Customizing Operator Implementations for SQL Processing via Large Language Models

Immanuel Trummer
Cornell University
itrummer@cornell.edu

Abstract

Recent advances in generative AI enable code synthesis at unprecedented levels of accuracy. In the context of data management, this opens up exciting opportunities to automatically customize core components of database management systems with the help of large language models such as OpenAI's GPT model series. This paper describes the vision of GenesisDB, a framework powered by large language models, that uses code synthesis to customize relational operators according to user specifications. The paper describes the first, simple prototype of GenesisDB that generates operator implementations in the Python programming language. The first experimental results demonstrate that the prototype is able to generate implementations for most relational operators, required for running standard benchmarks such as TPC-H. At the same time, the experiments reveal a multitude of research challenges that need to be solved to make this approach practical.

1 Introduction

Database management systems typically use a code base that evolves slowly over time, supported by large teams of software developers. End users may still influence system behavior via tuning knobs. However, the scope of such modifications is limited and insufficient for the examples outlined next.

Example 1: A user, trying to formulate a complex SQL query, would like to add custom SQL debugging support. For instance, this may take the form of customized output, including input/output samples, printed after each operation in the query plan.

Example 2: A developer wants to integrate SQL processing into a Python application that uses specialized in-memory data structures. To avoid conversion overheads and external dependencies, the developer would like to generate custom implementations of relational operators that are tailored to existing data structures.

Example 3: A database vendor would like to explore different types of progress updates while data is being processed. This can take the form, e.g., of a progress bar or of notifications detailing the amount of data processed at certain time intervals. The vendor would like to test various different variants in a user study before fully implementing the most popular version.

Example 4: The teacher of an introductory database class would like to create a specialized processing engine that generates educative content on used operators while processing. This output could take the form of operator descriptions, links to associated SQL tutorials or book chapters, or information on the complexity of each operator as it is processing.

Existing systems are typically unable to support all of the aforementioned specializations via parameter settings. Hence, supporting the example scenarios requires code changes of existing systems or implementing a new system from scratch. Such changes are beyond the capabilities of lay users without significant expertise in coding and databases. Even for experienced developers, some of the aforementioned changes may represent significant endeavors that consume large amounts of time.

This paper proposes the use of generative AI to customize core components of database management systems, thereby reducing or, in the best case, even eliminating customization overheads for developers and end users. As a proof of concept, it also reports on the first experimental results of a prototype system, GenesisDB, that uses generative AI to synthesize relational operator implementations.

The enabling technology of GenesisDB are large language models, based on the Transformer architecture. This architecture, together with pre-training approaches that leverage large amounts of unlabeled samples, has recently led to fundamental advances in the domain of natural and formal language processing. The newest generation of such models is often used without task-specific training, merely by specifying generation tasks as part of the input text (the so-called "prompt") while optionally providing a few solution samples. This is the approach taken by GenesisDB: it employs OpenAI's GPT models to synthesize custom code implementing relational operators.

GenesisDB comes with a set of prompt templates that describe the desired behavior of standard operators, as well as input and output formats. These prompt templates contain placeholders that can be substituted by user instructions (in natural language), thereby enabling customization. GenesisDB uses GPT to synthesize code for each operator. Then, it validates implementations by a sequence of more and more complex tests. Such tests include SQL queries, processed using newly generated operator implementations as well as a reference system (currently Postgres) to verify result consistency. Failed test cases initiate an automated debugging approach. Here, GenesisDB analyzes dependencies between operators and test cases as well as a probabilistic model to identify likely faulty operators. Next, GenesisDB tries to re-synthesize code for those operators, varying the prompt structure as well as generation settings to obtain a variety of alternative implementations to test. Specifically, GenesisDB may use code generated previously for other operators which passes a large number of test cases (and is therefore likely to be correct) as samples directly in prompts. This increases the chance to generate better code for other operators.

At run time, GenesisDB uses traditional query planning to translate queries into sequences of operator invocations. This process does not involve code synthesis by models and is therefore fast and robust. GenesisDB translates query plans into code that references custom operator implementations. While their implementation is dynamically synthesized, their input/output signature is known a-priori. More precisely, for each operator, the name of the function implementing it as well as the names of its parameters are known a-priori. On the other side, the type of each input parameter can be influenced by user commands. Despite of that, as GenesisDB uses a dynamically typed language for query execution, the code referencing custom operators does not require customization. Hence, GenesisDB synthesizes code only before run time. At run time, it uses a static component generating query-specific code that invokes previously synthesized operator implementations. Note that generated operator implementations can also be used outside of GenesisDB, e.g. in the context of an existing application storing data in specialized data structures.

GenesisDB is currently an early prototype and is restricted to generating operator code in the Python programming language. Despite its early stages, the experiments show that the current version already synthesizes correct operator implementations for a majority of relational operators. While the data processing performance is currently not competitive, the proof-of-concept results show that the resulting implementations can process even queries of elevated complexity of the TPC-H benchmark. In summary, the original scientific contributions in this paper are the following:

- The paper introduces the vision of using generative AI to synthesize customizable code for core components of database management systems.
- The paper describes an early prototype of GenesisDB, a code synthesis framework using OpenAI's GPT models, that follows this approach.
- The paper presents first experimental results, revealing the general feasibility of the approach while also pointing out limitations.
- The paper discusses next steps and future research challenges, based on observations with the current prototype.

The remainder of this paper is organized as follows. Section 2 provides background information and discusses prior, related work. Section 3 describes an early prototype of GenesisDB that is used for the experiments in the following section. Section 4 reports on proof-of-concept experiments, performed with the current prototype. Finally, Section 5 discusses next steps and research challenges that need to be solved to make the approach practical.

2 Background and Prior Work

The last few years have seen transformative advances in the domain of language processing, including natural as well as formal languages (while less relevant for this publication, recent Transformer models [13] can also be used for data of other modalities than text, such as images). These advances are due to novel neural network architectures, in particular the Transformer model [22, 24], as well as to the successful application of transfer learning methods in training. Transformer models, among other advantages, facilitate the creation of large models with hundreds of billions of trainable parameters. When trained on generic tasks (e.g., predicting the next token) on sufficiently large amounts of unlabeled training data, such models require little to no specialization to solve new tasks [2]. Transformer models can be used for code synthesis [4, 12], a feature exploited by GenesisDB. Trained on large amounts of code from repositories such as GitHub, such models complete prompts, i.e. short text documents containing partial code or natural language instructions, into fully specified code in general-purpose programming languages. In certain settings, their performance is nowadays comparable to the one of human developers [8].

These developments motivate the vision of database components, synthesized via language models. GenesisDB is a prototype implementing this approach. GenesisDB is based on OpenAI's GPT models [12]. Similar models power GitHub's CoPilot [5], an auto-completion tool offered on the GitHub platform. Unlike CoPilot, GenesisDB is however specialized to generating code for relational operators that, when combined, form a complete SQL processing engine. GenesisDB features prompt templates, specialized for generating relational operators, as well as strategies needed to create a complex system from parts generated in different code synthesis steps. The size of a typical engine, generated by GenesisDB, exceeds the code size typically generated in a single model invocation. GenesisDB relates most to CodexDB [20], a system synthesizing code for processing SQL queries that additionally implements natural language instructions. While CodexDB synthesizes code for processing *one* specific SQL query, GenesisDB aims at generating a system that can process *all* queries without run time code synthesis.

GenesisDB also relates to other recent applications of language models and, more broadly, machine learning in the context of database management systems. These include, for instance, prior work on natural language query interfaces [6, 10, 19, 23, 25], as well as approaches for entity matching [9], data wrangling [11, 16], database auto-tuning [18, 21], and data integration [3]. Several recent publications use language models or, generally, machine learning to implement relational operators directly [3, 7, 15, 17],

Table 1: Operators synthesized by GenesisDB.

Group	Operators
Tests	CheckColumnType, CheckTableType
I/O	LoadTable, StoreTable
Basic	GetColumn, GetValue, CreateTable, GetNull, IsNull, IsEmpty, SetColumn, AddColumn, FillIntColumn, FillFloatColumn, FillBoolColumn, FillStringColumn, NrRows, Map, ToInt, ToFloat, ToBool, ToString, Substring, Limit
Arithmetic	Addition, Subtraction, Multiplication, Division, Floor
Boolean	LessThan, GreaterThan, LessOrEqual, GreaterOrEqual, Equal, NotEqual, Case, And, Or, Not, Filter
Aggregates	Sum, Min, Max, Avg, Count, SumGrouped, MinGrouped, MaxGrouped, AvgGrouped
Complex	Sort, InnerJoin, LeftOuterJoin, RightOuterJoin, CartesianProduct

leading to approximate processing. Instead, GenesisDB uses machine learning before run time to synthesize code for processing.

3 Prototype Overview

Figure 1 shows an overview of the GenesisDB prototype. Section 3.1 discusses query processing. Section 3.2 describes how GenesisDB synthesizes its core engine using generative AI.

3.1 Query Processing

The **Processing** sub-system (left side in Figure 1) processes SQL queries and returns results to users. GenesisDB is an analytical SQL processing engine and supports all queries of the TPC-H benchmark (but no transactions). As discussed in more detail later, it uses relational operators for processing that are synthesized according to user instructions. A GenesisDB database is represented as a directory, containing a file with SQL commands creating the corresponding schema, as well as one .csv file for each table in the database (containing the data). The current prototype is restricted to processing data stored in .csv files. Expanding the scope to other data formats, likely enabling more efficient processing, is part of the future work plans.

The database catalog contains information on the database schema and file locations, extracted from the database directory. It is used to inform the query parser as well as the query planner. Currently, GenesisDB uses a simple, rule-base query planner, implemented by the Apache Calcite library [1]. The planner generates a logical query plan, determining the sequence of operations without selecting between operator implementations. Using cost-based optimization is challenging as the implementation (and, therefore, cost function) of operators changes dynamically. Learned cost models requiring a few samples

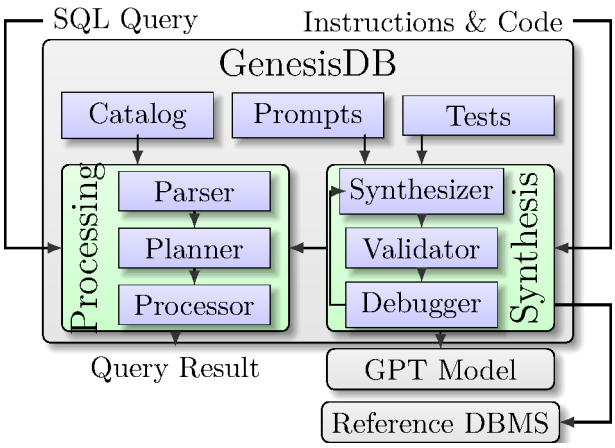

Figure 1: Overview of GenesisDB system.

Table 2: GenesisDB accepts natural language instructions or code snippets for the following SQL engine properties.

Property	**Semantics**
Prefix	Common prompt prefix (e.g., imports)
Table	How to represent relations
Column	How to represent columns
Null	How to represent the SQL NULL value
Boolean	How to represent Boolean values
Integer	How to represent integer values
Float	How to represent float values
String	How to represent string values
Suffix	Common prompt suffix (e.g., behavior)

may help in the future.

The SQL processor translates logical query plans into a sequence of steps, using a set of 54 operators, shown in Table 1. These operators are more fine-grained than the ones typically used in database engines (e.g., introducing separate operators for aggregation with and without grouping as well as for different join types). The goal of using relatively specialized operators is to make operator synthesis easier, reducing the number of cases that need to be handled per function. GenesisDB generates code for processing queries, combining synthesized code implementing operators with code invoking the latter to execute the query plan. In the current prototype, all code is formulated in Python (Version 3.8). Generating code in lower-level languages such as C is planned for future versions.

3.2 Operator Synthesis

The distinguishing feature of GenesisDB, compared to prior systems, is that it synthesizes operator implementations using generative AI. This opens up possibilities to customize generated components according to user instructions. In principle, such instructions may change the in-memory representation of tables, columns, or fields, as well as inject custom behavior during processing. For instance, this may include generating output that is useful for debugging or training, executing custom performance or data analysis, or writing checkpoints according to user instructions. GenesisDB generates code via

```
def not_equal(column_1, column_2):
    """ True where column_1 <> column_2.

    1. Return [Null] for rows where one input row is [Null].
    2. Return true iff first row <> second row otherwise.
    3. Ensure that the output is [Column].

    Args:
        column_1: [Column].
        column_2: [Column].

    Returns:
        [Column] containing Boolean values.
    """
```

Figure 2: Prompt template for generating NotEqual operator. The template contains placeholders (marked in color) that can be customized by users.

prompting [14], i.e. by submitting small text documents describing the generation task to generative models. Users influence the generation process by providing substitutes for placeholders in prompt templates or custom text that is added as a prefix or suffix to default prompts. Table 2 shows an overview of all the prompt components that can be influenced by users. An example of a prompt template containing placeholders from Table 2 follows.

Example 5: Figure 2 shows an example prompt template, used to generate the "NotEqual" operator. Prompt parts marked up in color represent snippets that can be customized by users.

Algorithm 1 shows high-level pseudo-code for the synthesis process. Given user instructions, a list of operators to synthesize, a set of test cases to validate synthesized operators, and, optionally, default implementations for each operator, it returns a custom engine (i.e., operator implementations) that follows the input instructions. The full list of operators, as well as their semantics and function signatures, remain fixed (to enable the query processor to use them appropriately to realize query plans). If desired, synthesis may only focus on a subset of operators (while using default implementations for others). To validate generated implementations, the current prototype uses a set of 172 test cases by default (users can easily add new test cases that are automatically used during synthesis). Test cases are either realized as SQL queries (then, GenesisDB compares query results generated by custom operators to the ones generated by a reference system) or as small code samples referencing operators (here, GenesisDB ensures that all assertions hold).

GenesisDB first sorts operators using a simple heuristic (based on the length of the associated prompt), prioritizing operators that are potentially easier to synthesize. As discussed more in the following sections, operator order matters as it influences, for instance, the order in which tests are performed. Also, code synthesized for operators that are ordered earlier may be used as a sample when synthesizing operators that appear later. After sorting, GenesisDB synthesizes the first version of the engine by generating one implementation for each required operator ($E.code[o]$ denotes the implementation of operator o).

Next, GenesisDB validates generated code via test cases and tries to fix problems via re-synthesis. This process continues until the generated engines passes all test cases or until a user-defined timeout is reached. Testing via Function RunTests stops at the first failed test case. The result of testing is a summary, reporting passed and failed tests. GenesisDB uses that summary to identify likely faulty

Algorithm 1 Generating SQL execution engines based on natural language instructions.

```
1: // Returns SQL processing engine using code synthesized
2: // according to natural language user instructions U or default
3: // implementations D to implement operator list O, validated
4: // via test cases T.
5: function GENESIS(U, O, T, D)
6:     // Choose order in which operators will be validated
7:     O ←SORTOPERATORS(O)
8:     // Synthesize initial code for each operator
9:     E ←ENGINE.INIT
10:    for o ∈ O do
11:        c ←SYNTHESIZE(U, E, O, o)
12:        E.code[o] ← c
13:    end for
14:    // Run tests and re-synthesize faulty operators
15:    while Not all tests pass and no timeout do
16:        // Validate synthesized engine
17:        r ←RUNTESTS(O, E, T)
18:        // Find operators likely to have bugs
19:        F ←FAULTYOPERATORS(r)
20:        // Try replacing faulty operators
21:        E ←FIXOPERATORS(U, O, D, E, F, r.tests)
22:    end while
23:    return E
24: end function
```

operators, and then tries to fix them. Fixing an operator involves re-synthesizing its code, possibly with a different prompt and different synthesis settings. If this approach fails to resolve previous problems, using a limited number of tries, GenesisDB uses default operator implementations instead. Those default operators should use the same data representation as the desired target engine (in order to be compatible). They may however not implement any custom behavior, requested by the user. Therefore, GenesisDB tries to minimize the number of default operators used. If default operators are not specified, GenesisDB ultimately notifies the user, hinting at operators that likely need replacement. After providing the corresponding code, the synthesis process can be restarted.

4 Proof-of-Concept Experiments

The goal of the experiments is to test whether the prototype is able to synthesize operator implementations that can process complex SQL queries.

4.1 Experimental Setup

All experiments are executed on a t2.2xlarge EC2 instance, featuring eight virtual CPUs, 32 GB of main memory, and 500 GB of EBS storage. The instance is running Ubuntu and GenesisDB uses Python 3.8 to execute queries. Postgres 10.22 is used to generate reference results for all test cases. All test queries are executed on a TPC-H database with a low scaling factor of 0.01 to speed up test evaluation. The following experiments use GPT-3 Codex for code synthesis. This model is relatively small, compared

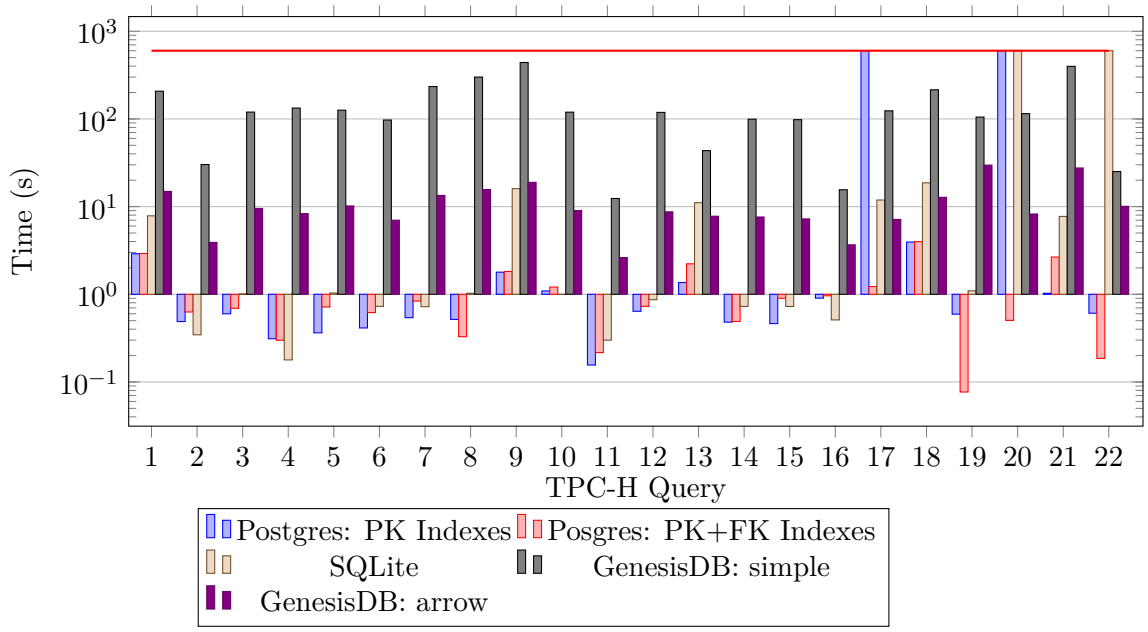

Figure 3: Performance on TPC-H queries with scaling factor 1. The red line marks the timeout of 10 minutes.

to recently released models, but has been specialized for code generation. GenesisDB retries operator synthesis five times in case of errors. It uses 172 test cases for validation, including all TPC-H queries, executed on a TPC-H database with a scaling factor of 0.01.

Tables 3 and 4 show values used to substitute placeholders in prompt templates in two separate experiments. The first experiment synthesizes a system that exploits standard Python classes for representing data in main memory. The second experiment generates an engine that relies on PyArrow to represent and process data. To increase the chances of successful generation, each prompt template was prefixed by the implementation of a simple operator (incrementing the values in a column by one) that follows the requested style (i.e., in the first experiment, the increment operator uses Python lists as input, in the second experiment it operates on Pyarrow data structures). Whenever GenesisDB is unable to synthesize an operator after at most five tries, a corresponding operator implementation is added manually. The ratio of successfully generated operators is one of the evaluation metrics discussed in the next subsection.

4.2 Experimental Results

Using Python data structure, GenesisDB is able to synthesize correct operator implementations in 89% of cases. On the other hand, GenesisDB was only able to generate correct implementations for 48% of operators when customizing synthesis for PyArrow data structures. Generating operators and running tests took about four hours for the first experiment and about three hours for the second experiment.

Figure 3 shows performance results for TPC-H queries, comparing GenesisDB (in arrow configuration and with simple Python lists as in-memory data representation) to Postgres 10.22 with primary key or primary and foreign key indexes and SQLite 3.36. Clearly, traditional database management systems achieve optimal performance. On the other hand, there are a few queries where the PyArrow implementation performs better than some of the traditional systems. Overall, while performance improvements are a primary goal of future work, the results show that the prototype can synthesize operator imple-

Table 3: Settings used to synthesize a simple SQL engine.

Property	**Value**
Prefix	import functools import operator import streamlit as st import time
Table	a list of rows where each row is a list
Column	a list
Null	None
Boolean	bool
Integer	int
Float	float
String	str

Table 4: Settings to synthesize SQL engine using Pyarrow.

Property	**Value**
Prefix	import pyarrow as pa import pyarrow.compute as pc import pyarrow.csv as csv
Table	a pyarrow Table
Column	a pyarrow Array
Null	None
Boolean	pa.bool_()
Integer	pa.int64()
Float	pa.float64()
String	pa.string()

mentations enabling processing of complex SQL queries. More importantly, the results provide evidence that customization has a significant impact on the properties of the generated implementations, notably run time.

5 Future Work

The proof-of-concept experiments demonstrate that language models can synthesize code for a majority of relational operators. At the same time, they provide the first evidence for the possibility of customizing the generated operator implementations, leading to significantly different behavior in the generated systems. While these first results are promising, they also reveal important limitations that need to be addressed in future work.

First, the performance results demonstrate the limitations of using Python for implementing relational operators. The choice of the Python programming language for those first experiments is motivated by several factors. First of all, Python is advertised as one of the languages in which GPT models are most proficient in (due to a large amount of corresponding training data in the corpora used for pre-training). At the same time, high-level programming languages such as Python enable relatively concise pieces of code that implement standard operators. Generating shorter pieces of code is oftentimes

easier for language models, compared to generating the more verbose operator implementations that are typical for traditional database management systems. However, as revealed in the experiments, the performance penalty of using Python is enormous, motivating future research that aims at generating code for relational operators in lower-level languages such as C.

Generating code in lower-level programming languages leads to new research challenges. First of all, whereas the current implementation aims to generate entire operator implementations in case of suspected errors, this may become inefficient in the case of larger operator implementations. Instead, it seems prudent to restrict re-synthesis efforts to parts of operator implementations that are likely to be incorrect. Here, language models can help to identify code parts, based on an analysis of error messages or inconsistent results, that likely require re-generation. Second, whereas the current implementation merely provides a description of desired operator behavior to the language model, future implementations could provide more information, facilitating the synthesis task. For instance, it might be possible to provide language models with a simple operator implementation as part of the prompt input, thereby facilitating the task of generating more sophisticated versions. Also, instead of generating operators from scratch, it may be easier to "morph" an operator implementation in multiple steps, running automated tests after each transformation step.

A complementary avenue to improve the performance of the generated implementations is to bias synthesis, based on automated performance tests. For instance, given user-defined performance goals, the system could regenerate operators whenever performance goals are not met. Of course, doing so introduces additional constraints that may make it harder to generate operator implementations that pass all correctness and performance tests.

Another source of future research challenges lies in increasing the success ratio of code synthesis. As shown in the experiments, the current prototype is not able to generate correct code in all cases. As a first step, a study evaluating the cost-quality tradeoff achieved by different OpenAI models (e.g., the recently released OpenAI o1 model) or models of other providers is interesting. Another possibility for future improvements is to replace generic language models by versions that are specialized for the task of generating relational operators. For instance, many of OpenAI's GPT model variants enable users to apply fine-tuning, meaning to re-train models on a corpora that is more representative of the specific tasks they are targeted at. In the specific scenario investigated in this paper, relevant training data could incorporate code from (open-source) database management systems. As the set of relational operators is fairly standardized across different database management systems, fine-tuning on existing code likely provides valuable information to the language model for operator synthesis. At the same time, this could enable GenesisDB to integrate advanced techniques that have been proven to improve performance for relational data processing, thereby benefitting performance as well.

Finally, future research could focus on improving the interaction between the user and the system. Language models cannot currently provide guarantees on generating accurate code. Therefore, coding assistants are typically considered tools for human-in-the-loop software development, rather than purely automated tools. As shown in the experiments, GenesisDB requires help to synthesize complete sets of operators as well. While some of the aforementioned approaches may push the boundaries in terms of the success ratio in code synthesis, it still seems likely that GenesisDB requires guidance from users to deal with complex scenarios. However, the time of users is precious which motivates research on how to get the most valuable insights from users with limited time investments from their side. For instance, users with an IT developer background could provide sample code that is requested by the language model. Here, the goal would be to carefully select which samples to request, maximizing the benefit for code synthesis. Alternatively, users without an IT background could still be helpful in verifying whether or not generated engine implementations satisfy the features requested via customization. Again, minimizing the number of questions addressed to users will make the system more practical.

6 Conclusion

This paper introduces the vision of highly customizable database management systems. Given recent developments in the domain of generative AI, it becomes possible to generate core components of database execution engines via generative AI. This opens up new possibilities for customization, according to user specifications.

This paper described the first prototype of GenesisDB, a code synthesis framework powered by language models that implements this approach. First experimental results show that GenesisDB is able to generate correct code for a majority of operator implementations. On the other hand, the current prototype is only able to generate correct code for a subset of operators and the performance of the generated code is not yet satisfactory. This opens up various opportunities for future research.

Acknowledgement

This project is supported by NSF CAREER grant IIS-2239326 ("Mining Hints from Text Documents to Guide Automated Database Performance Tuning").

References

[1] E. Begoli, J. Camacho-Rodríguez, J. Hyde, M. J. Mior, and D. Lemire. Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *SIGMOD*, pages 221–230, 2018.

[2] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, pages 1877–1901, 2020.

[3] Z. Chen, J. Fan, S. Madden, and N. Tang. Symphony: Towards Natural Language Query Answering over Multi-modal Data Lakes. In *CIDR*, pages 1–7, 2023.

[4] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, P. Schuh, K. Shi, S. Tsvyashchenko, J. Maynez, A. Rao, P. Barnes, Y. Tay, N. Shazeer, V. Prabhakaran, E. Reif, N. Du, B. Hutchinson, R. Pope, J. Bradbury, J. Austin, M. Isard, G. Gur-Ari, P. Yin, T. Duke, A. Levskaya, S. Ghemawat, S. Dev, H. Michalewski, X. Garcia, V. Misra, K. Robinson, L. Fedus, D. Zhou, D. Ippolito, D. Luan, H. Lim, B. Zoph, A. Spiridonov, R. Sepassi, D. Dohan, S. Agrawal, M. Omernick, A. M. Dai, T. S. Pillai, M. Pellat, A. Lewkowycz, E. Moreira, R. Child, O. Polozov, K. Lee, Z. Zhou, X. Wang, B. Saeta, M. Diaz, O. Firat, M. Catasta, J. Wei, K. Meier-Hellstern, D. Eck, J. Dean, S. Petrov, and N. Fiedel. PaLM: Scaling Language Modeling with Pathways. *CoRR*, abs/2204.0:1–87, 2022.

[5] G. D. Howard. GitHub Copilot: Copyright, Fair Use, Creativity, Transformativity, and Algorithms *. pages 1–13, 2021.

[6] G. Karagiannis, M. Saeed, P. Papotti, and I. Trummer. Scrutinizer: A Mixed-Initiative Approach to Large-Scale, Data-Driven Claim Verification. *PVLDB*, 13(12):2508–2521, 2020.

[7] T. Kraska, M. Alizadeh, A. Beutel, E. H. Chi, J. Ding, A. Kristo, G. Leclerc, S. Madden, H. Mao, and V. Nathan. SageDB: A learned database system. In *CIDR*, pages 1–10, 2019.

[8] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, and R. Leblond. Competition-Level Code Generation with AlphaCode. *DeepMind Technical Report*, pages 1–73, 2022.

[9] Y. Li, J. Li, Y. Suhara, A. Doan, and W. C. Tan. Deep entity matching with pre-trained language models. *Proceedings of the VLDB Endowment*, 14(1):50–60, 2020.

[10] Y. Luo, N. Tang, G. Li, C. Chai, W. Li, and X. Qin. Synthesizing Natural Language to Visualization (NL2VIS) Benchmarks from NL2SQL Benchmarks. In *SIGMOD*, pages 1235–1247, 2021.

[11] A. Narayan, I. Chami, L. Orr, and C. Ré. Can Foundation Models Wrangle Your Data? *PVLDB*, 16(4):738–746, 2022.

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

[13] OpenAI. GPT-4 Omni Release, 2024.

[14] T. L. Scao and A. M. Rush. How Many Data Points is a Prompt Worth? In *NAACL*, pages 2627–2636, 2021.

[15] S. Suri, I. Ilyas, C. Re, and T. Rekatsinas. Ember: No-Code Context Enrichment via similarity-based keyless joins. *PVLDB*, 15(3):699–712, 2021.

[16] N. Tang, J. Fan, F. Li, J. Tu, X. Du, G. Li, S. Madden, and M. Ouzzani. Rpt: Relational pre-trained transformer is almost all you need towards democratizing data preparation. *PVLDB*, 14(8):1254–1261, 2021.

[17] J. Thorne, M. Yazdani, M. Saeidi, F. Silvestri, S. Riedel, and A. Halevy. From natural language processing to neural databases. *Proceedings of the VLDB Endowment*, 14(6):1033–1039, 2021.

[18] I. Trummer. The Case for NLP-Enhanced Database Tuning: Towards Tuning Tools that "Read the Manual". *PVLDB*, 14(7):1159–1165, 2021.

[19] I. Trummer. BABOONS: Black-box optimization of data summaries in natural language. *PVLDB*, 15(11):2980 – 2993, 2022.

[20] I. Trummer. CodexDB: Synthesizing Code for Query Processing from Natural Language Instructions using GPT-3 Codex. *PVLDB*, 15(11):2921 – 2928, 2022.

[21] I. Trummer. DB-BERT: a Database Tuning Tool that "Reads the Manual". In *SIGMOD*, pages 190–203, 2022.

[22] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is All You Need. In *Advances in Neural Information Processing Systems*, pages 5999–6009, 2017.

[23] N. Weir, A. Crotty, A. Galakatos, A. Ilkhechi, S. Ramaswamy, R. Bhushan, U. Cetintemel, P. Utama, N. Geisler, B. Hättasch, S. Eger, and C. Binnig. DBPal: Weak Supervision for Learning a Natural Language Interface to Databases. pages 1–4, 2019.

[24] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. Le Scao, S. Gugger, M. Drame, Q. Lhoest, and A. Rush. Transformers: State-of-the-Art Natural Language Processing. In *EMNLP*, pages 38–45, 2020.

[25] V. Zhong, C. Xiong, and R. Socher. Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning. *CoRR*, abs/1709.0(1):1–12, 2017.