QueryArtisan: Generating Data Manipulation Codes for Ad-hoc Analysis in Data Lakes

Xiu Tang
Zhejiang University
Hangzhou High-Tech Zone (Binjiang) Blockchain and Data Security Research Institute
tangxiu@zju.edu.cn

Wenhao Liu
Zhejiang University
Hangzhou High-Tech Zone (Binjiang) Blockchain and Data Security Research Institute
wenhao.liu@zju.edu.cn

Sai Wu
Zhejiang University
Hangzhou High-Tech Zone (Binjiang) Blockchain and Data Security Research Institute
wusai@zju.edu.cn

Chang Yao
Zhejiang University
Hangzhou High-Tech Zone (Binjiang) Blockchain and Data Security Research Institute
changy@zju.edu.cn

Gongsheng Yuan
Zhejiang University
Hangzhou High-Tech Zone (Binjiang) Blockchain and Data Security Research Institute
ygs@zju.edu.cn

Shanshan Ying
ApeCloud
shanshan.ying@apecloud.com

Gang Chen
Zhejiang University
Hangzhou High-Tech Zone (Binjiang) Blockchain and Data Security Research Institute
cg@zju.edu.cn

ABSTRACT

Query processing over data lakes is a challenging task, often requiring extensive data pre-processing activities such as data cleaning, transformation, and loading. These tasks typically demand significant human intervention. However, the advent of Large Language Models (LLMs) has illuminated a new pathway to address these complexities by offering a unified approach to understanding the diverse datasets submerged in data lakes. In this paper, we introduce QueryArtisan, a novel LLM-powered analytic tool specifically designed for data lakes. QueryArtisan transcends traditional ETL (Extract, Transform, Load) processes by generating just-in-time code for dataset-specific queries. It eliminates the need for an intermediary schema, enabling users to query the data lake directly using natural language. To achieve this, we have developed a suite of heterogeneous operators capable of processing data across various modalities. Additionally, QueryArtisan incorporates a cost model-based query optimization technique, significantly enhancing its code generation capabilities for efficient query resolution. Our extensive experimental evaluations, conducted with real-life datasets, demonstrate that QueryArtisan markedly outperforms existing solutions in terms of effectiveness, efficiency and usability.

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/skyrise-l/QueryArtisan.

1 INTRODUCTION

A data lake is designed to serve as a comprehensive storage hub, engineered to accommodate, manage, and safeguard vast quantities of data, irrespective of its structure [30, 33]. Whether the data is meticulously organized, semi-structured, or entirely unstructured, a data lake is capable of preserving it in its original format [19]. Moreover, it is adept at processing an extensive array of data types, unbounded by size constraints.

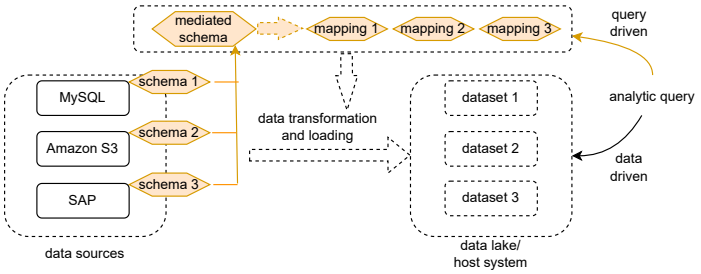


Figure 1: Conventional data lake processing.

However, the provision of data lake services necessitates intricate engineering efforts. Figure 1 illustrates the typical processing scenario in conventional data lake systems, where two distinct methodologies are commonly employed. In both strategies, it is essential to define a unified, mediated schema and establish mapping relationships between each source schema and this central schema.

In the query-driven approach, any query aimed at the mediated schema is translated into corresponding queries for each source dataset, based on these mapping relationships [48]. These queries are then processed individually by the source systems. The data lake's role is to amalgamate the results into a cohesive output.

Conversely, in the data-driven model, we transform all incoming data to align with the mediated schema and subsequently load them into our host system within the data lake [1, 14]. Here, the host system is responsible for processing all queries, providing a centralized solution for data handling.

Implementing a data lake system poses three primary challenges:

Expert Knowledge Requirement for Schema Definition. Crafting an appropriate mediated schema demands extensive domain expertise and a comprehensive understanding of all source data. Achieving this level of insight may not always be feasible.

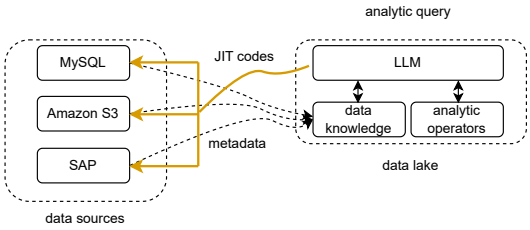


Figure 2: LLM-Powered data lake processing.

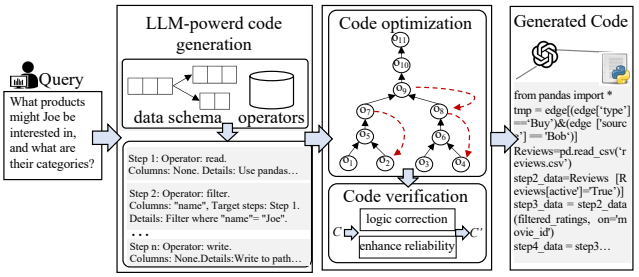


Figure 3: The architecture of QueryArtisan system.

Complexity in Schema Mapping. Even with the aid of semi-automatic tools, establishing schema mappings is a complex and labor-intensive task. This process often results in significant processing overheads and necessitates considerable human intervention.

The Data Transformation Dilemma. This challenge is twofold, depending on the approach chosen. In the data-driven approach, the process of data transformation and loading is resource-intensive and costly. On the other hand, the query-driven approach introduces complexities in query rewriting and result merging, making it challenging to ensure the correctness and tractability of query results.

To tackle these challenges, this paper introduces QueryArtisan, an innovative query engine for data lakes powered by Large Language Models (LLMs). QueryArtisan transforms the conventional data lake system in three ways.

- It allows users to submit queries in natural language, making the process more intuitive and accessible.
- By completely bypassing the requirement for a mediated schema, it simplifies the query process.
- There is no necessity for data transformation and loading, streamlining operations.

This is achieved through a Just-In-Time (JIT) code generation strategy as shown in Figure 2. For each specific query, the LLM (Large Language Model) is tasked with generating processing codes tailored to each source data. This approach notably reduces the cost and complexity of building and maintaining data lake services, offering a more efficient and user-friendly solution.

While directly utilizing LLMs for code generation can occasionally lead to unstable performance and inconsistent results, QueryArtisan addresses these concerns with several optimization strategies. First, QueryArtisan includes a meticulously defined repository of data processing operators, each semantically detailed to enhance understanding and application. Second, the system is fed with comprehensive metadata from source datasets, including schemas, data types, and distributions. This enriched context allows the LLM to generate more accurate and relevant code. Third, we direct the LLM to answer queries using our pre-defined operators, supplemented by any additional necessary code. This approach streamlines the code generation process.

To further refine the process, QueryArtisan incorporates a plan optimizer. This component is capable of translating generated codes into equivalent versions that are optimized for enhanced performance. Moreover, to ensure the reliability of the results, QueryArtisan employs a specialized code verification algorithm. This algorithm validates the correctness of the generated codes, providing a layer of assurance for accurate outcomes.

We evaluate the performance of QueryArtisan on multiple datasets. QueryArtisan is not only more effective in querying the complex relationships across the unstructured documents, semi-structured data and structured tables, but also more robust for the traditional query tasks over the structured data compared to the SOTA systems.

2 OVERVIEW

In this section, we define our problem and present an overview of our proposed solution. Our Just-In-Time (JIT) code generation strategy eliminates the need for costly Extract, Transform, Load (ETL) processes when handling data analytics tasks across heterogeneous datasets within a data lake. We define the JIT code generation process as follows:

Definition 2.1. Given an analytic query Q in natural language and a data lake consisting of multiple data sources $D = \cup_{i=1}^{k} D_i$, QueryArtisan generates executable code snippets $C = \cup_{i=1}^{k} C_i$ for each data source. These code snippets calculate results $R = \cup_{i=1}^{k} R_i$, individually. Subsequently, QueryArtisan combines all the results to provide the answer to query Q.

To enable Just-In-Time (JIT) code generation, QueryArtisan is equipped with two essential components: *Code Generation* and *Code Optimization and Validation*. Figure 3 provides an architectural overview of QueryArtisan.

Code Generation. We maintain a diverse set of heterogeneous operators denoted as $\boldsymbol{O}$. These operators encompass a wide array of data manipulation actions, effectively serving as the foundational building blocks for generating code to answer queries. Upon receiving a Natural Language (NL) query Q, LLM initiates the process by identifying query-related operators from the operator set $\boldsymbol{O}$. This initial step involves mapping Q to a subset O_q of operators in $\boldsymbol{O}$ that collectively capture the semantics of Q. Operators within O_q collaborate with one another. To facilitate this collaboration, we instruct the LLM to construct a query analysis graph $G_q = \{V, E\}$, where V represents nodes labeled as v_i, each corresponding to a selected operator in O_q. Edges $(v_i, v_j) \in E$ signify the dependency of operator v_i on the output produced by operator v_j. With G_q in place, the LLM proceeds to generate code $C(G_q)$, which is customized to the operators within O_q and the data they manipulate. This code generation process adheres to the identified dependencies within

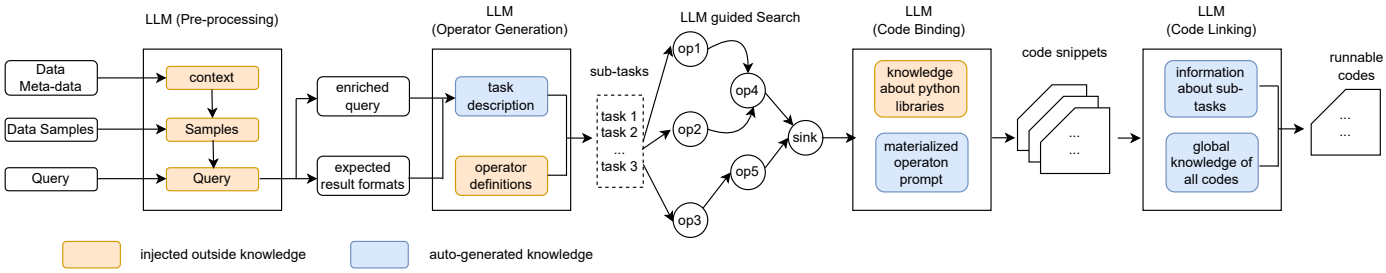


Figure 4: LLM-powered code generation. This pipeline includes pre-processing, abstract definition of operators, LLM-guided search on offline operator database, tasks to operator graph, code binding and linking.

G_q. This streamlined approach effectively narrows the semantic gap between natural language queries and machine-executable code.

Code Optimization and Validation. LLM-generated codes often exhibit common drawbacks, including inefficiency and potential invalidity, particularly in cases involving complex logic [12]. To enhance efficiency, when presented with the LLM-generated query analysis graph G_q and its initial codes $C(G_q)$, QueryArtisan takes a step further by refining G_q using our cost model-based plan optimizer. The optimizer is responsible for optimizing G_q by performing operator replacements or relocations, resulting in an equivalent form denoted as G'_q. This refined version, guided by our cost model, demonstrates significantly improved performance. To ensure the correctness of the generated codes, QueryArtisan conducts a comprehensive code verification process. This verification encompasses checks for syntactic correctness, logical consistency, and adherence to predefined data processing rules.

Finally, as observed by previous work [39], LLM cannot generate accurate code due to the absence of contextual information. Therefore, we have introduced knowledge injection modules throughout the two phases. First, it provides crucial dataset descriptions to the LLM, including schemas, data distributions, and data types. Second, to narrow the scope of LLM searches during code generation, we materialize the abstract operators O with extracted knowledge from the background datasets. The contents are provided to LLMs through chain-of-thought prompting [42], helping the LLM process complex information effectively. For datasets with thousands of columns, we use existing open-source methods [10, 11, 40, 41] to identify relevant tables and columns upon receiving a query.

3 LLM-POWERED CODE GENERATION

In this section, we delve into the process by which QueryArtisan generates code for a particular dataset in response to an incoming NL query. The overall workflow is illustrated in Figure 4, wherein we engage with the LLM by utilizing both external knowledge and auto-generated knowledge to yield precise and high-quality inference results.

3.1 Pre-processing

During the pre-processing stage, we merge three key components (meta-data, data samples and NL query) into an enriched query through the LLM. In particular, we perform an unbiased sampling for the target dataset to collect the meta-data M and key samples S.

As depicted in Figure 5, M consists of three distinct components. The "dataset info" encompasses comprehensive details about the entire dataset, including its format, size, and a general description. Through the process of scanning and sampling the dataset, we also acquire the "schema info" and "instance info". Furthermore, certain critical findings, such as primary-foreign key relationships and histograms, are derived using conventional algorithms [20, 53].

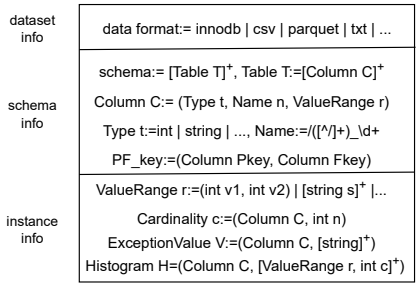


Figure 5: Meta-data for LLM.

Due to the constraint on prompt length, we are unable to provide an excessive number of samples to the Large Language Model (LLM). Therefore, we opt for a selection of the most representative samples. To achieve this, we employ a specific approach: we randomly select samples from the largest fact tables and subsequently follow the primary-foreign keys to retrieve joinable tuples from the associated dimension tables. Additionally, as part of our strategy, we intentionally introduce some tuples with noisy values, such as null values and values with invalid formats.

In the final step, both the meta-data M and the key samples S serve as contextual information for the NL query. We engage the LLM to produce an enriched query Q_e in the format of $[context, query]$. In this process, the LLM extracts essential details from M and S to refine and rewrite the query for proper declaration.

3.2 Operator-based Search

With the enriched query Q_e at hand, we aim to narrow down the scope of the LLM search, thereby encouraging the generation of more predictable code outputs. To achieve this objective, we introduce a set of abstract data processing operators. These operators encompass functions such as *read, filter, join, agg, reshape, group, sort, replace* and *update*. Figure 6 provides an abstract definition of the "filter" operator, outlining the fundamental knowledge necessary for its proper execution.

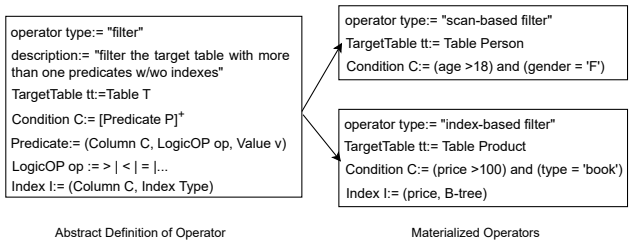


Figure 6: Abstract definition vs. materialized operators.

For an enriched query, denoted as Q_e, we task the LLM with generating a set of sub-tasks, denoted as $J_0, ..., J_k$, each of which is equivalent to a sub-query. These sub-tasks entail multiple steps, such as "Step 1: Scan the dataset to retrieve columns of AGE and SALARY; Step 2: ...". For each individual task, represented as J_i, we leverage the LLM to construct a processing graph, denoted as $G_i = \{V, E\}$, by providing it with operator definitions and task descriptions. In this context, V represents the set of operators, while E defines their dependencies. By executing the operators within G_i, we can effectively address the task at hand.

Throughout the graph construction process, the LLM not only creates an abstract operator but also translates it into one or more concrete operators, as depicted in Figure 6. These materialized operators, through the incorporation of specific configurations concerning tables, indexes, and data access methods, essentially represent a detailed execution plan guided by predefined priority rules. These rules are provided to the LLMs, enabling them to autonomously select the most appropriate physical operators.

3.3 Offline Operator Database

To reduce reliance on the LLM and minimize unnecessary calls, we developed an offline database containing typical natural language query patterns from the WikiSQL [54] and Spider [47] training datasets. This database, $\mathcal{D}$, consists of key-value pairs (J_i, G_i), which represent generic processing graphs for typical queries. To ensure generalizability and remove dataset-specific values, we have replaced detailed tables, columns, and variable names with placeholders. This setup enables quick matching of similar patterns for incoming queries, significantly reducing query processing time.

3.4 Tasks to Operator Graph

The availability of dataset $\mathcal{D}$ empowers us to reframe the query processing challenge as a sub-graph matching problem. When confronted with a query Q, we generate a set of sub-tasks $\{J_0, ..., J_k\}$ by consulting the Language Model (LLM). In addition, we capture the dependencies between these sub-tasks in the form of a graph denoted as G_{dep}, where the vertices represent individual tasks, and edges delineate their interdependencies. Subsequently, for each J_i, we embark on a search within $\mathcal{D}$ to pinpoint the most closely related historical tasks. The similarity between two tasks, represented as J_i and J_x, is quantified using the following formula:

$$sim(J_i, J_x) = cosine(f(J_i), f(J_k)). \quad (1)$$

Here, the function f makes use of a pre-trained Doc2Vec model [22], and we measure the similarity by computing the cosine distance between their respective vectors. A minimum similarity threshold, τ, is enforced to determine task-matching eligibility. If $\text{sim}(J_i, J_x) < \tau$, the match is deemed non-similar, and the LLM autonomously matches the operators and then generates the operator graph.

One straightforward approach involves searching for the most similar task from $\mathcal{D}$ for each sub-task and reusing their respective processing graphs. This results in a collection of graphs $\{G_0, ..., G_k\}$. To execute the overall task, it's essential to connect these sub-graphs and create a comprehensive graph denoted as $\mathcal{G}$. This process is known as operator linking. However, a significant challenge arises when attempting to link these sub-graphs due to discrepancies in their input and output formats. To overcome this issue, an additional operator, termed *reshape*, is introduced to align the formats, albeit incurring additional overhead.

To tackle this problem, we define the similarity between the new task $\mathcal{J} = \{J_0, ..., J_k\}$ and a sequence of database tasks $\bar{\mathcal{J}} = \{\bar{J}_0, ..., \bar{J}_k\}$ as follows:

$$sim(\mathcal{J}, \bar{\mathcal{J}}) = \sum_{J_i \in \mathcal{J}} sim(J_i, \bar{J}_i) - \sum_{\forall e \in G_{dep}} g(e.Start, e.End). \quad (2)$$

The function g returns 0 when the output of task $e.Start$ can directly serve as the input for $e.End$; otherwise, it returns 1.

Given a sub-task set $\mathcal{J}$, our goal is to retrieve a set of corresponding tasks $\bar{\mathcal{J}}$ from $\mathcal{D}$, satisfying:

$$\arg\max_{\forall \bar{\mathcal{J}}} sim(\mathcal{J}, \bar{\mathcal{J}}).$$

To find the best match, we consider the k matching slots as k matching arrays, and apply the variant of TA (threshold algorithm) [44] to find the top solution.

3.5 Code Generation

To address the challenges related to the accuracy and reliability of the LLM when generating code, we adopt a strategy that involves requesting the LLM to generate m sets of sub-tasks (typically, setting m to 3 suffices). For each set, we conduct a search within our offline database to obtain the highest-ranked graph. Subsequently, we generate code for each of these graphs and select the optimal one based on the criteria established in our optimization and validation module, as detailed in Section 4.

The process of generating code for a graph unfolds in a progressive manner. To be specific, we commence with the vertices (operators) within the graph that have no predecessors. Gradually, we traverse the edges of the graph, sequentially assembling the entire piece of code. We define a running state, denoted as $s = [\mathcal{J}', G', C]$, wherein $\mathcal{J}'$ represents the tasks that have already been covered by the existing sub-graph G', and C signifies the current collection of code snippets. Whenever a new operator is translated into a code snippet, we update the state s, which serves as the context prompt for future code generation.

Ultimately, we execute a code-linking procedure to guarantee that all code snippets can be seamlessly merged to yield the final result. When integration is not straightforward, we introduce the necessary complementary code to ensure the desired outcome.

4 CODE OPTIMIZATION AND VALIDATION

In this section, we will delve into our post-processing stage, encompassing the crucial components of *Code Optimization and Validation.*

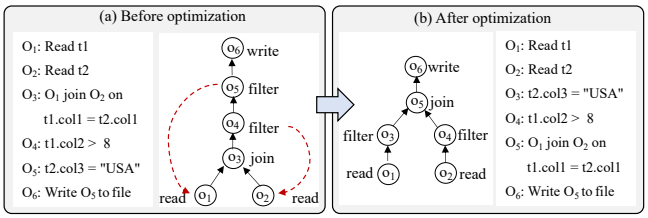


Figure 7: The example for *Filter* operator pushdown.

4.1 Code Optimization

Given the final processing graph $\mathcal{G}$ discussed in Section 3.4, it becomes evident that there is room for substantial improvement in the initial code generated. Specifically, we introduce two key adjustments for $\mathcal{G}$, known as *shuffle* and *collapse*.

The *shuffle* approach involves altering the order of two operators within $\mathcal{G}$. For instance, QueryArtisan aims to enhance query plans by strategically placing *filter* operators deeper into the execution sequence. Figure 7 serves as an illustrative example of this technique, where two *filter* operators are swapped with a join operator. Figure 7(a) outlines the original query plan steps, while Figure 7(b) showcases the optimized query plan resulting from two *shuffle* adjustments. It's important to note that the two adjacent operators within $\mathcal{G}$ can be switched only if their order has no impact on the processing results. This verification can be conducted by examining the inputs and outputs of the operators involved.

The *collapse* approach, on the other hand, aims to consolidate multiple operators and generate code for them simultaneously. For instance, it allows us to merge the two *filter* operators shown in Figure 7(a). Furthermore, we can combine the *filter* operator with the *read* operator on table $t1$. The *collapse* adjustment can often result in more efficient code generation. By considering multiple operators together, LLM can eliminate unnecessary data transformations, repeated reads, and other I/O operations during code generation. Additionally, it enhances the cache hit ratio by processing multiple operators for a single tuple simultaneously. Nonetheless, there is a trade-off involved; merging too many operators together can compromise our flexibility and render the processing logic too intricate for efficient optimization by LLM. In our case, we only allow to *collapse* at most two operators.

Moreover, in order to assess the extent to which these two adjustments decrease processing costs and enhance efficiency, QueryArtisan utilizes a cost model-based plan optimization technique, which is similar to the one adopted by the database system. This component adeptly transforms the query analysis graph G_q into performance-optimized equivalents denoted as G'_q.

Queries can manifest in various equivalent implementations, with their execution efficiency inherently tied to the specific data distribution, size, and selectivity. QueryArtisan constructs a cost model designed to assess the anticipated costs associated with these equivalent implementations. This cost model quantitatively evaluates data and query characteristics, as elaborated below:

$$C(G_q) = \sum_{i=1}^{n} \left(C_{op}(O_i) + C_{data}(O_i, O_{i+1}) \right) \tag{3}$$

where $C_{op}(O_i)$ denotes the operational cost of the i-th operator, and $C_{data}(O_i, O_{i+1})$ represents the cost associated with moving data from operator O_i to O_{i+1}. The cost model empowers QueryArtisan to flexibly adjust query plans, with a primary focus on enhancing efficiency and performance. Whether through adjustments such as *shuffle* and *collapse*, or by reconfiguring the plan, the ultimate objective remains consistent: to minimize execution cost $C(G_q)$ while ensuring accurate and consistent results.

4.2 Automatic Code Verification

We delve into empirical verification techniques crafted to thoroughly validate the generated code and minimize potential inaccuracies. Given a query analysis graph G_q, QueryArtisan systematically inspects each operator o_i in sequence for verification. Upon encountering an operator, QueryArtisan applies pre-defined rules to correct it based on its type. QueryArtisan then assesses the correctness of the operator linked to the current node. Different operators undergo specific verification methods. For instance, the verification of join operators involves evaluating join conditions' logic, leveraging either established primary-foreign key relationships or a column representation model [9] to assess the feasibility of joining two tables through the cosine similarity of column vectors. Should a join prove impractical, alternative columns are considered, and if this approach fails, LLMs are engaged to reconfigure G_q.

5 EVALUATION

We conducted an extensive evaluation of QueryArtisan across multiple datasets encompassing various types. Our primary objectives were to address the following key questions:

- Can QueryArtisan surpass state-of-the-art (SOTA) query systems in terms of query accuracy? (section 5.2.1)
- Can QueryArtisan outperform SOTA query systems in terms of query efficiency? (section 5.2.2)
- What are the primary functions and contributions of the core modules within QueryArtisan? (section 5.3)

5.1 Experimental Setting

5.1.1 Datasets. To conduct a comprehensive evaluation on five different datasets of two different categories:

Open-sourced Relational Data: Within this category, we leveraged three prominent open-source Text2SQL datasets, each of which provides natural language queries along with their corresponding SQL results, and experimented with their development sets:

(i) Spider [47] is a comprehensive and diverse text-to-SQL dataset, boasting 10,181 questions and 5,693 unique complex SQL queries. It encompasses a wide array of 200 databases with multiple tables, spanning across 138 different domains.

(ii) WikiSQL [54] is a well-recognized text-to-SQL dataset that includes 2,716 relational tables and 8,421 queries in its development set. It's worth noting that a portion of the natural language queries in the WikiSQL dataset have incorrect SQL statements [27].

(iii) Bird [24] is a benchmark dataset specifically designed for large-scale, database-grounded Text-to-SQL evaluation. It features an impressive collection of over 12,751 unique question-SQL pairs spread across 95 large databases, with a cumulative size of 33.4 GB.

A notable characteristic of Bird lies in the complexity of its data, as it often incorporates JSON and binary information within textual formats, necessitating specialized parsing techniques.

(iv) The Synthetic dataset [29] is the world's largest and most diverse open-source Text-to-SQL collection. It features 105,851 records, with 100,000 for training and 5,851 for testing, covering 100 domains and around 23 million tokens.

Heterogeneous Data Lake: We conducted validation of our method using two benchmark datasets from multimodal data lakes. These datasets encompass an open-source data lake with three distinct types of modal data and a self-constructed data lake featuring five modal data types.

(i) UniBench [51] is a comprehensive multimodal e-commerce benchmark . It represents customer and product information as two relational tables, totaling 19,425 tuples. Additionally, it includes order information and transaction relationships, which are represented through a JSON document comprising 240,726 objects, as well as a graph with 1,889,246 vertices and 5,949,887 edges.

(ii) DLBench: The benchmark was sourced from the e-commerce website eBay[1], including data related to products, goods and reviews. The resulting dataset comprises 7 relational data files with a total of 1,997,759 records, 5 JSON data files encompassing 1,209,993 objects, 5 graph data files with 1,136,288 vertices and 1,008,143 edges, 403,331 images, and 2,594 text entries.

For each dataset within the heterogeneous data lakes, we create a collection of 1,000 queries along with their corresponding result sets. This process begins by transforming the data into a relational schema. We then proceed to randomly generate and execute SQL queries to retrieve the desired results. Following this, we take these SQL queries and convert them into natural language queries. The resulting query set is grouped into categories such as Q1, Q2, Q3, and Q4, each denoting the number of tables involved in the join operation; e.g. Q3 signifies queries with three tables to join.

5.1.2 Baseline Approaches. In this work, the following four methods were used to answer queries, serving as comparative baselines: (i) GPT-3.5 [34]: a sophisticated evolution in the generative pre-trained transformer (GPT) series. (ii) ACT-SQL [52]: an automated CoT example generation method that extends the in-context learning methods to the multi-turn text-to-SQL tasks. (iii) DAIL-SQL [17]: an LLM-based Text2SQL approach improves the mapping between questions and queries, and analyzes the trade-off between example quality and quantity. It refreshes the Spider and Bird leaderboard and ranks the first place. (iv) CodexDB [39]: a framework on top of GPT-3 Codex that decomposes complex SQL queries into a series of simple processing steps, described in natural language. Codex translates the articulated steps into query processing code.

5.1.3 Evaluation Metrics. To evaluate query performance, we focus on query accuracy and efficiency. Query accuracy assesses the correctness of the query results, quantifying how precisely the queries retrieve the intended data. Query efficiency consists of two key indicators. First, $t1$ considers only query execution time. The average query time measures the execution time of the generated codes. Second, the end-to-end total time T includes code generation time $t2$ (Sec. 3), code optimization time $t3$ (Sec. 4), and $t1$.

[1]https://www.ebay.com

Table 1: Results of query accuracy.

Methods	Spider	Wikisql	Bird	Synthetic	UniBench	DLBench
GPT-3.5 (3.5)	65.9	66.3	30.9	53.6	68.2	61.1
ACT-SQL (3.5)	80.4	71.2	28.9	59.7	43.5	40.0
DAIL-SQL (3.5)	79.6	80.9	44.7	64.2	35.1	40.1
CodexDB (3.5)	78.2	80.5	19.7	62.4	63.2	69.3
Ours (Llama 3)	81.1	86.5	55.7	75.6	79.2	73.4
Ours (3.5)	**86.9**	**92.8**	**65.2**	**79.8**	**85.0**	**79.2**
Ours (3.5)$_{!OD}$	86.3	92.4	64.8	79.3	84.5	78.6
Champions	91.2	92.9	63.4	–	–	–
Ours (4)	**91.5**	**93.6**	**74.4**	**86.4**	**89.6**	**84.0**
Ours (4)$_{!OD}$	91.0	93.3	73.8	85.7	89.1	83.3

5.1.4 Implementation Details. The QueryArtisan system was crafted using Python for its core components, with the integration of a performance-enhancing optimizer written in C. To facilitate experimental integration into the QueryArtisan framework, we harnessed the advanced GPT-3.5-turbo-16k-0613 and GPT-4.0-turbo models developed by OpenAI [34], as well as Meta's leading open-source LLM, the meta-llama-3-70b-instruct model [28]. The MySQL version employed for our experiments was 5.7.26. Our experimentation environment was based on an Ubuntu Linux 20.04 system, equipped with 8 NVIDIA RTX A5000 GPUs and 384 GB of memory.

Table 2: Results of query efficiency. t1: average query code execution time, t2: code generation time (Sec. 3), t3: code optimization time (Sec. 4), T: end-to-end time.

Methods	Bird				UniBench			
MySQL	$t1/s$	$t2/s$	$t3/s$	T/s	$t1/s$	$t2/s$	$t3/s$	T/s
MySQL	2.90	0	0	2.90	2.86	0	0	2.86
GPT-3.5	0.65	0.93	0	1.58	0.91	1.04	0	1.95
Ours	0.46	4.08	0.008	4.55	0.65	4.01	0.007	4.67
Ours$_{!OD}$	0.46	4.63	0.008	5.10	0.65	4.64	0.007	5.30

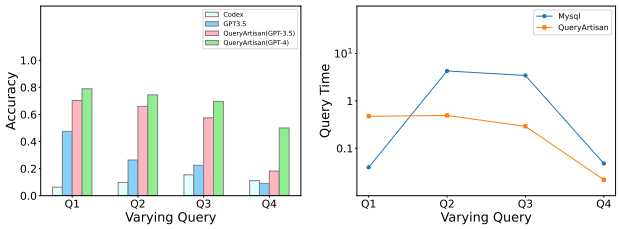
5.2 Experimental Results

5.2.1 Query Accuracy. In the query answering task, we conduct a comparative analysis of QueryArtisan's query accuracy against various baseline methods. The objective of this experiment is to assess whether the code generated by our approach effectively captures query semantics.

As the baseline methods only support relational data and not multimodal data, we had to transform multimodal data into relational data through the ETL process for the baseline approaches. In contrast, QueryArtisan completely circumvents the ETL process.

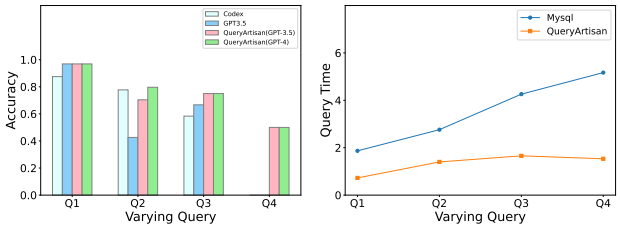
As illustrated in Table 1, QueryArtisan consistently outperforms all comparative methods in terms of accuracy across all datasets when utilizing GPT-3.5. Notably, the term "Champions" pertains to methods that currently hold top positions on public dataset leaderboards, leveraging GPT-4. It is evident from our results that QueryArtisan, when employing GPT-4 for queries, significantly surpasses the accuracy of all other methods.

Within the Spider dataset, both MiniSeek and DAIL-SQL (4.0) currently hold top positions. However, it's noteworthy that QueryArtisan significantly outperforms them by a substantial margin.



(a) Query accuracy (b) Query efficiency

Figure 8: Impact of query in Bird dataset.



(a) Query accuracy (b) Query efficiency

Figure 9: Impact of query in DLBench dataset.

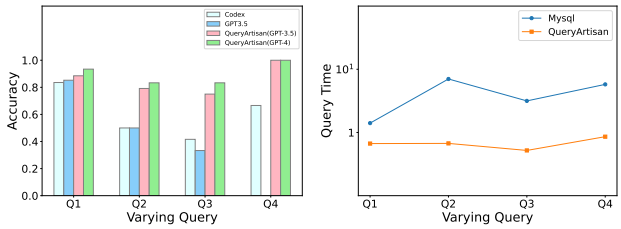
In the WikiSQL dataset, QueryArtisan continues to maintain its superiority over other approaches. Even the current leading approach (SeaD [45], denoted as "Champion") attains a precision rate of 92.9%, akin to QueryArtisan when utilizing GPT-4.0. However, it's important to highlight that a notable portion of natural language queries in the dataset contain inaccuracies. Upon the removal of these errors, our system achieves an accuracy of 96.7%, further widening the performance gap.

In the Bird dataset, QueryArtisan's knowledge injection feature proved to be a significant advantage, delivering an accuracy rate that surpasses the closest competitor, DAIL-SQL, by an impressive margin of 20.5%. It's worth noting that a substantial accuracy gain is achieved compared to the vanilla GPT-3.5. This result highlights the impracticality of directly employing Large Language Models (LLM) for code generation in complex tasks. ACT-SQL, which employs pre-trained models for chained thought prompts, struggles on the Bird dataset due to its limitations in deciphering complex logic.

In the UniBench and DLBench multimodal datasets, we put our system to the test with queries that involved complex group aggregations and multi-table joins on JSON, graph and text data. Remarkably, even in the face of this complexity, our system consistently delivered exceptional performance. Notably, it's crucial to highlight that while other methods required an ETL process to transform data, QueryArtisan, without altering the data format, maintained a significant advantage. Furthermore, it's worth mentioning that DALI-SQL and ACT-SQL necessitate training on specific datasets, and it's evident that their methods exhibit poor transferability.

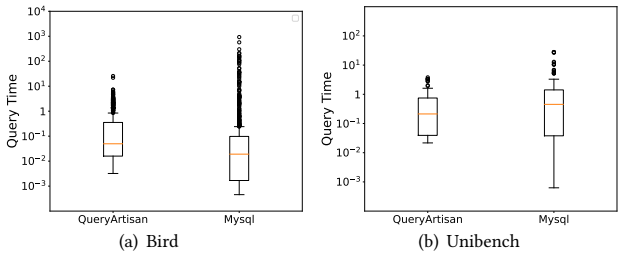
We found that GPT-4.0 exhibited greater advantages in datasets with higher complexity, and outperformed all GPT-3.5 powered approaches. When QueryArtisan was equipped with GPT-4, it surpassed the top-ranked methods (referred to as "Champions") in accuracy on the Spider[2], WikiSQL[3], and Bird[4] leaderboards. In the

[2]https://yale-lily.github.io/spider
[3]https://github.com/salesforce/WikiSQL
[4]https://bird-bench.github.io/



(a) Query accuracy (b) Query efficiency

Figure 10: Impact of query in UniBench dataset.



(a) Bird (b) Unibench

Figure 11: Distribution of query times.

Spider dataset, QueryArtisan outperforms the top-ranked methods by 4.6%. In the Bird dataset, the system surpasses the current leading method for Bird by 11%.

We further explored the impact of query complexity by varying the number of tables in join queries. As depicted in Figure 8, Figure 9, and Figure 10, QueryArtisan consistently demonstrates superior performance across queries of varying complexity. Note that Codex and GPT-3.5 results are unavailable for some Q4 category queries due to incorrect answers.

5.2.2 Query Efficiency. To assess the execution efficiency of the code generated by QueryArtisan, we compared its computational time to that of queries executed on a MySQL database. This runtime measurement encompasses all aspects, including file reading, data operations, and file writing. The results of this comparison are presented in Table 2.

An intriguing observation emerges: our code surpasses even the MySQL database, a well-optimized commercial database system. This can be attributed to a two-fold rationale. Firstly, MySQL, being a comprehensive database system, integrates numerous features that are not essential for processing a specific query. In contrast, the generated code exclusively contains the necessary processing logic, omitting extraneous modules such as concurrent control, logging, and transactions. Secondly, our definition of operators offers greater flexibility than traditional relational algebras. A single operator generated by our system can encompass the processing of multiple relational operators. For instance, a single filter operator in our case can handle all predicate processing. In such instances, we can produce code that is more cache-friendly and compiler-friendly, contributing to the enhanced efficiency observed.

As shown in Table 2, we analyzed end-to-end time comparisons. Due to the network latency and OpenAI's rate limits, our method doesn't excel in overall time; however, this latency is acceptable in data analysis scenarios. As illustrated in Figure 11, it's evident that QueryArtisan exhibits minimal query time variance, whereas

Table 3: Ablation test on query accuracy.

Methods	Spider	Wikisql	Bird	UniBench	DLBench
QueryArtisan$_{!DO}$	65.9	66.3	30.9	68.1	67.2
QueryArtisan$_{!CO}$	86.5	92.9	65.0	85.2	78.1
QueryArtisan$_{!CV}$	75.6	73.9	45.6	74.1	70.0
QueryArtisan	**86.9**	**92.8**	**65.2**	**85.0**	**79.2**

Table 4: Ablation test on query efficiency.

Methods	Spider	DLBench
QueryArtisan$_{!DO}$	2.66ms	233.6ms
QueryArtisan$_{!CO}$	2.59ms	224.9ms
QueryArtisan$_{!CV}$	2.49ms	155.8ms
QueryArtisan	**2.47ms**	**144.2ms**

the query time of MySQL displays substantial fluctuations. This distinction is due to QueryArtisan's query time primarily involving reading local files, with minimal time spent on data operations.

5.3 Ablation Studies

In this section, we perform ablation experiments over some facets of QueryArtisan in order to better understand their roles.

Effect of Pre-defined Operators. We begin by highlighting the significance of pre-defined operators, which entail knowledge injection into the system. Table 3 shows a significant drop in query accuracy when the operator module is removed, indicated as QueryArtisan$_{!DO}$, with the LLM generating code as a blackbox.

Table 4 presents a summary of the query execution times for each method, noting that the query response time is the sum of data file read/write time and query execution time. It demonstrates a notable reduction in query efficiency for QueryArtisan$_{!DO}$ when contrasted with QueryArtisan. It is due to the operator module providing a richer context, enabling more accurate and efficient code generation.

Effect of Code Optimization. The focus of code optimization (CO) is on achieving performance-optimized query analysis graphs. The results demonstrate that QueryArtisan$_{!CO}$ and QueryArtisan are equivalent in terms of accuracy performance. This module's primary objective is the enhancement of query efficiency, rather than query accuracy. Table 4 reveals a 3.0-fold improvement in query efficiency for QueryArtisan on the Spider dataset compared to QueryArtisan$_{!CO}$, despite similar average query times. Although optimization is effective, detecting significant differences in average query time is challenging due to their brevity. The Spider dataset's relatively small data volume and simple queries account for this observation. In the DLBench multimodal dataset, which features more complex queries, optimizing impact poses greater challenges. QueryArtisan exceeds QueryArtisan$_{!CO}$ by a factor of 1.9 in query efficiency, and this is also reflected in a significant reduction in average query time. This demonstrates that the method is effective even in complex query and data scenarios.

Effect of Code Verification. The necessity of the code verification module is further discussed through a comparison between QueryArtisan and QueryArtisan$_{!CV}$. QueryArtisan significantly outperforms QueryArtisan$_{!CV}$ in query accuracy, underscoring the importance of code verification in LLM-based code generation. The focus of code verification (CV) is the verification of the correctness of query analysis graphs. The results show that QueryArtisan$_{!CV}$ and QueryArtisan exhibit similar efficiency performance. This module's primary objective is enhancing query accuracy, rather than improving query efficiency.

6 RELATED WORK

Our work intersects with several lines of research, and we provide a brief overview of these areas.

Data Discovery over Data Lake. Recent research endeavors [2–5, 9, 26, 35] have primarily focused on automating discovery tasks within data lakes that exclusively contain relational tables. These approaches leverage various similarity signals, including exact value overlap [55], schema similarity [31], approximate hash sketches [16], ontology matches [15], transformations [1], and embeddings [8, 38]. Some systems, such as Aurum [13], D3L [4], and TURL [7], combine these signals to varying degrees.

Query on Data Lake. Supporting query processing within data lakes containing heterogeneous data sources is an inherently challenging problem that often necessitates significant human intervention during the data integration and query rewriting phases [6, 49, 50]. Pioneering research and systems [13, 14, 18, 21, 30] have made notable strides in streamlining the integration and querying of these diverse data repositories. Additionally, [48] explores using reinforcement learning to integrate schemas and answer queries across multiple data sources.

Automatic Programming. Recent works, such as those by Ni et al. [32], Feng et al. [12], and Li et al. [25], have made significant strides in advancing the automatic programming field [23, 43]. This approach has been effectively implemented in systems like ReAct [46] and Reflexion [37], which harness these prompts to construct logical flows and action plans using large language models (LLMs). Furthermore, ToolFormer [36] showcases the ability to learn interactions with external tools through simple APIs, thereby expanding the scope and utility of automatic programming.

7 CONCLUSION

In this paper, we introduce QueryArtisan, which uses Language Models (LLMs) to generate ad-hoc data manipulation code for data lakes. QueryArtisan streamlines the data processing pipeline by eliminating the need for cumbersome ETL processes and instead generates code in an ad-hoc manner. It enhances code quality with a repository of data processing operators and metadata, optimizes performance with a plan optimizer, and ensures reliability through code verification. Experimental results substantiate the superiority of QueryArtisan over state-of-the-art systems when handling complex data sources, underscoring its effectiveness and robustness.

ACKNOWLEDGMENT

This work was supported by the Key Research Program of Zhejiang Province (Grant No. 2023C01037), the Zhejiang Provincial Natural Science Foundation (Grant No. LZ21F020007) and the Fundamental Research Funds for the Central Universities (226-2024-00049, 226-2024-00145). This work was supported by Ant Group through CCF-Ant Research Fund.

REFERENCES

[1] Ziawasch Abedjan, John Morcos, Ihab F. Ilyas, Mourad Ouzzani, Paolo Papotti, and Michael Stonebraker. 2016. DataXFormer: A robust transformation discovery system. In *ICDE*. IEEE Computer Society, 1134–1145.

[2] Naser Ahmadi, Hansjorg Sand, and Paolo Papotti. 2022. Unsupervised Matching of Data and Text. In *ICDE*. IEEE, 1058–1070.

[3] Angelos-Christos G. Anadiotis, Oana Balalau, Catarina Conceição, et al. 2022. Graph integration of structured, semistructured and unstructured data for data journalism. *Inf. Syst.* 104 (2022), 101846.

[4] Alex Bogatu, Alvaro A. A. Fernandes, Norman W. Paton, and Nikolaos Konstantinou. 2020. Dataset Discovery in Data Lakes. In *ICDE*. IEEE, 709–720.

[5] Ursin Brunner and Kurt Stockinger. 2019. Entity Matching on Unstructured Data: An Active Learning Approach. In *SDS*. IEEE, 97–102.

[6] Dong Deng, Raul Castro Fernandez, Ziawasch Abedjan, Sibo Wang, Michael Stonebraker, et al. 2017. The Data Civilizer System. In *CIDR*.

[7] Xiang Deng, Huan Sun, Alyssa Lees, You Wu, and Cong Yu. 2022. TURL: Table Understanding through Representation Learning. *SIGMOD* (2022).

[8] Yuyang Dong, Kunihiro Takeoka, Chuan Xiao, and Masafumi Oyamada. 2021. Efficient Joinable Table Discovery in Data Lakes: A High-Dimensional Similarity-Based Approach. In *ICDE*. IEEE, 456–467.

[9] Mohamed Y Eltabakh, Mayuresh Kunjir, Ahmed Elmagarmid, and Mohammad Shahmeer Ahmad. 2023. Cross Modal Data Discovery over Structured and Unstructured Data Lakes. *arXiv preprint arXiv:2306.00932* (2023).

[10] Mohamed Y. Eltabakh, Mayuresh Kunjir, Ahmed K. Elmagarmid, and Mohammad Shahmeer Ahmad. 2023. Cross Modal Data Discovery over Structured and Unstructured Data Lakes. *Proc. VLDB Endow.* 16, 11 (2023), 3377–3390.

[11] Grace Fan, Jin Wang, Yuliang Li, et al. 2023. Semantics-aware Dataset Discovery from Data Lakes with Contextualized Column-based Representation Learning. *Proc. VLDB Endow.* 16, 7 (2023), 1726–1739.

[12] Zhangyin Feng, Daya Guo, Duyu Tang, et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. 1536–1547.

[13] Raul Castro Fernandez, Ziawasch Abedjan, Famien Koko, Gina Yuan, Samuel Madden, and Michael Stonebraker. 2018. Aurum: A Data Discovery System. In *ICDE*. IEEE Computer Society, 1001–1012.

[14] Raul Castro Fernandez and Samuel Madden. 2019. Termite: a system for tunneling through heterogeneous data. In *aiDM@SIGMOD*. ACM, 7:1–7:8.

[15] Raul Castro Fernandez, Essam Mansour, Abdulhakim Ali Qahtan, et al. 2018. Seeping Semantics: Linking Datasets Using Word Embeddings for Data Discovery. In *ICDE*. IEEE Computer Society, 989–1000.

[16] Raul Castro Fernandez, Jisoo Min, Demitri Nava, and Samuel Madden. 2019. Lazo: A Cardinality-Based Method for Coupled Estimation of Jaccard Similarity and Containment. In *ICDE*. IEEE, 1190–1201.

[17] Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, et al. 2023. Text-to-sql empowered by large language models: A benchmark evaluation. *arXiv preprint arXiv:2308.15363* (2023).

[18] Rihan Hai, Sandra Geisler, and Christoph Quix. 2016. Constance: An Intelligent Data Lake System. In *SIGMOD*. ACM, 2097–2100.

[19] Alon Y. Halevy, Flip Korn, Natalya Fridman Noy, Christopher Olston, Neoklis Polyzotis, Sudip Roy, and Steven Euijong Whang. 2016. Goods: Organizing Google's Datasets. In *SIGMOD*. ACM, 795–806.

[20] Felix Halim, Panagiotis Karras, and Roland H. C. Yap. 2009. Fast and effective histogram construction. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management, CIKM 2009, Hong Kong, China, November 2-6, 2009*, David Wai-Lok Cheung, Il-Yeol Song, Wesley W. Chu, Xiaohua Hu, and Jimmy Lin (Eds.). ACM, 1167–1176.

[21] Aamod Khatiwada, Roee Shraga, Wolfgang Gatterbauer, and Renée J. Miller. 2022. Integrating Data Lake Tables. *Proc. VLDB Endow.* 16, 4 (2022), 932–945.

[22] Jey Han Lau and Timothy Baldwin. 2016. An Empirical Evaluation of doc2vec with Practical Insights into Document Embedding Generation. In *Rep4NLP@ACL*.

[23] Guohao Li, Hasan Abed Al Kader Hammoud, et al. 2023. Camel: Communicative agents for" mind" exploration of large scale language model society. *arXiv preprint arXiv:2303.17760* (2023).

[24] Jinyang Li, Binyuan Hui, Ge Qu, Binhua Li, et al. 2023. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *arXiv preprint arXiv:2305.03111* (2023).

[25] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, et al. 2022. Competition-level code generation with alphacode. *Science* 378, 6624 (2022), 1092–1097.

[26] Lizi Liao, Le Hong Long, Zheng Zhang, Minlie Huang, and Tat-Seng Chua. 2021. MMConv: An Environment for Multimodal Conversational Search across Multiple Domains. In *SIGIR*. ACM, 675–684.

[27] Qian Liu, Bei Chen, Jiaqi Guo, Morteza Ziyadi, et al. 2022. TAPEX: Table Pre-training via Learning a Neural SQL Executor. In *ICLR*.

[28] Meta. [n.d.]. Meta Llama 3. https://huggingface.co/meta-llama/Meta-Llama-3-70B-Instruct.

[29] Yev Meyer, Marjan Emadi, Dhruv Nathawani, et al. 2024. *Synthetic-Text-To-SQL: A synthetic dataset for training language models to generate SQL queries from natural language prompts.*

[30] Fatemeh Nargesian, Erkang Zhu, Renée J. Miller, Ken Q. Pu, and Patricia C. Arocena. 2019. Data Lake Management: Challenges and Opportunities. *Proc. VLDB Endow.* 12, 12 (2019), 1986–1989.

[31] Fatemeh Nargesian, Erkang Zhu, Ken Q. Pu, and Renée J. Miller. 2018. Table Union Search on Open Data. *Proc. VLDB Endow.* 11, 7 (2018), 813–825.

[32] Ansong Ni, Srini Iyer, Dragomir Radev, et al. 2023. Lever: Learning to verify language-to-code generation with execution. In *International Conference on Machine Learning*. PMLR, 26106–26128.

[33] Natasha F. Noy. 2020. When the Web is your Data Lake: Creating a Search Engine for Datasets on the Web. In *SIGMOD*. ACM, 801.

[34] OpenAI. [n.d.]. OpenAI API. https://api.openai.com/.

[35] Marnith Peng, Jose Luis Beltran, and Ravigopal Vennelakanti. 2020. Entity Matching from Unstructured and Dissimilar Data Collections: Semantic and Content Distribution Approach. In *IMMS*. ACM, 29–33.

[36] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Lomeli, et al. 2023. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761* (2023).

[37] Noah Shinn, Beck Labash, et al. 2023. Reflexion: an autonomous agent with dynamic memory and self-reflection. *arXiv preprint arXiv:2303.11366* (2023).

[38] Sahaana Suri, Ihab F. Ilyas, et al. 2021. Ember: No-Code Context Enrichment via Similarity-Based Keyless Joins. *Proc. VLDB Endow.* 15, 3 (2021), 699–712.

[39] Immanuel Trummer. 2022. CodexDB: Synthesizing Code for Query Processing from Natural Language Instructions using GPT-3 Codex. *Proc. VLDB Endow.* 15, 11 (2022), 2921–2928.

[40] Bailin Wang, Richard Shin, Xiaodong Liu, et al. 2020. RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers. In *ACL*. ACL, 7567–7578.

[41] Lihan Wang, Bowen Qin, Binyuan Hui, et al. 2022. Proton: Probing Schema Linking Information from Pre-trained Language Models for Text-to-SQL Parsing. In *KDD*. ACM, 1889–1898.

[42] Jason Wei, Xuezhi Wang, Dale Schuurmans, et al. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *NeurIPS*.

[43] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *NeurIPS* 35 (2022), 24824–24837.

[44] Dong Xin, Jiawei Han, and Kevin Chen-Chuan Chang. 2007. Progressive and selective merge: computing top-k with ad-hoc ranking functions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*. ACM, 103–114.

[45] Kuan Xu, Yongbo Wang, Yongliang Wang, et al. 2022. SeaD: End-to-end Text-to-SQL Generation with Schema-aware Denoising. In *NAACL*. ACL, 1845–1853.

[46] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629* (2022).

[47] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, et al. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *arXiv preprint arXiv:1809.08887* (2018).

[48] Qin Yuan, Ye Yuan, Zhenyu Wen, et al. 2023. An effective framework for enhancing query answering in a heterogeneous data lake. In *SIGIR*. 770–780.

[49] Ye Yuan, Guoren Wang, Lei Chen, and Haixun Wang. 2013. Efficient Keyword Search on Uncertain Graph Data. *TKDE* 25, 12 (2013), 2767–2779.

[50] Ye Yuan, Guoren Wang, Haixun Wang, and Lei Chen. 2011. Efficient Subgraph Search over Large Uncertain Graphs. *Proc. VLDB Endow.* 4, 11 (2011), 876–886.

[51] Chao Zhang. 2018. Parameter Curation and Data Generation for Benchmarking Multi-model Queries. In *VLDB (CEUR Workshop Proceedings)*, Vol. 2175.

[52] Hanchong Zhang, Ruisheng Cao, Lu Chen, Hongshen Xu, and Kai Yu. 2023. ACT-SQL: In-Context Learning for Text-to-SQL with Automatically-Generated Chain-of-Thought. In *EMNLP*.

[53] Meihui Zhang, Marios Hadjieleftheriou, Beng Chin Ooi, Cecilia M. Procopiuc, and Divesh Srivastava. 2010. On Multi-Column Foreign Key Discovery. *Proc. VLDB Endow.* 3, 1 (2010), 805–814.

[54] Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103* (2017).

[55] Erkang Zhu, Dong Deng, Fatemeh Nargesian, and Renée J. Miller. 2019. JOSIE: Overlap Set Similarity Search for Finding Joinable Tables in Data Lakes. In *SIGMOD*. ACM, 847–864.