

CodexDB: Generating Code for Processing SQL Queries using GPT-3 Codex

Immanuel Trummer

Cornell University

Ithaca, NY

itrummer@cornell.edu

ABSTRACT

CodexDB is an SQL processing engine whose internals can be customized 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 a majority of queries of the WikiSQL benchmark and can be customized in various ways.

Artifact Availability:

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

1 INTRODUCTION

Modifying a database management system is hard. Systems such as Postgres feature millions of code lines. Understanding and changing that code requires expert knowledge in databases on top of advanced coding skills. This prevents all but the most experienced developers from creating customized versions.

This paper presents the vision behind CodexDB, a novel database management system that can be customized 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 Tableau 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 novice database user wants to gain a deeper understanding of how database management systems work. To

that purpose, the user would like to generate customized output after each processing step (e.g., summarizing steps performed and showing a small sample of intermediate results). Integrating such changes into traditional systems is beyond the user’s capabilities. With CodexDB, the user specifies natural language queries (which, internally, are translated into SQL), together with a natural language description of desired per-step output.

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 a majority of cases for a popular text-to-SQL benchmark. Also, it is able to customize generated code using simple instructions, inspired by the use cases outlined before.

In summary, the original scientific contributions in this paper are the following:

- The paper presents the vision behind CodexDB, an analytical SQL engine that can be customized 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 next steps and concludes.

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 ideas: a novel neural network architecture, the Transformer [23],

and new training paradigms, implementing the idea of transfer learning [15]. The Transformer is nowadays the dominant architecture in the domain of language processing [26]. 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 [4]). 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 [1]. 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 [2, 14]. 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 connects to prior work on natural language interfaces in the database community [9, 16, 25]. So far, the focus was on “democratizing access to data”, i.e. enabling lay users to work with database systems. CodexDB goes one step further by “democratizing” the design of database system internals. The goal is to enable lay users to change system behavior to a degree that goes beyond the configuration scope of traditional database systems (as well as making such changes easier for more advanced users).

CodexDB relates to prior work exploiting machine learning [6, 7, 13] and specifically Transformers [20, 21] in the context of database systems. It connects broadly to prior work using GPT-3 for program synthesis [5, 11, 12]. It differs by its focus on customizable SQL query processing. Prior work on code generation for query processing [8, 24] cannot integrate natural language instructions.

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 [10, 18, 28].

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.

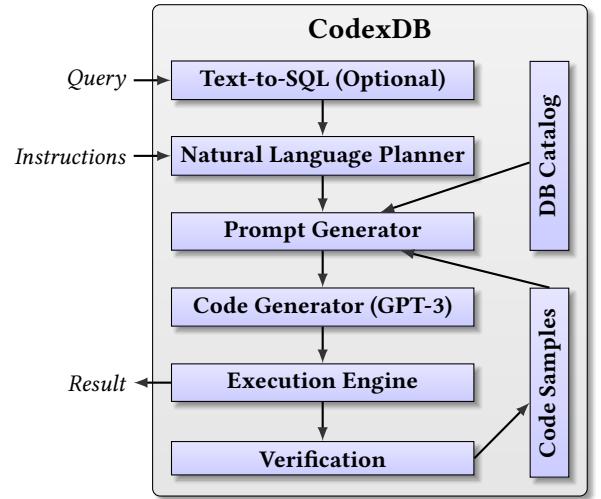


Figure 1: Overview of CodexDB prototype.

Table 1: Text templates used during planning.

Pattern	Text Template
X = Y	Check if T(X) equals T(Y)
from X where Y	Filter T(X) using T(Y)
X as Y	T(X) (aka. T(Y))
select X from Y	Create table with columns T(X) from T(Y)

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. Table 1 shows example templates ($T(X)$ and $T(Y)$ denote the text representation of expressions X and Y respectively). 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).

```
data = pd.read_csv('Data.csv')
print('Loaded Data')
is_dell_curry = data['Player'] == 'dell curry'
print('Checked if Player equals dell curry')
dell_curry_data = data[is_dell_curry]
print('Filtered data for dell curry')
result = dell_curry_data['Nationality']
print('Created table with Nationality column')
result.to_csv('result.csv', header=True)
print('Wrote to file result.csv')
```

Figure 3: Code generated by GPT-3 in response to prompt from Figure 2 (code was shortened for readability by removing empty lines and comments).

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. Figure 3 shows the generated code. It processes the query and writes the result into a file. While doing so, it prints out progress updates summarizing steps performed.

4 EXPERIMENTS

The goal of the experiments is threefold. First, to verify that CodexDB generates correct code in most 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 [1, 19]. The generated code is in Python, the language both models are most capable in [14].

The following experiments compare CodexDB to baselines that try translating natural language queries directly to code. This is the most direct method of using GPT-3, making the comparison interesting. Doing so requires a text-to-SQL benchmark that features natural language questions, along with corresponding queries. We consider a subset of the WikiSQL benchmark [28], a popular benchmark featuring over 80,000 queries with examples. The experiments only consider up to the first hundred queries as treating all queries is prohibitively expensive¹. 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 benchmark are available [28] with recent methods achieving a precision of over 90 % [27]. 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 [17, 28]. 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).

¹At 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.

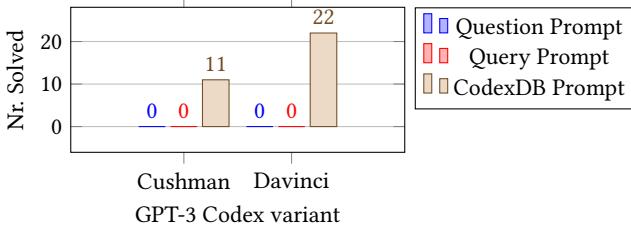


Figure 4: Number of test cases solved out of 100 without prior training (“zero-shot”) for different models and prompts.

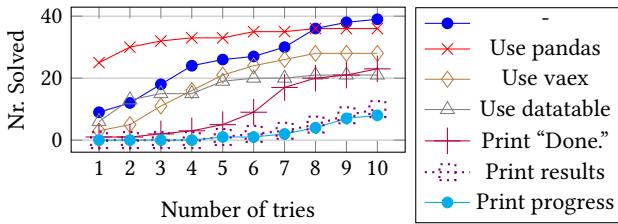


Figure 5: Number of test cases out of 50 solved during training for different instructions as function of the number of tries.

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

Figure 4 reports results of an experiment comparing different prompt generation methods (on 100 queries from the WikiSQL test set). “CodexDB Prompt” refers to prompts generated by CodexDB (integrating, in particular, natural language query plans). “Question Prompt” and “Query Prompt” integrate the same description of the data source as CodexDB (i.e., table and column names) but replace the natural language query plan by the natural language question or the correct SQL query respectively. 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.

Figure 4 reports a success rate of 22% without prior training. 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 [1] in the following. Here, examples with solutions are integrated as part of the prompt.

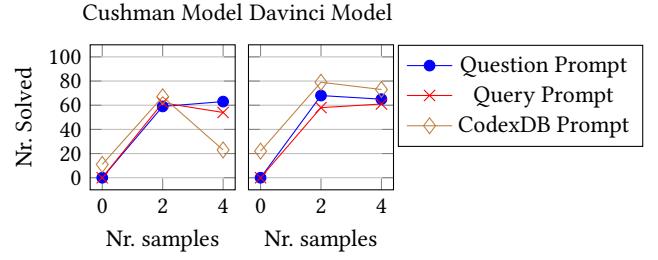


Figure 6: Number of test cases solved out of 100 as a function of the number of training samples in prompt (“few-shot”).

Figure 5 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 6 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.

Figure 6 reports number of test cases solved (out of 100 queries from the WikiSQL *test set*, i.e. no overlap with pre-generated samples) as a function of the number of samples included in the prompt. It compares the previously introduced prompt styles. Clearly, performance improves significantly (e.g., from around 20 to around 80% for Davinci) when adding samples. Adding samples decreases the gap between CodexDB’s and other prompts. Still, the CodexDB prompt performs best except for four samples and the Cushman model. The reason is the slightly longer prompts of CodexDB (featuring query plans) that exceed the maximum input size for the Cushman model for 67 test cases. Davinci supports larger inputs and does not suffer from this problem. Unless noted otherwise, the remaining experiments use two samples and the Davinci model (the configuration leading to maximal performance in Figure 6).

We test customization by adding the instructions described in Section 4.1. Figure 7 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. Table 2 reports statistics on the size of generated code (and on the size of the corresponding SQL queries), measured in characters. The average size of code generated by CodexDB is larger by up to one order of magnitude, compared to SQL queries. This illustrates the difficulty of the task. Adding instructions on logging increases code size (due to print statements after processing steps).

Table 2: Code length in characters for different languages and instructions (only considering executable programs).

Language	Instructions	Min	Median	Max
SQL	-	42	77	227
Python	-	276	545	1110
	Use pandas library	284	438.5	782
	Use vaex library	355	637	1018
	Use datatable library	307	437	848
	Print “Done.”	390	724	1388
	Print intermediate results	458	875	1734
	Print progress updates	585	836	1458

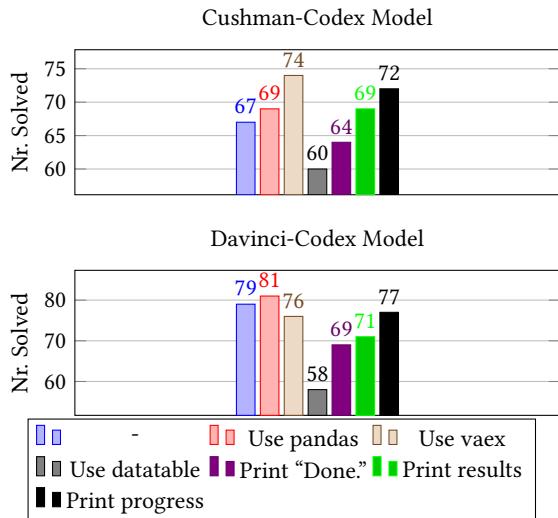


Figure 7: Number of test cases solved out of 100 for different natural language instructions.

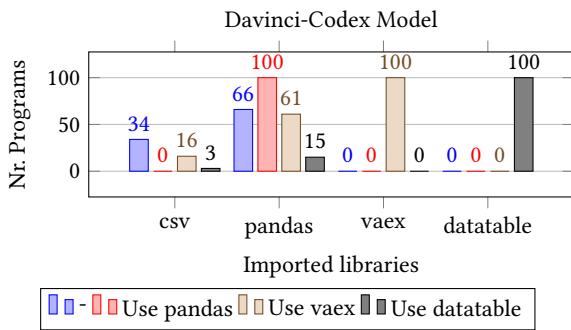


Figure 8: Number of generated programs out of 100 importing specific libraries for library-related instructions.

So far, we discussed correctness. Next, we examine whether additional instructions are reflected in the generated programs.

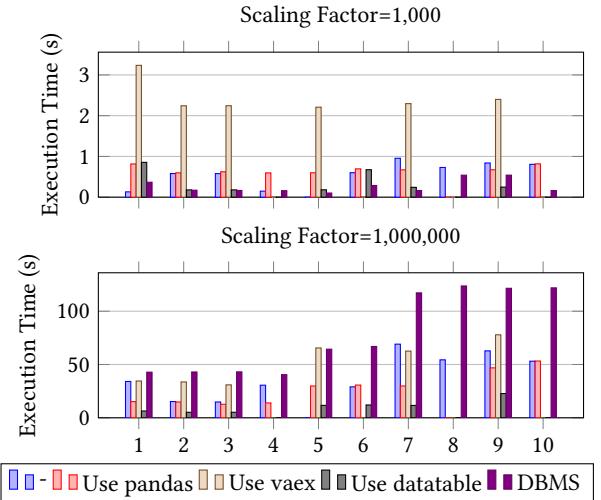


Figure 9: Execution time of programs generated by CodexDB with different instructions and of one traditional DBMS.

Table 3: Total run time for queries solved by all baselines.

Baseline	Time (s)	
	SF: 1K	SF: 1M
CodexDB: -	3	196
CodexDB: Use pandas library	3	119
CodexDB: Use vaex library	12	240
CodexDB: Use datatable library	2	51
DBMS	1	368

Figure 8 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.

Figure 9 reports execution time measurements for programs generated with different instructions for the ten first queries. Missing bars indicate that no correct program was generated for the corresponding query. The data sets of the WikiSQL benchmark are too small for meaningful performance measurements. Hence, data were scaled by factor 1,000 and by factor 1,000,000 (by simply duplicating rows). The resulting data sets have an average size of 1.2 GB and 15 million rows. Table 3 reports total execution time for all of the aforementioned queries for which correct programs were generated for all possible instructions. Clearly, instructing CodexDB to

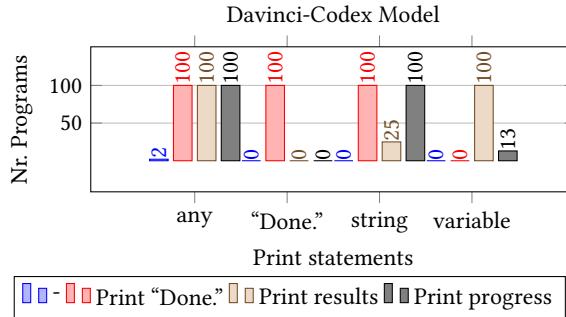


Figure 10: Number of generated programs containing specific types of print statements for output-related instructions.

use different libraries has significant impact on performance. This indicates that the generated code is fundamentally different. Finally, time measurements are provided for a traditional, widely used, database management system. To ensure a fair comparison, time measurements include time for loading data from disk, processing the query, and writing the result back to disk (the generated code implements the same tasks). While performance is not the primary goal of CodexDB, the generated code is reasonably efficient.

Figure 10 refers to logging-related instructions. The figure shows how many out of 100 generated programs contain certain types of 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. Figure 3 from Section 3 shows a corresponding example.

5 CONCLUSION AND OUTLOOK

CodexDB blurs the line between user and developer. It enables far-ranging customization via natural language commands. Experiments with a first prototype are promising but also hint at significant potential for improvements.

First, CodexDB generates correct code in most but not in all cases (up to 81% of queries are solved, depending on scenario and model). A success rate of 100% is illusory for any kind of natural language interfaces. Still, the newest generation of text-to-SQL methods achieves a precision of more than 90% on the same benchmark. Hence, increasing the precision of CodexDB will be a primary research goal in the near term.

Second, customizing code via natural language instructions works but sometimes in unexpected ways. For instance, given instructions to use specific libraries, CodexDB always imports (“uses”) them indeed. However, in a minority of cases, imported libraries are not ultimately “used” for processing query steps. This motivates stronger mechanisms allowing users to enforce a specific interpretation of their natural language input.

The natural language query planner does not yet use cost-based optimization. This is acceptable for the simple queries of the WikiSQL benchmark. To handle complex queries with many joins, future versions will integrate optimization according to cost models (e.g., based on the number of tuples processed) that have been shown to work quite well across different physical operator implementations [3]. Alternatively, machine learning can be used to optimize generated query plans for specific workloads [13, 22].

REFERENCES

- [1] 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
- [2] 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 Khaaf, 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 Heben 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> <http://arxiv.org/abs/2107.03374>
- [3] Andrey Gubichev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *PVLDB* 9, 3 (2015), 204–215.
- [4] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrz̄bski, Bruna Morrone, Quentin de Laroussilhe, Andrea Gesmundo, Mona Attaryan, and Sylvain Gelly. 2019. Parameter-efficient transfer learning for NLP. *36th International Conference on Machine Learning, ICML 2019-June* (2019), 4944–4953. arXiv:1902.00751
- [5] Naman Jain, Skanda Vaidyanathan, 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>
- [6] 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>
- [7] 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
- [8] 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>
- [9] 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>
- [10] Fei Li and HV Jagadish. 2016. Understanding natural language queries over relational databases. *SIGMOD Record* 45, 1 (2016), 6–13.
- [11] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittweis, and Rémi Leblond. 2022. Competition-Level Code Generation with AlphaCode. *DeepMind Technical Report* (2022), 1–73.
- [12] 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>

- [13] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papernmanoil, 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
- [14] OpenAI. 2021. <https://openai.com/blog/openai-codex/>.
- [15] Sebastian Ruder, Matthew E Peters, Swabha Swamyamdipta, and Thomas Wolf. 2019. Transfer Learning in Natural Language Processing. In *ACL: Tutorials*. 15–18.
- [16] 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.
- [17] 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
- [18] Jaydeep Sen, Chuan Lei, Abdul Quamar, Fatma Özcan, Vasilis Eftymiou, 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>
- [19] Richard Shin and Benjamin Van Durme. 2021. Evaluating the Text-to-SQL Capabilities of Large Language Models. *arXiv preprint arXiv:2112.08696* (2021).
- [20] 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
- [21] 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>
- [22] Immanuel Trummer, Junxiong Wang, Deepak Maram, Samuel Moseley, Saehan Jo, and Joseph Antonakakis. 2019. SkinnerDB: regret-bounded query evaluation via reinforcement learning. In *SIGMOD*. 1039–1050.
- [23] 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
- [24] Skye Wanderman-Milne and Nong Li. 2014. Runtime Code Generation in Cloud-era Impala. *IEEE Data Engineering Bulletin* 37, 1 (2014), 31–37. <http://dblp.uni-trier.de/db/journals/debu/debu37.html#Wanderman-Milne14>
- [25] Nathaniel Weir, Andrew Crotty, Alex Galakatos, Amir Ilkhechi, Shekar Ramaswamy, Rohin Bhushan, Ugur Cetintemel, Prasetya Utama, Nadja Geisler, Benjamin Härttasch, 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>
- [26] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierrette 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
- [27] 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>
- [28] 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>