A Demonstration of QueryArtisan: Real-Time Data Lake Analysis via Dynamically Generated Data Manipulation Code

Wenhao Liu
Zhejiang University
wenhao.liu@zju.edu.cn

Xiu Tang*
Zhejiang University
tangxiu@zju.edu.cn

Sai Wu
Zhejiang University
wusai@zju.edu.cn

Chang Yao
Zhejiang University
changy@zju.edu.cn

Gongsheng Yuan
Zhejiang University
ygs@zju.edu.cn

Gang Chen
Zhejiang University
cg@zju.edu.cn

ABSTRACT

Querying and analyzing data in data lakes requires substantial manual intervention, including numerous data preprocessing steps, and often demands complex domain expertise. However, the advent of Large Language Models (LLMs) has introduced a promising solution to these challenges by providing a unified framework for interpreting the heterogeneous datasets within data lakes. In this paper, we demonstrate QueryArtisan, a novel LLM-powered analytical system tailored for data lakes. It enables users to issue complex queries in natural language without the need for domain-specific expertise. The system automatically executes user-submitted queries and performs data processing and analysis based on the query results. QueryArtisan extends beyond traditional ETL (Extract, Transform, Load) processes by generating just-in-time code customized for dataset-specific tasks. A suite of heterogeneous operators is developed to process data across various modalities. In addition, a cost-based query optimization mechanism is integrated to improve the efficiency of the generated code. Furthermore, QueryArtisan can dynamically instantiate multiple agents in response to user-defined analytical requirements to perform further in-depth analysis of the retrieved data.

PVLDB Reference Format:
Wenhao Liu, Xiu Tang, Sai Wu, Chang Yao, Gongsheng Yuan, and Gang Chen. A Demonstration of QueryArtisan: Real-Time Data Lake Analysis via Dynamically Generated Data Manipulation Code . PVLDB, 18(12): 5263 - 5266, 2025.
doi:10.14778/3750601.3750647

PVLDB Artifact Availability:
The source code and data have been made available at https://github.com/skyrise-l/QueryArtisan_demo.

1 INTRODUCTION

A data lake serves as a comprehensive storage hub, specifically designed to accommodate, manage, and safeguard vast quantities of data, regardless of its structure [3, 4]. Whether the data is meticulously organized, semi-structured, or entirely unstructured, a data lake is capable of preserving it in its original formats [3]. Moreover, it supports the processing of a wide variety of data types without size constraints.

However, providing data lake services requires complex engineering efforts. In conventional data lake systems, two distinct methodologies for processing are typically employed. In both strategies, the definition of a unified mediated schema and the establishment of mapping relationships between each source schema and the central schema are crucial.

In the query-driven approach, queries targeting the mediated schema are translated into equivalent queries for each source dataset, utilizing these mapping relationships [7]. These queries are subsequently processed individually by the source systems. The data lake's role involves amalgamating the results into a cohesive output.

Conversely, within the data-driven model, all incoming data is transformed to conform to the mediated schema and is subsequently loaded into a centralized host system in the data lake [1]. In this model, the host system assumes responsibility for processing all queries, thus offering a centralized solution for data handling.

Existing data lake tools typically do not provide built-in support for analyzing the data retrieved through queries. As a result, once users obtain the queried data, they must resort to external tools to perform further analysis. This disconnect disrupts workflows, increases complexity, and adds to the user's burden of managing both queries and separate analytical code, making the analysis process more fragmented, labor-intensive, and costly.

As shown in Figure 1, the implementation of a data lake system presents three primary challenges:

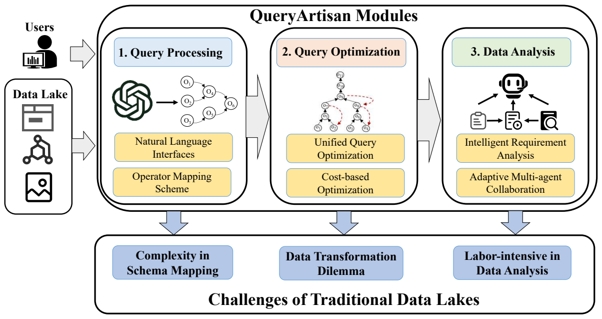

QueryArtisan Modules
Users
Data Lake
1. Query Processing
Natural Language Interfaces
Operator Mapping Scheme
2. Query Optimization
Unified Query Optimization
Cost-based Optimization
3. Data Analysis
Intelligent Requirement Analysis
Adaptive Multi-agent Collaboration
Complexity in Schema Mapping
Data Transformation Dilemma
Labor-intensive in Data Analysis
Challenges of Traditional Data Lakes

Figure 1: Comparison of data analysis limitations in traditional data lake systems and the QueryArtisan system.

*Xiu Tang is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 18, No. 12 ISSN 2150-8097.
doi:10.14778/3750601.3750647

- **Complexity in Schema Mapping.** Establishing schema mappings, even with semi-automatic tools, remains complex and resource-intensive, involving considerable manual intervention and overhead.
- **Data Transformation Dilemma.** The data-driven approach demands intensive data transformation and loading efforts, while the query-driven approach complicates query rewriting and result aggregation, compromising accuracy and manageability.
- **Labor-intensive Data Analysis.** Existing data lake systems lack integrated analytical capabilities, requiring users to manually bridge the gap between data retrieval and subsequent analysis.

To address these challenges, we introduce QueryArtisan, a real-time data lake analysis system built upon our prior work [5]. The system integrates all our proposed techniques to enable accurate and efficient real-time analytics over data lakes. As shown in Figure 1, QueryArtisan comprises three modules:

- **Query Processing Module:** Utilizes Natural Language Interfaces to interpret user queries and employs heterogeneous operators to translate these queries into executable sub-tasks, effectively resolving the complexity of schema mapping.
- **Query Optimization Module:** Applies a unified, cost-based query optimization approach to holistically optimize the generated Query Processing Graph. By globally optimizing query execution plans, this module effectively reduces and even eliminates complex data transformation processes, thus addressing the Data Transformation Dilemma.
- **Data Analysis Module:** Adopts adaptive multi-agent collaboration to automatically integrate analytical tasks with query execution, bridging the gap between data retrieval and analysis, and simplifying labor-intensive workflows.

Based on QueryArtisan, users can execute complex real-time analytics on data lakes using natural language queries, significantly reducing both human resource requirements and dependency on expert knowledge.

2 SYSTEM OVERVIEW

In this section, we provide an overview of QueryArtisan's architecture. Figure 2 illustrates the three key components of QueryArtisan:

- **User-Friendly Interface:** QueryArtisan provides three intuitive interfaces for users: Query Chat, Query Process, and Final Report. These interfaces enhance transparency by presenting intermediate processes from analysis modules.
- **QueryArtisan Modules:** This component forms the core of the QueryArtisan system and consists of three primary sub-modules: Operator-based Query Graph Generation, Query Optimization and Code Correction, and Multi-agent-based In-depth Analysis.
- **Data Lake:** This module is responsible for data storage and organizational management within the system. It comprises an offline operator database for query graph construction, heterogeneous data sources and schema definitions, and a distributed processing framework for code execution.

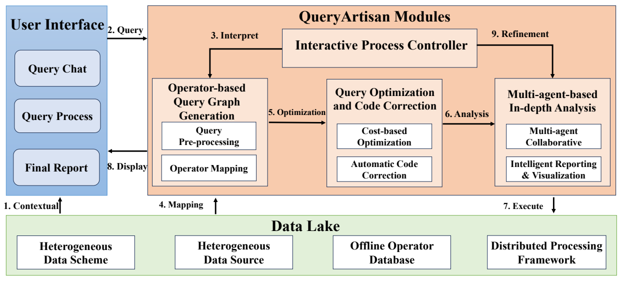

User Interface
Query Chat
Query Process
Final Report
2. Query
8. Display
1. Contextual
QueryArtisan Modules
3. Interpret
Interactive Process Controller
9. Refinement
Operator-based Query Graph Generation
Query Pre-processing
Operator Mapping
5. Optimization
Query Optimization and Code Correction
Cost-based Optimization
Automatic Code Correction
6. Analysis
Multi-agent-based In-depth Analysis
Multi-agent Collaborative
Intelligent Reporting & Visualization
4. Mapping
7. Execute
Data Lake
Heterogeneous Data Scheme
Heterogeneous Data Source
Offline Operator Database
Distributed Processing Framework

Figure 2: The architecture of QueryArtisan system.

Operator-based query graph generation. Given a query, QueryArtisan first retrieves relevant metadata and data samples from the data lake. Leveraging this contextual information, the LLM reformulates the original query into an enriched query. Subsequently, the LLM decomposes the enriched query into a set of subtasks, consisting of sub-query tasks and sub-analysis tasks.

To effectively interpret each sub-query task, we introduce a set of abstract data-processing operators, where each operator represents a fundamental data operation and is annotated by detailed semantics to enhance interpretability and applicability. We then instruct the LLM to utilize these operators to capture the data processing relationships for each sub-query task, constructing a processing graph consisting of operators and their dependencies. By executing the operators in the processing graph according to their dependencies, we effectively achieve the desired task.

Query optimization and code correction. Given the processing graph, there remains significant potential to optimize the initially generated code. We introduce two key adjustments, known as shuffle and collapse. The shuffle approach entails altering the order of operators within the processing graph. For example, QueryArtisan seeks to optimize query plans by strategically positioning filter operators later in the execution sequence. In contrast, the collapse approach consolidates multiple operators and generates code for them simultaneously, thereby facilitating the merging of operators such as multiple filter stages. Furthermore, to evaluate how significantly these two adjustments reduce processing costs and increase efficiency, QueryArtisan employs the cost model-based plan optimization, similar to those used in traditional database systems. This component transforms the query analysis graph into a performance-optimized execution plan.

Subsequently, QueryArtisan generates executable code from the optimized graph. Recognizing that LLM-based code generation may introduce inaccuracies, QueryArtisan validates each operator's code according to correction rules, using a dedicated auditing agent to detect and repair such issues. If irreparable errors are encountered, the system initiates an error-handling mechanism involving backtracking to a stable prior state and retrying the execution.

Multi-agent-based in-depth analysis For sub-analysis tasks, QueryArtisan utilizes multi-agent collaboration to handle each task. Initially, a task interpreter agent is employed to parse the task requirements and to ensure that prerequisite queries are completed before proceeding. The system then dynamically generates domain-specific agent prompts based on the task context, enhancing overall scalability and applicability. These prompts are used to invoke an execution agent that performs the task, while a supervision agent oversees the entire process to ensure correctness. Finally, the visualization results from the analysis trigger an Analysis Agent for producing a comprehensive report that integrates insights within the query context.

3 IMPLEMENTATION DETAILS

Query pre-processing. During pre-processing, we leverage an LLM to integrate three key components (meta-data, samples, and NL query) into an enriched query. Specifically, we perform unbiased sampling on the target dataset to collect representative meta-data and key samples. The meta-data includes "Dataset info", covering general details such as format, size, and descriptions, as well as "Schema info" and "Data info", obtained through sampling and scanning the dataset. Additionally, essential features like primary-foreign key relationships and histograms are derived using conventional algorithms [2, 8].

Due to prompt length constraints of the LLM, we prioritize selecting representative samples by randomly sampling the largest fact tables and retrieving related tuples from dimension tables via primary-foreign key relationships. Intentionally introducing tuples with noise, such as null or invalid values, is also part of our strategy.

Finally, the meta-data and selected samples provide contextual information for refining and rewriting the original NL query. The LLM then generates a concise and accurate enriched query, incorporating essential extracted details.

Offline operator database. To diminish reliance on the LLM and minimize unnecessary calls, an offline operator database has been developed. Specifically, a collection of typical NL queries has been gathered from the WikiSQL [9] and Spider [6] datasets.

We have observed that many data analytic queries incorporate similar sub-tasks, culminating in a unified processing graph. Consequently, to reduce dynamic requests to the LLM, an offline key-value database has been developed. This database contains pairs representing typical queries derived from the WikiSQL and Spider datasets. To promote generalizability and eliminate dataset-specific biases, detailed tables, columns, and variable names within these queries have been substituted with generic placeholders.

Cost-based optimization. Queries can be represented by multiple equivalent implementations, with execution efficiency closely tied to factors such as data distribution, size, and selectivity. To address this, QueryArtisan develops a cost model to evaluate the expected costs associated with different query plan implementations, based on the characteristics of both the data and the query. The total cost of a query plan is derived by summing the operational costs of individual operators and the costs incurred when transferring data between successive operators. This cost model enables QueryArtisan to optimize query plans, focusing on reducing execution costs while maintaining accuracy and consistency in the results.

All possible pairs of operators are systematically evaluated through recursive testing, with the potential advantages of altering their sequence considered based on the cost model. Ultimately, a plan is arrived at which, though not necessarily optimal, effectively mitigates significant.

Automatic code correction. QueryArtisan employs a specialized algorithm to detect and correct errors in code generated by large language models (LLMs). For a given query analysis graph, the algorithm validates the mapping of each operator in the code using predefined detection schemes. When errors, such as missing read operations or incorrect references to table or column names, are identified, predefined correction rules are applied. If the error cannot be fixed, the system invokes a correction agent or reverts to a previous step to reanalyze the task or regenerate the code.

Multi-agent task interpretation, execution, and supervision. For sub-analysis tasks, QueryArtisan assigns a Task Interpreter Agent to each task, which is responsible for interpreting task-specific requirements, identifying the necessary prerequisite queries, and defining relevant constraints based on user inputs and system configurations, including algorithms, visualization tools, model parameters and other elements. Once these steps are completed and all prerequisite queries have been verified, the system generates a task-specific prompt using the gathered information. The prompt then triggers the Analysis Executor Agent to generate the corresponding analysis code. The entire process is overseen by the Process Supervisor Agent, which monitors for errors, reports them to the relevant agents for correction. Once verification is complete, it executes the code to obtain the final output.

Distributed processing framework. QueryArtisan employs a distributed framework that decomposes queries and analyses into fine-grained subtasks executed in parallel across multiple nodes, markedly boosting throughput and slashing response times for the large-scale, complex workloads common in data-lake environments. By continuously tracking node health, efficiently aggregating results, and scaling on demand, it delivers reliable, real-time insights from data lakes.

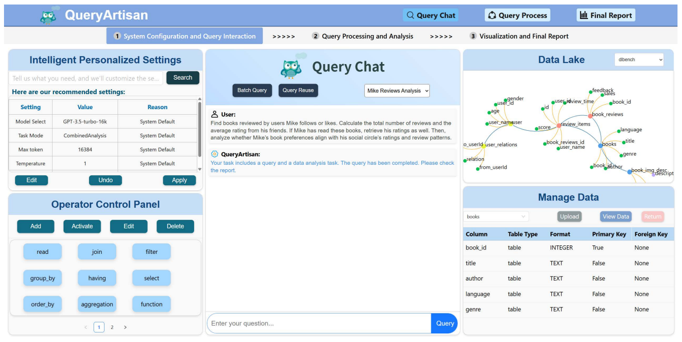
4 DEMONSTRATION SCENARIOS

Figure 3 presents a screenshot of the web-based visualization interface of QueryArtisan, enabling users to interact with queries and observe the data analysis process through the following steps:

Step 1 (Query Chat): Figure 3(a) shows the query interaction interface of the system. The *Intelligent Personalized Settings* panel enables users to customize various system parameters, including selecting the LLM model type and specifying algorithms for specific tasks. Users can either input their specific requirements to let the system automatically generate the appropriate configuration or manually modify and refine the system-recommended settings to better align with their analytical objectives.

In the *Operator Control Panel*, users can manage standard and custom operators used in queries. Standard operators are essential for ensuring the proper execution of queries. These operators can be modified but cannot be deleted. Custom operators can be added by specifying their functions and providing corresponding code examples. The LLM attempts to incorporate these custom operators into subsequent data analysis tasks. For operators that are not needed temporarily, users can set them to an inactive state.

Once the configuration is complete, users can design queries based on the data source information displayed in the *Data Lake panel.* Users also have the option to switch between data sources via the top-right corner. They can click on any node in the displayed graph to query detailed information about that data node. Additionally, users can manage the selected data source in the *Manage Data panel.*



Here are our recommended settings:

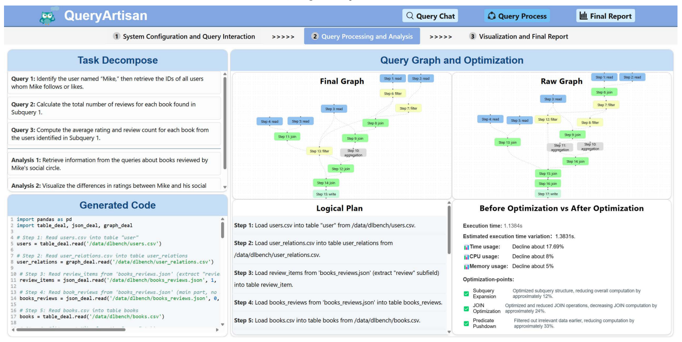
Setting	Value	Reason
Model Select	GPT-3.5-turbo-16k	System Default
Task Mode	CombinedAnalysis	System Default
Max token	16384	System Default
Temperature	1	System Default

User:
Find books reviewed by users Mike follows or likes. Calculate the total number of reviews and the average rating from his friends. If Mike has read these books, retrieve his ratings as well. Then, analyze whether Mike's book preferences align with his social circle's ratings and review patterns.

QueryArtisan:
Your task includes a query and a data analysis task. The query has been completed. Please check the report.

Column	Table Type	Format	Primary Key	Foreign Key
book_id	table	INTEGER	True	None
title	table	TEXT	False	None
author	table	TEXT	False	None
language	table	TEXT	False	None
genre	table	TEXT	False	None

(a) Query Chat


QueryArtisan
Query Chat
Query Process
Final Report
1 System Configuration and Query Interaction
>>>>>
2 Query Processing and Analysis
>>>>>
3 Visualization and Final Report
Task Decompose
Generated Code
Query Graph and Optimization
Final Graph
Raw Graph
Logical Plan
Before Optimization vs After Optimization

Query 1: Identify the user named "Mike," then retrieve the IDs of all users whom Mike follows or likes.

Query 2: Calculate the total number of reviews for each book found in Subquery 1.

Query 3: Compute the average rating and review count for each book from the users identified in Subquery 1.

Analysis 1: Retrieve information from the queries about books reviewed by Mike's social circle.

Analysis 2: Visualize the differences in ratings between Mike and his social

```
import pandas as pd
import table_deal, json_deal, graph_deal

# Step 1: Read users.csv into table "user"
users = table_deal.read('/data/dlbench/users.csv')

# Step 2: Read user_relations.csv into table user_relations
user_relations = graph_deal.read('/data/dlbench/user_relations.csv')

 # Step 3: Read review_items from 'books_reviews.json' (extract "revie
 review_items = json_deal.read('/data/dlbench/books_reviews.json', 1,

 # Step 4: Read book_reviews from 'books_reviews.json' (main part, no
 books_reviews = json_deal.read('/data/dlbench/books_reviews.json', 0,

 # Step 5: Read books.csv into table books
 books = table_deal.read('/data/dlbench/books.csv')

```

Step 1: Load users.csv into table "user" from /data/dlbench/users.csv.

Step 2: Load user_relations.csv into table user_relations from /data/dlbench/user_relations.csv.

Step 3: Load review_items from 'books_reviews.json' (extract "review" subfield) into table review_item.

Step 4: Load books_reviews from 'books_reviews.json' into table books_reviews.

Step 5: Load books.csv into table books from /data/dlbench/books.csv.

Execution time: 1.1384s

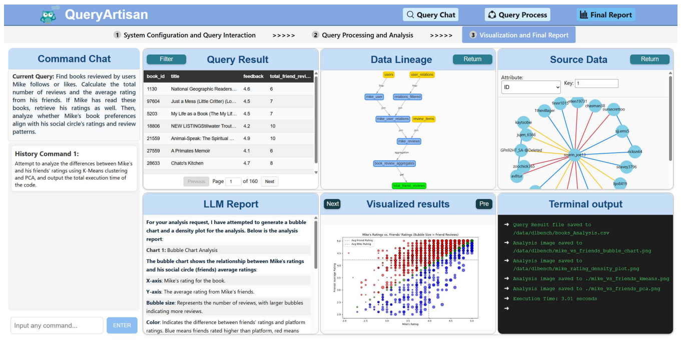
Estimated execution time variation: 1.3831s.

Time usage:	Decline about 17.69%
CPU usage:	Decline about 8%
Memory usage:	Decline about 5%

Optimization-points:

✓	Subquery Expansion	Optimized subquery structure, reducing overall computation by approximately 12%.
✓	JOIN Optimization	Optimized and reduced JOIN operations, decreasing JOIN computation by approximately 24%.
✓	Predicate Pushdown	Filtered out irrelevant data earlier, reducing computation by approximately 33%.

(b) Query Process



Current Query: Find books reviewed by users Mike follows or likes. Calculate the total number of reviews and the average rating from his friends. If Mike has read these books, retrieve his ratings as well. Then, analyze whether Mike's book preferences align with his social circle's ratings and review patterns.

History Command 1:
Attempt to analyze the differences between Mike's and his friends' ratings using K-Means clustering and PCA, and output the total execution time of the code.

book_id	title	feedback	total_friend_revi...
1130	National Geographic Readers...	4.6	6
97604	Just a Mess (Little Critter) (Lo...	4.5	7
5203	My Life as a Book (The My Lif...	4.5	7
18806	NEW LISTINGStillwater Trout...	4.2	10
21559	Animal-Speak: The Spiritual ...	4.9	10
27559	A Primates Memoir	4.1	6
28633	Chato's Kitchen	4.7	8

Previous Page 1 of 160 Next

For your analysis request, I have attempted to generate a bubble chart and a density plot for the analysis. Below is the analysis report:

Chart 1: Bubble Chart Analysis

The bubble chart shows the relationship between Mike's ratings and his social circle (friends) average ratings:

X-axis: Mike's rating for the book.

Y-axis: The average rating from Mike's friends.

Bubble size: Represents the number of reviews, with larger bubbles indicating more reviews.

Color: Indicates the difference between friends' ratings and platform ratings. Blue means friends rated higher than platform, red means

```
➜ Query Result file saved to
  /data/dlbench/books_Analysis.csv
➜ Analysis image saved to
  /data/dlbench/mike_vs_friends_bubble_chart.png
➜ Analysis image saved to
  /data/dlbench/mike_rating_density_plot.png
➜ Analysis image saved to ./mike_vs_friends_kmeans.png
➜ Analysis image saved to ./mike_vs_friends_pca.png
➜ Execution Time: 3.01 seconds
➜
```

(c) Final Report

Figure 3: A screenshot of QueryArtisan with an example data analysis task on the data lake.

The *Query Chat panel* enables users to initiate interactive query and analysis sessions, with the system maintaining a history of previous queries to support continuation from earlier results. Users can also request additional information related to the queries, such as receiving suggestions for query optimization. The Batch Query button provides an interface where users can input a large number of queries simultaneously. After execution, the system generates a comprehensive report that users can download. The Query Reuse button enables users to efficiently search for and reuse historical queries, further enhancing productivity and reducing repetitive workload.

Step 2 (Query Process): As shown in Figure 3(b), the second interface presents detailed intermediate results generated during the query analysis process. The *Task Decompose panel* illustrates how the system automatically decomposes the user's query and analytical tasks into sub-tasks. The *Generated Code panel* displays the final executable code corresponding to the analyzed query.

The *Query Graph and Optimization panel* visualizes the intermediate optimization stages of the query. Within this panel, the Original Graph shows the initial query graph before optimization, while the Final Graph represents the optimized query graph. Additionally, the Logical Plan provides a natural language explanation of the query's logical execution plan. The Before Optimization vs. After Optimization part compares key metrics before and after optimization, highlighting improvements in query execution time, CPU utilization, and other critical system resources. This part also details specific optimization points, offering users deeper insights into the effectiveness of optimization strategies.

Step 3 (Final Report): As shown in Figure 3(c), the third interface displays the final results of the query. Notably, the *Command Chat panel* allows users to execute optimization commands for the current query, such as requesting execution time statistics, modifying the data analysis algorithm or visualization method. The *Query Result panel* shows the final query results. Users can select any column to view its corresponding Data Lineage, which is visualized in the *Data Lineage panel.* Furthermore, clicking the starting point of the data source within the *Data Lineage panel* displays the corresponding source content in the *Source Data panel*, thereby facilitating in-depth analysis and traceability.

The *LLM Report panel* displays a comprehensive summary of the entire query and analysis process, including an analysis of the visualized results in the *Visualized Results panel.* The *Terminal Output panel* shows the system's code execution output, including intermediate results, providing support for debugging.

ACKNOWLEDGMENT

This paper is supported by the National Regional Innovation and Development Joint Fund (No. U24A20254) and Zhejiang Provincial Natural Science Foundation of China under Grant (No.LQN25F020009). The authors are supported by Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security.

REFERENCES

[1] Raul Castro Fernandez and Samuel Madden. 2019. Termite: a system for tunneling through heterogeneous data. In *aiDM@SIGMOD*. ACM, 7:1–7:8.
[2] Felix Halim, Panagiotis Karras, and Roland H. C. Yap. 2009. Fast and effective histogram construction. In *CIKM*. ACM, 1167–1176.
[3] 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.
[4] Natasha F. Noy. 2020. When the Web is your Data Lake: Creating a Search Engine for Datasets on the Web. In *SIGMOD*. ACM, 801.
[5] Xiu Tang, Wenhao Liu, Sai Wu, Chang Yao, Gongsheng Yuan, Shanshan Ying, and Gang Chen. 2024. QueryArtisan: Generating Data Manipulation Codes for Ad-hoc Analysis in Data Lakes. *Proc. VLDB Endow.* 18, 2 (2024), 108–116.
[6] Tao Yu, Rui Zhang, Kai Yang, and et al. 2018. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. In *EMNLP*. 3911–3921.
[7] Qin Yuan, Ye Yuan, Zhenyu Wen, He Wang, and Shiyuan Tang. 2023. An effective framework for enhancing query answering in a heterogeneous data lake. In *SIGIR*. 770–780.
[8] 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. https://doi.org/10.14778/1920841.1920944
[9] Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103* (2017).