NAT-NL2GQL: A Novel Multi-Agent Framework for Translating Natural Language to Graph Query Language

Anonymous ACL submission

Abstract

There has been increasing interest in using Large Language Models (LLMs) for translating natural language into graph query language (NL2GQL). While progress has been made, current approaches often fail to fully exploit the potential of LLMs to autonomously plan and collaborate on complex NL2GQL tasks. To address this gap, we propose NAT-NL2GQL, an innovative multi-agent framework for NL2GQL translation. The framework consists of three complementary agents: the Preprocessor agent, the Generator agent, and the Refiner agent. The Preprocessor agent handles tasks such as entity recognition, query rewriting, and schema extraction. The Generator agent, a fine-tuned LLM trained on NL-GQL data, generates corresponding GQL statements based on queries and their related schemas. The Refiner agent refines the GQL or context using error feedback from the GQL execution results. In the absence of high-quality open-source NL2GQL datasets based on nGQL syntax, we developed StockGQL, a Chinese dataset derived from a Chinese financial market graph database, which will be made publicly available to support future research. Experiments on the StockGQL and SpCQL datasets demonstrate that our approach significantly outperforms baseline methods, underscoring its potential to drive advancements in NL2GQL research.

1 Introduction

Graph data is gaining prominence in modern data science for its ability to reveal complex relationships, enhance information connectivity, and support intelligent decision-making. It is particularly valuable in fields such as finance, healthcare, and social networks, where managing highly connected and structurally complex data is crucial (Zhao et al., 2022a; Sui et al., 2024). Graph data requires specialized graph databases (DBs) for efficient storage and processing (Pavliš, 2024). Popular graph DBs, including Neo4j, NebulaGraph, and JanusGraph, offer distinct features but share similar query graph languages (GQLs) (e.g., Cypher, nGQL, and Gremlin), enabling users to analyze data efficiently.

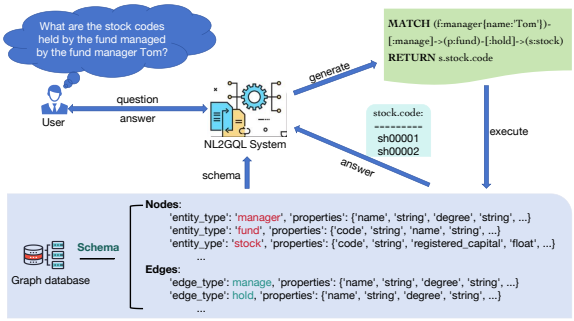

Figure 1: The demonstration of the NL2GQL task transforming the user's natural language into a graph query language that can be executed on a NebulaGraph.

Despite the growing importance of graph data, ordinary users often struggle with graph DBs due to their complex operations and lack of technical expertise, limiting their adoption in real-world applications (Guo et al., 2022). Additionally, the complex syntax of GQL creates further obstacles, especially for users attempting to translate natural language (NL) into GQL, a task known as NL2GQL. These challenges make NL2GQL a particularly demanding problem (Liang et al., 2024b; Zhou et al., 2024). Figure 1 shows an NL2GQL example for NebulaGraph, highlighting key components like natural language understanding, DB schema comprehension, and GQL generation. This emphasizes the need for a system that automates NL2GQL, simplifying graph data queries and analysis to promote wider adoption.

NL2GQL is a specialized application of the Seq2Seq task. Modern methods have moved from template-based approaches to generative models, offering more flexibility and accuracy in handling complex queries. The study (Guo et al., 2022) first applied a Seq2Seq framework to NL2GQL and

introduced the SpCQL dataset. The work (Zhao et al., 2023) developed a SQL2Cypher algorithm for mapping SQL to Cypher, though the approach is limited by the differences between GQL and SQL. The paper (Tran et al., 2024) proposes the CoBGT model, combining BERT, GraphSAGE (Hamilton et al., 2017), and Transformer for key-value extraction, relation-property prediction, and Cypher query generation.

LLMs have revolutionized performance in NLP tasks, with applications extending to DB research (Zhu et al., 2024; Ren et al., 2024; Peng et al., 2024; Zhou et al., 2023; Lao et al., 2023), where they bridge natural language and structured query languages for more intuitive DB interactions. Research on LLMs for graph DBs, especially NL2GQL, is growing. Tao et al. (2024) uses heuristic prediction and LLM revision, showing effectiveness in some domains. Zhou et al. (2024) combines smaller models for ranking and rewriting with larger models for the final NL-to-GQL transformation. Liang et al. (2024b) proposes constructing an NL2GQL dataset using domain-specific graph DBs and tokenization to enhance accuracy.

LLM-based methods exhibit some effectiveness in solving NL2GQL tasks, but their streamlined approach carries a major challenge—error accumulation. Incorrect extraction of the related schema can lead to flawed GQL generation. For example, as shown in Figure 1, the correct related schema for the query should include the nodes "manager," "fund," and "stock," and edges "manage" and "hold." If the extracted schema omits "stock" and "hold," the generated GQL, such as *MATCH (f:manager{name:'Tom'})-[:manage]->(p:fund) RETURN s.fund.code*, will produce incorrect results.

In this study, we propose **NAT-NL2GQL**, a multi-agent framework for translating NL2GQL, as shown in Figure 3. The framework consists of three agents: the *Preprocessor*, *Generator*, and *Refiner*. The Preprocessor agent handles data preprocessing tasks, such as extracting values from the graph DB, performing named entity recognition (NER), rewriting user queries, linking paths, and extracting related schemas. The Generator agent, a fine-tuned LLM trained on the NL-GQL dataset, generates the GQL based on the context and user queries. The Refiner agent refines the GQL or context using error information from GQL execution results. These agents interact iteratively for up to three rounds. Given that different graph DBs have varying GQL syntaxes, we propose a general framework to handle these differences. To address the lack of high-quality NL2GQL datasets, we developed StockGQL, derived from a financial market NebulaGraph DB. We evaluated our framework using the StockGQL and SpCQL datasets (Guo et al., 2022), showing significant improvements over baseline methods in NL2GQL accuracy. Ablation experiments further confirm the importance of each module in enhancing task performance.

Key Contributions. To summarize, this paper makes the following contributions:

- First, to alleviate error accumulation inherent in streamlined methods, we designed a collaborative and iterative multi-agent framework to tackle the NL2GQL task.
- Second, based on a Chinese financial market NebulaGraph DB, we constructed the StockGQL dataset, which can serve as a testbed for future NL2GQL research.
- Third, our proposed method surpassed the baseline methods on both StockGQL and SpCQL datasets, which denotes the new state-of-the-art NL2GQL results in both general and specific domains.

2 Related works

NL2GQL is a typical NLP task that has emerged with the widespread adoption of graph data and can be classified as a seq-to-seq task (Guo et al., 2022; Zhao et al., 2023). Its primary function is to convert users' NL questions into GQL queries that can be executed on a graph DB. This task involves user queries understanding, graph schema linking, and GQL generation (Liang et al., 2024b; Zhou et al., 2024). Early efforts focused on using hand-crafted rules to translate NL into GQL (Zhao et al., 2022b). Modern approaches primarily incorporate state-of-the-art (SOTA) models to optimize performance. We categorize LLM-based NL2GQL methods into two types: PLMs-based methods and LLMs-based Methods.

PLMs-based methods. Fine-tuning PLMs within a sequence-to-sequence framework is one of the most widely used approaches for generative tasks in NLP. Initially, Guo et al. (2022) constructed a text-to-Cypher dataset and designed three baselines: seq2seq, seq2seq + attention (Dong and Lapata, 2016), and seq2seq + copying (Gu et al., 2016).

However, the results on the two evaluation metrics, EX and EX, were not satisfactory. Reference (Tran et al., 2024) employs the BERT (Kenton and Toutanova, 2019) model for key-value extraction and uses GraphSAGE (Hamilton et al., 2017) to analyze the relational properties of the DB. These features are then fed into a transformer to generate the Cypher query. Their proposed small Text-to-Cypher dataset outperforms seq2seq models like T5 (Raffel et al., 2020) and GPT-2 (Radford et al., 2019). Reference (Liang et al., 2024a) introduces the KEI-CQL framework, a heuristic-like approach that utilizes pre-trained language models to extract semantic features from natural language queries and populate predefined slots in Cypher query sketches, effectively addressing the NL2GQL challenge.

LLM-based methods. Leveraging the powerful understanding and generation capabilities of LLMs to tackle the NL2GQL task has become a recent research hotspot. Reference (Tao et al., 2024) attempts to combine heuristic methods with LLM-based approaches. They first extract GQL clauses using heuristic rules, then concatenate these clauses to form a complete GQL, and finally use an LLM for refinement. Reference (Zhou et al., 2024) deconstructs the NL2GQL task into individual subtasks, using a combination of smaller models and LLMs for each stage. Specifically, smaller models are employed during the initial ranking and rewriting phases, while an LLM is used for the final generation step. In contrast, Liang et al. (2024b) aligns LLMs with domain-specific graph DBs to address NL2GQL tasks within those DBs. They construct an NL2GQL dataset based on a domain-specific graph DB, then fine-tune an LLM with this dataset, enabling the LLM to effectively tackle NL2GQL tasks in the specific domain. However, streaming-based task decomposition methods often struggle with error accumulation. In response to the observed challenge, we introduce the NAT-NL2GQL framework. Detailed comparisons with similar tasks (e.g., Text2SQL, KBQA) are provided in Appendix 8.9.

3 PRELIMINARIES

NL2GQL Task Definition. The input consists of an NL query $\mathcal{X}$ and a graph DB $\mathcal{G}$, which is represented as $\mathcal{G} = \{(s, r, o) \mid s, o \in \mathcal{V}, r \in \mathcal{E}\}$. Here, $\mathcal{V}$ and $\mathcal{E}$ denote the sets of vertices and edges, respectively. The objective is to generate a correct GQL query based on the provided question and the graph DB.

LLM-based NL2GQL Systems. The in-context learning approach enables LLMs to generate accurate answers by incorporating a few examples into the prompt. This can be formalized as follows:

$$\hat{\mathcal{Q}} = LLM_{ICL}(\mathcal{I}, \mathcal{D}, \text{NL})$$

Here, $\mathcal{I}$ represents the task description, $\mathcal{D}$ consists of demonstrations from annotated datasets, and NL refers to the input question.

4 StockGQL Dataset Build

We use the self-instruct method (Wang et al., 2022) to create StockGQL, based on a real-world financial stock NebulaGraph DB, with privacy processing applied to named entities. Figure 2 illustrates our approach. Next, we will provide a detailed explanation of each step's functionality.

Schema Extraction. As shown in Step 1 of Figure 2, we extract the schema from the graph DB, identifying the nodes, edges, and their attributes. This forms the foundation for creating a structured representation of the graph, enabling subsequent query generation and processing.

Subschema Extraction. A subschema is a subset of the graph's schema, containing only partial information. Step 2 of Figure 2 involves extracting a subschema by applying specific rules to identify all possible path combinations, from 0-hop to 6-hop paths.

Data Generation. Step 3 in Figure 2 shows the data generation module, detailed in Algorithm 1. Using the ICL method, we sample K data points from the pool, which initially contains 16 manually crafted examples. These are used to create masked NL-GQL pairs, where entity names are replaced with placeholders in both the query and the GQL. An example is shown below:

Masked query : What is the code of stock [s]?

Masked GQL : MATCH (s:stock{name:'[s]'})

RETURN s.stock.code

We use the placeholder `[s]` to represent stock entity names in both the natural language query and the corresponding GQL.

We generate each subschema for m times to cover as many attributes of all entities as possible. Using the self-instruct approach, the process

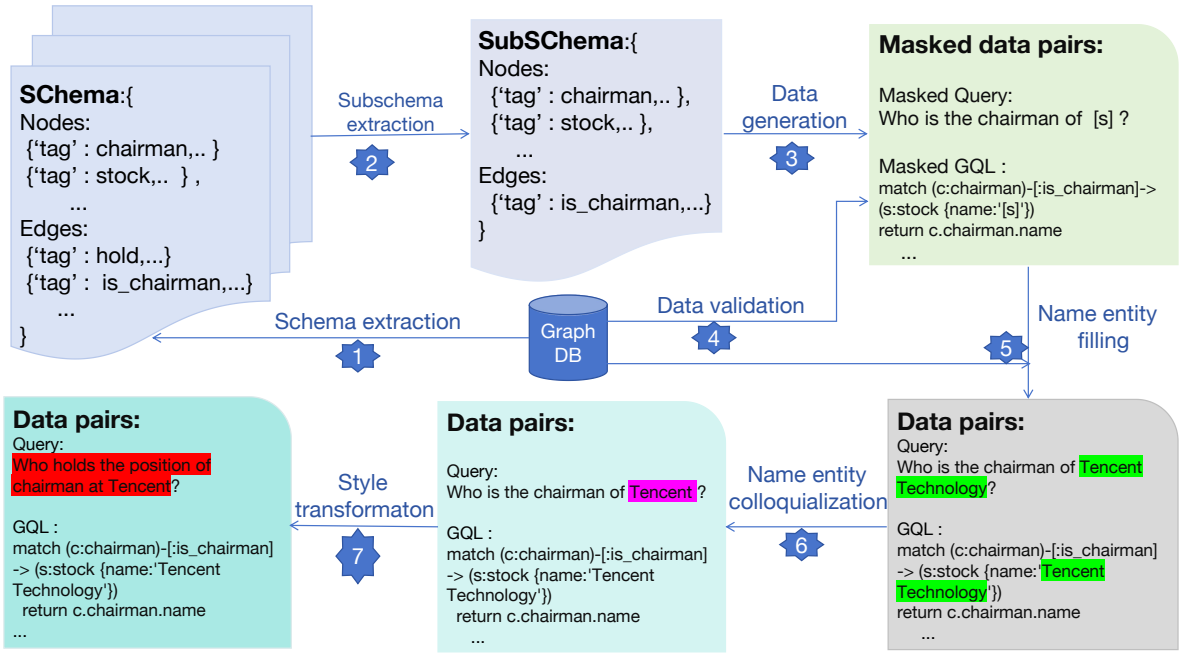

Figure 2: This is the flowchart for constructing the dataset, where the parts of the data that have changed relative to the previous step in Step 5, Step 6, and Step 7 are highlighted. The GQL is based on the nGQL syntax.

Algorithm 1: Masked NL-GQL Data Pairs Generation

Input: A set of *subschemas*; Data pool $\mathcal{D}$; Number of demonstrations K; Iterations number m; Task description I

foreach s *in subschemas* **do**
  for $i = 1$ **to** m **do**
    Sample K items from Data pool;
    Build demonstrations $\mathcal{E}$ using the sampled items;
    Generate Masked NL-GQL Data Pairs;
    $d\_list \leftarrow LLM_{ICL}(I, \mathcal{E}, s)$;
    Add d_list to $\mathcal{D}$;
return $\mathcal{D}$

iterates until all subschemas have been covered, at which point it will terminate.

Data Validation. This step filters out erroneous data where NL and GQL are inconsistent. We follow the approach outlined in (Liang et al., 2024b), using an entity-filled, CoT-based GQL2NL method to generate NL' from GQL. The data is then filtered based on low embedding similarity between NL and NL'. As a result, we obtain a large number of high-quality masked NL-GQL data pairs.

Name Entity Filling. This step involves filling in the previously masked data by extracting relevant named entities from the graph DB based on the mask type. For example, `[s]` corresponds to stock entity names.

Name Entity Colloquialization. In this step, we randomly select a dataset with named entities and manually rewrite the entities in both the NL and GQL as abbreviated forms. This simulates real-world scenarios where users commonly use the abbreviation of the entity names. For example, in Step 6 of Figure 2, the colloquialization of Tencent Technology has been changed to Tencent.

Style Transformation. The final step is to enhance query diversity by rephrasing the NL, adjusting its wording and phrasing to suit different contexts or user needs while preserving the original meaning. This includes tailoring the NL to different linguistic preferences, tones, or levels of formality, ensuring clarity and relevance across various scenarios without changing its intended purpose.

It is worth noting that our method is generalizable to both general and domain-specific areas, enabling the generation of NL2GQL datasets in various languages, based on different Graph DBs, across a wide range of domains. We have constructed the Chinese StockGQL dataset. A statistical analysis of the data, shown in Table 1, reveals that 63% of the queries involve more than 2 hops, with 26% involving more than 3 hops. The dataset includes 12 types of nodes, 13 types of edges, and 62 types of properties. StockGQL is an NL2GQL dataset based on the nGQL syntax, designed for complex multi-hop, multi-type

queries. We hope its open-source release will advance NL2GQL research and model development. The dataset format is shown in Appendix 8.1.

Dataset	0-hop	1-hop	2-hop	3-hop	4-hop	Others
Train (4572)	528	1167	1666	942	253	16
Dev (655)	70	163	207	155	54	6
Test (1229)	172	378	414	194	62	9

Table 1: Statistics on the number of hops contained in StockGQL dataset.

5 Method

In this section, we explain the NAT-NL2GQL workflow. As shown in Figure 3, it consists of three agents: *Preprocessor*, *Generator*, and *Refiner*. The agents work together iteratively to complete the task. Next, we will provide a detailed description of the specific functions of each module.

5.1 Preprocessor Agent

As highlighted in (Liang et al., 2024b; Zhou et al., 2024), extracting the NL-relevant schema from the full graph DB schema offers three main benefits: reducing schema size to avoid context length issues, eliminating irrelevant noise to improve GQL accuracy, and speeding up GQL generation. The Preprocessor agent extracts relevant schemas, aligns named entities in the query with those in the DB, and rewrites the query as needed, including tasks like NER, entity alignment, schema revision, linking completion, and query rewriting.

LLM-based NER. Extracting named entities from NL is crucial for identifying the related schema. Previous studies have shown that LLMs can effectively recognize named entities (Xie et al., 2023; Xiao et al., 2024; Xu et al., 2023). Building on this, the Preprocessor agent uses LLM-based NER to extract entities from the query, helping pinpoint relevant schema parts. This reduces the schema search space and ensures accurate mappings between query entities and graph DB counterparts for precise GQL generation. We use ChatGPT-4 for entity extraction, following the prompt structure in Appendix 8.2.

Entity Alignment. After extracting named entities, we align them with corresponding entity names in the graph DB. This ensures accurate mapping to relevant nodes or edges, enabling precise query generation. We first build a dictionary $\mathcal{D}$, where each key is an entity type and its value is a list of names. We then compare extracted entity names with those in the dictionary. If an exact match is found, the entity type name is assigned. For unmatched entities, we use locality-sensitive hashing (LSH) (Datar et al., 2004) to select the most similar entity name. This process is formulated as:

$$\hat{\mathcal{D}} = LSH(\mathcal{Z}, \mathcal{D}, \gamma)$$

$$\hat{d} = \arg\max_{d_i \in \hat{\mathcal{D}}} \text{Cosine}(Emb(\mathcal{X}), Emb(d_i))$$

Here, $\mathcal{Z}$ denotes the extracted named entities from the NL using LLM-based NER, $\mathcal{D}$ is the entity dictionary from the graph DB, and $\hat{\mathcal{D}}$ consists of entities retrieved using LSH similarity to $\mathcal{X}$ with threshold γ. *Emb*($\mathcal{X}$) represents the embedding of $\mathcal{X}$ encoded via all-MiniLM-L12-v1, and $\hat{d}$ denotes the entity names extracted based on cosine similarity to $\mathcal{X}$. After alignment, we obtain the entity names with their corresponding types.

Linking Completion. While multiple entity types are extracted, they may not necessarily form a connected subgraph. To handle queries that require reasoning across different entity types, we link related entities. We begin by extracting entity and attribute names from the graph database schema, matching them with those in the query, and eliminating duplicates. To obtain a relevant subgraph, we use the search algorithm from (Liang et al., 2024b) to identify the smallest subgraph that includes all the extracted entities. Finally, we apply the algorithm in Appendix 8.4 to complete the intermediate entities and relationships, resulting in a candidate related schema.

Related Schema Revision. Due to various factors, such as potential errors in NER, entities with identical names, inconsistent attribute naming in the graph database , the candidate related schema may include redundant nodes and edges. We apply further filtering using ChatGPT-4 to retain only the most relevant entities and relationships. The specific prompt is provided in Appendix 8.3, and experimental results show that this significantly improves accuracy.

Question Rewriting. Queries may include colloquial terms or abbreviations, which need to be aligned with graph DB entities for accurate GQL generation. After aligning named entities, mismatches are replaced with corresponding entity names. Some entities may not match exactly, but the related schema revision step filters out irrelevant ones. This process rewrites NL entity names for consistency with the graph DB. For example, the original query:

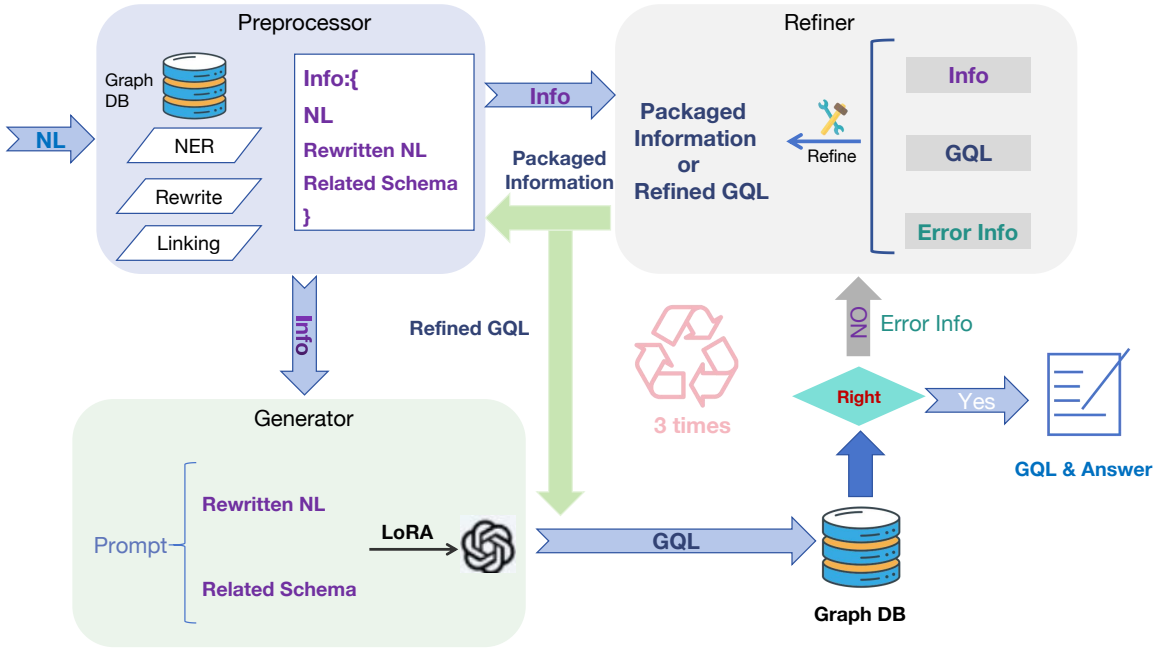

Figure 3: Our NAT-NL2GQL framework consists of three synergistic agents: the Preprocessor agent, the Generator agent, and the Refiner agent. The entire process follows a cyclic and iterative flow, with the three agents collaboratively handling data preprocessing, GQL generation, and GQL refinement.

梁*dong* 是董事长的股票关联的产业下游的产业有哪些?

(What are the downstream industries related to the industries associated with the chairman Liang Dong's stock?)

can be revised to :

梁东是董事长的股票关联的产业下游的产业有哪些?

5.2 Generator Agent

Once data pre-processing is complete, we generate the GQL using the obtained information. To optimize memory usage while maintaining performance, we apply Parameter Efficient Fine-Tuning (PEFT) techniques, such as LoRA (Hu et al., 2021), which fine-tunes only a small subset of parameters. This approach is ideal for GQL generation, balancing model size and computational efficiency. We fine-tune the selected base LLMs using LoRA. As shown in Figure 3, the fine-tuning prompt combines the original NL, rewritten NL, and related schema. During training, the golden related schema from labeled GQL is used, while during inference, the Preprocessor agent predicts the related schema.

5.3 Refiner Agent

Many studies show that rewriting queries with syntax errors improves query accuracy (Pourreza et al., 2024a; Talaei et al., 2024; Zhou et al., 2024). However, these methods often rely on LLMs to correct syntax errors, which usually involve only minor modifications to the original query and may not address more complex issues. Additionally, error information typically highlights only the first error encountered, making it unsuitable for queries with multiple errors. Most importantly, if the related schema or query from earlier steps is incorrect, fixing the GQL syntax alone may not resolve the issue, as it may still not align with the original query. In such cases, the error information should prompt a review of the auxiliary information from earlier steps.

As shown in the refine prompt in Appendix 8.5, our approach differs by using the question, preprocessed data, GQL, and error information to determine whether the related schema is correct. If the schema is correct, we directly rewrite the GQL. If it's wrong, this indicates an error in the previous data preprocessing step. In that case, we package the information and send it to the Preprocessor agent, treating both the GQL and error details as historical data for re-execution. We also set an iteration limit, terminating the process if the GQL remains incorrect after several attempts. The Refiner agent then decides whether to modify the GQL or save the historical data to restart the process, as shown in Figure 4.

Method	Backbones	StockGQL		SpCQL	
		EM(%)	EX(%)	EM(%)	EX(%)
ICL(K=4)	GLM-4-9B-Chat	30.43	28.23	7.03	8.22
	Qwen2.5-14B-Instruct	31.16	27.18	7.87	8.92
	LLaMA-3.1-8B-Instruct	24.74	23.35	7.42	8.21
	LLaMA-3.2-3B-Instruct	8.62	8.46	6.03	7.27
	ChatGPT-3.5-Turbo	29.29	29.21	7.37	7.62
	ChatGPT-4o	40.68	38.08	9.22	10.26
Fine-Tuning	GLM-4-9B-Chat	80.96	81.77	53.86	52.12
	Qwen2.5-14B-Instruct	82.67	83.65	53.91	51.57
	LLaMA-3.1-8B-Instruct	82.75	83.91	54.16	50.57
	LLaMA-3.2-3B-Instruct	82.51	82.99	49.18	48.83
Others' approach	SpCQL	1.4	1.8	2.3	2.6
	Align-NL2GQL	82.99	84.06	54.21	52.86
	R^3-NL2GQL	81.86	84.13	55.06	53.06
Ours	Qwen2.5-14B-Instruct & ChatGPT-4o	**85.44** ↑2.45	**86.25** ↑2.12	**59.99** ↑4.93	**58.69** ↑5.63

Table 2: Comparison between our method and the baseline method. The bold numbers indicate the best results . The red upward arrow denotes an improvement, and the red number in parentheses indicates the exact improvement value compared to the best baseline method.

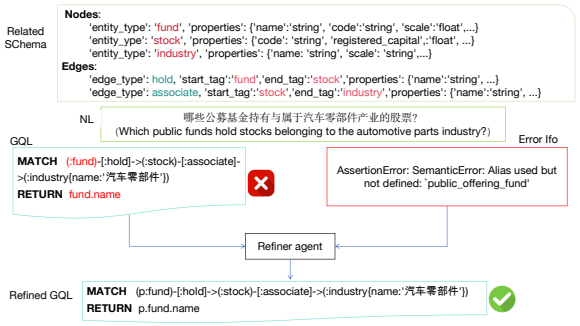

Figure 4: A refined example.

6 Experiment Results

6.1 Experimental Setup

Datasets. We conducted experiments on the StockGQL and SpCQL (Guo et al., 2022) datasets. The SpCQL dataset uses **Cypher** GQL, while StockGQL follows **nGQL** syntax.

Baseline Methods. To validate and compare the effectiveness of our method, as shown in Table 2. we selected three types of baseline methods: ICL approaches, fine-tuning approaches, and a method from previous related work. For the ICL approaches, the prompt format we designed is illustrated in Appendix 8.6. In the fine-tuning approaches, the complete schema is incorporated into the input.

Evaluation Metrics. We follow the approach in (Guo et al., 2022; Liang et al., 2024b), using exact-set-match accuracy (**EM**) and execution accuracy (**EX**)to evaluate our method. EM measures the consistency of individual components, segmented by keywords, between the predicted query and its corresponding ground truth, while EX assesses the consistency of the execution results in the DB.

Implementation Details. Experiments were conducted on an A800 GPU, using GLM-4-9B-Chat, Qwen2.5-14B-Instruct, LLaMA-3.1-8B-Instruct, LLaMA-3.2-3B-Instruct, ChatGPT-3.5-Turbo, and ChatGPT-4 as the LLMs. The number of demonstrations k was set to 4, and the LSH threshold γ was set to 0.6.

6.2 Main Results

An analysis of the results in Table 2 leads to the following conclusions: First, our approach outperforms all baselines. On the StockGQL dataset, it surpasses the best baseline by 2.45% on the EM metric and 2.12% on the EX metric. On the SpCQL dataset, it improves by 4.93% on EM and 5.63% on EX. Second, the ICL method performs poorly for NL2GQL, likely due to the lack of high-quality GQL corpora during model training. A possible solution is to gather high-quality GQL data to retrain base LLMs. Third, the SpCQL dataset is more challenging than StockGQL due to more entity name mismatches, which explains the larger improvement seen in SpCQL, where our entity alignment approach is particularly effective. Lastly, while LLaMA-3.1-8B-Instruct outperforms LLaMA-3.2-3B-Instruct with ICL, their performances are nearly identical after fine-tuning. This suggests smaller models are less suited for ICL but more effective with fine-tuning when enough data is available.

6.3 Further Analysis

Breakdown Analysis. We analyzed the model's performance on StockGQL by hop count, as shown in Figure 5. The accuracy generally decreases as the hop number increases, indicating higher difficulty. Lower accuracy for 0-hop queries is due to their flexible, conversational nature, while multi-hop queries become more challenging for the model, highlighting the complexity of handling intricate dependencies and the need for advanced techniques to manage these queries.

Impact of the Related Schema. As emphasized by previous works (Liang et al., 2024b; Zhou et al., 2024) and Table 4, related schema extraction is crucial. We compared our method with R^3-NL2GQL and Align-NL2GQL on StockGQL. The results in Table 3 show that our method achieves the highest accuracy. While it performs well, improving the related schema extraction further remains a promising direction for future work.

Performance with Various Base LLMs. The Preprocessor and Refiner agents use ChatGPT-4o, while the Generator is fine-tuned with LoRA on Qwen2.5-14B-Instruct. We experimented with various base LLMs for each agent and compared the results, summarized in Table 5. The findings indicate that the Generator agent is more robust to base LLM choices after fine-tuning, while the Preprocessor and Refiner agents, using unmodified base LLMs, are more sensitive to model choice, significantly affecting overall performance. Further analysis is provided in Appendix 8.7 and Appendix 8.8.

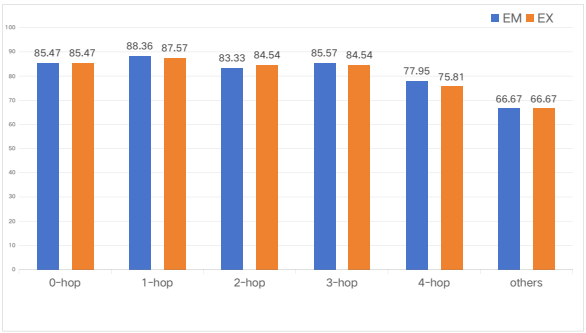

Figure 5: The EM and EX accuracy of our method on StockGQL, statistically by hop count.

6.4 Ablation Study

The ablation study in Table 6 shows that removing any component reduces performance. Replacing the fine-tuned generator with ChatGPT-4o's ICL method causes the largest drop. The "Without Regeneration" setting highlights the Refiner's role in detecting schema errors and initiating re-extraction, leading to some improvement.

Method	Acc(%)
Ours	**86.00**
Ours(w/o filtering)	72.50
Align-NL2GQL	80.15
R^3-NL2GQL	81.04

Table 3: Comparison of accuracy across different methods for extracting related schemas on StockGQL.

Method	EM(%)	EX(%)
Error Related Schema	20.34	21.16
All Schema	82.67	84.05
Golden Related Schema	91.46	92.84

Table 4: The table shows the impact of the related schema on GQL accuracy for StockGQL.

Agent	LLM	EM(%)	EX(%)
Preprocessor	Qwen2.5-14B-Instruct	77.95	79.01
	ChatGPT-3.5-Turbo	80.88	79.98
	ChatGPT-4o	**85.44**	**86.25**
Generator	GLM-4-9B-Chat	85.03	85.84
	LLaMA-3.1-8B-Instruct	85.35	86.09
	LLaMA-3.2-3B-Instruct	85.19	85.92
	Qwen2.5-14B-Instruct	**85.44**	**86.25**
Refiner	Qwen2.5-14B-Instruct	84.21	84.95
	ChatGPT-3.5-Turbo	84.87	85.68
	ChatGPT-4o	**85.44**	**86.25**

Table 5: Impact of base LLMs on NAT-NL2GQL performance on StockGQL.

Method	EM(%)	EX(%)
Ours	**85.44**	**86.25**
Without Preprocessor	80.55 ↓(4.88)	80.06 ↓(6.19)
Generator -> ChatGPT-4o	50.04 ↓(35.40)	47.93 ↓(38.32)
Without Refiner	83.40 ↓(2.04)	83.16 ↓(3.09)
Without Regeneration	84.21 ↓(1.23)	84.13 ↓(2.12)

Table 6: Ablation study on StockGQL, with the green downward arrow showing the decrease and the number in parentheses indicating the value.

7 Conclusion

In this paper, we introduce the NAT-NL2GQL framework to address the NL2GQL task. Our framework comprises three synergistic agents: the Preprocessor Agent, the Generator Agent, and the Refiner Agent. Additionally, we have developed a NL2GQL dataset, named StockGQL. Experimental results show that our approach significantly outperforms baseline methods.

Limitations

Although we have already constructed the StockGQL dataset for the NL2GQL task, the English version is still being prepared. We will release it as open source once it is complete.

References

Hasan Alp Caferoğlu and Özgür Ulusoy. 2024. E-sql: Direct schema linking via question enrichment in text-to-sql. *arXiv preprint arXiv:2409.16751*.

Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262.

Li Dong and Mirella Lapata. 2016. Language to logical form with neural attention. *arXiv preprint arXiv:1601.01280*.

Jiatao Gu, Zhengdong Lu, Hang Li, and Victor OK Li. 2016. Incorporating copying mechanism in sequence-to-sequence learning. *arXiv preprint arXiv:1603.06393*.

Aibo Guo, Xinyi Li, Guanchen Xiao, Zhen Tan, and Xiang Zhao. 2022. Spcql: A semantic parsing dataset for converting natural language into cypher. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*, CIKM '22, page 3973–3977, New York, NY, USA. Association for Computing Machinery.

Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*.

Jacob Devlin Ming-Wei Chang Kenton and Lee Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of naacL-HLT*, volume 1, page 2. Minneapolis, Minnesota.

Yunshi Lan, Gaole He, Jinhao Jiang, Jing Jiang, Wayne Xin Zhao, and Ji-Rong Wen. 2021. A survey on complex knowledge base question answering: Methods, challenges and solutions. *arXiv preprint arXiv:2105.11644*.

Jiale Lao, Yibo Wang, Yufei Li, Jianping Wang, Yunjia Zhang, Zhiyuan Cheng, Wanghu Chen, Mingjie Tang, and Jianguo Wang. 2023. Gptuner: A manual-reading database tuning system via gpt-guided bayesian optimization. *arXiv preprint arXiv:2311.03157*.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. 2024. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems*, 36.

Yuan-Lin Liang, Chih-Yung Chang, and Shih-Jung Wu. 2024a. Kei-cql: A keyword extraction and infilling framework for text to cypher query language translation. *International Journal of Design, Analysis & Tools for Integrated Circuits & Systems*, 13(1).

Yuanyuan Liang, Keren Tan, Tingyu Xie, Wenbiao Tao, Siyuan Wang, Yunshi Lan, and Weining Qian. 2024b. Aligning large language models to a domain-specific graph database for nl2gql. In *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management*, pages 1367–1377.

Karime Maamari, Fadhil Abubaker, Daniel Jaroslawicz, and Amine Mhedhbi. 2024. The death of schema linking? text-to-sql in the age of well-reasoned language models. *Preprint*, arXiv:2408.07702.

Robert Pavliš. 2024. Graph databases: An alternative to relational databases in an interconnected big data environment. In *2024 47th MIPRO ICT and Electronics Convention (MIPRO)*, pages 247–252. IEEE.

Gan Peng, Peng Cai, Kaikai Ye, Kai Li, Jinlong Cai, Yufeng Shen, Han Su, and Weiyuan Xu. 2024. Online index recommendation for slow queries. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pages 5294–5306. IEEE.

Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan O Arik. 2024a. Chase-sql: Multi-path reasoning and preference optimized candidate selection in text-to-sql. *arXiv preprint arXiv:2410.01943*.

Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan O. Arik. 2024b. Chase-sql: Multi-path reasoning and preference optimized candidate selection in text-to-sql. *Preprint*, arXiv:2410.01943.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*, 21(140):1–67.

Tonghui Ren, Yuankai Fan, Zhenying He, Ren Huang, Jiaqi Dai, Can Huang, Yinan Jing, Kai Zhang, Yifan Yang, and X Sean Wang. 2024. Purple: Making a large language model a better sql writer. *arXiv preprint arXiv:2403.20014*.

Yongduo Sui, Qitian Wu, Jiancan Wu, Qing Cui, Longfei Li, Jun Zhou, Xiang Wang, and Xiangnan He. 2024. Unleashing the power of graph data augmentation on covariate distribution shift. *Advances in Neural Information Processing Systems*, 36.

Shayan Talaei, Mohammadreza Pourreza, Yu-Chen Chang, Azalia Mirhoseini, and Amin Saberi. 2024. Chess: Contextual harnessing for efficient sql synthesis. *arXiv preprint arXiv:2405.16755*.

Wenbiao Tao, Hanlun Zhu, Keren Tan, Jiani Wang, Yuanyuan Liang, Huihui Jiang, Pengcheng Yuan, and Yunshi Lan. 2024. Finqa: A training-free dynamic knowledge graph question answering system in finance with llm-based revision. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 418–423. Springer.

Quoc-Bao-Huy Tran, Aagha Abdul Waheed, and Sun-Tae Chung. 2024. Robust text-to-cypher using combination of bert, graphsage, and transformer (cobgt) model. *Applied Sciences*, 14(17):7881.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2022. Self-instruct: Aligning language models with self-generated instructions. *arXiv preprint arXiv:2212.10560*.

Yinlong Xiao, Zongcheng Ji, Jianqiang Li, and Mei Han. 2024. Chinese ner using multi-view transformer. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*.

Tingyu Xie, Qi Li, Jian Zhang, Yan Zhang, Zuozhu Liu, and Hongwei Wang. 2023. Empirical study of zero-shot ner with chatgpt. *arXiv preprint arXiv:2310.10035*.

Derong Xu, Wei Chen, Wenjun Peng, Chao Zhang, Tong Xu, Xiangyu Zhao, Xian Wu, Yefeng Zheng, Yang Wang, and Enhong Chen. 2023. Large language models for generative information extraction: A survey. *arXiv preprint arXiv:2312.17617*.

Tong Zhao, Wei Jin, Yozen Liu, Yingheng Wang, Gang Liu, Stephan Günnemann, Neil Shah, and Meng Jiang. 2022a. Graph data augmentation for graph machine learning: A survey. *arXiv preprint arXiv:2202.08871*.

Ziyu Zhao, Wei Liu, Tim French, and Michael Stewart. 2023. Cyspider: A neural semantic parsing corpus with baseline models for property graphs. In *Australasian Joint Conference on Artificial Intelligence*, pages 120–132. Springer.

Ziyu Zhao, Michael Stewart, Wei Liu, Tim French, and Melinda Hodkiewicz. 2022b. Natural language query for technical knowledge graph navigation. In *Australasian Conference on Data Mining*, pages 176–191. Springer.

Xuanhe Zhou, Guoliang Li, Zhaoyan Sun, Zhiyuan Liu, Weize Chen, Jianming Wu, Jiesi Liu, Ruohang Feng, and Guoyang Zeng. 2023. D-bot: Database diagnosis system using large language models. *arXiv preprint arXiv:2312.01454*.

Yuhang Zhou, Yu He, Siyu Tian, Yuchen Ni, Zhangyue Yin, Xiang Liu, Chuanjun Ji, Sen Liu, Xipeng Qiu, Guangnan Ye, and Hongfeng Chai. 2024. r^3-NL2GQL: A model coordination and knowledge graph alignment approach for NL2GQL. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 13679–13692, Miami, Florida, USA. Association for Computational Linguistics.

Jun-Peng Zhu, Peng Cai, Kai Xu, Li Li, Yishen Sun, Shuai Zhou, Haihuang Su, Liu Tang, and Qi Liu. 2024. Autotqa: Towards autonomous tabular question answering through multi-agent large language models. *Proceedings of the VLDB Endowment*, 17(12):3920–3933.

8 Appendix

8.1 Dataset Format

The dataset includes the following fields:

- **Qid**: A unique identifier for each query in the dataset.
- **Query_masked**: The masked version of the original query, where entity names and other sensitive information are replaced with placeholders (e.g., [s] for stock names, [i] for industry names).
- **GQL_masked**: The masked version of the corresponding GQL (Graph Query Language) query. Similar to the query, the entity names in the GQL are replaced with placeholders.
- **Query**: The original, unmasked natural language query, which is the input that a user would typically provide.
- **GQL**: The corresponding Graph Query Language (GQL) query based on nGQL syntax.
- **SubSchema**: A part of the overall graph schema that is relevant to the specific query. It includes the nodes, edges, and properties involved in the query, providing a structured representation of the relevant subgraph from the graph DB.
- **Masked_name**: A list of entity names that were masked in the query and GQL.

- **Oral_name**: Users often use shortened or informal terms when querying DBs; this field represents the formal version of the colloquial name.

- **Answer**: The result or output generated by executing the corresponding GQL query on the graph DB.

Here is an example. Since our dataset is in Chinese, we have provided the corresponding English translation below the Chinese text for easier reading.

- ***Qid***: 10

- ***Query_masked***:
  [c]是董事长的股票关联的产业下游的产业有哪些?
  (What are the downstream industries related to the industries associated with the chairman [c]'s stock?)

- ***GQL_masked***:
  MATCH (c:chairman{name:'[c]'})
  -[:is_chairman_of]->(s:stock)-[:associate]-
  >(i1:industry)-[:affect]->(i2:industry) RETURN
  i2.industry.name

- ***Query***:
  梁dong是董事长的股票关联的产业下游的产业有哪些?
  (What are the downstream industries related to the industries associated with the chairman Liang Dong's stock?)

- ***GQL***:
  MATCH (c:chairman{name:'梁东'})
  -[:is_chairman_of]->(s:stock)-[:associate]-
  >(i1:industry)-[:affect]->(i2:industry) RETURN
  i2.industry.name

- ***SubSchema***:
  nodes : ["chairman", "stock",, "industry"]
  edges : ["is_chairman_of", "associate", "affect"]

- ***Masked_name***:
  [c] : 梁东'

- ***Oral_name***:
  梁dong': 梁东'}

- ***Answer***:
  i2.industry.name : ["电脑硬件(Computer Hardware)", "汽车(Car)", "金融服务(Financial services)"]

8.2 NER Prompt

The prompt we use for the NER task is shown in 6.

8.3 Related Schema Revision Prompt

The prompt used for revising the related schema is illustrated in Figure 7.

8.4 Linking Completion Algorithm

The algorithm for link completion is described in Algorithm 2.

8.5 GQL Refinement Prompt

The prompt for refining the GQL is shown in Figure 8.

8.6 In-Context Learning Prompt

The prompt used for in-context learning is shown in Figure 9.

8.7 Error Analysis

To further evaluate our method, we analyzed the errors on StockGQL, categorized by hop count and error type. As shown in Figure 10, most errors occur within the 0 to 3 hop range, reflecting the higher proportion of test data in this range. The "Error Statistics by Type" show that nearly half of the errors are due to schema extraction issues. Thus, improving schema extraction accuracy is a key strategy. Additionally, many errors stem from misunderstandings of the query, indicating that question comprehension, especially for colloquial or ambiguous queries, remains a critical challenge. In Appendix 8.8, we present a case study to compare our approach with the baseline method.

8.8 Case Study

To further demonstrate the strengths of our method, we present a detailed case study in Table 7. From the case, we observe that baseline methods either extract the wrong related schema, generate GQL with syntax errors, or fail to recognize colloquial variations of named entities. In contrast, our approach accurately extracts the related schema, even for multi-hop queries, and effectively interprets colloquial variations of named entities. This ensures that entity names are recognized and accurately reflected in the generated GQL, even when the input deviates from standard formal representations. This highlights the robustness and adaptability of our method in handling complex and varied queries, further reinforcing its effectiveness in real-world applications.

Instruction:
You are an expert in the NLP field. I would appreciate your assistance with an NER task. Given entity label set: label set. Refer to the given example. Based on the provided entity label set, please recognize the named entities in the given Question. Please directly output the answer.

Output Format:
In JSON format, for example: {Entity Name: Entity Type, Entity Name: Entity Type}.

Here are some examples:
{EXAMPLES}
======== **Predict** ========
Question:
{QUESTION}
Answer:

Figure 6: Prompt for performing Name Entity Recognition on questions using ChatGPT-4o.

Instruction:
You are an expert in the NLP field. I am working on an information extraction task that involves identifying the related schema potentially relevant to a given question from a graph DB schema. I have already extracted the Candidate Related Schema. Please assist me in verifying whether the Candidate Related Schema contains any redundancies and ensure that each one is necessary. Based on the provided examples, kindly provide the correct Candidate Related Schema.

Candidate Related Schema:
-The complete Schema structure of Candidate Related Nodes and Candidate Related Edges.

Output Format:
Please follow the format in the Examples. Directly output the result you consider correct after "Related Schema:" .

Here are some examples:
{EXAMPLES}
======== **Predict** ========
Question:
{QUESTION}
Candidate Related Schema:
{Candidate_related_schema}

Related Schema:

Figure 7: Prompt for revising the related schema.

8.9 Comparison with Similar Tasks

Text2SQL

Text2SQL is a task in NLP that is quite similar to NL2GQL, as both involve transforming user queries into statements that can be executed on a DB. Recently, there have been many efforts to apply LLMs to solve Text2SQL, and these methods have achieved good results (Pourreza et al., 2024b; Maamari et al., 2024; Li et al., 2024; Caferoğlu and Ulusoy, 2024). However, there are significant differences between the two.

- The diversity inherent in GQL presents a series of challenges. Unlike SQL, which has a well-established and standardized query language for relational DBs, GQL lacks a unified standard (Zhou et al., 2024). This deficiency

Algorithm 2: Linking Completion Algorithm

Input: Graph Schema $G = (V, E)$; Identified Entities $E_{\text{identified}}$; Identified Edges $R_{\text{identified}}$
Output: Connected Subgraph $SG = (V_{\text{subgraph}}, E_{\text{subgraph}})$

Function `LinkCompletion`$(G, E_{identified}, R_{identified})$**:**
  $V_{\text{subgraph}} \leftarrow \emptyset$
  $E_{\text{subgraph}} \leftarrow \emptyset$
  foreach *entity* $v_i \in E_{identified}$ **do**
    $V_{\text{subgraph}} \leftarrow V_{\text{subgraph}} \cup \{v_i\}$
  foreach *edge* $r_j \in R_{identified}$ **do**
    $E_{\text{subgraph}} \leftarrow E_{\text{subgraph}} \cup \{r_j\}$
  foreach *edge* $e_k \in E_{subgraph}$ **do**
    foreach *neighbor* $v_l \in neighbors(e_k)$ **do**
      $V_{\text{subgraph}} \leftarrow V_{\text{subgraph}} \cup \{v_l\}$
      $E_{\text{subgraph}} \leftarrow E_{\text{subgraph}} \cup \{e_k\}$
  while $V_{subgraph}$ *is not connected* **do**
    Find the minimum edge to add that connects two disconnected components
    $E_{\text{subgraph}} \leftarrow E_{\text{subgraph}} \cup \{\text{min edge}\}$
  return $SG = (V_{\text{subgraph}}, E_{\text{subgraph}})$

NL	持有华强股票且持仓比例超过7%的基金经理? (The fund manager who manages the fund that holds Huaqiang stock with a holding ratio exceeding 7%?)	
Formal expression	持有华强科技股票且持仓比例超过7%的基金由哪位基金经理管理? (Which fund manager manages the fund that holds Huaqiang Technology stock with a holding ratio exceeding 7%?)	
Method	**Related Nodes and Edges**	**Output**
ICL(ChatGPT-4o)	full schema	MATCH (s:stock{name: '华强'})<-[h:manage]-(fm:fund_manager) WHERE h.position_ratio > 7% RETURN fm.name
Fine-Tuning(full schema) Qwen2.5-14B-Instruct	full schema	MATCH (s:stock{name: '华强'})<-[h:hold]- (pof:fund)<-[:manage]-(fm:fund_manager) WHERE h.hold.position_ratio > 7% RETURN fm.fund_manager.name
Align-NL2GQL	Nodes:[fund_manager,fund,stock] Edges: [manage,hold]	MATCH (s:stock{name: '华强'})<-[h:hold]- (pof:fund)<-[:manage]-(fm:fund_manager) WHERE h.position_ratio > 7% RETURN fm.fund_manager.name
R^3-NL2GQL	Nodes:[fund_manager,stock] Edges: [hold]	MATCH (s:stock{name: '华强科技'})<-[h:hold]-(fm:fund_manager) WHERE h.position_ratio > 7% RETURN fm.fund_manager.name
Ours	Nodes:[fund_manager,fund,stock] Edges: [manage,hold]	MATCH (s:stock{name: '华强科技'})<-[h:hold]- (pof:fund)<-[:manage]-(fm:fund_manager) WHERE h.position_ratio > 7% RETURN fm.fund_manager.name

Table 7: A case study in the StockGQL dataset is presented, displaying the results of both our method and the baseline methods. Due to space limitations, the table uses "Related Nodes and Edges" rather than listing the full details of the related schema. The segments with predicted errors are highlighted in red, while the correct ones are marked in blue.

creates obstacles in various areas, including dataset construction, the development of models capable of generalizing across different DBs, and the establishment of consistent training paradigms. There is a difference in query objectives. NL2GQL aims to execute queries on graph DBs, whereas Text2SQL targets relational DBs. Graph DBs feature more flexible data structures and complex relationships, requiring NL2GQL to manage a wider variety of queries and data relationships (Liang et al., 2024b).

- The flexibility of query languages differs. GQL is more flexible compared to SQL, allowing for complex queries on nodes and edges

Instruction:
You are an expert in NebulaGraph DBs, with specialized expertise in nGQL. A prior attempt to execute a query did not produce the expected results, either due to execution errors or because the returned output was empty or incorrect. Your task is to analyze the issue using the provided related schema of query and the details of the failed execution. Based on this analysis, you should offer a corrected version of the nGQL. Ensure adherence to the nGQL conventions for naming variables, entities, and attributes (e.g., ‘s.stock.name‘) and verify that all conditional filters use ‘==‘ syntax, such as ‘s.stock.name == ’[s]’‘.

Procedure:
1. Analyze Query Requirements:
- Question: Consider what information the query is supposed to retrieve.
- Info: The preprocessed data information. - nGQL: Review the nGQL query that was previously executed and led to an error or incorrect result.
- Error: Analyze the outcome of the executed query to identify why it failed (e.g., AssertionError).
2. Determine whether the Related Schema is correct.
- Based on the above information, first determine whether the extracted related schema is correct.
- If related schema is not correct, directly output "Info Error". Otherwise, modify the nGQL query to address the identified issues, ensuring it correctly fetches the requested data according to the graph DB schema and query requirements.

Output Format:
Based on whether the determined Related Schema is correct, output either "Info Error" or your corrected query. The corrected query as a single line of nGQl code. Ensure there are no line breaks within the query.

Here are some examples:
{EXAMPLES}
======== **Predict** ========
Question:
{QUESTION}
Related Schema:
{RELATED_SCHEMA}
nGQL:
{nGQL}
Error:
{ERROR}
Output:

Figure 8: The prompt used for GQL refine.

in a graph DB, while SQL is constrained by the fixed structure and syntax of relational DBs. There is a greater variety of keyword types in GQL compared to SQL. GQL encompasses more keyword types, reflecting the diverse data structures and query requirements in graph DBs. NL2GQL must recognize and process these different types of keywords, further complicating the task.

- The complexity of query paths is notable. Queries in graph DBs often involve intricate paths between multiple nodes and edges. NL2GQL must handle these complex paths and translate natural language questions into corresponding GQL queries, adding to the overall complexity of the task.

The following examples highlight scenarios where NL2GQL excels while Text2SQL faces limitations due to relational model constraints.

1. Multi-hop Path Query Question: Find the shortest collaboration path from User A to User B, where all participants in the path belong to the same department.

Cypher Implementation:

```
MATCH (a:User {name: "UserA"}),
 (b:User {name: "UserB"}),
```

Instruction:
You are an expert in NebulaGraph DBs, please write the nGQL query corresponding to the given Question directly based on the provided knowledge graph Schema and Examples. Ensure adherence to the nGQL conventions for naming variables, entities, and attributes (e.g., ‘s.stock.name‘) and verify that all conditional filters use ‘==‘ syntax, such as ‘s.stock.name == ’[s]’‘. Please provide the answer directly without any additional explanation. Please provide the answer directly without any additional explanation. Please provide the answer directly without any additional explanation.

Output Format:
Please output nGQL directly.

Schema:
{SCHEMA}

Here are some examples:
{EXAMPLES}
======== **Predict** ========
Question:
{QUESTION}
nGQL:

Figure 9: Prompt for In-Context Learning.

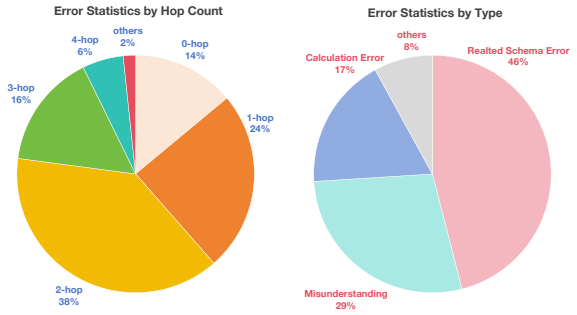

Figure 10: Error analysis statistics chart. On the left is the error statistics based on the hop count of the jump in the NL, and on the right is the error statistics based on the error type.

```
path = shortestPath((a)
 -[:COLLABORATED_WITH*]-(b))
WHERE ALL(node IN nodes(path)
WHERE node.department = a.department)
RETURN path
```

Text2SQL Challenges: In relational databases, such multi-hop path queries require recursive JOINs (e.g., using WITH RECURSIVE), which have poor performance and complex syntax. It is not possible to directly express the "shortest path" semantics, relying on stored procedures or external algorithms.

2. Cyclic Relationship Detection Question: Detect if there exists a collaboration cycle: User A → User B → User C → User A.

Cypher Implementation:

```
MATCH (a:User {name: "UserA"})
 -[:COLLABORATED_WITH]->(b:User),
    (b)-[:COLLABORATED_WITH]->(c:User),
    (c)-[:COLLABORATED_WITH]->(a)
RETURN a, b, c
```

Text2SQL Challenges: Requires self-joins on the same table multiple times (e.g., Users AS u1 JOIN Users AS u2 ...), leading to exponential query complexity. It is not possible to directly express cyclical structures and requires manually hardcoding the path length (e.g., 3 hops in this example).

3. Dynamic Aggregation and Graph Pattern Matching Question: Count the managers in each department who have more than 10 subordinates and whose subordinates have participated in cross-department projects.

Cypher Implementation:

```
MATCH (m:Manager)-[:MANAGES]
 ->(e:Employee)
WITH m, COUNT(e) AS subordinates
WHERE subordinates > 10
MATCH (e)-[:PARTICIPATED_IN]->
 (p:Project{is_cross_department:true})
RETURN m.name, subordinates,
 COLLECT(p.name) AS projects
```

Text2SQL Challenges: Requires combining aggregation (COUNT) with existence checks (EXISTS subqueries), leading to high complexity in nested queries. It is difficult to efficiently handle graph pattern matching for "cross-department projects" (requires multi-table JOINs and complex filtering conditions).

4. Recursive Relationship Query Question: Find all indirect subordinates of User A, including the subordinates' subordinates.

Cypher Implementation:

```
MATCH (a:User {name: "UserA"})
-[:MANAGES*1..]->(sub:Employee)
RETURN sub.name
```

Text2SQL Challenges: In relational databases, recursive CTEs (WITH RECURSIVE) must be used, but the syntax is obscure and the performance is poor. It is difficult to control the recursion depth flexibly (e.g., the *1.. notation in this example represents an arbitrary depth).

5. Graph Embedding-Based Semantic Similarity Query Question: Find users whose interests are similar to User A's, with at least three common interests.

Cypher Implementation:

```
MATCH (a:User {name: "UserA"})
-[:INTERESTED_IN]->(i:Interest)
WITH a, COLLECT(i) AS interests
MATCH (u:User)-[:INTERESTED_IN]
 ->(i:Interest)
WHERE u <> a AND SIZE([x IN interests
 WHERE x IN u.interests]) >= 3
RETURN u.name
```

Text2SQL Challenges: Requires handling set intersection (common interests), which in SQL must be implemented with INTERSECT and subqueries, making the syntax cumbersome. It is not possible to directly express graph embedding-based similarity calculations (which require external extension libraries).

6. Temporal Graph Analysis Question: List all stocks that experienced a drop of more than 5% in a single day after five consecutive days of price increase.

Cypher Implementation:

```
MATCH (s:Stock)-[r:HAS_DAILY_DATA]
 ->(d:DailyData)
WITH s, d ORDER BY d.date ASC
WITH s, COLLECT(d) AS data
WHERE size(data) >= 6
  AND ANY(i IN RANGE(0, size(data)-6)
    WHERE REDUCE(rising = true,
     j IN [0..4] | rising AND
    data[i+j+1].close > data[i+j].close)
     AND
     (data[i+5].close - data[i+6].close)
     /data[i+5].close >= 0.05
RETURN s.name
```

Text2SQL Challenges: Requires window functions (e.g., LAG/LEAD) and complex condition combinations, reducing readability. It is difficult to efficiently handle dynamic time-series patterns (e.g., "consecutive N days of increase").

The scenario types that can be achieved by NL2GQL but are difficult to implement with Text2SQL is shown in Table 8. In summary, NL2GQL is more complex than Text2SQL due to its handling of graph DB queries, the flexibility of GQL, the complexity of data paths and the variety of keyword types. Given these differences, it is challenging to directly transplant methods from the Text2SQL task to the NL2GQL task.

KBQA

Knowledge-Based Question Answering (KBQA) systems leverage structured knowledge bases (KBs) to answer user queries. SP-based methods, commonly known as NL2SPARQL, first translate natural language questions into SPARQL queries, which are then executed on the KB to retrieve answers (Lan et al., 2021). This approach is similar to NL2GQL; however, a significant difference between NL2GQL and NL2SPARQL in the KGQA domain lies in the complexity of data storage and query languages. Graph databases (Graph DBs), which manage data with intricate relationships, introduce additional complexity (Liang et al., 2024b). Moreover, NL2GQL requires a deeper focus on schema information, as entities in graph DBs may have a diverse range of attribute types (Zhou et al., 2024). NL2GQL is also characterized by complex graph modalities, a wide variety of query types, and the unique nature of GQLs (Zhou et al., 2024). As a result, directly applying KBQA methods to the NL2GQL task is impractical.

The following are additional examples that showcase the unique capabilities of NL2GQL and its corresponding Cypher implementations, which traditional KBQA methods struggle to handle:

1. Multi-hop Relationship and Co-participation Count Question: Find friends of the user 'Alice' who have at least three common

Capability	NL2GQL	Text2SQL
Multi-hop Path Traversal	✓	× (Requires recursive CTE)
Cyclic Structure Detection	✓	× (Complex self-joins)
Recursive Relationship Query	✓	× (Syntax limitations)
Dynamic Graph Pattern Matching	✓	× (Exploding JOINs)
Temporal Graph Analysis	✓	× (Relies on window functions)
Set and Graph Embedding Operations	✓	× (Limited functionality)

Table 8: Summary of scenario types that can be achieved by NL2GQL but are difficult to implement with Text2SQL.

projects with her.

Cypher Implementation:

```
MATCH (alice:User {name: "Alice"})
  -[:FRIEND_OF]->(f1:User)-[:FRIEND_OF]
  ->(f2:User)
MATCH (f2)-[:PARTICIPATED_IN]->
(p:Project)<-[:PARTICIPATED_IN]-(alice)
WITH f2, COUNT(DISTINCT p)
  AS common_projects
WHERE common_projects >= 3
RETURN f2.name AS mutual_friend,
  common_projects
```

KBQA Challenges: Dynamic traversal of multi-hop social relationships (2-hop friends) and association with common projects. KBQA methods generally cannot flexibly combine multi-hop paths with aggregation and filtering conditions (e.g., COUNT >= 3).

2. Temporal Event Combination Filtering Question: Identify all users who purchased Product A in 2023 and rated it five stars within the last six months. **Cypher Implementation:**

```
MATCH (u:User)-[:PURCHASED]
 ->(p:Product {name: "ProductA"})
WHERE p.purchase_date >= '2023-01-01'
 AND p.purchase_date <= '2023-12-31'
WITH u
MATCH (u)-[r:RATED]
 ->(p:Product {name: "ProductA"})
WHERE r.rating = 5 AND
 r.date >= date().duration("-6 months")
RETURN u.name, r.date AS rating_date
```

KBQA Challenges: Combining temporal windows (2023 purchase + recent 6-month rating) and cross-event associations (purchase and rating). KBQA struggles with dynamic time-based calculations.

3. Aggregation and Nested Subqueries Question: Count the managers in each department whose salary is above the department's average and who manage at least two subordinates. **Cypher Implementation:**

```
MATCH (d:Department)
WITH d, AVG(e.salary) AS avg_salary
MATCH (m:Manager)-[:MANAGES]
 ->(e:Employee {department: d.name})
WHERE m.salary > avg_salary
WITH m, COUNT(e) AS subordinates
WHERE subordinates >= 2
RETURN d.name AS department,
 m.name AS manager, m.salary
 , subordinates
```

KBQA Challenges: First, calculating the department's average salary, which then serves as a filtering condition. KBQA cannot dynamically execute nested aggregation (department-level aggregation + individual-level filtering).

4. Cyclic Subgraph Pattern Detection Question: Find all collaborative networks that form cycles with at least four nodes. **Cypher Implementation:**

```
MATCH path = (a:User)
 -[:COLLABORATES_WITH*3..]->(a)
WHERE length(path) >= 3
 AND ALL(n IN nodes(path)
 WHERE size(apoc.coll.duplicates
        (nodes(path))) = 0)
RETURN path
```

KBQA Challenges: Detecting cyclic structures in graph theory (path starts and ends at the same node without repeated nodes). KBQA lacks subgraph pattern matching capability.

5. Consecutive Temporal Event Detection Question: Identify all customers who placed two consecutive orders with decreasing amounts in the last three months. **Cypher Implementation:**

Scenario	NL2GQL	KBQA
Multi-hop Dynamic Path	✓	× (Relies on predefined paths)
Temporal Event Combinations	✓	× (Time logic is rigid)
Nested Aggregation	✓	× (Only single-layer aggregation)
Cyclic Pattern Detection	✓	×
Continuous Event Sequence Analysis	✓	×

Table 9: Summary of scenario types that can be achieved by NL2GQL but are difficult to implement with KBQA.

```
MATCH (c:Customer)-[o:ORDERED]
 ->(order:Order)
WHERE o.date >= date()
 .duration("-3 months")
WITH c, order ORDER BY o.date ASC
WITH c, COLLECT(order) AS orders
WHERE size(orders) >= 2
  AND ANY(i IN RANGE(0, size(orders)-2)
      WHERE orders[i].amount
       > orders[i+1].amount
        AND orders[i+1].amount
         > orders[i+2].amount
  )
RETURN c.name, [order IN orders
 | {date: order.date, amount:
   order.amount}]
 AS order_history
```

KBQA Challenges: Detecting consecutive event patterns (decreasing order amounts). KBQA cannot handle dynamic temporal sequence aggregation analysis.

The scenario types that can be achieved by NL2GQL but are difficult to implement with Text2SQL is shown in Table 9.