GQLBench: A Large-Scale Cross-Domain, Cross-Dialect Benchmark for NL2GQL

Yanning Su[1*] **Yuhang Zhou**[1,2*] **Yang Fang**[3]
Sen Liu[1] **Guangnan Ye**[1,2†] **Hongfeng Chai**[1]

[1]Fudan University [2]Shanghai Innovation Institute [3]Vesoft Inc.

Abstract

Despite growing interest in NL2GQL, benchmarking progress has been constrained by the lack of resources that are simultaneously large-scale, cross-domain, and cross-dialect. To address this gap, we present **GQLBench**, a new benchmark built through an automated and scalable framework that integrates NL2SQL-to-NL2GQL conversion with graph-native data generation. GQLBench supports execution-based evaluation on both Cypher and ISO GQL, covering hundreds of graph databases and over 20k natural language questions for each dialect. By combining converted data from mature NL2SQL resources with synthetic graph-specific queries, it captures both schema diversity from real-world relational sources and graph-native reasoning challenges, including long paths and cycles. Beyond overall performance comparison, GQLBench also enables fine-grained evaluation across dialects, graph patterns, and query complexity. Experiments on advanced LLMs show that even strong proprietary models struggle on GQLBench, with gemini-3-flash achieving only 35.40% average execution accuracy across the two dialects. Our data and code are available at `https://github.com/qxssadf/GQLBench`.

1 Introduction

With the growing use of Graph Database Management Systems (GDBMS) in domains such as recommendation systems (Stanescu, 2021), QA systems incorporating retrieval-augmented generation (RAG) (Edge et al., 2024; Guo et al., 2024; Gutiérrez et al., 2024, 2025), and biological data storage (Mazein et al., 2024), the task of translating natural language into graph query languages, known as NL2GQL, has attracted increasing attention (Zhou et al., 2024; Liang et al., 2024a,b). However, despite the rapid progress of NL2SQL benchmarks (Lei et al., 2024; Li et al., 2023; Yu et al., 2018), NL2GQL benchmarking still lacks resources that are simultaneously large-scale, cross-domain, and cross-dialect, which limits robust evaluation and model development. As a result, current NL2GQL evaluation often remains fragmented, making it difficult to systematically compare model performance across databases, domains, and query dialects.

These limitations mainly arise from three challenges. First, many existing benchmarks focus on specific domains, such as finance, law, or healthcare, which restricts their ability to evaluate the generalization capability of LLMs across diverse scenarios (Liang et al., 2024b; Nie et al., 2022; Zhong et al., 2025; Feng et al., 2025; Ozsoy et al., 2025). Second, with the recent release of the ISO GQL standard[1], existing resources provide limited support for evaluating models on standardized and emerging graph query dialects. This limitation is especially problematic because real-world graph ecosystems are inherently heterogeneous, and a practical NL2GQL benchmark should evaluate whether models can generalize across both widely used and emerging standardized dialects. Third, building high-quality NL2GQL benchmarks typically requires expensive manual data construction and annotation, making scalable evaluation difficult in practice.

To address these challenges, we develop an automated and scalable framework for constructing NL2GQL benchmarks. Instead of relying on expensive manual collection and annotation, our framework leverages mature NL2SQL resources and augments them with graph-native synthetic data, making it possible to construct execution-supported benchmarks at scale. The framework consists of two complementary modules. The first is a converter from NL2SQL resources to

*These authors contributed equally to this work.
†Corresponding author (yegn@fudan.edu.cn).

[1] `https://www.iso.org/standard/76120.html`.

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 31989–32014
July 2-7, 2026 ©2026 Association for Computational Linguistics

NL2GQL data, which automatically migrates relational databases into graph databases and translates SQL queries into executable GQL queries across dialects through a unified GQL Abstract Syntax Tree (AST). The second is a graph generator, which synthesizes graph-specific structures that are rarely available in relational sources, such as long paths and cycles, and further produces corresponding GQL queries and natural language questions.

Built with this framework, **GQLBench** is a large-scale, cross-domain, and cross-dialect benchmark for NL2GQL. It supports execution-based evaluation on both Cypher and ISO GQL, covering hundreds of graph databases and over 20k natural language questions for each dialect. Table 1 presents a comparison of dataset sizes across benchmarks. GQLBench surpasses all others in terms of the number of entities, indicating its substantially larger scale. Beyond scale, GQLBench is designed to capture both schema diversity from real-world relational resources and graph-specific reasoning challenges introduced through synthetic generation, such as long paths and cycles. As a result, it supports not only overall performance comparison, but also fine-grained analysis of model behavior across dialects, graph patterns, and query complexity. Experiments on advanced LLMs show that even strong proprietary models struggle on GQLBench, with gemini-3-flash achieving only 35.40% average execution accuracy across the two dialects.

In summary, the main contributions of this paper are as follows:

- We introduce **GQLBench**, a large-scale, cross-domain, and cross-dialect benchmark for NL2GQL, with execution support for both Cypher and ISO GQL.

- We develop an automated and scalable framework for constructing the benchmark, combining relational-to-graph migration, SQL-to-GQL translation via a unified GQL AST, and synthetic generation of graph-specific *(GQL, Question)* pairs.

- We show through extensive experiments that GQLBench is highly challenging for current LLMs and supports fine-grained evaluation across diverse graph patterns, return specifications, and advanced function usage.

Benchmark	# graphs	Data Size
Text-to-Cypher		
MetaQA-Cypher	1	43k entities,135k relations
SpCQL	1	25M entities,140M relations
MedT2C	2	91k entities,2.5M relations
CypherBench	11	7.8M entities,14.7M relations
Text-to-nGQL		
R^3-NL2GQL	3	46k entities,298k relations
StockGQL	1	650k entities,4k relations
NL2GQL		
GQLBench(Cypher)	311	140.7M entities,15.0M relations
GQLBench(ISO GQL)	260	56.0M entities,12.9M relations

Table 1: Data size comparison with other benchmarks.

2 Related Work

2.1 NL2GQL Benchmarks

Recent years have seen growing interest in benchmarking NL2GQL, with prior work making valuable progress from different perspectives. Early datasets such as MetaQA-Cypher (Nie et al., 2022), MedT2C (Zhong et al., 2025), and StockGQL (Liang et al., 2024b) provide important testbeds for studying NL2GQL in domain-specific settings. Other efforts, including Neo4j Text2Cypher (Ozsoy et al., 2025), CypherBench (Feng et al., 2025), and SpCQL (Guo et al., 2022), further expand the scope of benchmark construction by exploring larger graph data, realistic graph settings, or graph-oriented semantic parsing.

These benchmarks have substantially advanced the study of NL2GQL. However, existing resources still mainly focus on particular domains, specific graph settings, or individual query dialects, and therefore do not yet provide a benchmark that is simultaneously large-scale, cross-domain, cross-dialect, and execution-supported. In addition, most prior work centers on Cypher or nGQL (Liang et al., 2024b; Zhou et al., 2024), while support for ISO GQL-aligned dialects remains limited. In contrast, GQLBench is designed to fill this gap by providing a large-scale, cross-domain, and cross-dialect benchmark with execution support on both Cypher and NebulaGraph GQL[2].

2.2 Automated Data Migration

Automated migration from existing NL2SQL resources provides a promising direction for scalable benchmark construction, and prior studies have laid important foundations for this line of work. For example, Rel2Graph (Zhao et al., 2023) explores

[2]In this work, we use NebulaGraph GQL(Nebula) as the implementation of ISO GQL.

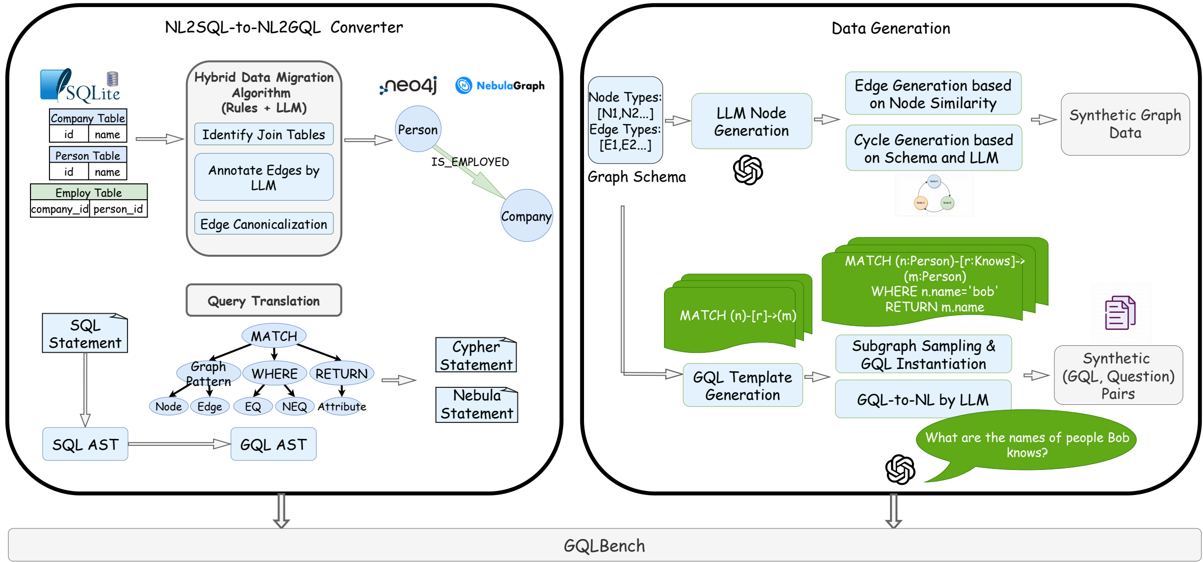

Figure 1: This figure illustrates the two main data construction pipelines within our framework. On the left is a pipeline based on existing NL2SQL datasets, which includes data migration and query translation. On the right is a pipeline based on data generation, which automatically generates nodes and edges, and creates *(GQL, Question)* pairs by combining GQL templates and LLM-based question generation. The final benchmark is formed by combining data and queries from both methods.

automatic transformation from relational databases to graph representations, while SQL2CYPHER (Li et al., 2021) studies migration from SQL workloads to Cypher queries. These efforts show that leveraging mature relational resources can effectively reduce the cost of constructing graph-oriented datasets.

Building on this line of research, our work places particular emphasis on benchmark construction for NL2GQL. We combine rule-based structure preservation with LLM-based semantic annotation to support relational-to-graph migration, SQL-to-GQL translation, and the construction of graph data that is both scalable and semantically natural.

3 Overview

In this paper, we construct a large-scale, cross-domain, and cross-dialect NL2GQL benchmark. To mitigate the expensive manual collection and annotation, we propose a data migration algorithm that combines rule-based structure preservation with LLM-based semantic annotation to transform relational databases into property graphs. The algorithm first identifies join tables, which are subsequently transformed into edges in the graph schema, and further determines the overall edge structure of the graph by prompting LLMs to annotate and canonicalize edges. Building upon the migrated data, we further develop a unified GQL Abstract Syntax Tree (AST) and a corresponding SQL-to-GQL translation algorithm. This design enables systematic transformation from SQL AST to GQL AST, facilitating the generation of semantically equivalent and executable GQL queries across multiple dialects, including Cypher and NebulaGraph GQL(Nebula), an implementation of ISO GQL used in this work. These two modules constitute a converter from NL2SQL resources to NL2GQL data. Furthermore, to capture graph-specific structures that are rarely available in relational sources, such as long paths and cycles, we introduce an automated graph generation pipeline that synthesizes complex graph structures along with their corresponding *(GQL, Question)* pairs. The overall pipeline is demonstrated in Figure 1.

4 NL2SQL-to-NL2GQL Converter

4.1 Data Migration: Relational-to-Graph Conversion

To efficiently leverage the high-quality data available in existing NL2SQL datasets and avoid the need for manual data collection, we propose a hybrid, automatic relational-to-graph conversion algorithm that combines rule-based heuristics with LLMs. The algorithm employs deterministic rules to preserve the structural integrity of relational databases, while simultaneously utilizing LLMs to ensure the semantic completeness and natural-

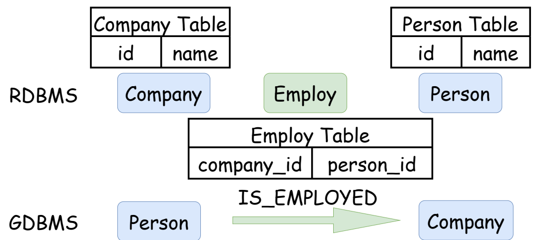

Figure 2: An example of a join table. The `Employ` table is identified as a join table and then converted to an edge type in GDBMS.

ness of the conversion process.

4.1.1 Identify Join Tables

A join table is a table in the relational databases that semantically represents many-to-many relationships between entities and should therefore be transformed into one or more edge types rather than a node type. A non-join table (i.e. an entity table) is represented as a node type in the graph.

We propose a hybrid classification strategy combining structural heuristics with semantic analysis to identify join tables.

Deterministic Exclusion. Any table containing fewer than two foreign keys is definitively classified as a non-join table (i.e., an entity table).

Deterministic Inclusion. Any table whose primary keys consist solely of foreign keys, with at least two such keys, is classified as a join table.

LLM-Based Semantic Analysis. For tables that do not satisfy these strict structural rules, we prompt an LLM to analyze the table's semantic role and determine whether it should be regarded as a join table.

4.1.2 Annotate Edges by LLM

In addition to edge types derived from join tables, foreign key relationships between entity tables are also transformed into graph edge types. To support this process, we employ an LLM to analyze the database schema, performing two key tasks: (1) annotating the directions and names of edge types arising from direct foreign key constraints, and (2) analyzing the identified join tables to determine the precise edge definitions, including edge names and the corresponding source and target entity tables.

During the annotation process, we also determine the columns used to establish edges between entities. With each row in an entity table representing a distinct node instance, we construct edges based on these columns. An example of edge annotation is shown in Figure 3. More details are provided in Appendix D.1.2.

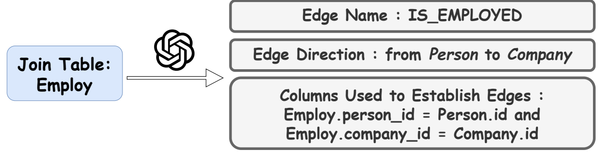

Figure 3: An example of annotating edges by LLM. The edge name, edge direction and columns used to establish edges are provided by LLM.

4.1.3 Edge Canonicalization

Despite the strong semantic capabilities of LLMs, edge annotation across multiple dimensions may introduce semantically redundant edge types. To address this issue, we apply an edge canonicalization process, following prior work in automatic knowledge graph construction (Zhang and Soh, 2024; Vashishth et al., 2018). Specifically, we prompt an LLM to identify and consolidate redundant edge types that exhibit semantic equivalence between the same pair of tables, thereby ensuring a concise and unambiguous graph schema. The detailed results of semantic naturalness are provided in Appendix D.1.3.

4.2 Query Translation: Transforming SQL Queries to GQL Queries

To enable automated translation from SQL queries to GQL queries, we construct a unified GQL Abstract Syntax Tree (AST) that supports multiple graph query dialects, including Cypher and Nebula. Building upon this unified intermediate representation, we implement a translation algorithm that systematically maps SQL AST to GQL AST, ensuring alignment with the transformations defined in our aforementioned database migration algorithm. Furthermore, we develop a query generation framework that converts the unified GQL AST into executable queries across multiple dialects.

4.2.1 Constructing the GQL AST

To enable cross-dialect translation from SQL to GQL, we construct a unified GQL Abstract Syntax Tree (AST), which provides a foundation for extending support to additional GQL dialects.

Relation. In relational databases, connections are implicitly defined through foreign key constraints and join tables. We reify these implicit links into explicit *Relation* structures in the AST. This abstraction effectively encapsulates edge semantics

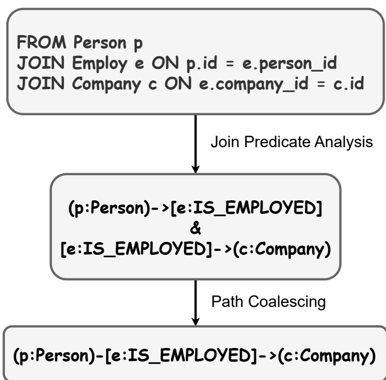

Figure 4: An example of translating the `JOIN` predicates to a graph pattern with Join Predicate Analysis and Path Coalescing.

(e.g., types and aliases), decoupling them from the underlying physical storage schema.

Graph Pattern. A fundamental distinction between SQL and GQL lies in their structural representation. While SQL relies on `FROM` and `JOIN` clauses to establish data connectivity, GQL employs *Graph Patterns* within `MATCH` clauses to define the topology of the queried graph. In our AST, the graph pattern serves as a central component for bridging SQL query structures and GQL semantics. It is represented as a sequence of nodes and relations, enabling a systematic mapping from SQL join operations to graph traversal patterns. In particular, a graph pattern consisting of a strictly alternating sequence of nodes and relations is formally defined as a *Path*.

4.2.2 Translating SQL AST to GQL AST

The core of the translation lies in systematically mapping SQL clauses to their GQL counterparts: `SELECT` and `WHERE` are mapped to `RETURN` and `WHERE`, respectively. More importantly, the data connectivity expressed through `FROM` and `JOIN` operations is transformed into graph patterns within the `MATCH` clause.

Join Predicate Analysis. Our translation system handles join predicates in two scenarios: atomic equality constraints (e.g., `Table_A.a = Table_B.b`) and conjunctive predicates. For each join predicate, the system analyzes the schema to determine whether it corresponds to an edge type derived from either a join table or a foreign key constraint. When a valid correspondence is identified, the relational join is mapped to the corresponding graph edge. The details of converting a `JOIN` to a graph pattern are described in Appendix D.2.1.

Path Coalescing. In scenarios involving multiple `JOIN` clauses, the translation process initially generates a set of discrete edges. To optimize the resulting query, the algorithm evaluates the topological connectivity among these edges and coalesces individual edges into a continuous, multi-hop traversal. Figure 4 illustrates the translation of the `JOIN` predicates into a graph pattern.

4.2.3 Generating GQL Queries for Multiple Dialects

Leveraging the unified GQL AST, we develop a suite of dialect-specific generation modules, each tailored to the unique syntax of a target GQL dialect, to automate the translation of executable queries. The translation system currently supports multiple target dialects, including Cypher and Nebula, which conforms to the ISO GQL standard.

We adopt the *Visitor* design pattern to decouple query generation logic from the structural representation of the AST, enabling modular and extensible generation across different dialects via dialect-specific generator classes. Consequently, support for emerging ISO GQL dialects can be seamlessly integrated by simply implementing new *visitors*, without necessitating modifications to the core syntax tree. The semantic preservation validation is discussed in Appendix D.2.4.

5 Data Generation

To increase data diversity and better capture graph-specific structures such as cycles and long paths, we further generate synthetic data, including both graph structures and their corresponding *(GQL, Question)* pairs.

5.1 Graph Data Generation

We manually curate 30 distinct graph schemas spanning 10 diverse domains, including finance, education, and law. To simulate realistic data distributions, we employ LLMs to populate these schemas with node instances based on the defined attributes. Subsequently, inspired by the generation strategy used in the LDBC Social Network Benchmark (Erling et al., 2015), we generate edges based on the homophily principle. Specifically, we compute vector embeddings for node attributes and calculate pairwise similarities. The probability of edge creation between nodes is determined via a geometric distribution function derived from these similarity scores. For each node u, we rank all other nodes v by their embedding similarity in descending order.

The probability of establishing an edge (u, v), denoted as P_{uv}, decays geometrically with the rank r_{uv}:

$$P_{uv} = p(1-p)^{r_{uv}-1} \quad (1)$$

where p is a hyperparameter controlling the connectivity sparsity, and r_{uv} denotes the rank of node v in u's similarity list. This mechanism assigns higher connection probability to semantically similar entities while allowing for occasional long-range associations. By applying this similarity-based selection to every possible source node under a specific configuration of p, the generator naturally facilitates the formation of multi-hop semantic chains (see Appendix D.3.3 for a detailed analysis). To further enhance topological diversity, we pre-define specific cyclic patterns and leverage LLMs to generate corresponding structures, enriching the dataset with complex graph features. Figure 6 in Appendix D.3.4 shows an example of long paths and cycles in the generated data.

5.2 Query Generation

We adopt a template-based approach to guarantee grammatical correctness while maintaining fine-grained control over query complexity and structural diversity. The query construction follows the fundamental GQL backbone: pattern definition (`MATCH`), constraint application (`WHERE`), and result projection (`RETURN`). More details about templates are in Appendix D.3.1.

By combining these modular components, we synthesize complete GQL query templates. These templates are then instantiated by randomly sampling entities and property values from the generated graph. To ensure the utility of the benchmark, we execute the generated queries and prune instances that yield empty result sets. Finally, we prompt an LLM to perform reverse-annotation, generating natural language questions corresponding to each valid GQL query. Detailed dataset composition and leakage analysis are in Appendix D.3.5.

6 Experiments

6.1 Data Migration and Query Translation from NL2SQL Benchmark

The translated GQL queries, executed on Neo4j and NebulaGraph 5.x, are semantically equivalent to the original SQL queries and achieve more than a twofold improvement in execution efficiency compared to the SQLite baseline. More details of the execution time comparison are shown in Appendix D.2.3.

6.2 Human Evaluation on NL-GQL Alignment

To evaluate the reliability of the automated data generation pipeline, we conduct a human evaluation via random spot-checking. Specifically, we randomly sample 400 *(NL, GQL)* pairs covering diverse graph patterns and assess their semantic alignment using a Likert scale ranging from 1 to 5, where 1 denotes *completely irrelevant* and 5 denotes *perfectly aligned*. The evaluation yields an average score of 4.78 out of 5.0, indicating a high degree of semantic alignment.

6.3 Evaluation Metrics

Following standard NL2SQL evaluation, we use Execution Accuracy (EX) to determine whether the predicted query retrieves the correct answer:

$$\text{EX}(q, \hat{q}) = \mathbb{I}(V = \hat{V}) \quad (2)$$

where V and $\hat{V}$ denote the execution results of the ground truth query q and the predicted query $\hat{q}$, respectively, and $\mathbb{I}(\cdot)$ is the indicator function. To handle graph-specific data types (i.e., nodes or edges), we extract their unique internal identifiers to verify equivalence. The final metric is calculated as the average EX score across the entire dataset.

Moreover, we adopt BLEU(Papineni et al., 2002) and edit distance(Levenshtein et al., 1966) as complementary metrics, as two semantically different queries may coincidentally produce identical results (e.g., empty result sets) on a given database, thereby inflating execution accuracy. Therefore, structural metrics are necessary to capture syntactic and compositional differences between queries.

6.4 Statistics

Our benchmark is constructed as a large-scale, cross-domain corpus, covering both Cypher and Nebula dialects across diverse databases. Detailed statistics of the dataset are listed in Table 2.

For both Cypher and Nebula dialects, the benchmark provides more than 10,000 queries for both training and testing phases, ensuring sufficient data for robust training and evaluation. The high property density reflects the rich semantic information embedded in the graphs, demonstrating the structural diversity and complexity of the underlying

Split	# Queries	# DBs	Avg. Schema Size per Graph		
			# Node Types	# Edge Types	# Props
Train-Cypher	14,073	211	4.98	4.08	36.27
Train-Nebula	10,182	174	4.40	3.59	30.07
Test-Cypher	13,279	100	3.86	3.55	31.47
Test-Nebula	12,053	86	3.79	4.05	63.07

Table 2: Statistics of GQLBench.

Model	Cypher EX (%)	Nebula EX (%)	Average EX (%)	Cypher BLEU	Nebula BLEU	Average BLEU	Cypher Edit Sim	Nebula Edit Sim	Average Edit Sim
Zero-Shot									
gemini-3-flash	**53.23**	**17.56**	**35.40**	**0.0596**	0.0396	**0.0496**	**0.5264**	**0.5134**	**0.5199**
Qwen3-Max	45.18	15.41	30.30	0.0565	0.0369	0.0467	0.4983	0.4878	0.4931
GPT-4o-mini	35.30	14.35	24.83	0.0508	**0.0453**	0.0481	0.4926	0.4518	0.4722
Qwen3-32B	36.41	11.02	23.71	0.0576	0.0392	0.0484	0.5095	0.4981	0.5038
DeepSeek-V3.2-chat	37.59	8.81	23.20	0.0573	0.0346	0.0460	0.4811	0.4635	0.4723
Qwen3-8B	26.15	16.06	21.11	0.0466	0.0362	0.0414	0.4430	0.4309	0.4370
GPT-OSS-20B	38.72	2.80	20.76	0.0472	0.0095	0.0284	0.4821	0.2112	0.3467
Llama-3.1-8B-Instruct	20.45	10.30	15.38	0.0311	0.0360	0.0336	0.4194	0.4269	0.4232
Fine-Tuned with LoRA									
Qwen3-32B	**31.07**	**30.32**	**30.69(+29.44%)**	**0.2111**	**0.1815**	**0.1963(+305.58%)**	**0.5159**	**0.5582**	**0.5371(+6.61%)**
Llama-3.1-8B-Instruct	28.92	26.92	27.92(+81.53%)	0.1978	0.1716	0.1847(+449.70%)	0.5072	0.5455	0.5264(+24.39%)
Qwen3-8B	29.26	19.58	24.42(+15.68%)	0.1922	0.1671	0.1797(+334.06%)	0.5049	0.5289	0.5169(+18.28%)
GPT-OSS-20B	24.63	21.45	23.04(+10.98%)	0.1701	0.1504	0.1603(+464.44%)	0.4847	0.5116	0.4982(+43.70%)

Table 3: Main results of NL2GQL generation. We calculate Execution Accuracy (EX), BLEU and Edit Similarity (Edit Sim) based on edit distance. **Bold** indicates the best performance in each column within each setting. For all metrics, higher scores indicate greater similarity between the predicted GQL and the ground truth GQL.

schemas. More details about the queries are provided in Appendix B. Table 1 presents a comparison of dataset sizes across benchmarks. GQLBench surpasses all others in terms of the number of entities, indicating its substantially larger scale.

6.5 Main Results

We aggregate the converted data from the training splits of both Spider and BIRD to form our training set, while utilizing the remaining development and test splits, augmented with our synthesized data, as the test set. Table 3 shows the results of NL2GQL generation using different metrics on the test set. To provide a comprehensive evaluation, we conduct experiments under two distinct settings: (1) a Zero-Shot setting, and (2) a Fine-Tuning setting using LoRA (Hu et al., 2022).

Performance in Zero-Shot Setting. Gemini-3-flash exhibits exceptional strength in both dialects, reaching a peak average EX of 35.40%, although its performance drops on the less common Nebula dialect. Generally, zero-shot models struggle with NebulaGraph queries, likely due to the scarcity of Nebula and ISO GQL in their pre-training corpora compared to the widely adopted Cypher. These results highlight the difficulty of our benchmark.

Efficacy of LoRA Fine-Tuning. We fine-tune the model using LoRA for 3 epochs, with rank set to 8, scaling factor $\alpha = 16$, and a learning rate of 1×10^{-5}. LoRA fine-tuning results in substantial performance gains across open-source models. The fine-tuned Qwen3-32B attains an average EX of 30.69%, significantly outperforming its base model (23.71%) and effectively surpassing the proprietary GPT-4o-mini (24.83%). Similarly, Llama-3.1-8B-Instruct achieves a remarkable gain, nearly doubling its EX from 15.38% to 27.92%. The improvement in Nebula shows that fine-tuning bridges the gap for the low-resource dialect, effectively compensating for its scarcity in pre-training corpora. In addition to EX, structural similarity also improves consistently, as evidenced by the gains in BLEU and edit similarity.

Some models (e.g., Qwen3-32B) exhibit a decline in EX on Cypher after LoRA fine-tuning. This observation suggests that, although LoRA

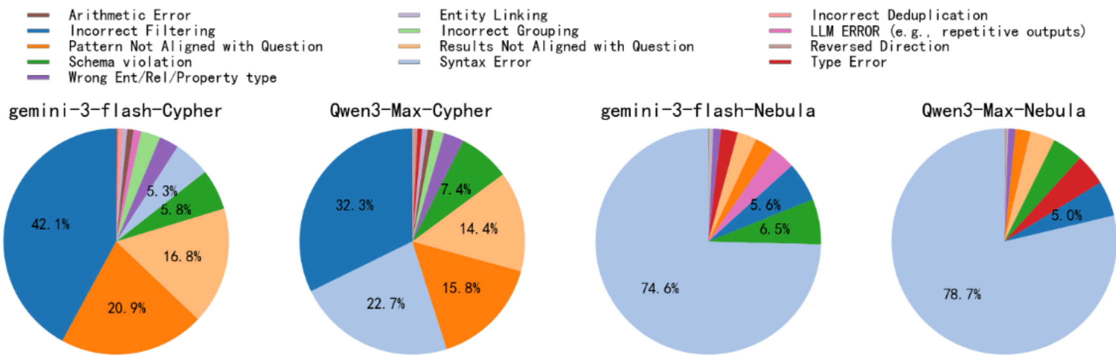

Figure 5: Error Analysis on gemini-3-flash and Qwen3-Max.

effectively reinforces surface-level syntactic patterns and formatting, it may bias the model toward training-specific patterns and schemas, thereby constraining its ability to generalize across domains and query dialects. A detailed error analysis, including a comparison of error patterns before and after LoRA, is provided in the section 6.6.1.

6.6 Error Analysis

Cypher. We conduct an error analysis on gemini-3-flash and Qwen3-Max. As shown in Figure 5, both models struggle with the *Incorrect Filtering*, *Pattern Not Aligned with Question* and *Results Not Aligned with Question* issues. Furthermore, Qwen3-Max exhibits a higher frequency of syntax-related errors in Cypher. These findings indicate that improving NL2GQL performance requires not only enhanced semantic understanding of natural language queries but also stronger modeling of graph query structures and dialect-specific syntax.

Nebula. Our error analysis regarding Nebula on gemini-3-flash and Qwen3-Max reveals that syntax errors constitute more than 70% of the failures. This large proportion of syntactic issues highlights a deficiency in the LLM's understanding of the Nebula and the ISO GQL standard, likely due to limited exposure during pre-training. More details about the definitions and corresponding examples of each error category are provided in Appendix C.

6.6.1 Error Analysis Before and After LoRA

We expand our error analysis to compare our best-performing fine-tuned model (Qwen3-32B) and the most-improved model (Llama-3.1-8B-Instruct) before and after fine-tuning. Detailed results are shown in Table 12. The results shows that:

- **Syntax Errors are Substantially Reduced.** For Nebula, fine-tuning reduces Syntax Errors from 79.61% to 7.60% (Qwen3-32B) and from 74.32% to 8.19% (Llama3.1-8B-Instruct). This proves that while base models struggle with the relatively low-resource Nebula syntax, fine-tuning effectively handles the "language" barrier.
- **Errors Shift to Domain Logic.** As syntax errors are significantly mitigated through fine-tuning, the performance bottleneck shifts toward *Incorrect Filtering* and *Schema Violation*. This transition indicates that while SFT excels at helping LLMs memorize surface-level syntax patterns and formatting, it struggles to instill the underlying graph-native reasoning required for complex queries. The fact that simple SFT fails to resolve these core errors proves that GQLBench is a high-quality, non-trivial dataset that requires more advanced reasoning capabilities rather than mere pattern imitation.

6.7 Fine-Grained Breakdown of Performance across Query Complexity

We include a comprehensive fine-grained breakdown of performance across three key dimensions as detailed in Table 7, 8, and 9: graph patterns, return specifications and advanced functions. This analysis provides deeper insights into the models' EX performance. Details are in Table 4, 5, 6. All models are before LoRA and we randomly sample 4000 questions.

- **Graph Pattern Complexity.** Our results show that LLMs perform relatively well on simple patterns such as single node or edge, with an average EX around 40%. However, as the structural complexity increases, the performance drops significantly. This validates that

Category	gemini-3-flash	Qwen3-Max	GPT-4o-mini	Qwen3-32B	DeepSeek-V3.2-chat	Qwen3-8B	GPT-OSS-20B	Llama-3.1-8B-Instruct	Avg
(node)	50.21%	43.08%	38.67%	33.99%	36.08%	34.13%	30.70%	25.80%	36.58%
(node),(node)	9.62%	8.22%	6.57%	5.87%	7.75%	6.10%	5.16%	4.69%	6.75%
(node)-[edge(optional match)]->(node)	0.00%	25.00%	0.00%	12.50%	0.00%	0.00%	0.00%	0.00%	4.69%
(node)-[edge]	27.41%	29.63%	20.00%	20.74%	26.67%	17.04%	22.96%	20.74%	23.15%
(node)-[edge]-(node)	37.08%	30.72%	24.89%	25.33%	22.11%	18.45%	20.10%	14.27%	24.12%
(node)-[edge]-(node)-[edge]-(node)	34.12%	26.27%	22.94%	20.20%	20.00%	18.04%	18.04%	13.92%	21.69%
[edge]	57.45%	44.68%	34.04%	40.43%	38.30%	29.79%	44.68%	14.89%	38.03%
cycle	12.50%	18.75%	12.50%	6.25%	3.12%	6.25%	3.12%	3.12%	8.20%
long path	20.34%	18.62%	15.47%	13.18%	12.03%	10.89%	11.17%	6.59%	13.54%

Table 4: Fine-grained EX categorized by graph pattern.

Category	gemini-3-flash	Qwen3-Max	GPT-4o-mini	Qwen3-32B	DeepSeek-V3.2-chat	Qwen3-8B	GPT-OSS-20B	Llama-3.1-8B-Instruct	Avg
Aggregate Function	33.72%	30.77%	25.33%	25.87%	27.51%	21.13%	23.62%	18.73%	25.84%
Argmax	35.49%	27.47%	24.57%	24.06%	25.43%	17.92%	18.94%	8.87%	22.85%
Arithmetic	32.46%	20.39%	17.98%	16.67%	10.09%	12.94%	11.18%	6.36%	16.01%
Distinct	31.17%	31.44%	24.12%	23.58%	24.12%	22.49%	21.95%	15.45%	24.29%
Entity	33.56%	23.76%	27.38%	17.18%	17.05%	15.97%	16.24%	19.19%	21.29%
Property	37.32%	31.22%	25.21%	23.90%	23.77%	22.64%	20.15%	13.33%	24.69%
Sort	28.86%	22.34%	17.79%	18.14%	15.98%	14.38%	14.53%	8.97%	17.62%

Table 5: Fine-grained EX categorized by return specification.

Category	gemini-3-flash	Qwen3-Max	GPT-4o-mini	Qwen3-32B	DeepSeek-V3.2-chat	Qwen3-8B	GPT-OSS-20B	Llama-3.1-8B-Instruct	Avg
Groupby Operation	32.26%	6.45%	0.00%	0.00%	51.61%	3.23%	6.45%	22.58%	15.32%
String Manipulation Function	30.99%	26.12%	25.32%	21.75%	21.13%	19.29%	20.52%	13.86%	22.37%
Subquery	4.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.50%
Temporal Function	21.88%	20.83%	13.54%	6.25%	14.58%	14.58%	11.46%	4.17%	13.41%
No Advanced	41.76%	35.66%	29.03%	27.69%	27.20%	24.89%	22.89%	19.28%	28.55%
Set Operation	18.75%	37.50%	12.50%	18.75%	6.25%	12.50%	12.50%	6.25%	15.62%

Table 6: Fine-grained EX categorized by advanced function usage.

GQLBench effectively evaluates a model's structural reasoning and its ability to navigate complex graph topologies.

- **Return Specifications.** The performance variance across categories like Aggregate Function (25.84%) and Arithmetic (16.01%) further demonstrates the benchmark's ability to test the coordination between graph traversal and data post-processing.

- **Advanced Function Challenges.** The data reveals that models achieve their highest performance (28.55%) when no advanced functions are involved. However, when faced with subqueries, performance plummets(0.50% avg). This confirms that our benchmark poses significant challenges in logical composition and functional integration, going far beyond simple keyword translation.

7 Conclusion

In this work, we present **GQLBench**, a large-scale, cross-domain, and cross-dialect benchmark for NL2GQL, with execution support on both Cypher and NebulaGraph GQL, an ISO GQL-aligned dialect. To enable scalable benchmark construction, we develop an automated framework that combines NL2SQL-to-NL2GQL conversion with graph-native data generation, allowing GQLBench to cover both diverse real-world relational schemas and graph-specific reasoning patterns such as long paths and cycles. Our experiments show that GQLBench is highly challenging for current LLMs, while also supporting fine-grained analysis across dialects, graph patterns, return specifications, and advanced function usage. We hope GQLBench can serve as a useful benchmark for future research on robust, generalizable, and dialect-aware NL2GQL systems.

Limitations

One limitation of this work is that, due to the high cost of inference, the evaluation is confined to a representative subset of LLMs. In addition, although GQLBench is designed as a cross-dialect benchmark supporting both Cypher and Nebula, it does not yet encompass other widely used graph query language dialects.

Acknowledgements

This work was supported by grants from the National Natural Science Foundation of China(72595845).

References

Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, Dasha Metropolitansky, Robert Osazuwa Ness, and Jonathan Larson. 2024. From local to global: A graph rag approach to query-focused summarization. *arXiv preprint arXiv:2404.16130*.

Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. 2015. The ldbc social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 619–630.

Yanlin Feng, Simone Papicchio, and Sajjadur Rahman. 2025. Cypherbench: Towards precise retrieval over full-scale modern knowledge graphs in the llm era. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8934–8958.

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*, pages 3973–3977.

Zirui Guo, Lianghao Xia, Yanhua Yu, Tian Ao, and Chao Huang. 2024. Lightrag: Simple and fast retrieval-augmented generation. *arXiv preprint arXiv:2410.05779*, 2(3).

Bernal J Gutiérrez, Yiheng Shu, Yu Gu, Michihiro Yasunaga, and Yu Su. 2024. Hipporag: Neurobiologically inspired long-term memory for large language models. *Advances in neural information processing systems*, 37:59532–59569.

Bernal Jiménez Gutiérrez, Yiheng Shu, Weijian Qi, Sizhe Zhou, and Yu Su. 2025. From rag to memory: Non-parametric continual learning for large language models. *arXiv preprint arXiv:2502.14802*.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Liang Wang, Weizhu Chen, and 1 others. 2022. Lora: Low-rank adaptation of large language models. *Iclr*, 1(2):3.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pages 611–626.

Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, Zhaoqing Suo, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, and 1 others. 2024. Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows. *arXiv preprint arXiv:2411.07763*.

Vladimir I Levenshtein and 1 others. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710. Soviet Union.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, and 1 others. 2023. 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*, 36:42330–42357.

Shunyang Li, Zhengyi Yang, Xianhang Zhang, Wenjie Zhang, and Xuemin Lin. 2021. Sql2cypher: automated data and query migration from rdbms to gdbms. In *International Conference on Web Information Systems Engineering*, pages 510–517. Springer.

Yuanyuan Liang, Keren Tan, Tingyu Xie, Wenbiao Tao, Siyuan Wang, Yunshi Lan, and Weining Qian. 2024a. 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.

Yuanyuan Liang, Tingyu Xie, Gan Peng, Zihao Huang, Yunshi Lan, and Weining Qian. 2024b. Nat-nl2gql: a novel multi-agent framework for translating natural language to graph query language. *arXiv preprint arXiv:2412.10434*.

Ilya Mazein, Adrien Rougny, Alexander Mazein, Ron Henkel, Lea Gütebier, Lea Michaelis, Marek Ostaszewski, Reinhard Schneider, Venkata Satagopam, Lars Juhl Jensen, and 1 others. 2024. Graph databases in systems biology: a systematic review. *Briefings in Bioinformatics*, 25(6):bbae561.

Lunyiu Nie, Shulin Cao, Jiaxin Shi, Jiuding Sun, Qi Tian, Lei Hou, Juanzi Li, and Jidong Zhai. 2022. Graphq ir: Unifying the semantic parsing of graph query languages with one intermediate representation. In *Proceedings of the 2022 conference on empirical methods in natural language processing*, pages 5848–5865.

Makbule Gulcin Ozsoy, Leila Messallem, Jon Besga, and Gianandrea Minneci. 2025. Text2cypher: Bridging natural language and graph databases. In *Proceedings of the Workshop on Generative AI and Knowledge Graphs (GenAIK)*, pages 100–108.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318.

Liana Stanescu. 2021. A comparison between a relational and a graph database in the context of a recommendation system. In *FedCSIS (Position Papers)*, pages 133–139.

Shikhar Vashishth, Prince Jain, and Partha Talukdar. 2018. Cesi: Canonicalizing open knowledge bases

using embeddings and side information. In *Proceedings of the 2018 World Wide Web Conference*, pages 1317–1327.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, and 1 others. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In *Proceedings of the 2018 conference on empirical methods in natural language processing*, pages 3911–3921.

Bowen Zhang and Harold Soh. 2024. Extract, define, canonicalize: An llm-based framework for knowledge graph construction. In *Proceedings of the 2024 conference on empirical methods in natural language processing*, pages 9820–9836.

Ziyu Zhao, Wei Liu, Tim French, and Michael Stewart. 2023. Rel2graph: Automated mapping from relational databases to a unified property knowledge graph. *arXiv preprint arXiv:2310.01080*.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, and Zheyan Luo. 2024. Llamafactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd annual meeting of the association for computational linguistics (volume 3: system demonstrations)*, pages 400–410.

Zijie Zhong, Linqing Zhong, Zhaoze Sun, Qingyun Jin, Zengchang Qin, and Xiaofan Zhang. 2025. Synthet2c: Generating synthetic data for fine-tuning large language models on the text2cypher task. In *Proceedings of the 31st international conference on computational linguistics*, pages 672–692.

Yuhang Zhou, Yu He, Siyu Tian, Yuchen Ni, Zhangyue Yin, Xiang Liu, Chuanjun Ji, Sen Liu, Xipeng Qiu, Guangnan Ye, and 1 others. 2024. r3-nl2gql: A model coordination and knowledge graph alignment approach for nl2gql. In *Findings of the association for computational linguistics: EMNLP 2024*, pages 13679–13692.

A Prompts

A.1 Data Migration

We provide the prompts employed in the Data Migration stage. Prompt 1 guides the LLM to infer edge labels derived from foreign keys, while Prompt 2 determines the direction of these edges. Regarding join tables, Prompt 3 identifies whether a table in the RDBMS functions as a join table, and Prompt 5 facilitates converting these tables into edges. Finally, Prompt 4 performs edge canonicalization.

A.2 Data Generation

We prompt the LLM to generate graph data adhering to the schema and constraints. Specifically, we employ the Prompt 6 to create entity instances, while Prompt 7 is used to construct complex cyclic structures. After constructing the GQL queries, we utilize Prompt 8 to generate natural language questions.

A.3 Evaluation

The evaluation is based on both zero-shot and LoRA settings, both of which utilize the Prompt 9 as input.

B Query Characteristics

GQLBench encompasses a broad spectrum of graph patterns in the `MATCH` clause, alongside diverse output specifications in the `RETURN` clause and a suite of advanced functions and operations, thereby providing a comprehensive evaluation.

B.1 Graph Patterns in `MATCH` Clause

Table 7 provides a taxonomy of representative graph patterns, whereas actual queries may manifest as intricate combinations of these elements.

B.2 Output Specifications in `RETURN` Clause

Table 8 details the diverse output specifications in the `RETURN` clause. GQLBench covers a wide range of return formats, extending from the retrieval of basic entities and properties to complex analytical operations, including sorting mechanisms, argmax, and aggregation functions.

B.3 Advanced Functions and Operations

To assess the model's capability in detailed data processing, GQLBench further integrates a variety of advanced functions and operations, including set operations, temporal functions, string manipulation functions, `Group By` operations, and subqueries. Details are in Table 9.

C NL2GQL Error Types

Drawing inspiration from CypherBench (Feng et al., 2025), we categorize NL2GQL errors into 13 distinct classes. Table 11 details 12 of these, while the remaining category is `LLM ERROR`, which includes generation failures such as infinite repetition.

D Experiment Details

D.1 Data Migration

D.1.1 Experiment Settings

In the data migration phase, we employed the DeepSeek-V3.2-chat model for tasks requiring LLM prompting, configured with a temperature setting of 0, while Qwen-text-embedding-v3 was utilized to generate embeddings. We deployed two distinct graph database systems for our experiments: NebulaGraph 5.2.0 and Neo4j Desktop 2.0.5. The latter was selected specifically because the standard Neo4j Community Edition does not support multiple databases. We selected NebulaGraph because it is one of the most widely adopted graph databases globally, currently ranking second only to Neo4j among dedicated graph databases on DB-Engines[3], surpassing all other ISO GQL implementations. Meanwhile, NebulaGraph 5.x is the pioneer graph database to natively support the ISO GQL standard. As a representative system that actively supports the ISO GQL standard, NebulaGraph provides industry-leading performance and distributed scalability for large-scale storage, making it a highly practical choice for an execution-based benchmark. The migration processes were executed on a server environment equipped with 503 GB memory.

D.1.2 More Details about Annotating Edges by LLM

Given a foreign key constraint `Table_A.a = Table_B.b`, the corresponding edge is constructed via `Table_A.a` and `Table_B.b`. For each row of the join results of `Table_A` and `Table_B`, if `Table_A.a` equals `Table_B.b`, an edge instance is created between the corresponding node instances of `Table_A` and `Table_B`, which are determined by the values of `Table_A.a` and `Table_B.b`. Similarly, for edges derived from join tables, we prompt

[3] https://db-engines.com/en/

the LLM to identify which columns should be used to establish edges. We then utilize an embedding model to map the identified columns in join tables to their corresponding columns in the source and target entity tables by selecting the candidates with the highest embedding similarity, thereby enabling the precise identification of node instances.

D.1.3 Results & Ablation Study

We organized a human evaluation to assess the semantic naturalness of the migrated graph schemas. Participants were asked to assign scores on a scale of 0 to 5, where 0 represents the lowest degree of naturalness and 5 indicates the highest. To provide a comprehensive assessment, we calculated the average scores across all migrated schemas. As illustrated in Table 10, our data migration algorithm consistently maintains semantic naturalness across diverse datasets and achieves the best performance. Furthermore, the ablation study confirms that `identify join tables` and `edge canonicalization` modules contribute to the final performance.

Migration Methods	Naturalness
Ours	4.6
Rel2Graph	3.6
Ours(w/o identify join tables)	4.1
Ours(w/o edge canonicalization)	4.3

Table 10: Performance comparison of different data migration methods and ablation study.

D.2 Query Translation

D.2.1 Details about Join Predicate Analysis

For complex predicates composed of `AND` operators, given the nested structure of the AST, we first recursively flatten the `AND` hierarchy into a linear sequence of independent atomic equalities. Subsequently, for each atomic predicate, the system interrogates the schema to determine if the relationship corresponds to an edge type derived from either a join table or a foreign key constraint. If a valid mapping is identified, the relational join is transformed into the corresponding graph edge. While `INNER JOIN` is directly converted to the graph pattern in the `MATCH` clause, we employ `OPTIONAL MATCH` to simulate `LEFT JOIN` or `RIGHT JOIN`.

The details of converting a `JOIN` to a graph pattern are described in the Algorithm 1.

Algorithm 1 Find_Edge_from_Join

Input: A single SQL JOIN statement S_{join}; Edge schema $\mathcal{E}$, where each entry e specifies the source node src, target node tgt, the pair of columns src_col, tgt_col used to link them and a flag $from_jt$ indicating whether the edge is derived from a join table
Output: $graph\ pattern$
Algorithm:

1: $T_L, T_R, C_L, C_R \leftarrow ParseJoin(S_{\text{join}})$ ▷ L: left, R: right, T: table, C: column
2: **for** $e \in \mathcal{E}$ **do**
3: **if** $e.src = T_L \wedge e.tgt = T_R \wedge e.src_col = C_L \wedge e.tgt_col = C_R$ **then**
4: $r \leftarrow CreateRelation(e.type_name)$
5: **return** $GraphPattern([T_L, r, T_R])$
6: **else if** $T_L = e.src \wedge C_L = e.src_col \wedge e.from_jt$ **then**
7: $r \leftarrow CreateRelation(e.type_name)$
8: **return** $GraphPattern([T_L, r])$
9: **else if** $T_R = e.tgt \wedge C_R = e.tgt_col \wedge e.from_jt$ **then**
10: $r \leftarrow CreateRelation(e.type_name)$
11: **return** $GraphPattern([r, T_R])$
12: **end if**
13: **end for**
14: **return** $[T_L, T_R]$ ▷ Cartesian Product (find no edge)

D.2.2 Details about SQL AST

For the SQL parsing, we leverage sqlglot[4] to generate the SQL AST.

D.2.3 Execution Time Comparison between GQL and SQL

The translated GQL queries, executed on Neo4j, are 2.72 times faster, and on NebulaGraph 5.x, yield a 5.46 times faster speedup compared to the SQLite baseline. These varying speedup ratios stem from the different language implementations of the two databases, which lead to different subsets of successfully translated SQL queries being used for each comparison. To verify these results, we conducted five independent runs for both the SQL and GQL engines. We then performed a t-test on the execution times of these runs. The resulting p-value < 0.01 confirms that the speedup is statistically significant.

D.2.4 Semantic Equivalence in Translation

- **Automated Verification.** To verify the semantic equivalence between the original SQL and the translated GQL, we developed an automated validation pipeline and conducted experiments on the BIRD dev set. We executed both the SQL and the translated GQL, then converted the raw outputs into a unified Python data structure to eliminate formatting discrepancies. Our results demonstrate that

[4]Sqlglot is an existing open-source library, available at https://github.com/tobymao/sqlglot.

for all queries, the execution outputs are 100% identical. However, certain queries, such as those including a LIMIT clause without an ORDER BY, were excluded from automated matching. In these cases, the potentially different results between SQL and GQL do not necessarily imply semantic non-equivalence.

- **Human Evaluation.** To complement our automated verification, we conducted a manual audit to evaluate semantic equivalence from a human perspective. We randomly sampled 500 (SQL, GQL) pairs from the migrated dataset and scored the alignment of their query intents. We employed a 5-point Likert scale, where a score of 5 represents "perfectly identical intent" and 1 represents "completely divergent intent." The evaluation yielded an average score of 4.99/5.0, with 99% of the pairs receiving a perfect score. The very rare cases of semantic divergence occurred in complex SQL queries involving non-standard join conditions (e.g., JOIN A ON A.a=B.b OR A.aa=XXX).

D.3 Data Generation

D.3.1 Details about Templates

`Match` Clause Template. A GQL query typically starts with a `Match` clause, which serves as the fundamental component for defining the target subgraph structure. To ensure comprehensive coverage of graph query capabilities, we systematically implement diverse patterns ranging from node-only patterns to complex traversals:

- **Single-Node and Multi-Node Patterns:** Templates targeting individual nodes or multiple disconnected entities to evaluate basic retrieval performance.
- **Linear Path Patterns:** Templates generating contiguous traversal paths, ranging from single-hop interactions to deep multi-hop chains (e.g., 5 hops). These patterns enforce schema consistency to ensure valid connectivity between consecutive nodes.
- **Cyclic Patterns:** Templates designed to identify closed-loop structures, where the starting and ending nodes of a traversal path are identical.

`Where` Clause Template. The `Where` clause imposes filtering rules on the retrieved subgraph. A critical feature of our generation engine is **Context-Aware Variable Binding**, which ensures that all generated predicates exclusively reference entities and relationships bound in the preceding `MATCH` clause. We construct filtering logic covering three key dimensions:

- **Simple Predicates:** Fundamental constraints involving standard comparison operators (e.g., $=$, $<$, $>$), set membership checks (`IN`), and nullability assertions (`IS NULL`).
- **Composite Boolean Logic:** Complex predicates constructed by combining constraints with logical operators (`AND`, `OR`, `NOT`). The generator supports sophisticated logical structures to rigorously evaluate the LLM's capability in parsing intricate boolean logic.
- **String-Based Constraints:** Advanced filtering conditions derived from complex string manipulations, specifically highlighting operations such as substring extraction and regular expression matching.

`Return` Clause Template. The `Return` clause defines the final projection of the query results. Similar to the filtering logic, our generator enforces context-aware projection, ensuring that the returned fields are strictly derived from the entities and relationships bound in the `MATCH` clause. The templates support the retrieval of nodes and properties, the application of aggregation functions (e.g., `COUNT`, `SUM`) and mathematical calculations, as well as the enforcement of uniqueness via the `DISTINCT` keyword. Furthermore, `ORDER BY` and `LIMIT` clauses are integrated to capture the complexity of varied data constraints.

D.3.2 Experiment Settings

Consistent with the data migration phase, the data generation phase utilized the DeepSeek-V3.2-chat LLM with a temperature setting of 0. During graph structure generation, the hyperparameter for the geometric distribution, denoted as p, was set to 0.5.

D.3.3 Analysis of the Similarity-Based Edge Creation

In practice, we restricted the edge-creation process to the top-10 most similar candidates for each source node and set p to 0.5.

Adhering to the notation established in Equation 1, we derive the expected number of edges from a

source node u as Equation 3. The result of Equation 3 shows that for every possible source node, approximately one edge will be created, thus facilitating the formation of multi-hop semantic chains.

$$\begin{aligned} E(\#\text{ of edges from } u) &= \sum_{r_{uv}=1}^{N} P_{uv} \\ &= \sum_{r_{uv}=1}^{N} p(1-p)^{r_{uv}-1} \\ &= \frac{p(1-(1-p)^N)}{1-(1-p)} \\ &= \frac{p(1-(1-p)^N)}{p} \\ &= 1-(1-p)^N \\ &\approx 1 \quad (\text{for } p = 0.5, N = 10) \end{aligned} \tag{3}$$

D.3.4 An Example of Long Paths and Cycles

Figure 6 shows an example of long paths and cycles in the generated data.

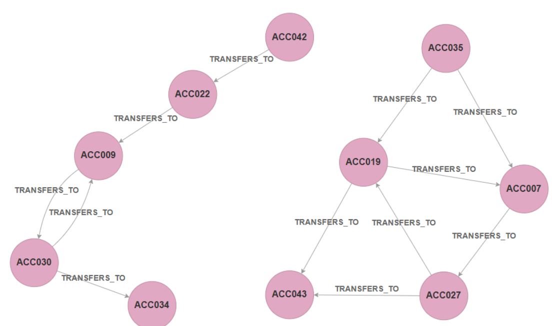

Figure 6: An example of long paths and cycles in the generated data. This figure visualizes a financial transaction network where red nodes represent entities of type `ACCOUNT` and directed edges represent the relationship type `TRANSFERS_TO`.

D.3.5 Dataset Composition & Leakage Analysis

Synthetic queries account for about 70% of the test set and are created exclusively for evaluation, with no overlap with the training set. We ensure that the synthetic test set remains entirely disjoint from the training set to prevent any risk of leakage.

D.4 Evaluation

D.4.1 Experiment Settings

Our evaluation framework leveraged LLaMA-Factory (Zheng et al., 2024) for LoRA fine-tuning tasks and vLLM (Kwon et al., 2023) for efficient, high-throughput inference. Specifically, LLaMA-Factory was employed to fine-tune smaller models, including Qwen3-32B, Llama-3.1-8B-Instruct, GPT-OSS-20B, and Qwen3-8B. We subsequently performed inference on both the base and LoRA-adapted versions of these models using vLLM and LLaMA-Factory. All local training and inference workloads were executed on a single server node equipped with four NVIDIA A800 GPUs. In contrast, for larger models such as gemini-3-flash, Qwen3-Max, GPT-4o-mini, and DeepSeek-V3.2-chat, inference was conducted via their respective APIs.

Graph Pattern	Question	GQL
	How many clubs are there?	MATCH (`club`:`club`) RETURN COUNT(*) AS `_col_0`
	Among the schools with the average score in Math over 560 in the SAT test, how many schools are directly charter-funded?	MATCH (`t1`:`satscores`) MATCH (`t2`:`frpm`) WHERE `t1`.`cds` = `t2`.`cdscode` AND (`t1`.`avgscrmath` > 560 AND `t2`.`charter funding type` = 'Directly funded') RETURN COUNT(`t2`.`school code`) AS `_col_0`
	What is the phone number of the school that has the highest number of test takers with an SAT score of over 1500?	MATCH (`t1`:`satscores`)-[sat_data_for_0:sat_data_for]->(`t2`:`schools`) WHERE `t1`.`cds` = `t2`.`cdscode` ORDER BY `t1`.`numge1500` IS NULL ASC, `t1`.`numge1500` DESC RETURN `t2`.`phone` AS `phone` LIMIT 1
	State the district and region for loan ID '4990'.	MATCH (`t3`:`loan`)-[for_account_0:for_account]->(`t1`:`account`)-[located_in_0:located_in]->(`t2`:`district`) WHERE `t1`.`district_id` = `t2`.`district_id` AND `t1`.`account_id` = `t3`.`account_id` AND (`t3`.`loan_id` = 4990) RETURN `t2`.`a2` AS `a2`, `t2`.`a3` AS `a3`
	Find all circular transfer chains where an account transfers funds through three intermediate accounts and eventually receives money back from the final account, returning the starting account and all three intermediate accounts involved in the cycle.	MATCH (n:`Account`)-[r1:`TRANSFERS_TO`]->(m0:`Account`)-[r2:`TRANSFERS_TO`]->(m1:`Account`)-[r3:`TRANSFERS_TO`]->(m2:`Account`)-[r4:`TRANSFERS_TO`]->(n:`Account`) RETURN n, m0, m1, m2
	List all the reviews which rated a business less than 1	MATCH ()-[review:reviewed]->() WHERE `review`.`rating` < 1 RETURN `review`.`text` AS `text`
	list all the reviews by Niloofar	MATCH (`t2`:`user`)-[t1:reviewed]->() WHERE `t2`.`user_id` = `t1`.`user_id` AND (`t2`.`name` = 'Niloofar') RETURN `t1`.`text` AS `text`",
	What is the average score in writing for the schools that were opened after 1991 or closed before 2000? List the school names along with the score. Also, list the communication number of the schools if there is any.	MATCH (`t2`:`schools`) OPTIONAL MATCH (`t1`:`satscores`)-[sat_data_for_0:sat_data_for]->(`t2`:`schools`) WHERE CASE WHEN `t2`.`opendate` is not NULL THEN apoc.temporal.format(`t2`.`opendate`,'yyyy') ELSE NULL END > '1991' OR CASE WHEN `t2`.`closeddate` is not NULL THEN apoc.temporal.format(`t2`.`closeddate`,'yyyy') ELSE NULL END < '2000' AND `t2`.`cdscode` = `t1`.`cds` RETURN `t2`.`school` AS `school`, `t1`.`avgscrwrite` AS `avgscrwrite`, `t2`.`phone` AS `phone`

Table 7: Various graph patterns. This table presents representative graph patterns in GQLBench. Circles denote nodes, lines denote relationships, dashed lines indicate optional matches, and different node colors represent distinct nodes (i.e., nodes with the same color represent the same node). Note that queries in GQLBench may involve more complex combinations of these patterns.

RETURN Output Specification	Question	GQL
Property	how big is new mexico	MATCH (\`state\`:\`state\`) WHERE \`state\`.\`state_name\`='new mexico' RETURN \`state\`.\`area\` AS \`area\`
Entity	Find all circular transfer chains where an account transfers funds through three intermediate accounts and eventually receives money back from the final account, returning the starting account and all three intermediate accounts involved in the cycle.	MATCH (n:\`Account\`)-[r1:\`TRANSFERS_TO \`]->(m0:\`Account\`)-[r2:\`TRANSFERS_TO \`]->(m1:\`Account\`)-[r3:\`TRANSFERS_TO \`]->(m2:\`Account\`)-[r4:\`TRANSFERS_TO \`]->(n:\`Account\`) RETURN n, m0, m1, m2
Aggregate function	How many clubs are there?	MATCH (\`club\`:\`club\`) RETURN COUNT(*) AS \`_col_0\`
Distinct	What venues are for Neuroscience ?	MATCH (\`venue\`:\`venue\`) WHERE \`venue\`.\`venuename\` = ' Neuroscience' RETURN DISTINCT \`venue\`.\`venueid\` AS \` venueid\`
Sort	List the name of clubs in ascending alphabetical order.	MATCH (\`club\`:\`club\`) RETURN \`club\`.\`name\` AS \`name\` ORDER BY \`name\` IS NULL DESC, \`name\`
Argmax	What is the name of the player with the highest earnings?	MATCH (\`player\`:\`player\`) ORDER BY \`player\`.\`earnings\` IS NULL ASC , \`player\`.\`earnings\` DESC RETURN \`player\`.\`name\` AS \`name\` LIMIT 1
Arithmetic	Find the total checking and saving balance of all accounts sorted by the total balance in ascending order.	MATCH (\`t1\`:\`checking\`) MATCH (\`t2\`:\`savings\`) WHERE \`t1\`.\`custid\` = \`t2\`.\`custid\` RETURN \`t1\`.\`balance\` + \`t2\`.\`balance\` AS \`_col_0\` ORDER BY \`t1\`.\`balance\` + \`t2\`.\`balance\` IS NULL DESC, \`t1\`.\`balance\` + \`t2\`.\` balance\`

Table 8: Various RETURN output specifications.

Advanced Function and Operation	**Question**	**GQL**
Set Operaion	Find the name of the campuses that is in Northridge, Los Angeles or in San Francisco, San Francisco.	MATCH (`campuses`:`campuses`) WHERE `campuses`.`location`='Northridge' AND `campuses`.`county`='Los Angeles' RETURN `campuses`.`campus` AS `campus` UNION DISTINCT MATCH (`campuses`:`campuses`) WHERE `campuses`.`location`='San Francisco' AND `campuses`.`county`='San Francisco' RETURN `campuses`.`campus` AS `campus`
Temporal Function	Name all employees who were hired before year 1990.	MATCH (`employee`:`employee`) WHERE CASE WHEN `employee`.`hire_date` is not NULL THEN apoc.temporal.format(` employee`.`hire_date`,'yyyy') ELSE NULL END < '1990' RETURN `employee`.`fname` AS `fname`, ` employee`.`lname` AS `lname`
String Manipulation Function	What are the dates of the exams whose subject code contains the substring ̈data?̈ Return them in descending order of dates.	MATCH (`exams`:`exams`) WHERE `exams`.`subject_code` =~ '(?i)(?s).*data.*' RETURN `exams`.`exam_date` AS `exam_date ` ORDER BY `exam_date` IS NULL ASC, ` exam_date` DESC
Group By Operation	What are the type of questions and their counts?	MATCH (`questions`:`questions`) WITH `questions`.`type_of_question_code` AS _groupby_col_0, COUNT(*) AS _groupby_col_1 RETURN _groupby_col_0 AS ` type_of_question_code`, _groupby_col_1 AS `_col_1`
Subquery	What are the product price and the product size of the products whose price is above average?	CALL(*) { MATCH (`products_sub0`:`products`) RETURN avg(`products_sub0`.` product_price`) AS `_col_0`} MATCH (`products`:`products`) WHERE `products`.`product_price` > ` _col_0` RETURN `products`.`product_price` AS ` product_price`, `products`.`product_size ` AS `product_size`

Table 9: Various advanced functions and operations.

Error Type	Question	Ground-truth GQL	Predicted GQL
Syntax Error The predicted gql is not executable.	What is the average score in writing for the schools that were opened after 1991 or closed before 2000? List the school names along with the score. Also, list the communication number of the schools if there is any.	MATCH (`t2`:`schools`) OPTIONAL MATCH (`t1`:`satscores`)-[sat_data_for_0:sat_data_for]->(`t2`:` schools`) WHERE CASE WHEN `t2`.`opendate` is not NULL THEN apoc.temporal.format(`t2`.` opendate`,'yyyy') ELSE NULL END > '1991' OR CASE WHEN `t2`.`closeddate` is not NULL THEN apoc.temporal.format(`t2`.` closeddate`,'yyyy') ELSE NULL END < '2000' AND `t2`.`cdscode` = `t1`.`cds` RETURN `t2`.`school` AS `school`, `t1`.` avgscrwrite` AS `avgscrwrite`, `t2`.` phone` AS `phone`	MATCH (s:schools) WHERE s.opendate > date ('1991-12-31') OR s.closeddate < date('2000-01-01') MATCH (sat:satscores)-[: sat_data_for]->(s) WITH s, sat.avgscrwrite AS writing_score, s.phone AS communication_number WHERE writing_score IS NOT NULL RETURN s.school AS school_name, writing_score, communication_number, avg(writing_score) OVER() AS average_writing_score
Wrong Ent/Rel/Property Type The predicted gql uses a valid but not intended entity/relation/property type.	Which active district has the highest average score in Reading?	MATCH (`t2`:`satscores`)-[sat_data_for_0 :sat_data_for]->(`t1`:`schools`) WHERE `t1`.`cdscode` = `t2`.`cds` AND (` t1`.`statustype` = 'Active') ORDER BY `t2`.`avgscrread` IS NULL ASC, `t2`.`avgscrread` DESC RETURN `t1`.`district` AS `district` LIMIT 1	MATCH (s:satscores)-[: sat_data_for]->(sch:schools) WHERE sch.statustype = 'Active' WITH s.dname AS district, AVG(s. avgscrread) AS avgReadingScore RETURN district, avgReadingScore ORDER BY avgReadingScore DESC LIMIT 1
Reversed Direction A relation is used in the opposite direction.	What is the last name and position of the student that bought pizza on 09/10/2019?	MATCH (`t2`:`expense`)-[incurred_by_0: incurred_by]->(`t1`:`member`) WHERE `t1`.`member_id` = `t2`.` link_to_member` AND (`t2`.`expense_date` = '2019-09-10' AND `t2`.`expense_description` = 'Pizza ') RETURN `t1`.`last_name` AS `last_name`, `t1`.`position` AS `position`	MATCH (e:expense)<-[:incurred_by]-(m:member) WHERE e.expense_description CONTAINS 'Pizza' AND e. expense_date = '2019-09-10' RETURN m.last_name, m.position
Entity Linking An entity in the predicted gql does not match the intended entity in the database.	How many connections does the atom 19 have?	MATCH ()-[t:connected_to_atom]->() WHERE CASE WHEN -2-1 < 0 THEN substring (`t`.`atom_id`,size(`t`.`atom_id`) +1+(-2-1)) ELSE substring(`t`.`atom_id `,-2-1) END = '19' RETURN COUNT(`t`.`bond_id`) AS `_col_0`	MATCH (a:atom {atom_id: 'TR%_19 '})<-[:connected_to_atom]-(b: bond) RETURN count(DISTINCT b.bond_id) AS connection_count
Schema Violation The MATCH pattern does not conform to the graph schema (aside from reversed relations). For example, the predicted gql contains a hallucinated entity/relation/property type, or an invalid relation between entities.	Which client issued his/her card in 1994/3/3, give his/her client id.	MATCH (`t3`:`card`) MATCH (`t1`:`client`)-[t2: has_account_disposition]->() WHERE `t1`.`client_id` = `t2`.`client_id `\nAND `t2`.`disp_id` = `t3`.`disp_id`\ nAND (`t3`.`issued` = localdatetime ('1994-03-03T00:00:00')) RETURN `t2`.`client_id` AS `client_id`	MATCH (c:client)-[: has_account_disposition]->(a: account)<-[:for_account]-(card: card) WHERE card.issued = date ('1994-03-03') RETURN c.client_id
Pattern Not Aligned with Question The MATCH pattern conforms to the graph schema, but does not align with the question.	What is the number of SAT test takers of the schools with the highest FRPM count for K-12 students?	CALL(*) { MATCH (`frpm_sub0`:`frpm`) ORDER BY `frpm_sub0`.`frpm count (k-12)` IS NULL ASC, `frpm_sub0`.`frpm count (k -12)` DESC RETURN `frpm_sub0`.`cdscode` AS `cdscode ` LIMIT 1 } MATCH (`satscores`:`satscores`) WHERE `satscores`.`cds` = `cdscode` RETURN `satscores`.`numtsttakr` AS ` numtsttakr`	MATCH (f:frpm)-[:meal_data_for]->(s:schools)<-[:sat_data_for]-(sat:satscores) WITH f, sat, s ORDER BY f.`frpm count (k-12)` DESC WITH collect(DISTINCT s.cdscode) AS schoolIds, max(f.`frpm count (k-12)`) AS maxFrpm MATCH (f2:frpm)-[:meal_data_for]->(s2:schools)<-[:sat_data_for]-(sat2:satscores) WHERE f2.`frpm count (k-12)` = maxFrpm AND s2.cdscode IN schoolIds RETURN s2.cdscode, s2.school, sat2.numtsttakr

Table 11: Error types in the NL2GQL task. Examples are demonstrated using Cypher.

Error Type	Question	Ground-truth GQL	Predicted GQL
Incorrect Grouping The grouping operation in the predicted gql is incorrect.	What are the two most common first names among the school administrators? Indicate the district to which they administer.	CALL (*) {MATCH (`schools_sub0`:`schools`) WITH `schools_sub0`.`admfname1` AS _groupby_col_0, COUNT(`schools_sub0`.`admfname1`) AS _groupby_col_1 RETURN _groupby_col_0 AS `admfname1` ORDER BY _groupby_col_1 IS NULL ASC, _groupby_col_1 DESC LIMIT 2} MATCH (`t1`:`schools`) WHERE `t1`.`admfname1` = `admfname1` RETURN DISTINCT `t1`.`admfname1` AS `admfname1`, `t1`.`district` AS `district`	MATCH (s:schools) WHERE s.admfname1 IS NOT NULL AND s.admfname1 <> '' WITH s.admfname1 AS firstName, s.district AS district, COUNT(*) AS nameCount ORDER BY nameCount DESC LIMIT 2 RETURN firstName, district, nameCount
Incorrect Filtering The filtering operation in the predicted gql is incorrect.	Please list the lowest three eligible free rates for students aged 5-17 in continuation schools.	MATCH (`frpm`:`frpm`) WHERE `frpm`.`educational option type` = 'Continuation School' AND NOT CASE WHEN `frpm`.`enrollment (ages 5-17)`=0 THEN NULL ELSE `frpm`.`free meal count (ages 5-17)` / `frpm`.`enrollment (ages 5-17)` END IS NULL RETURN CASE WHEN `frpm`.`enrollment (ages 5-17)`=0 THEN NULL ELSE `frpm`.`free meal count (ages 5-17)` / `frpm`.`enrollment (ages 5-17)` END AS `_col_0` ORDER BY CASE WHEN `frpm`.`enrollment (ages 5-17)`=0 THEN NULL ELSE `frpm`.`free meal count (ages 5-17)` / `frpm`.`enrollment (ages 5-17)` END IS NULL DESC, CASE WHEN `frpm`.`enrollment (ages 5-17)`=0 THEN NULL ELSE `frpm`.`free meal count (ages 5-17)` / `frpm`.`enrollment (ages 5-17)` END LIMIT 3	MATCH (f:frpm)-[:meal_data_for]->(s:schools) WHERE f.`school type` CONTAINS 'Continuation' AND f.`enrollment (ages 5-17)` > 0 RETURN s.school AS school_name, f.`percent (%) eligible free (ages 5-17)` AS eligible_free_rate ORDER BY eligible_free_rate ASC LIMIT 3
Results Not Aligned with Question The RETURN clause does not align with the question, resulting in incorrect outputs.	List the top five schools, by descending order, from the highest to the lowest, the most number of Enrollment (Ages 5-17). Please give their NCES school identification number.	MATCH (`t2`:`frpm`)-[meal_data_for_0:meal_data_for]->(`t1`:`schools`) WHERE `t1`.`cdscode` = `t2`.`cdscode` ORDER BY `t2`.`enrollment (ages 5-17)` IS NULL ASC, `t2`.`enrollment (ages 5-17)` DESC RETURN `t1`.`ncesschool` AS `ncesschool` LIMIT 5	MATCH (f:frpm)-[:meal_data_for]->(s:schools) RETURN s.ncesschool AS NCESSchoolID, f.`enrollment (ages 5-17)` AS Enrollment ORDER BY f.`enrollment (ages 5-17)` DESC LIMIT 5
Incorrect Deduplication The predicted gql produces duplicated or missing results compared to the ground truth.	How many types of cards does the artist Aaron Boyd illustrated about card art?	MATCH (`cards`:`cards`) WHERE `cards`.`artist`='Aaron Boyd' RETURN COUNT(`cards`.`type`) AS `_col_0`	MATCH (c:cards) WHERE c.artist = 'Aaron Boyd' RETURN COUNT(DISTINCT c.type)
Type Error There is a type error in the predicted gql.	What's the code for the set which was released on 2017/7/14?	MATCH (`sets`:`sets`) WHERE `sets`.`releasedate` = localdatetime('2017-07-14T00:00:00') WITH `sets`.`releasedate` AS _groupby_col_0, `sets`.`code` AS _groupby_col_1 RETURN _groupby_col_1 AS `code`	MATCH (s:sets) WHERE s.releasedate = '2017-07-14' RETURN s.code
Arithmetic Error There is an arithmetic error in the predicted gql, such as division by zero.	Among the schools with an SAT excellence rate of over 0.3, what is the highest eligible free rate for students aged 5-17?	MATCH (`t1`:`frpm`) MATCH (`t2`:`satscores`) WHERE `t1`.`cdscode` = `t2`.`cds` AND (CASE WHEN `t2`.`numtsttakr`=0 THEN NULL ELSE toFloat(`t2`.`numge1500`) / `t2`.`numtsttakr` END > 0.3) RETURN MAX(CASE WHEN `t1`.`enrollment (ages 5-17)`=0 THEN NULL ELSE toFloat(`t1`.`free meal count (ages 5-17)`) / `t1`.`enrollment (ages 5-17)` END) AS `_col_0`	MATCH (f:frpm)-[:meal_data_for]->(s:schools)<-[:sat_data_for]-(sat:satscores) WHERE sat.numge1500 / toFloat(sat.numtsttakr) > 0.3 RETURN MAX(f.`percent (%) eligible free (ages 5-17)`) AS highest_eligible_free_rate

Table 11: Error types in the NL2GQL task (Continued).

Error Type	Llama3.1-8B-instruct (Cypher)	Llama3.1-8B-instruct (Nebula)	Qwen3-32B (Cypher)	Qwen3-32B (Nebula)
Incorrect Filtering	2565 (24.28%) / 5099 (54.02%)	647 (5.98%) / 2992 (33.97%)	2224 (26.34%) / 5111 (55.84%)	545 (5.08%) / 3251 (38.71%)
Schema violation	1365 (12.92%) / 1244 (13.18%)	1341 (12.40%) / 4015 (45.58%)	525 (6.22%) / 1104 (12.06%)	413 (3.85%) / 3278 (39.03%)
Pattern Not Aligned with Question	1051 (9.95%) / 986 (10.45%)	156 (1.44%) / 292 (3.32%)	1015 (12.02%) / 954 (10.42%)	309 (2.88%) / 447 (5.32%)
Syntax Error	4042 (38.27%) / 940 (9.96%)	8035 (74.32%) / 721 (8.19%)	2803 (33.20%) / 865 (9.45%)	8538 (79.61%) / 638 (7.60%)
Wrong Ent/Rel/Property type	378 (3.58%) / 415 (4.40%)	80 (0.74%) / 185 (2.10%)	262 (3.10%) / 311 (3.40%)	92 (0.86%) / 157 (1.87%)
Results Not Aligned with Question	624 (5.91%) / 326 (3.45%)	196 (1.81%) / 263 (2.99%)	1264 (14.97%) / 346 (3.78%)	505 (4.71%) / 282 (3.36%)
Arithmetic Error	64 (0.61%) / 116 (1.23%)	6 (0.06%) / 44 (0.50%)	54 (0.64%) / 43 (0.47%)	14 (0.13%) / 22 (0.26%)
LLM ERROR (e.g., repetitive outputs)	77 (0.73%) / 106 (1.12%)	30 (0.28%) / 50 (0.57%)	1 (0.01%) / 194 (2.12%)	112 (1.04%) / 119 (1.42%)
Incorrect Grouping	75 (0.71%) / 77 (0.82%)	6 (0.06%) / 37 (0.42%)	98 (1.16%) / 91 (0.99%)	16 (0.15%) / 49 (0.58%)
Incorrect Deduplication	9 (0.09%) / 54 (0.57%)	1 (0.01%) / 20 (0.23%)	14 (0.17%) / 66 (0.72%)	0 (0.00%) / 21 (0.25%)
Entity Linking	55 (0.52%) / 44 (0.47%)	43 (0.40%) / 27 (0.31%)	65 (0.77%) / 43 (0.47%)	17 (0.16%) / 23 (0.27%)
Type Error	130 (1.23%) / 27 (0.29%)	264 (2.44%) / 152 (1.73%)	103 (1.22%) / 15 (0.16%)	160 (1.49%) / 101 (1.20%)
Reversed Direction	128 (1.21%) / 5 (0.05%)	6 (0.06%) / 10 (0.11%)	16 (0.19%) / 10 (0.11%)	4 (0.04%) / 11 (0.13%)

Table 12: Error analysis before and after LoRA. Each cell shows error count and percentage before fine-tuning (left) / after fine-tuning (right).

Foreign Key Edge Annotation Prompt

```
You are a database expert. I will show you a relational database schema including the overview of the database and the
description of every table and column.
Then I will give you the `foreign key information` and I want to transform the relational database to graph database. You
should only focus on what edges should be constructed given the foreign keys.
Your task is to analyze and inference the names of the edges and fill them into the `label` field.
Here are the rules you should obey:
    1. if there is a foreign key relationship between 2 tables, you should construct an edge or several edges bewteen the 2
tables.
    2. Do not change anything outside `label` field. Your output must remain the same as the `foreign key information` input
except `label` field, which you are asked to fill.
    3. Make sure that the edge is natural and reasonable. The direction of edges may not be the same as that of the `foreign
key information`. The direction is semantically determined.
### Note: Your output should be the same format as `foreign key information`, but use a list to fill the `label` field. Even
if a table T do not have foreign key, it should be included in the output.

### database_name:
    {{database_name}}
### Overview of the database:
    {{database_overview_json}}
### description of every table and column:
{% for table_name, table_desc in tables-%}
    {{ table_name }}:
    {{ table_desc }}
{% endfor %}
### foreign key information:
    Each item has the following structure:
        table_name: [{'to_table': XXX, 'from_to_columns': [[[XXX],[YYY]]], 'label': null}]
        Here is the explanation:
            table_name: the table which the foreign key starts from
            to_table: the table referenced by the foreign key
            from_to_columns: each item is in the format of [[XXX],[YYY]] indicating that columns in the current table (
from_columns,i.e. [XXX]) references columns in the target table (to_columns,i.e.[YYY]).
            label : name of edge which you should fill,the number of `label` should equal to len(from_to_columns)
    {{fk_info}}

### output json format:
    {table_name: [{'to_table': XXX, 'from_to_columns': [[[XXX,YYY]]], 'label': [edge_name]}]}

### output:
```

Foreign Key Edge Direction Annotation Prompt

```
Here is a database.
### database_name:
    {{database_name}}
### Overview of the database:
    {{database_overview_json}}
### description of every table and column:
{% for table_name, table_desc in tables-%}
    {{ table_name }}:
    {{ table_desc }}
{% endfor %}

Given the following two entities and the semantic meaning of their relationship, determine the more reasonable direction of
the edge between them.
The pattern I give you is (a)-[:edge_name]->(b) or (b)-[:edge_name]->(a). `a` and `b` represent nodes. `edge_name` represents
 the edge between `a` and `b`. The arrow represents if the edge is from `a` to `b` or from `b` to `a`.
if {{dir1}} is more semantically reasonable than the pattern {{dir2}}, output 1. Otherwise, output 0.
You only need to output 0 or 1 without reasoning.
```

Join Table Classification Prompt

```
You are a database expert. I will show you a relational database schema including the overview of the database and the
description of every table and column.
Then I will give you the `foreign key information` and I want to transform the relational database to graph database.
An `Entity Table` typically represents a real-world entity, and should be modeled as a Node in a graph database, while a `
Join Table` typically represents a many-to-many relationship between two or more `Entity Table` and should be modeled as an
Edge or several Edges in a graph database.
### `Join Table` Examples:
user_friends (stores user_id and friend_id to link users).
order_items (stores order_id and product_id to link orders to products).

### database_name:
    {{database_name}}
### Overview of the database:
    {{database_overview_json}}
### description of every table and column:
{% for table_name, table_desc in tables-%}
    {{ table_name }}:
    {{ table_desc }}
{% endfor %}
### foreign key information:
    Each item has the following structure:
        table_name: [{'to_table': XXX, 'from_to_columns': [[[XXX],[YYY]]], 'label': null}]
        Here is the explanation:
            table_name: the table which the foreign key starts from
            to_table: the table referenced by the foreign key
            from_to_columns: each item is in the format of [[XXX],[YYY]] indicating that columns in the current table (
from_columns,i.e. [XXX]) references columns in the target table (to_columns,i.e.[YYY]).
            label : not important which can be ignored
    {{fk_info}}

Please analyze whether table {{table_to_pred}} is a `Join Table`. If table {{table_to_pred}} is a `Join Table`, please output
 1. Otherwise, output 0.
You only need to output 0 or 1 without reasoning.
```

Merge Redundant Edges Prompt

```
You are a database expert. I will show you a graph database schema including some edges from the source node to the target
node.
Your task is to canonicalize the duplicate edges. You should merge multiple edges if they are considered sementically
equivalent or they can be generalized to the same edge.
You should only consider the relationship between source node and target node, `other related node description` is for
reference only.
### source node
{{source}}
### source node description
{{source_desc}}
### target node
{{target}}
### target node description
{{target_desc}}
### other related node description
{% for table_name, table_desc in other_nodes_desc-%}
    {{ table_name }}:
    {{ table_desc }}
{% endfor %}
### edges from source node to target node
{{jt_edges}}
{{fk_edges}}

### output json format:
{origin_edge_label : new_edge_label}
### example
    [{'edge_label':'first_mentorship'},{'edge_label':'second_mentorship'}] should be merged to {'first_mentorship':'
mentorship','second_mentorship':'mentorship'}
### output:
```

Join Table Edge Annotation Prompt

```
You are a database expert. I will show you a relational database schema including the overview of the database and the
description of every table and column.
I want to transform the relational database to graph database.
An `Entity Table` typically represents a real-world entity, and should be modeled as a Node in a graph database, while a `
Join Table` typically represents a many-to-many relationship between two or more `Entity Table` and should be modeled as an
Edge or several Edges in a graph database.
### `Join Table` Examples:
user_friends (stores user_id and friend_id to link users).
order_items (stores order_id and product_id to link orders to products).

I have already identified all the `Join Tables`. Here is the list of `Join Table`: {{jointables}}.
Your task is to :
    For each `Join Table`, analyze what edges should be constructed and provide the following:
        1. The names of the `Join Table`. The `Join Table` must be one of the `Join tables` I provide.
        2. The suggested names of the edges, i.e., edge_label. The edge_label should focus on the relationship between source
 entity and target entity.
        3. The source node and target node for each edge.
            3.1 If multiple fields refer to the same entity type (e.g., multiple employees), treat them as separate roles and
 create multiple edges accordingly, but they may share the same edge_label if they are semantically identical.
                e.g.
                    'mentorship': # edge_label , multiple edges share one edge_label, do not use `first_mentorship`,`
second_mentorship` for edge_label
                    [
                        {
                        'source':{
                            'Entity_type': 'employees',
                            'filter_columns': ['mentor_id']
                        },
                        'target': {
                            'Entity_type': 'employees',
                            'filter_columns': ['first_employee_id']
                        }
                        },
                        {
                        'source':{
                            'Entity_type': 'employees',
                            'filter_columns': ['mentor_id']
                        },
                        'target': {
                            'Entity_type': 'employees',
                            'filter_columns': ['second_employee_id']
                        }
                        }
                    ]
            3.2 Both source node and target node contain `Entity_type` and `filter_columns` to filter the node in the `Join
Table`.
            3.3 Bidirectional edges are allowed. They should be included twice in the output JSON.(e.g. a relate_to b and b
relate_to a)


### database_name:
    {{database_name}}
### Overview of the database:
    {{database_overview_json}}
### description of every table and column:
{% for table_name, table_desc in tables-%}
    {{ table_name }}:
    {{ table_desc }}
{% endfor %}

### Note: Your output should be in the json format as follows without extra explanation.
### output json format:
    {
        join_table_name:
        {
            edge_label:
            [
                {
                'source':{
                    'Entity_type': XXX , # must be one of the `Entity tables`
                    'filter_columns': [XXX,YYY]  # columns in the `Join Table` which can filter the source node
                },
                'target': {
                    'Entity_type': XXX,  # must be one of the `Entity tables`
                    'filter_columns': [XXX,YYY]  # columns in the `Join Table` which can filter the target node
                }
                }
            ]
        }
    }
### output:
```

Node Generation Prompt

```
Generate {current_batch_size} realistic {node_type['name']} instances for the {domain} domain.
Node Type: {node_type['name']}
Properties: {', '.join(prop_names)}
Property Types: {prop_types}

Constraints:
{constraints_info}

Please generate {current_batch_size} diverse, realistic instances that satisfy all constraints.
Return the data as a JSON array, where each object has the properties: {', '.join(prop_names)}.

Example format:
[
  {{
    "property1": "value1",
    "property2": 123,
    "property3": 45.67
  }},
  {{
    "property1": "value2",
    "property2": 456,
    "property3": 78.90
  }}
]

Return only the JSON array, no additional text.
```

Cycle Generation Prompt

```
Generate {node_count_per_type} complete cycle instances for the {schema['domain']} domain.

Cycle Pattern: {cycle_pattern['name']}
Description: {cycle_pattern['description']}
Cycle Path: {' -> '.join(cycle_path)}
Explanation: {cycle_pattern['explanation']}

{cycle_description}

Constraints:
{constraints_info}

Requirements:
1. Generate {node_count_per_type} complete cycle instances
2. Each cycle must be a true instance-level cycle (start and end at the same node instance)
3. Use 'cycle_' prefix for all node and relationship IDs
4. Ensure all unique properties are unique
5. Follow the exact cycle path: {' -> '.join(cycle_path)}

Return the data as a JSON object with structured cycles:
{{
  "cycles": [
    {{
      "cycle_id": "cycle_1",
      "path": [
        {{"type": "NodeType", "id": "cycle_NodeType_0", "properties": {{...}}}},
        {{"type": "RelType", "id": "cycle_RelType_0", "from_node": "cycle_NodeType_0", "to_node": "cycle_NodeType_1", "
properties": {{...}}}},
        {{"type": "NodeType", "id": "cycle_NodeType_1", "properties": {{...}}}},
        {{"type": "RelType", "id": "cycle_RelType_1", "from_node": "cycle_NodeType_1", "to_node": "cycle_NodeType_2", "
properties": {{...}}}},
        ...
        {{"type": "NodeType", "id": "cycle_NodeType_0", "properties": {{...}}}}  // return to the start node
      ]
    }},
    ...
  ]
}}

IMPORTANT:
- Each cycle's path must follow the cycle_path pattern exactly
- The last element in each cycle's path must be the same node instance as the first element
- The path alternates between nodes and relationships: [node, rel, node, rel, ..., node]

Return only the JSON object, no additional text.
```

GQL2NL Prompt

```
You are a professional database query expert. Please generate a natural, fluent English query question based on the given GQL
 query and database schema.

Database Schema:
{schema_desc}

GQL Query:
{gql_query}

Please generate an English query question that:
1. Is natural and fluent
2. Accurately reflects the intent of the GQL query
3. Includes specific business scenario descriptions

Note:
1. Please note the string operations in the `WHERE` clause, such as TOUPPER. You should include the string operation in the
query question rather than only the value of the property.
2. Please note the content in the `RETURN` clause. You should include all the content in the `RETURN` clause in your query
question. If the `RETURN` clause contains entities or relations, you should include them in the query question.


Please only return the query question:
```

Zero-shot Evaluation Prompt

```
You are a graph database expert. I will show you a natural language query and a graph database schema. Your task is to
generate {gql_dialect_str} that is semantically equivalent to the natural language query.
Natural Language Query: {nl_query}
Graph Database Schema: {schema}
Please only return the query in plain text format but not code block format, do not include any other content.
Evidence: {evidence}
```