In-Context Examples for GPT-3? *http://arxiv.org/abs/2101.06804* 3 (2021). arXiv:2101.06804 http://arxiv.org/abs/2101.06804

[21] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2018. Neo: A Learned query optimizer. *PVLDB* 12, 11 (2018), 1705–1718. https://doi.org/10.14778/3342263.3342644 arXiv:1904.03711

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

[23] Sebastian Ruder, Matthew E Peters, Swabha Swayamdipta, and Thomas Wolf. 2019. Transfer Learning in Natural Language Processing. In *ACL: Tutorials*. 15–18.

[24] Diptikalyan Saha, Avrilia Floratou, Karthik Sankaranarayanan, Umar Farooq Minhas, Ashish R Mittal, and Fatma Ozcan. 2016. ATHENA: An ontology-driven system for natural language querying over relational data stores. *VLDB* 9, 12 (2016), 1209–1220.

[25] Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. PICARD: Parsing Incrementally for Constrained Auto-Regressive Decoding from Language Models. (2021), 9895–9901. https://doi.org/10.18653/v1/2021.emnlp-main.779 arXiv:2109.05093

[26] PG G Selinger, MM M Astrahan, D D Chamberlin, R A Lorie, and T G Price. 1979. Access path selection in a relational database management system. In *SIGMOD*. 23–34. http://dl.acm.org/citation.cfm?id=582095.582099

[27] Jaydeep Sen, Chuan Lei, Abdul Quamar, Fatma Özcan, Vasilis Efthymiou, Ayushi Dalmia, Greg Stager, Ashish Mittal, Diptikalyan Saha, and Karthik Sankaranarayanan. 2020. ATHENA++: natural language querying for complex nested SQL queries. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2747–2759. https://doi.org/10.14778/3407790.3407858

[28] Richard Shin and Benjamin Van Durme. 2021. Evaluating the Text-to-SQL Capabilities of Large Language Models. *arXiv preprint arXiv:2112.08696* (2021).

[29] Taylor Shin, Yasaman Razeghi, Robert L. Logan, Eric Wallace, and Sameer Singh. 2020. AUTOPROMPT: Eliciting knowledge from language models with automatically generated prompts. *EMNLP 2020 - 2020 Conference on Empirical Methods in Natural Language Processing, Proceedings of the Conference* (2020), 4222–4235. https://doi.org/10.18653/v1/2020.emnlp-main.346 arXiv:2010.15980

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

[31] 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. In *Proceedings of the VLDB Endowment*, Vol. 14. 1254–1261. https://doi.org/10.14778/3457390.3457391 arXiv:2012.02469

[32] US Patent and Trademark Office. 2019. Request for Comments on Intellectual Property Protection for Artificial Intelligence Innovation. *Federal Register* 84, 210 (2019), 58141–2. https://technologyreview.us11.list-manage.com/track/click?u=47c1a9cec9749a8f8cbc83e78&id=807d031d0a&e=eb6fe22ad6

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

[34] Skye Wanderman-Milne and Nong Li. 2014. Runtime Code Generation in Cloudera Impala. *IEEE Data Engineering Bulletin* 37, 1 (2014), 31–37. http://dblp.uni-trier.de/db/journals/debu/debu37.html#Wanderman-MilneL14

[35] Nathaniel Weir, Andrew Crotty, Alex Galakatos, Amir Ilkhechi, Shekar Ramaswamy, Rohin Bhushan, Ugur Cetintemel, Prasetya Utama, Nadja Geisler, Benjamin Hättasch, Steffen Eger, and Carsten Binnig. 2019. DBPal: Weak Supervision for Learning a Natural Language Interface to Databases. (2019), 1–4. arXiv:1909.06182 http://arxiv.org/abs/1909.06182

[36] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *EMNLP*. 38–45. https://doi.org/10.18653/v1/2020.emnlp-demos.6 arXiv:arXiv:1910.03771v5

[37] Kuan Xuan, Yongbo Wang, Yongliang Wang, Zujie Wen, and Yang Dong. 2021. SeaD: End-to-end Text-to-SQL Generation with Schema-aware Denoising. (2021). arXiv:2105.07911 http://arxiv.org/abs/2105.07911

[38] 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

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

[40] Albert Ziegler. 2022. A first look at rote learning in GitHub Copilot suggestions. *https://docs.github.com/en/github/copilot/research-recitation* (2022).