Towards Database-Free Text-to-SQL Evaluation: A Graph-Based Metric for Functional Correctness

Yi Zhan[1]*, **Longjie Cui**[2]*, **Han Weng**[3]*, **Guifeng Wang**[3], **Yu Tian**[3], **Boyi Liu**[3], **Yingxiang Yang**[3], **Xiaoming Yin**[3], **Jiajun Xie**[3], **Yang Sun**[3]†,
[1]School of Computer Science, Peking University, Beijing, China
[2]Department of Computer Science, The University of Hong Kong, Hong Kong
[3]ByteDance Inc., Beijing, China
zhanyi@stu.pku.edu.cn
longjie.cui@connect.hku.hk
{wenghan, wangguifeng, yutian.yt, boyi.liu01, yingxiang.yang, yinxiaoming, xiejiajun.666, sunyang.46135}@bytedance.com

Abstract

Execution Accuracy and Exact Set Match are two predominant metrics for evaluating the functional correctness of SQL queries in modern Text-to-SQL tasks. However, both metrics have notable limitations: Exact Set Match fails when SQL queries are functionally equivalent but syntactically different, while Execution Accuracy is prone to false positives due to inadequately prepared test databases, which can be costly to create, particularly in large-scale industrial applications. To overcome these challenges, we propose a novel graph-based metric, *FuncEvalGMN*, that effectively overcomes the deficiencies of the aforementioned metric designs. Our method utilizes a relational operator tree (ROT), referred to as *RelNode*, to extract rich semantic information from the logical execution plan of SQL queries, and embed it into a graph. We then train a graph neural network (GNN) to perform graph matching on pairs of SQL queries through graph contrastive learning. FuncEvalGMN offers two highly desired advantages: (i) it requires only the database schema to derive logical execution plans, eliminating the need for extensive test database preparation, and (ii) it demonstrates strong generalization capabilities on unseen datasets. These properties highlight FuncEvalGMN's robustness as a reliable metric for assessing functional correctness across a wide range of Text-to-SQL applications. The source code can be found at https://github.com/Leon00/FuncEvalGMN.

1 Introduction

Text-to-SQL (Zelle and Mooney, 1996) (Zhong et al., 2017) (Qin et al., 2022) is an important task in Natural Language Processing (NLP) that converts queries described in natural language into executable SQL queries. It enables users to interact with structured databases without requiring expertise in SQL syntax. With the advance of Large Language Models (LLMs) and their impressive capabilities, there has been a significant surge in interest towards generating SQL queries directly from natural language prompts (Gao et al., 2023) (Zhang et al., 2023). Consequently, the development of robust evaluation methods that ensure both syntax and functional correctness of the generated SQL queries has become increasingly important.

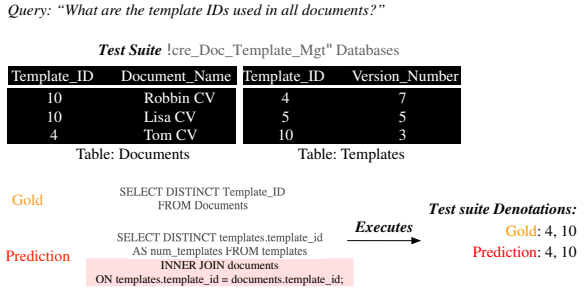


Figure 1: A false positive example may arise from redundant JOIN operators, particularly when the test suite constructs tables with foreign key relationships. Here, the `template_id` in the `Documents` table references the `template_id` in the `Templates` table, ensuring every `template_id` in `Documents` exists in `Templates`. This results in identical test suite denotations between golden and predicted SQL queries. However, they are semantically different, causing a false positive.

Two evaluation metrics are prevalent in existing listerature. The first one is *Execution Accuracy* (Yu et al., 2018), which directly assesses the similarity in functionality of SQL queries by comparing their execution outputs. The second metric, *Exact Set Match* introduced by Yu et al. (2018), is a specific example of a broader range of matching-based met-

*Work was done during the internship at ByteDance.
†Corresponding Author: sunyang.46135@bytedance.com

Proceedings of the 31st International Conference on Computational Linguistics, pages 4586–4610
January 19–24, 2025.

rics. This range includes adaptation of the BLEU metric (Papineni et al., 2002), specifically tailored for code evaluation (Eghbali and Pradel, 2022; Ren et al., 2020). Like others, *Exact Set Match* emphasizes the importance of textual similarity.

Despite their predominance, both aforementioned metric designs have notable limitations. Exact Set Match tends to generate false negatives, as it may incorrectly identify two functionally equivalent SQL queries as different due to their syntactic variations (Zhong et al., 2020). Conversely, Execution Accuracy is prone to generating false positives, identifying functionally different SQL queries as equivalent when their outputs on unit tests match due to inadequately prepared tests or databases.

Since their introduction, various efforts have been made to improve both metrics. For example, Zhong et al. (2020) propose *Test Suite Accuracy* to improve on Execution Accuracy by using a distilled and fuzzed database, achieving comprehensive code coverage and distinguishing neighboring queries. This effectively reduces the false positive rate of the Execution Accuracy metric. However, certain cases, such as those involving redundant JOIN operator (as illustrated in Figure 1), cannot be completely eliminated. On the other hand, researchers transform the query equivalence problem into a constraint satisfaction problem and utilize a generic verifier to determine query equivalence (Zhou et al., 2022; Wang et al., 2022; Chu et al., 2018). Nevertheless, some SQL keywords cannot be converted into equivalent symbolic representations, which limits its applicability to all queries. In summary, designing a well-rounded metric with both low false positive and false negative rates remains an open problem.

In this paper, we propose a novel graph-based evaluation metric that effectively mitigates both false negatives and false positives. Specifically, we parse SQL queries into a Relational Operator Tree (ROT), referred to as *RelNode*. Since RelNode is based on the logical plans of SQL queries, it can simulate SQL query execution without requiring a physical database. Moreover, functionally equivalent SQL queries with different syntactic structures yield similar logical plans. Inspired by RelNode's robust logical representation, we formulate the evaluation of the functional equivalence of SQL query pairs as a graph matching problem. In the following section, we first introduce *RelNode Partial Matching (RelPM)*, a rule-based graph matching approach that serves as a prototype of our idea. To enhance generalizability, we then leverage the Graph Matching Network (GMN) (Li et al., 2019) and incorporate a global positional embedding into its cross-attention mechanism to increase its expressive power. Additionally, we pre-train the GMN using graph contrastive learning, which ensures better generalization and transferability of graph representations.

To facilitate the evaluation of SQL functional correctness, we further develop a dataset called Spider-pair dev, which is derived from Spider (Yu et al., 2018). Each data point consists of a prompt constructed from table schemas and a corresponding question, along with a pair of SQL queries (reference and generated SQL queries), and a binary score (0 or 1) indicating the functional equivalence of the SQL query pair. We evaluate our trained network on the held-out validation set of Spider and on out-of-sample test sets created from WikiSQL (Zhong et al., 2017), BIRD (Li et al., 2024) and Spider-DK (Gan et al., 2021). Compared to other metrics, our trained graph network, FuncEvalGMN, achieves a higher AUC on the held-out validation set and comparable performance on out-of-sample test sets. This highlights possibilities for a robust evaluation metric when test databases are not available.

Contributions. The main contributions of this paper are as follows: (i) We propose a novel graph-based evaluation metric that effectively reduces false negatives and false positives across various datasets and complexity levels. (ii) We innovatively introduce RelNodes to represent SQL queries, ensuring functionally equivalent queries yield similar logical execution plans. (iii) With RelNodes, we introduce a rule-based RelPM method. To enhance generalizability, we use a graph matching neural network with global positional embeddings and pre-train it through graph contrastive learning, which improves its ability to capture differences between graphs and generalize effectively. (iv) We develop the Spider-pair dataset for evaluating SQL functional correctness. Our model, named as *FuncEvalGMN*, achieves superior AUC on validation and external test sets, demonstrating its robustness both with and without databases.

2 Related Work

This section discusses research that focus on code representation with embeddings. It aligns with our approach that incorporates pre-trained models and

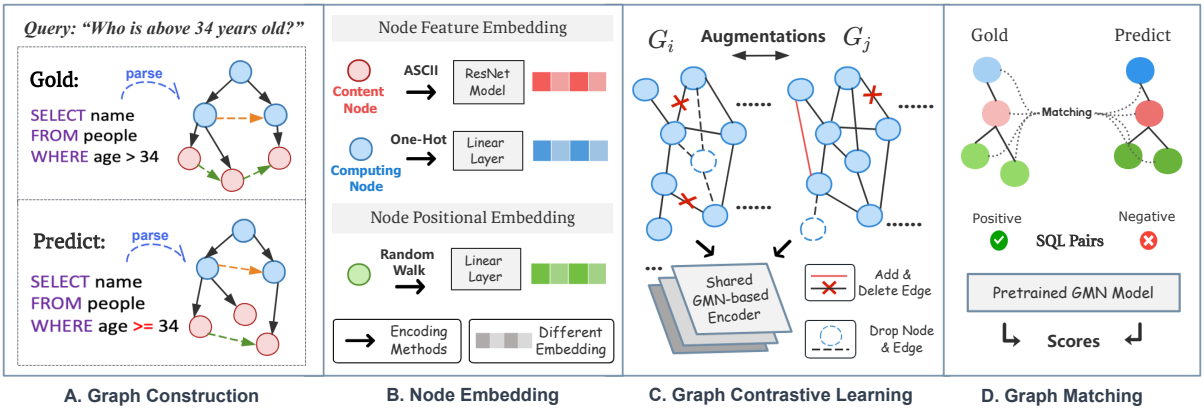


Figure 2: Overview of FuncEvalGMN approach. We first parse the SQL query pairs into graphs and embed their nodes, and then train the Graph Matching Network using graph contrastive learning. Finally, the correctness of the generated SQL is evaluated based on the similarity of the graph representations.

graph-based representations.

Code representation with pre-trained models. Pre-trained models demonstrate exceptional code comprehension, significantly enhancing code evaluation capabilities. CodeBERTScore (Zhou et al., 2023) utilizes CodeBERT (Feng et al., 2020) to encode codes into contextual embeddings and calculates their vector similarity, providing a robust metric for code quality assessment. Meanwhile, CodeScore (Dong et al., 2023) introduces a unified code generation learning framework for pre-trained models to effectively learn from code execution. Additionally, GraphCodeBERT (Guo et al., 2020) incorporates data flow in the pre-training stage. This approach tracks the flow of variables and yields meaningful results on downstream tasks, illustrating the powerful adaptability of these models to diverse coding contexts.

Graph-based code representations. Effectively representing source code while preserving crucial information is essential for various code analysis tasks (Wang and Li, 2021; Tang et al., 2021). Abstract Syntax Trees (ASTs) offer a tree-based representation that emphasizes structural and content-related aspects while omitting specific syntax details. To enhance ASTs, Mi et al. (2023); Wang et al. (2020) integrate explicit semantic edges, such as data flow and control dependencies, creating more comprehensive program graphs.

Beyond ASTs, other graphical representations like Control Flow Graphs (CFGs) (Cota et al., 1994), Data Flow Graphs (DFGs) (Orailoglu and Gajski, 1986), and Program Dependence Graphs (PDGs) (Ottenstein and Ottenstein, 1984) are widely used in program analysis to capture different aspects of program behavior. For example, Fang et al. (2020); Shi et al. (2023) leverage CFGs to address the limitations of ASTs in capturing semantic features and PDGs' coarse granularity in detailed code analysis.

However, SQL lacks the inherent structures provided by CFGs, DFGs, and PDGs. Current methods for SQL primarily focus on extracting its syntax and structure from ASTs (Cao et al., 2023a; Zhuo et al., 2021; Cao et al., 2023b). Given this limitation, a specialized representation is needed to capture SQL's unique logical and data flow characteristics. The Relational Operator Tree (ROT) addresses this by representing SQL queries as a hierarchy of relational operators, effectively capturing both the sequence and structure of the logical execution plan (Cyganiak, 2005). Our research pioneers the combination of ROT with logic and data flow to extract SQL syntax and semantics for more advanced code analysis.

3 From SQL to RelNode

Ren et al. (2020); Eghbali and Pradel (2022) argue that evaluating model-generated code purely as natural language may overlook its complex syntactic structure. Conversely, graph-based methods, as highlighted by Mi et al. (2023); Allamanis et al. (2018); Wang et al. (2020), effectively abstract this syntactic structure without omitting crucial information. In this section, we introduce ROT, which adeptly represents SQL queries based on their logical execution plans. Additionally, we explore a rule-based method utilizing ROT to assess the partial correctness of a SQL query in comparison to the golden SQL query.

Relational Operator Tree (ROT). To convert a SQL query into an ROT, we use relational algebra

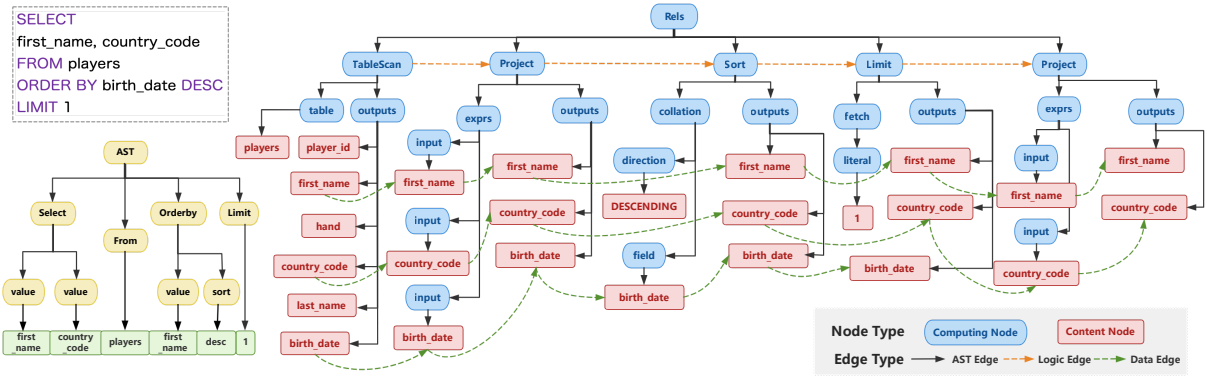


Figure 3: SQL representations using AST (left) and RelNode (right). The AST abstracts SQL purely from a syntactic perspective, while RelNode offers more semantic information from an execution standpoint. In RelNode, orange edges represent logic flows. Connected at the second layer of RelNode, they portray the logical sequence of clause execution. Green-colored data flows represent the pathways of data across various clauses, connecting nodes representing column names.

to parse the query into a sequence of logical operations and construct a tree-structured graph based on their execution order and dependencies (Cyganiak, 2005). By leveraging Apache Calcite (Begoli et al., 2018), we obtain a canonical ROT, known as RelNode. It refines the logic plan through operation reordering and redundant clause elimination, allowing RelNode to uncover similar logical execution patterns beneath varied syntactic forms (an example is shown in Appendix D). To facilitate representation, we categorize RelNode nodes into Computing Nodes and Content Nodes. This distinction arises because Computing Nodes are countable and serve as operators that form the syntactic structure of SQL queries, while Content Nodes are uncountable and represent the query's parameter variables as operands, functioning as the leaf nodes of the tree. Figure 3 shows an example.

RelNode Partial Matching (RelPM). Upon converting the SQL query to the RelNode, we establish an unbiased rule-based partial matching algorithm to serve as the benchmark for the matching-based evaluation (see details in Appendix A). In short, RelPM is a general matching algorithm applied to the tree structure, calculating the score for every matching node. In the Node Matching Algorithm 3, the score of each node is calculated as a weighted sum of the node's own score and the scores of its children. Algorithm 1 is used to compare the attributes of two nodes. The parameter α is a globally adjustable free parameter, subject to fine-tuning to ensure a balanced allocation of matching weights between the root node and its child nodes. Note that Algorithm 3 is asymmetric with respect to S and T. In Algorithm 2, we use an adjustable parameter β to calculate the weighted geometric mean to control the focus on the semantics completion of the source tree.

4 RelNode Graph Matching Network

As a matching-based method, RelPM still tends to fail to recognize identical semantics of SQL with syntactic structure change. Therefore, we propose a novel approach, FuncEvalGMN, based on the graph matching network, to further predict the functional correctness of generated SQLs.

As shown in Figure 3 with a specific SQL query, RelNode already provides a hierarchical organization of relational operators via the RelNode edges. To thoroughly capture syntactic and semantic information of SQL, we integrate both data and logic flows into this graph. Hence, the graph is able to accurately analyze the complex interactions and dependencies within SQL subclauses. Therefore, a SQL query can be represented as a heterogeneous program graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ are nodes consisting of computing nodes and content nodes, while $\mathcal{E}$ are edges including the edges of RelNode, data and logic edges.

4.1 Node Feature Embedding

Computing nodes. In SQL, many equivalent syntactic structures can perform the same functional operations. For instance, a combination of *ORDER BY* and *LIMIT* indicate an equivalent *MAX* or *MIN* operation. Assuming there are M distinct types in *Computing Nodes*, we compute the embeddings of node v with feature $\mathbf{x}_v$ as $h_v^{(0)} = \text{Embedding}(\mathbf{x}_v)$, where the superscript (0) represents the initial state, which will be updated as message propagates through the graph neural network.

Content nodes. The *Content Nodes* serve as placeholders for the query's parameter variables, which correspond to specific elements of the database

schema. One of the key challenges in text-to-SQL is the potential for SQL queries that fails due to incorrect usage of parameter variables, such as table and column names. In order to better represent Content Nodes, we avoid the use of word embedding models, such as *Word2Vec* (Mikolov et al., 2013) and *FastText* (Bojanowski et al., 2017), as they tend to produce similar embeddings for entities that are semantically related but distinct in meaning (e.g., column names 'kid' and 'child'). Instead, we introduce a string-aware embedding method to distinguish different entities. Specifically, we have represented each Content Node as a fixed-length string $S = (s_1, s_2, ..., s_n)$, where each element is encoded as an ASCII value ranging from 0 to 127. This string is then transformed into an n-dimensional vector $X \in \mathbb{R}^{1\times n}$. We then apply one-hot encoding, expanding the vector to $X \in \mathbb{R}^{n\times 128}$. Finally, we adopt a 1D text convolution model (Kim, 2014) with ResNet structure (He et al., 2015) to extract the string-level feature. For the specific architecture, please refer to Appendix B due to page limitations.

4.2 Positional Embedding

Graph positional encoding injects specific location information of nodes into a graph neural network (Dwivedi et al., 2021; Chen et al., 2022; Dwivedi et al., 2022). Actual functions of nodes in our setting vary due to unique hierarchical positions or locations within subtrees. To address such an issue, we incorporate positional encoding, specifically using the Random Walk Positional Encoding (RWPE) (Dwivedi et al., 2021), which is defined using a k-step random walk:

$$p_v^{\text{rw}} = [T_{vv}^1, T_{vv}^2, \ldots, T_{vv}^k] \in \mathbb{R}^k, \quad k \in [K],$$

where $T = AD^{-1}$ is the state transition matrix of random walk. A and D represent the adjacency matrix and degree matrix of the graph. T^k represents the k-th power of T. This method employs a simple random walk matrix approach, concentrating solely on the probability T_{vv} of node v returning to itself. This strategy offers a distinct node representation, based on the premise that each node possesses a unique k-hop topological neighborhood, particularly when k is sufficiently large.

We propose two methods for computing node positional encoding. The first method computes positional encoding for nodes within each graph separately. The second method connects corresponding program graphs through *Rels* and predefined

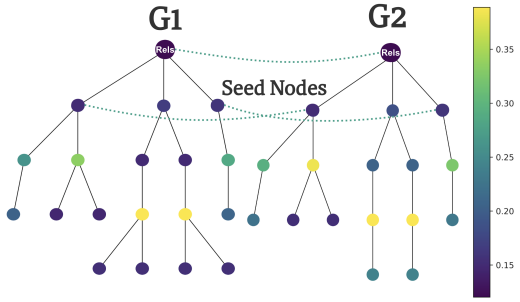


Figure 4: Positional Encoding: We compute positional encoding for graph pairs by connecting Rels nodes and seed nodes across the graphs.

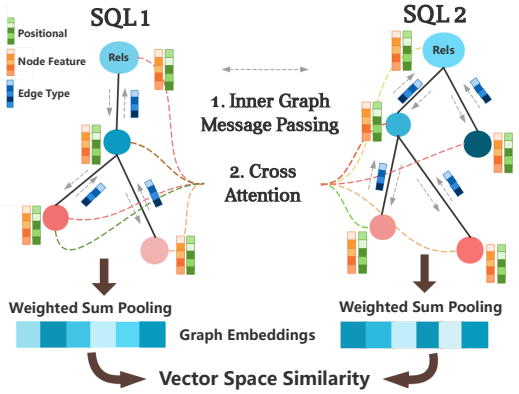


Figure 5: Graph Embedding: Graph pair embedding is computed with integrating inner graph message passing and a cross attention computation.

seed nodes to create a merged graph, as shown in Figure 4. This merged graph allows us to calculate global positional encoding, improving the assessment of functional consistency between the SQLs in terms of local substructures. The *Rels* nodes are the top nodes in each RelNode, and *Seed* nodes are those that match between two RelNodes with their corresponding subtrees S_1 and S_2, where $\text{RelPM}(S_1, S_2) = 1$. The node colors in Figure 4 represent the average probability of self-return after $K = 4$ random walks, highlighting feature similarities for nodes with similar substructures. Additionally, a merged graph provide its nodes with more random walk paths, reducing self-returns and emphasizing hierarchical positions.

4.3 Graph Embedding

Consider two graphs constructed from SQL queries, $\mathcal{G}_1 = (\mathcal{V}, \mathcal{E})$ and $\mathcal{G}_2 = (\mathcal{V}, \mathcal{E})$. We utilize a GMN based on (Li et al., 2019) to generate graph representations and assess their similarity. We focus on GMN's message-passing mechanism (Figure 5) in this section, deferring details to Appendix C.

Inner-graph message passing. The message passed at time t for the neighbor set $N(v)$ of node

v is:

$$m_v^{(t+1)} = \sum_{u \in N(v)} f_{\text{inner}}(h_v^{(t)}, h_u^{(t)}, e_{uv}),$$

where $m_v^{(t+1)}$ represents the message received by node v at setp $t+1$, $h_v^{(t)}$ and $h_u^{(t)}$ are representations of node v and u in step t. e_{uv} is the representation of edge between u and v. We compute the embeddings of each edge $\mathbf{e} \in \{0, 1, 2\}$ as $e_{uv} = \text{Embedding}(\mathbf{e})$, since there are three types of edges (See Figure 3).

Cross-graph message passing. In this stage, cross-attention is used to capture the semantic similarity between two graphs jointly. Specifically, for a node v in graph $\mathcal{G}_1$, we consider $\mathcal{G}_2(v)$ to represent the corresponding set of nodes in graph $\mathcal{G}_2$. Different from inner-graph message passing, the node representation r_v is a nonlinear combination of the node's feature embedding x_v and positional embedding p_v. Furthermore, the node's positional encoding is updated in this process (Dwivedi et al., 2021), giving more expressive node representation on top of Li et al. (2019). Specifically, μ_v is cross-graph message between node v to another graph, which is computed as:

$$\mu_v^{(t+1)} = \sum_{u \in G_2(v)} a_{u \to v}(r_v^{(t)} - r_u^{(t)}), \quad r_v^{(t)} = \text{MLP}(h_v^{(t)} \oplus p_v^{(t)})$$

$$a_{u \to v} = \frac{\exp(s(r_v^{(t)}, r_u^{(t)}))}{\sum_{u \in G_2(v)} \exp(s(r_v^{(t)}, r_u^{(t)}))}, \quad s(r_v, r_u) = \frac{r_v \cdot r_u}{\sqrt{d}},$$

where $a_{u \to v}$ is the attention weight of node v in G_1 to u in G_2, $s(\cdot, \cdot)$ is the similarity function, d is the dimension of the node embedding h, and the factor $1/\sqrt{d}$ follows from that in (Vaswani et al., 2017).

4.4 Graph Contrastive Learning

Given the limited availability of labels in our dataset, we adopt a graph pretraining approach to enhance generalization, following You et al. (2020). We generate positive and negative samples using techniques like node dropping and edge perturbation, and apply the same contrastive loss in You et al. (2020). Further details are in Appendix C.4.

5 Experiment

5.1 Datasets

In this section, we describe the composition of our dataset. Each entry consists of a SQL pair, where the generated SQL query is matched with its corresponding ground truth, labeled as positive (functionally identical) or negative (functionally different) (1/0). Further details on the training dataset can be found in Appendix E.

Training dataset. We construct the training dataset from the Spider dataset (Yu et al., 2018), consisting of approximately 18K SQL query pairs. Keyword statistics are provided in Table 6. We refer to this dataset as Spider-pair train.

Test dataset. Our primary test dataset, named Spider-pair dev, is derived from the Spider dev set, which uses a distinct database not overlapping with the training data. To assess the generalizability of FuncEvalGMN to unseen datasets, we also prepare test datasets from widely used sources, including WikiSQL (Zhong et al., 2017), BIRD (Li et al., 2024), and Spider-DK (Gan et al., 2021), referred to as WikiSQL-pair dev, BIRD-pair dev, and Spider-DK-pair dev, respectively. These datasets range from 0.8K to 3K in size, as shown in Table 4, and are collectively referred to as the test datasets.

Dataset labeling process. The labeling process for the training dataset is outlined in Figure 13 (Appendix E.2) and consists of three steps: (i) Denotation with *test suite*, (ii) GPT-4 Evaluation, and (iii) Human Annotation. The denotation step assigns initial labels to each SQL pair using the *test suite*, GPT-4 performs a consistency check, and human annotators review inconsistent labels to make the final judgment. For the test dataset, each sample is manually labeled.

5.2 Evaluation Results

We validate evaluation metrics against the functional correctness using correlation evaluation metrics such as *AUC*, *Spearman R* (r_s) and *Pearson R* (r_p), as discussed in Appendix I. To comprehensively evaluate our model, we compare our approach with four different kinds of methods.

Matching-based. We baseline the comparisons with commonly used matching-based metrics for code evaluation including CrystalBLEU (Eghbali and Pradel, 2022), CodeBLEU (Ren et al., 2020), as well as our developed RelPM and ASTPM.

Pre-trained model-based. We use our fine-tuned CodeScore and CodeBERTScore as baselines. We use GPT-4 as an evaluator (G-eval) with prompt format from CriticGPT (McAleese et al., 2024).

Equivalence verifier-based. To evaluate from the perspective of SQL equivalence, we compare our approach with the state-of-the-art SQL equivalence verifier, SPES (Zhou et al., 2022). The verifier can identify samples with the same functionality with the ground truths as positives.

Execution accuracy-based. Using *test suite* with augmentation of the existing database up to 20

Method Type	Method	Dataset											
		Spider-pair dev			BIRD-pair dev			Spider-DK-pair dev			WikiSQL-pair dev		
		AUC	τ	r_s	AUC	τ	r_s	AUC	τ	r_s	AUC	τ	r_s
Matching-based	CrystalBLEU	0.6721	0.2424	0.2966	0.7617	0.3130	0.3831	0.7364	0.2963	0.3627	0.7095	0.2869	0.3373
	CodeBLEU (with ROT)	0.8395	0.4852	0.5864	0.7776	0.3917	0.4796	0.7489	0.3241	0.3852	0.6425	0.1884	0.2270
	ASTPM	0.8281	0.4824	0.5718	0.8038	0.3683	0.4457	0.8712	0.4838	0.5723	0.6658	0.2284	0.2667
	RelPM (Ours)	0.8442	0.5028	0.5967	0.8357	0.4054	0.4927	0.8872	0.4929	0.5959	0.7419	0.3427	0.3910
Pre-trained Model-based	CodeBertScore	0.7044	0.2877	0.3522	0.7405	0.2875	0.3521	0.7522	0.3160	0.3869	0.6707	0.2216	0.2713
	CodeScore	0.8637	0.5123	0.6268	0.8051	0.4296	0.5257	0.7107	0.2643	0.3233	-	-	-
	G-eval (GPT-4)	0.6386	0.3504	0.3504	0.7042	0.3685	0.3685	0.7212	0.4103	0.4103	0.6139	0.2857	0.2857
Equivalence Verifier-based	SPES	0.7109	0.4805	0.4805	0.7496	0.5565	0.5565	0.7577	0.5805	0.5805	0.6083	0.2103	0.2103
Execution Accuracy-based	Test Suite	0.9637	**0.9316**	**0.9316**	0.9267	**0.8994**	**0.8994**	0.9277	**0.8958**	**0.8958**	-	-	-
GNN	FuncEvalGMN(Ours)	**0.9750**	0.8529	0.8529	**0.9272**	0.7563	0.7563	**0.9753**	0.8741	0.8741	**0.9910**	**0.9155**	**0.9155**

Table 1: The performance for different methods on four dev Datasets. The best results for each dataset are highlighted in bold, with FuncEvalGMN achieving the highest overall AUC score.

Query Difficulty	Methods	Dataset								
		Spider-pair dev			BIRD-pair dev			Spider-DK-pair dev		
		AUC	τ	r_s	AUC	τ	r_s	AUC	τ	r_s
Easy	FuncEvalGMN	**0.9908**	**0.9291**	**0.9291**	0.9371	0.7745	0.7745	**0.9921**	**0.9227**	**0.9227**
	Test Suite	0.9712	0.9093	0.9093	**0.9381**	**0.9093**	**0.9093**	0.9406	0.9051	0.9051
Medium	FuncEvalGMN	**0.9767**	0.8621	0.8621	**0.9218**	0.7010	0.7010	**0.9898**	**0.9200**	**0.9200**
	Test Suite	0.9554	**0.9113**	**0.9113**	0.8806	**0.8513**	**0.8513**	0.9183	0.8762	0.8762
Hard	FuncEvalGMN	**0.9730**	0.8260	0.8260	0.6810	0.5159	0.5159	0.9490	0.7768	0.7768
	Test Suite	0.9715	**0.9533**	**0.9533**	**0.8871**	**0.8643**	**0.8643**	**0.9643**	**0.9503**	**0.9503**
Extra Hard	FuncEvalGMN	**0.9517**	0.7425	0.7425	-	-	-	**0.9348**	0.7249	0.7249
	Test Suite	0.9494	**0.9317**	**0.9317**	-	-	-	0.8910	**0.8667**	**0.8667**

Table 2: Performance comparison of FuncEvalGMN and *test suite* on different datasets for various query difficulties. The best results for each dataset are highlighted in bold.

iterations, we further reduce false positives, leading to a more accurate assessment of the gold query's functionality. Note that our approach does not rely on any existing database for testing.

Results. As in Table 1, our proposed RelPM significantly surpasses all matching-based methods in terms of AUC.[1] By comparison, ASTPM's performance is lower by about 1.61%. This shows that RelNode can mitigate the impact of syntactic differences. To better exploit the potentials of CodeBLEU, we replace AST with RelNode. However, CodeBLEU only performs coarse-grained subtree matching on syntax trees, and its performance is merely on par with ASTPM. This highlights the superiority of our proposed partial matching algorithm. Additionally, since CrystalBLEU disregards the syntactic structure and semantic information of SQL queries, it performs the worst among all matching-based algorithms.

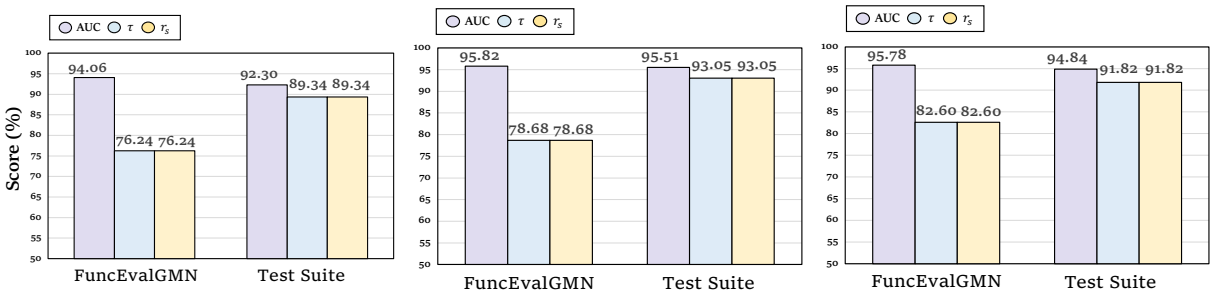
In pre-trained model-based approaches, CodeScore treats codes as token sequences and concatenates the generated code with reference code. A projection head predicts functional consistency between the ground truth and the predicted code. Despite the strength of pre-training, CodeScore scores 11.13% lower than FuncEvalGMN and outperforms non-finetuned G-Eval and CodeBERTScore on Spider-pair dev. However, it performs poorly on Bird-pair dev and Spider-DK-pair dev.

[1]FuncEvalGMN outputs a continuous score.

SPES generates symbolic query representations and establishes equivalence by assessing their containment relationships. However, it lacks full support for all SQL keywords and struggles with case-sensitive strings, leading to an AUC of only 71.09% on Spider-pair dev.

Generalization. Generalization on unseen datasets is of key importance for FuncEvalGMN. To highlight distribution differences, we assess the complexity of Spider-pair dev, BIRD-pair dev, and WikiSQL-pair dev, with keyword distributions shown in Table 6 (see Appendix M for details). BIRD-pair dev has a significantly higher usage of JOIN and WHERE, indicating more complex table and query structures. Despite a drop in correlation on BIRD-pair dev due to this domain gap, FuncEvalGMN, trained only on Spider-pair, still outperforms other metrics, including the *test suite*, suggesting its potential as an evaluation metric for Text-to-SQL applications in general.

In industrial practice, SQL queries are mainly categorized into DDL, DML, and DQL. Given that DDL statements generally have simpler logical structures, our focus is on the more complex DML and DQL forms, particularly SELECT statements, which constitute a substantial part of SQL queries. Our chosen datasets, including Spider and BIRD, cover nearly all commonly used keywords in SELECT queries, such as GROUP BY, JOIN, and ORDER BY, as indicated by Table 6 in Ap-



(a) Join Queries (b) Nested Queries (c) Group By Queries

Figure 6: Performance comparison of FuncEvalGMN and *test suite* across different datasets for Join Queries, Nested Queries, and Group By Queries using the combination of BIRD-pair dev, Spider-pair dev, and Spider-DK-pair dev datasets. It can be observed that the AUC scores of the FuncEvalGMN method are consistently higher than those of *test suite*.

pendix M. The BIRD dataset, in particular, spans 37 specialized domains and closely mirrors real-world contexts, ensuring that the logical forms in our datasets encompass a wide range of SQL query scenarios.

However, beyond merely covering SQL keywords, it is essential to consider the structural complexity that arises in real-world scenarios. The complexity of SQL queries often results from nested subqueries, which correspond to subgraphs in our graph-based representation. Importantly, such nested structures do not significantly affect our model's evaluation capabilities. This is demonstrated by the model's strong performance on the more complex BIRD dataset, despite being trained solely on the simpler Spider dataset (see Table 1).

Additionally, when extending our evaluation to real-world industrial settings, it is important to recognize that such datasets often contain a significantly larger number of tables and columns, resulting in greater structural complexity, which may affect the accuracy of Text-to-SQL generation tasks. However, despite this complexity, the logical structures and syntax of real-world queries remain similar to those found in datasets such as Spider and BIRD. This similarity minimizes the impact of increased structural complexity on our evaluation method, further supporting its effectiveness in diverse and complex real-world SQL scenarios.

Observation. In Figure 6 of Appendix J, we show a comparision of the performance for FuncEvalGMN and *test suite* on queries with important keywords. On queries related to *Join*, our FuncEvalGMN approach is superior to *test suite* on all test datasets. When there are foreign key relationships in certain tables, *test suite* cannot effectively identify these redundancies. A further decomposition into datasets is shown in Table 5. For more information, please refer to Appendix F and Appendix G. However, our approach shows performance deterioration in difficult cases in BIRD-pair dev dataset, which is demonstrated in Table 2. This suggests further fine-tuning data is needed.

5.3 Ablation Studies

This section presents ablation studies on variations of FuncEvalGMN and other graph matching methods. We summarize the results in Table 3.

Type	Method	AUC Score
GMN	(0): ASTGM	0.9205
	(1): *RelNodeGM*	0.9514
	(2): (1) + logic + data	0.9565
	(3): (2) + separated PE	0.9560
	(4): (2) + global PE	0.9707
	(5): (4) + graphCL	**0.9750**
Other GMNNs	(5): (2) + MGMN	0.8839
	(6): (2) + EGSC	0.9127
	(7): (2) + ERIC	0.9303

Table 3: Experiment (0) parses SQL into an AST, whereas (1) parses SQL into a RelNode; (2) incorporates control flow and data flow into the RelNode to capture more semantic information. Experiments for (0), (1), and (2) are conducted on the original version of GMN; (3) and (4) build upon (2), and introduce different Positional Embeddings (PE) into the GMN's cross attention. The PE in (3) is calculated on separate graphs, while the PE in (4) is derived from the merged graph obtained by connecting seed nodes. Experiment (5) introduces graph contrastive learning for model pre-training.

AST Graph Matching (ASTGM). We train FuncEvalGMN based on ASTs.[2] In the Spider-pair dev test dataset achieves an AUC of 92.05%.

RelNode Graph Matching (RelGM). FuncEvalGMN based on RelNode improves performance by 3.09% on Spider-pair dev. Further integrating logic

[2]We adopt this implementation: https://github.com/klahnakoski/mo-sql-parsing.

and data flow edges into RelNode boosts AUC by an additional 0.51%.

RelGM with Positional Encoding. Further enhancements to the GMN include learnable Positional Embeddings (PE) to include structural and positional information. Experiment (3), applying PE to two separate graphs, AUC decreases by 0.05% on Spider-pair dev, which indicates that emphasizing structural differences in SQL pairs with similar syntactic structures fails to provide clear positional inductive bias. In contrast, Experiment (4) calculates global PE on a merged graph pair, leading to gains of 1.47%. Global PE not only highlights differences between nodes at different levels in RelNode but also brings features of potentially matching subtrees closer, thereby facilitating generalization beyond the training dataset.

RelGM with GCL. GCL facilitates unsupervised representation learning on graph-structured data by leveraging contrastive learning. This approach maximizes feature consistency across different augmented views, improving the robustness of graph representations, and leads to a 0.43% performance increase for FuncEvalGMN. Figure 18 shows a faster convergence and significant improvement with this pretraining process.

Other Graph Matching Networks. In addition, we experiment with various Graph Matching Neural Networks (GMNNs), but their performance is inferior to that of GMN. We speculate that in SQL evaluation task, compared to MGNN (Ling et al., 2021), our optimized GMN is better at understanding SQL ROT structure via global positional encoding . Furthermore, compared to EGSC (Qin et al., 2021) and ERIC (Zhuo and Tan, 2022), it can better focus on feature differences between nodes through cross-attention.

6 Discussion

While our metric demonstrates strong performance in evaluating the functional equivalence of SQL queries, it still has limitations. These limitations arise primarily due to two key factors: the lack of column type information and insufficient diversity of training data. In the following, we detail these issues and propose potential solutions.

As shown in Figure 7, the unnecessary CAST to DOUBLE can mislead FuncEvalGMN, as it does not have access to the original data type of the `mpg` column.

As Figure 8 shows, the absence of similar data in the training data set means that our model has not learned that ">2" and ">=3" are semantically equivalent, resulting in false negatives.

Reference SQL:
```
SELECT MAX(CAST(mpg AS DOUBLE))
FROM cars_data;
```
Generated SQL:
```
SELECT MAX(mpg) FROM cars_data;
```

Figure 7: False Positive: Missing Column Information

Reference SQL:
```
SELECT Name FROM country GROUP
BY Name HAVING COUNT() > 2;
```
Generated SQL:
```
SELECT Name FROM country GROUP BY
Name HAVING COUNT() >= 3;
```

Figure 8: False Negative: Insufficient Training Dataset

There are two potential improvements. First, enhancing our graph representation by embedding schema information, such as column data types, key type (foreign keys, primary keys, etc.) and data lineage constraints, could provide the model with critical contextual information. Second, by prompting LLMs to allow for generation of more hard positive/negative cases, a more robust data distribution is provided. We believe these methods will potentially solve issues of the misjudged cases, and we will leave it for future work.

7 Conclusion

This paper presents FuncEvalGMN, a novel graph-based approach for accurate evaluation of functional correctness of text-to-SQL. This method involves transforming SQL queries into a graph that captures its syntactic and semantic essence, using a combination of node types and encoding strategies to represent SQL structures and parameters. We leverage Graph Matching Networks to assess graph similarity, incorporating positional embedding to improve structural understanding. We develop a training and test dataset, for the purpose of training FuncEvalGMN and verifying generalization. Looking forward, we aim to extend this methodology to additional programming languages, exploring its potential to generalize across diverse coding paradigms.

References

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to represent programs with graphs. In International Conference on Learning Representations.

Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J Mior, and Daniel Lemire. 2018. Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In Proceedings of the 2018 International Conference on Management of Data, pages 221–230.

Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. Enriching word vectors with subword information. Transactions of the association for computational linguistics, 5:135–146.

Ruisheng Cao, Lu Chen, Jieyu Li, Hanchong Zhang, Hongshen Xu, Wangyou Zhang, and Kai Yu. 2023a. A heterogeneous graph to abstract syntax tree framework for text-to-sql. IEEE Transactions on Pattern Analysis and Machine Intelligence.

Ruisheng Cao, Hanchong Zhang, Hongshen Xu, Jieyu Li, Da Ma, Lu Chen, and Kai Yu. 2023b. Astormer: An ast structure-aware transformer decoder for text-to-sql. arXiv preprint arXiv:2310.18662.

Dexiong Chen, Leslie O'Bray, and Karsten Borgwardt. 2022. Structure-aware transformer for graph representation learning. In Proceedings of the 39th International Conference on Machine Learning, volume 162 of Proceedings of Machine Learning Research, pages 3469–3489. PMLR.

Shumo Chu, Brendan Murphy, Jared Roesch, Alvin Cheung, and Dan Suciu. 2018. Axiomatic foundations and algorithms for deciding semantic equivalences of sql queries. Preprint, arXiv:1802.02229.

Bruce A Cota, Douglas G Fritz, and Robert G Sargent. 1994. Control flow graphs as a representation language. In Proceedings of Winter Simulation Conference, pages 555–559. IEEE.

Richard Cyganiak. 2005. A relational algebra for sparql. Digital Media Systems Laboratory HP Laboratories Bristol. HPL-2005-170, 35(9).

Xiang Deng, Ahmed Hassan Awadallah, Christopher Meek, Oleksandr Polozov, Huan Sun, and Matthew Richardson. 2021. Structure-grounded pretraining for text-to-sql. In Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. Association for Computational Linguistics.

Yihong Dong, Jiazheng Ding, Xue Jiang, Zhuo Li, Ge Li, and Zhi Jin. 2023. Codescore: Evaluating code generation by learning code execution. arXiv preprint arXiv:2301.09043.

Vijay Prakash Dwivedi, Chaitanya K. Joshi, Anh Tuan Luu, Thomas Laurent, Yoshua Bengio, and Xavier Bresson. 2022. Benchmarking graph neural networks. Preprint, arXiv:2003.00982.

Vijay Prakash Dwivedi, Anh Tuan Luu, Thomas Laurent, Yoshua Bengio, and Xavier Bresson. 2021. Graph neural networks with learnable structural and positional representations. arXiv preprint arXiv:2110.07875.

Aryaz Eghbali and Michael Pradel. 2022. Crystalbleu: precisely and efficiently measuring the similarity of code. In Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, pages 1–12.

Chunrong Fang, Zixi Liu, Yangyang Shi, Jeff Huang, and Qingkai Shi. 2020. Functional code clone detection with syntax and semantics fusion learning. In Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis, pages 516–527.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155.

Catherine Finegan-Dollak, Jonathan K. Kummerfeld, Li Zhang, Karthik Ramanathan, Sesh Sadasivam, Rui Zhang, and Dragomir Radev. 2018. Improving text-to-sql evaluation methodology. In Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). Association for Computational Linguistics.

Yujian Gan, Xinyun Chen, and Matthew Purver. 2021. Exploring underexplored limitations of cross-domain text-to-sql generalization. Preprint, arXiv:2109.05157.

Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2023. Text-to-sql empowered by large language models: A benchmark evaluation. Preprint, arXiv:2308.15363.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. arXiv preprint arXiv:2009.08366.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep residual learning for image recognition. Preprint, arXiv:1512.03385.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 770–778.

Jin Huang and Charles X Ling. 2005. Using auc and accuracy in evaluating learning algorithms. IEEE Transactions on knowledge and Data Engineering, 17(3):299–310.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, Jayant Krishnamurthy, and Luke Zettlemoyer. 2017. Learning a neural semantic parser from user feedback. Preprint, arXiv:1704.08760.

Maurice G Kendall. 1938. A new measure of rank correlation. Biometrika, 30(1/2):81–93.

Yoon Kim. 2014. Convolutional neural networks for sentence classification. Preprint, arXiv:1408.5882.

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.

Yujia Li, Chenjie Gu, Thomas Dullien, Oriol Vinyals, and Pushmeet Kohli. 2019. Graph matching networks for learning the similarity of graph structured objects. In International conference on machine learning, pages 3835–3845. PMLR.

Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2015. Gated graph sequence neural networks. arXiv preprint arXiv:1511.05493.

Xiang Ling, Lingfei Wu, Saizhuo Wang, Tengfei Ma, Fangli Xu, Alex X Liu, Chunming Wu, and Shouling Ji. 2021. Multilevel graph matching networks for deep graph similarity learning. IEEE Transactions on Neural Networks and Learning Systems.

Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. 2023. G-eval: Nlg evaluation using gpt-4 with better human alignment. Preprint, arXiv:2303.16634.

Nat McAleese, Rai Michael Pokorny, Juan Felipe Ceron Uribe, Evgenia Nitishinskaya, Maja Trebacz, and Jan Leike. 2024. Llm critics help catch llm bugs. Preprint, arXiv:2407.00215.

Qing Mi, Yi Zhan, Han Weng, Qinghang Bao, Longjie Cui, and Wei Ma. 2023. A graph-based code representation method to improve code readability classification. Empirical Software Engineering, 28(4):87.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781.

Alex Orailoglu and Daniel D Gajski. 1986. Flow graph representation. In Proceedings of the 23rd ACM/IEEE Design Automation Conference, pages 503–509.

Karl J Ottenstein and Linda M Ottenstein. 1984. The program dependence graph in a software development environment. ACM Sigplan Notices, 19(5):177–184.

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 on Association for Computational Linguistics, ACL '02, page 311–318, USA. Association for Computational Linguistics.

A Pranklin. 1974. Introduction to the Theory of Statistics.

Bowen Qin, Binyuan Hui, Lihan Wang, Min Yang, Jinyang Li, Binhua Li, Ruiying Geng, Rongyu Cao, Jian Sun, Luo Si, Fei Huang, and Yongbin Li. 2022. A survey on text-to-sql parsing: Concepts, methods, and future directions. Preprint, arXiv:2208.13629.

Can Qin, Handong Zhao, Lichen Wang, Huan Wang, Yulun Zhang, and Yun Fu. 2021. Slow learning and fast inference: Efficient graph similarity computation via knowledge distillation. Advances in Neural Information Processing Systems, 34:14110–14121.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. arXiv preprint arXiv:2009.10297.

Chaochen Shi, Borui Cai, Yao Zhao, Longxiang Gao, Keshav Sood, and Yong Xiang. 2023. Coss: leveraging statement semantics for code summarization. IEEE Transactions on Software Engineering.

Ze Tang, Chuanyi Li, Jidong Ge, Xiaoyu Shen, Zheling Zhu, and Bin Luo. 2021. Ast-transformer: Encoding abstract syntax trees efficiently for code summarization. In 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 1193–1195. IEEE.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. Advances in neural information processing systems, 30.

Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. 2020. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), pages 261–271. IEEE.

Yanlin Wang and Hui Li. 2021. Code completion by modeling flattened abstract syntax trees as graphs. In Proceedings of the AAAI conference on artificial intelligence, volume 35, pages 14015–14023.

Zhaoguo Wang, Zhou Zhou, Yicun Yang, Haoran Ding, Gansen Hu, Ding Ding, Chuzhe Tang, Haibo Chen, and Jinyang Li. 2022. Wetune: Automatic discovery and verification of query rewrite rules. In Proceedings of the 2022 International Conference on Management of Data, SIGMOD '22, page 94–107, New York, NY, USA. Association for Computing Machinery.

Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. Sqlizer: query synthesis from natural language. Proceedings of the ACM on Programming Languages, 1(OOPSLA):1–26.

Yuning You, Tianlong Chen, Yongduo Sui, Ting Chen, Zhangyang Wang, and Yang Shen. 2020. Graph contrastive learning with augmentations. Advances in neural information processing systems, 33:5812–5823.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. arXiv preprint arXiv:1809.08887.

John M. Zelle and Raymond J. Mooney. 1996. Learning to parse database queries using inductive logic programming. In Proceedings of the Thirteenth National Conference on Artificial Intelligence - Volume 2, AAAI'96, page 1050–1055. AAAI Press.

Hanchong Zhang, Ruisheng Cao, Lu Chen, Hongshen Xu, and Kai Yu. 2023. Act-sql: In-context learning for text-to-sql with automatically-generated chain-of-thought. Preprint, arXiv:2310.17342.

Ruiqi Zhong, Tao Yu, and Dan Klein. 2020. Semantic evaluation for text-to-sql with distilled test suites. arXiv preprint arXiv:2010.02840.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. Preprint, arXiv:1709.00103.

Qi Zhou, Joy Arulraj, Shamkant B Navathe, William Harris, and Jinpeng Wu. 2022. Spes: A symbolic approach to proving query equivalence under bag semantics. In 2022 IEEE 38th International Conference on Data Engineering (ICDE), pages 2735–2748. IEEE.

Shuyan Zhou, Uri Alon, Sumit Agarwal, and Graham Neubig. 2023. Codebertscore: Evaluating code generation with pretrained models of code. arXiv preprint arXiv:2302.05527.

Wei Zhuo and Guang Tan. 2022. Efficient graph similarity computation with alignment regularization. Advances in Neural Information Processing Systems, 35:30181–30193.

Zhongliu Zhuo, T Cai, Xiaosong Zhang, and Fengmao Lv. 2021. Long short-term memory on abstract syntax tree for sql injection detection. IET Software, 15(2):188–197.

A RelNode Partial Matching (RelPM)

RelPM calculates the matched nodes and their scores for each RelNode, using the F-beta score to determine the similarity between the two SQLs. We divide the process into three parts: Node Matching, RelNode Scoring, and Similarity Evaluation.

Algorithm 1 ROT Node Comparison

```
function CALC(node, Node)
    if node.val == Node.val then
        return 1
    end if
    return 0
end function
```

Algorithm 2 RelPM Score

```
Let S be the source tree.
Let T be the target tree.
function RELPM(S, T)
    recall ← NODEMATCH(S, T)
    precision ← NODEMATCH(T, S)
    F_β = (1+β²)×precision×recall / (β²×precision+recall)
    return F_β
end function
```

Algorithm 3 Node Match

```
procedure NODEMATCH(S, T)
    m_root ← CALC(S, T)
    if S or T is a leaf node then
        return m_root
    else
        scores ← an empty list
        for each child s in S.children do
            m ← 0
            for each child t in T.children do
                m ← max(m, (
                    (NODEMATCH(s, t)))
            end for
            scores.add(m)
        end for
        m_children = Sum(scores) / Length(scores)
        return m_root * α + (1 − α) * m_children
    end if
end procedure
```

A.1 Node Matching

When matching two SQL queries' Relational Operator Trees, we can designate one as the source tree and the other as the target tree, then use depth-first search to find their matching node sets. When evaluating the match for the target tree, the source tree is treated as a reference to measure how well the nodes of the target tree align with it. The success of node matching is determined by scoring against l candidate matching nodes, with the highest-scoring node being selected as the final match. The formula for this process is $m = \max\{m^j\}, j \in \{0, 1, ...l\}$ where m^j represents matching score of node n with a candidate matching node n_j in another tree, which can be calculated as:

$$m^j = \alpha \times m^j_{\text{self}} + (1-\alpha)\frac{\sum_{i=0}^{N} m^j_{\text{child}^i}}{N} \quad \alpha \in (0,1)$$

The m^j_{self} represents the matching score of the node itself with the candidate matching node, and m^j_{child} indicates the maximum matching score of child node between two trees. The calculation of m_{child} is recursive, following the same method as m, until it reaches leaf nodes. Additionally, as Figure 9 illustrates that α serves as a weighting coefficient to balance the significance between the scores of a node and those of its children. A node colored in red signifies a successful match, whereas a gray node denotes an unsuccessful one. When α falls below 0.5, the emphasis shifts towards the matching outcomes of the child nodes, leading us to circumvent the "OR" and "AND" logic in favor of matching a greater number of child nodes. Conversely, when α exceeds 0.5, the focus is placed on matching the parent node, resulting in the failure to match for all corresponding subtree nodes beneath it.

$$m^j_{\text{self}} = \begin{cases} 1 & \text{if val}(n) = \text{val}(n_j) \\ 0 & \text{otherwise} \end{cases}$$

For a given node, its matching score is determined by the candidate matching node n_j. If the attributes of both nodes are identical, it is considered that a match can be established.

A.2 RelNode Scoring

For trees that exhibit matching information, we adjust the weights of clauses and key nodes according to their importance, thereby determining the overall score. The scoring equation is defined as:

$$s = \omega \times s_{\text{self}} + \frac{\sum_{i=0}^{N} \omega^i \times s^i_{\text{child}}}{N},$$

$$\text{where} \quad \omega + \textstyle\sum_{i=0}^{N} \omega^i = 1, \quad s_{\text{self}} = \sigma, \quad \sigma \in (0,1)$$

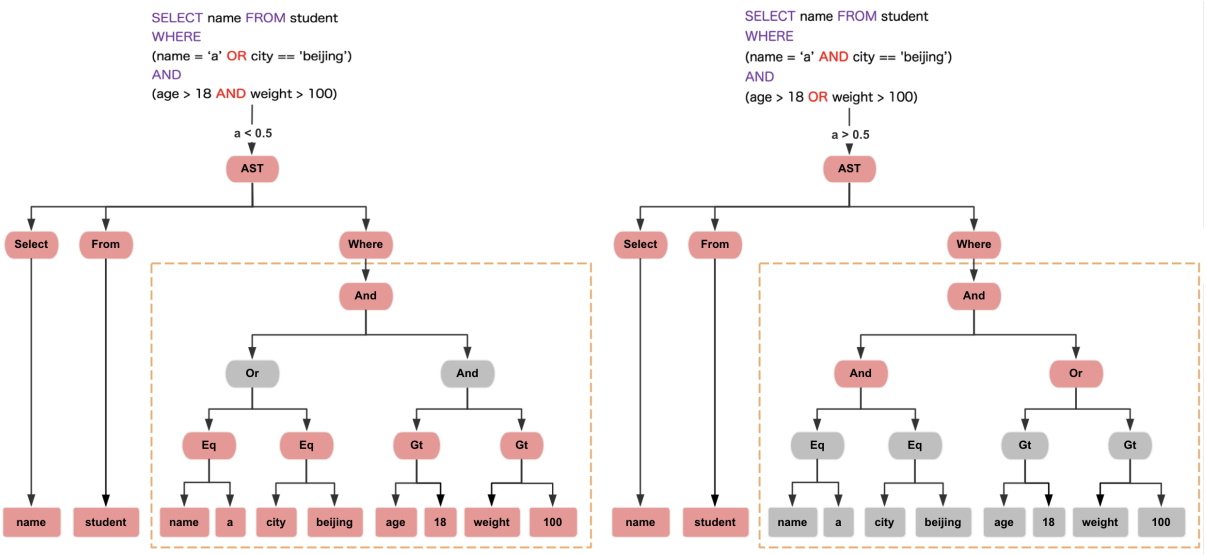


Figure 9: Partial Matching

Here, ω represents the weighting factor, and s_{self} denotes the node's own score, which is calculated during the process of matching stage. The calculation starts from the root node and proceeds recursively, combining the scores of the node itself and its children through a weighted sum.

A.3 Similarity Evaluation

By performing a cross-comparison between the source and target trees, "Precision" calculates the percentage of nodes in the source tree that successfully find matches in the target tree, while "recall" measures the percentage of nodes in the target tree that are matched in the source tree. We compute the weighted geometric mean,

$$F_\beta = \frac{(1+\beta^2) \times \text{Precision} \times \text{Recall}}{\beta^2 \times \text{Precision} + \text{Recall}}$$

When assessing code generation models, our main focus is on verifying if the generated code fully aligns with the semantics of the reference code. In this context, the reference code acts as the target tree, while the generated code represents the source tree. Hence, we give priority to recall in our evaluation, indicating that we assign a relatively high value to β.

B Content Node Feature Embedding

Figure 10 depicts the process of representing a *Content Node*, using "abc_cba" as an example. Initially, the string is encoded into a one-dimensional vector through ASCII encoding. Following this, each element of the vector is transformed via one-hot encoding. As a result, "abc_cba" is encoded into a 64×128 matrix. In this context, 64 denotes the string's length, with any deficiency up to 64 being compensated by zero padding. The 128 dimension reflects the ASCII one-hot encoding's dimensionality. To enhance the characterization of *Content Nodes*, we used a ResNet (He et al., 2016) model. This model comprises eight residual blocks (organized into 4 layers), with each block primarily employing 1D convolutional layers as its core components.

C RelNode Graph Matching Network

In addition to message passing, the other implementation details of GMN are as follows:

C.1 Update Function

The update function integrates all gathered messages to update each node's representation at every iteration step. This process is mathematically formulated as:

$$h_v^{(t+1)} = f_{\text{update}}(h_v^{(t)}, m_v^{(t+1)}, \mu_v^{(t+1)}),$$

In this equation, $h_v^{(t+1)}$ is the updated representation of node v at step $t+1$. The function f_{update}, which in our implementation is a Gated Recurrent Unit (GRU), updates the node's feature representation using its previous state $h_v^{(t)}$, the inner-graph message $m_v^{(t+1)}$, and cross-graph communication $\mu_v^{(t+1)}$.

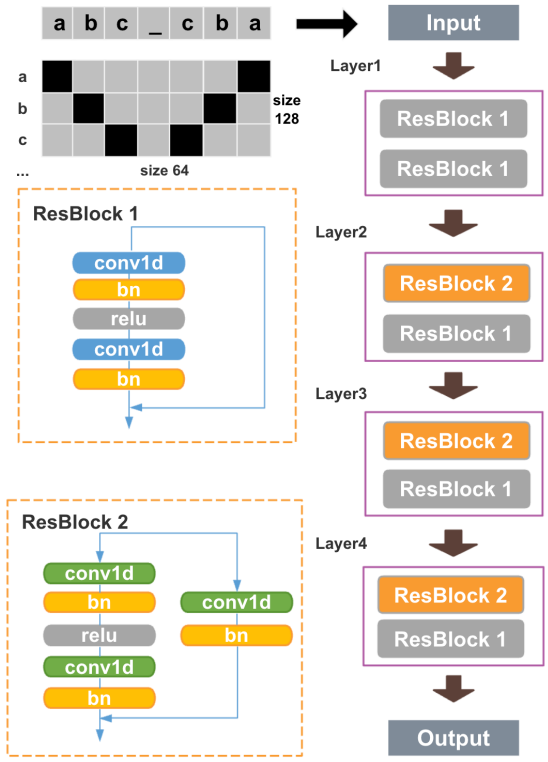


Figure 10: The Architecture of Content Node Feature Embedding Model.

C.2 Aggregator

Aggregation is to calculate a representation for the entire graph. Following T propagation steps, an aggregation function processes the set of node representations to produce a graph-level representation h_G. The aggregation method we utilize is proposed in Li et al. (2015):

$$h_G = MLP_G\left(\sum_{v \in G(v)} \sigma(MLP_{\text{gate}}(h_v^{(T)})) \odot MLP(h_v^{(T)})\right)$$

In this method, a gated weighted sum, which aggregates information across all nodes and filters out irrelevant data, proves to be more effective than a simple summation.

C.3 Similarity Metric

Once we obtain the graph representations, h_{G_1} and h_{G_2}, for the graph pair (G_1, G_2), we evaluate their similarity using a vector space metric. Suitable metrics include Euclidean, cosine, or Hamming similarities. Here, we employ the Euclidean distance as the similarity metric, which is defined as:

$$s(h_{G_1}, h_{G_2}) = \|h_{G_1} - h_{G_2}\|_2.$$

C.4 Graph Contrastive Learning

Inspired by the work of GraphCL (You et al., 2020), we have attempted various graph augmentation strategies on each RelNode graph, for instance, node dropping, edge perturbation, attribute masking and subgraph. The details of these augmentation strategies are given below:

Node dropping. For graph G, a specified number of random nodes and their incident edges will be removed. The dropout probability for each node is determined by an independent and identically distributed (i.i.d.) uniform distribution by default.

Edge perturbation. The connectivity of graph G will be disrupted by randomly adding or removing a certain proportion of edges. We also utilize an independent and identically distributed (i.i.d.) uniform distribution to decide whether to retain or remove each edge.

Attribute masking. The attribute masking requires the model to use the contextual information from the remaining known attributes to predict the masked vertex attributes.

Subgraph. We use random walk to find the subgraph within the graph G and use this subgraph as an augmentation graph to compare with other augmented graphs.

According to the findings reported in GraphCL, it was observed that the combination of multiple augmentation strategies yields superior gains in results. As such, we conducted several experiments with different combinations and ultimately identified the most effective combination, which involves applying attribute masking with a ratio of 0.1 on node features and node dropping with a ratio of 0.1. Therefore, for a batch of N graphs, $2N$ augmented graphs are generated, we choose the augmented graph from the same original graph as the positive samples, while the other $N-1$ graphs in the batch serve as negative samples. Noting that after the augmentation, the representation of each sample is still obtained through the graph matching network.

D Equivalent Conversion Capabilities of RelNode

Calcite converts relational algebra expressions generated by the parser into an execution plan, applying several optimization rules in the process. These optimization rules enable equivalent transformations of relational expressions, such as Predicate Pushdown, Constant Folding and Column Pruning.

In our work, we utilize execution plans gener-

```
SELECT COUNT(*)
FROM has_pet
JOIN student ON has_pet.stuid = student.stuid
JOIN pets ON has_pet.petid = pets.petid
WHERE pets.pettype = 'dog' AND student.sex = 'F'
```

```
SELECT COUNT(*)
FROM has_pet
JOIN ( SELECT * FROM student WHERE student.sex = 'F' ) as s1
ON has_pet.stuid = s1.stuid
JOIN ( SELECT * FROM pets WHERE pets.pettype = 'dog' ) as p1
ON has_pet.petid = p1.petid
```

Figure 11: An Example of Predicate Pushdown

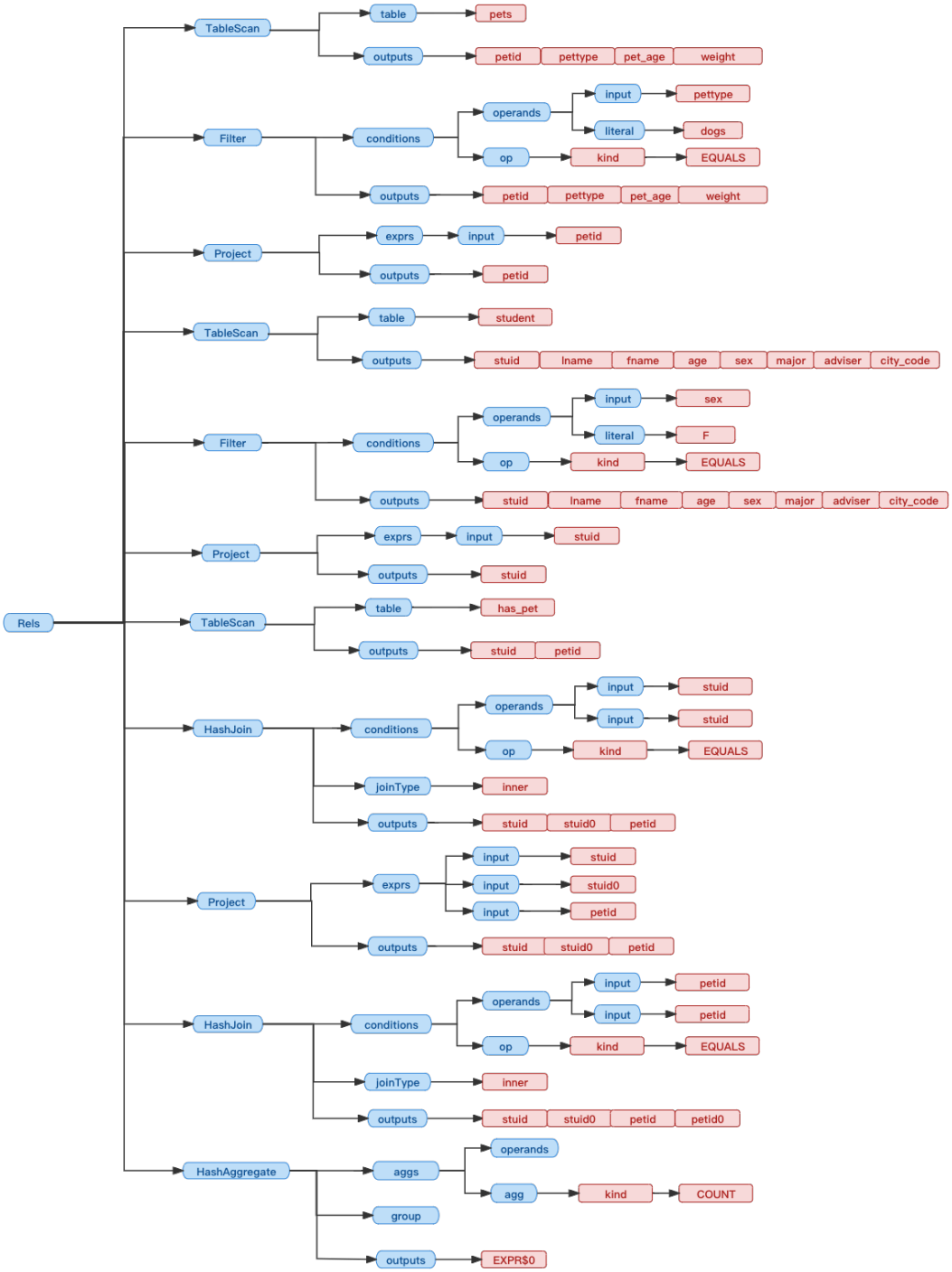


Figure 12: The RelNode Structure of above SQLs.

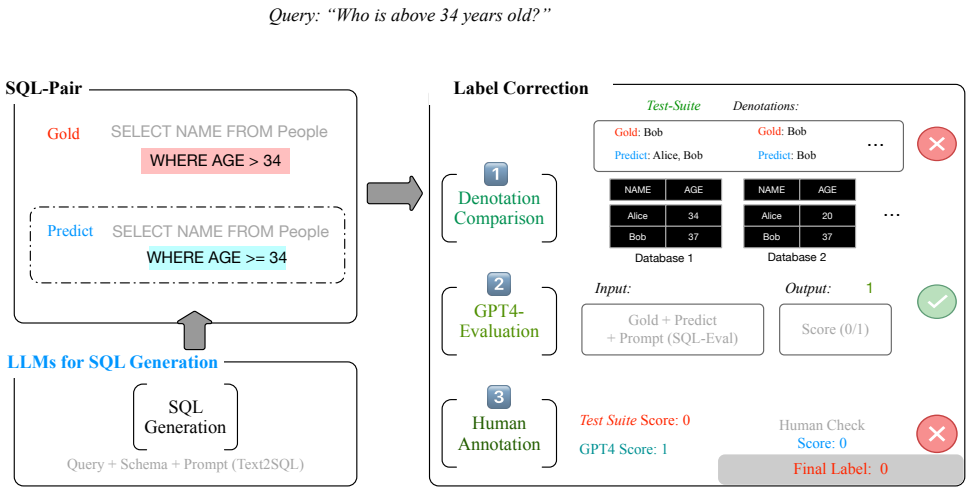


Figure 13: Dataset processing pipeline: 1) Generate predicted SQL queries using large language models. 2) Pair each generated SQL query with its corresponding ground truth SQL query to form a SQL-pair. 3) Apply label correction operations to associate each pair with a label indicating whether the predicted SQL query has the same functionality as the ground truth SQL query.

ated by Calcite to construct graphs. It abstracts the syntactic structure of SQL and provides rich semantic information from the perspectives of logical execution and variable usage. Furthermore, its optimization of execution plans standardizes SQL, uncovering the same functionality under different syntactic structures, thereby reducing the difficulty of judgment in determining functional equivalence.

In the following discussion, we explore a case of predicate pushdown optimization. Predicate pushdown is a strategy that involves relocating predicates from the `WHERE` clause of an outer query block to a more granular query block where the predicate is relevant. This approach enables earlier data filtering and enhances index utilization. Figures 11 demonstrates two SQL queries that are functionally equivalent, while Figure 12 illustrates their identical RelNode.

The initial SQL query conducts a θ-join across the `has_pet`, `student`, and `pets` tables, followed by applying a filter based on the conditions `pets.pettype = 'dog'` and `student.sex = 'F'`.

Through predicate pushdown optimization, the overarching condition of the outer query is decomposed into sub-conditions that are applied directly within the inner queries. Consequently, in the modified SQL, the condition `student.sex = 'F'` is applied to the `student` table, and `pets.pettype = 'dog'` is applied to the `pets` table, prior to executing a θ-join on these tables. This optimization allows for more efficient data processing by leveraging early filtering and improved index performance.

E Spider-pair Dataset

Text-to-SQL refers to the process of translating natural language queries into precise SQL commands (Yu et al., 2018; Iyer et al., 2017; Deng et al., 2021; Yaghmazadeh et al., 2017; Finegan-Dollak et al., 2018). Numerous datasets, including Spider (Yu et al., 2018), have been developed for this purpose, where each entry comprises a reference SQL query, a corresponding natural language question, and the relevant database. However, there is no dataset available to validate the consistency between the quality of generated SQL code and evaluation metrics. Our proposed Spider-pair fills this gap. It consists of a training set and a testing set, which we refer to as train and dev. Each entry in the dataset includes a pair of SQL queries (reference and generated SQL), a prompt, and the functional correctness of the generated SQL. In the following sections, we will introduce our construction approach.

E.1 SQL Pairs Auto-generated by LLMs

To generate SQL pairs, we utilize Spider (Yu et al., 2018) as our source dataset. It comprises $10,181$ queries and $5,693$ unique, complex SQL queries spanning 200 databases across 138 distinct domains. We utilized $8,659$ examples from 146 databases as our train set, while the test set contains $1,034$ examples from 20 different databases. This separation ensures fairness in evaluation by having

LLM	BIRD-pair dev	Spider-DK-pair dev	WikiSQL-pair dev
Code Llama 7b	-	262	102
Code Llama 13b	152	-	-
GPT-3.5	701	304	-
GPT-4	915	-	207
GPT-4 32k	748	228	-
DeepSeek 1.3b	-	298	152
DeepSeek 6.7b	387	291	129
DeepSeek 33b	-	283	159
Llama2 13b	74	211	74
Total	2977	1877	823

Table 4: Number of Predicted SQL Generated by Each LLM for BIRD-pair dev, Spider-DK-pair dev and WikiSQL-pair dev Datasets

distinct databases for train and test sets.

As the capabilities of Large Language Models (LLMs) continue to advance, surpassing human performance in various tasks, we leverage them as an intermediary to generate SQLs. We carefully designed the prompts formed from the questions and the Data Definition Language (DDL) of the required databases. The DDL is crafted from the schemas of all tables within our database, where each schema outlines the table's structure in markdown format. The LLMs used to generate SQLs include GPT-3.5, GPT-4, and CodeLLaMA-13B.

For executable SQL queries, we use 1 to indicate fidelity to the prompt instructions and 0 to indicate the opposite. Alternatively, 1 signifies equivalence in functionality to the reference SQL, while 0 indicates non-equivalence.

E.2 Data Labeling Process

We utilize three components for this process: original database, test suite augmentation, and distillation for the original database which involves language model evaluation and human annotation to ensure the quality of labels.

- **Denotation Comparison**: Using the *test suite* from (Zhong et al., 2020), SQL pairs with exactly the same denotations, up to column permutations are labeled as 1 (functionally consistent) and 0 (functionally inconsistent) otherwise. This helps us to eliminate a decent amount of *false positives*.

- **GPT4 Evaluation**: GPT-4 is known to have strong capacity of evaluation of other language model (Liu et al., 2023) on various tasks. We leverage GPT-4 as a judge for the generated predicted SQL. When GPT-4 determines that the SQL pairs are functionally identical, it will output 1; otherwise, it will output 0. The prompt is shown in the Appendix N.2.

- **Human Annotation**: Although the *test suite* has a high code coverage, it might still introduce false positives as shown in the Appendix F. Therefore, we further check samples with inconsistent labels derived from GPT-4 Evaluation and *test suite*, and finally perform manual verification to ensure the accuracy of the labels.

E.3 Data distribution of the test dataset

In order to evaluate the generalization ability of the model on unseen dataset, we select Spider-DK, BIRD, and WikiSQL datasets to prepare the test dataset. Spider-DK is built based on the Spider development set. It introduces five different domain knowledge to modify the query or query-SQL pair of the Spider dataset to ensure that the new expression conforms to the domain knowledge required by the existing Spider samples and does not cause ambiguity (Gan et al., 2021). WikiSQL is a large-scale hand-annotated semantic parsing dataset containing 80,654 samples covering 24,241 tables in Wikipedia. In the WikiSQL dataset, each sample includes a table, an SQL query, and the natural language question corresponding to the query (Zhong et al., 2017). The BIRD dataset is a comprehensive benchmark designed for text-to-SQL tasks, containing 12,751 SQL query pairs and 95 databases across 37 professional domains. It emphasizes the quality of database values and the challenges of generating efficient and accurate SQL queries in large-scale database environments, making its SQL

Reference SQL:

```
SELECT COUNT(DISTINCT Template_ID) AS EXPR$0 FROM Documents
```

Generated SQL:

```
SELECT COUNT(DISTINCT templates.template_id)
AS num_templates FROM templates
INNER JOIN documents ON templates.template_id =
documents.template_id;
```

Figure 14: False positive case a

Reference SQL:

```
SELECT id FROM TV_Channel GROUP BY Country, id HAVING
COUNT(*) > 2
```

Generated SQL:

```
select id from tv_channel group by id having count(1) > 2
```

Figure 15: False positive case b

queries more complex than those in Spider-DK and WikiSQL (Li et al., 2024). Specifically, the Spider-DK dataset contains 535 query-SQL pairs, the BIRD validation set includes 1201 query-SQL pairs, and the WikiSQL validation set contains 8243 query-SQL pairs. To ensure the fairness of the experiment, we keep the sample size of the three datasets roughly consistent. Therefore, we extract 502 query-SQL pairs from the WikiSQL validation set to prepare the test dataset. We generated predicted SQL queries through several different large language models. The number of predicted SQL generated by each LLM is shown in the Table 4.

We combine these generated predicted SQL with ground truth SQL into pairs. The labeling process is the same as in section 5.1, but it is worth noting that WikiSQL cannot perform database enhancement through *test suite*, so we directly execute SQL queries in WikiSQL to determine the label of the SQL pair. Finally, we still use GPT-4 to find possible false positive cases, and then get the confirmed label after manual review.

F False positive cases on *test suite*

In this section, we will introduce two typical cases of false positives after using test suite and discusses the causes of false positives.

In the first case, the key difference between the two SQL queries is that the second query includes an additional inner join. Specifically, the second query counts unique `template_id` in the `documents` table that also have matching entries in the `templates` table. In contrast, the first query directly counts unique `template_id` from the `documents` table without checking for their presence in the `templates` table, providing the count of all unique `template_id` used in the `documents` table, regardless of whether they exist in the `templates` table.

The execution outcomes are consistent across the augmented database because every `template_id` found in the `documents` table also exists in the `templates` table. However, if there are template ids within the `documents` table that lack corresponding entries in the `templates` table, the execution outcomes of the two SQL queries will differ.

For the second case, the semantic difference between the two SQL queries is that the first SQL's group by parameter is less than the second SQL's. The first SQL will return all ids that appear more than two times in the `tv_channel` table. The second SQL will return all ids that appear more than two times for each (`country`, `id`) combination in the `tv_channel` table. Therefore, if an `id` appears more than twice but belongs to different countries, and the (`country`, `id`) combination does not appear more than twice, the execution results of the two SQL queries will be different.

G False negatives cases on *test suite*

Reference SQL:

```
SELECT COUNT(*) FROM concert
WHERE CAST(year AS INTEGER) IN
(2014, 2015);
```

Generated SQL:

```
SELECT COUNT(concert_id) FROM
concert WHERE year IN ('2014',
'2015');
```

Figure 16: False Negative: Insufficient Training Dataset

As shown in Figure 16, since test suite randomly generated data may include entries like "2014FAS" in the year column, causing the two queries to produce different outputs and leading to false negatives.

H Attention Visualization For Explanation

In this section, we analyze a hard positive case that two SQLs are functionally equivalent but differ in their syntactic structure. We visualize the initial and final attention graphs propagated by our FuncEvalGMN to examine the its capacity to identify key features within the graph. Furthermore, by observing the modifications in the attention graph from initial to final propagation, we elucidate

how FuncEvalGMN detect and match nodes with strong correlations. This analysis demonstrates that our model is adept at recognizing equivalent substructures and functional nodes across both graphs, thereby facilitating a thorough comprehension of the SQL's syntactic structure and semantics.

H.1 Attention Map from the First Propagation

Figure 17a shows a notable feature: attention is primarily focused on the `ts_date` and `MAX` nodes. This is because `ts_date` is the field retrieved by the SQL query, and `MAX` is the logic used to extract it. Together, they essentially define the core functionality expressed by the SQL query. This indicates that our model has successfully captured the key semantics of SQL from the beginning.

H.2 Attention Map From the Final Propagation

Figure 17b displays the attention map from the model's final propagation, where we can identify several key features:

1. **Equivalent Substructures Captured**
   It is observed that all corresponding nodes of the `TableScan` subtree, except for the `other_details` node, are successfully matched by attention edges in both graphs. This phenomenon can be explained from two perspectives:
   a. In terms of similarity, the `TableScan` subtree, being a common element in both SQL queries, exhibits the highest degree of similarity.
   b. Furthermore, nearly all nodes within the `Project`, `Sort`, and `Limit` subtree in the left graph are matched with attention edges to the `HashAggregate` subtree in the right graph. This is because the combination of `Project`, `Sort`, and `Limit` operations in the left graph is functionally equivalent to the `HashAggregate` operation in the right graph. From this analysis, it is evident that our model possesses a strong capability to extract equivalent functional structures from entirely different structures.
2. **Equivalent Functional Nodes Captured**
   Observation reveals that the `LAST` node in the `Sort` subtree of the left graph draws attention to the `MAX` node in the right graph. This is because the operation of sorting in descending order and selecting the last data entry is equivalent to directly taking the `MAX` in an aggregate function. Additionally, `ts_date`, as the field resulting from the SQL execution, is precisely captured: the `1` node within the `limit` subtree in the left graph draws attention to the `ts_date` node in the right graph. This occurs because using `limit` to retrieve the last record extracts the `ts_date` field, and this node, being the only `ts_date` node in the right graph, is accurately identified.

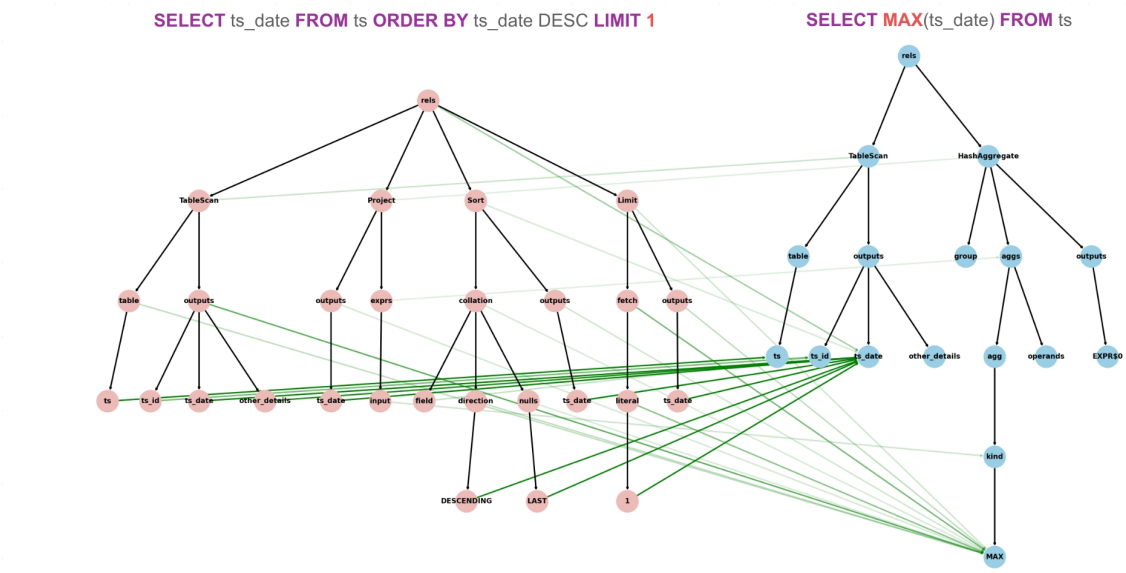
H.3 Attention Map Comparison

In the following, we compare the attention maps from the initial and final propagations to analyze the trends in the attention map changes throughout the model's propagation process, ultimately discerning the capabilities and characteristics of the model's attention component in feature extraction.

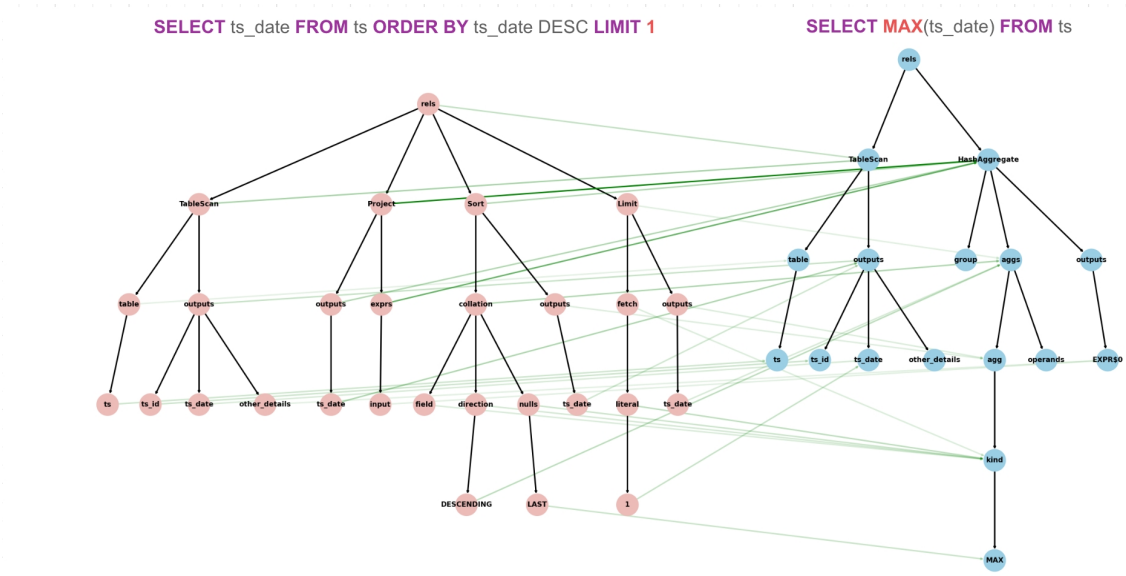
Initially, the node embeddings have only been processed by the encoder layer and have not yet integrated neighborhood information and structural features. At this stage, the model quickly captures the SQL's core features, `ts_date` and `MAX`, but overlooks other SQL details. However, after the final propagation, by observing the direction and opacity of the attention edges, we can discern that the attention map exhibits the following four characteristics:

1. The distribution of attention is more uniform.
2. The attention weights are more balanced.
3. Equivalent substructures within the two graphs are captured.
4. Functionally equivalent nodes across different structures are identified.

These observations indicate that as propagation progresses, the model begins to consider a broader range of features within the SQL graph, moving beyond the initial focus on key elements to a more comprehensive understanding of the SQL's structure and semantics. Additionally, the model is capable of extracting functionally consistent information from both equivalent functional nodes and equivalent substructures. This capability is, to some extent, due to the use of the relational operator tree (ROT), as each subtree (substructure) within the ROT represents an atomic functional operation in the execution plan.



(a) Attention map from the first propagation



(b) Attention map from the final propagation

Figure 17: Attention maps from different propagation steps. In the graph, the labels on the nodes denote their content. Black edges illustrate the connections in the RelNode, while green edges represent attention links. The transparency level of the green edges reflects the magnitude of the attention weights. Attention links are drawn from nodes in the left graph to the node that receives the highest attention out of all nodes in the right graph.

I Correlation Evaluation

The performance of different models can be evaluated using the following metrics:

Area Under the Curve (AUC) (Huang and Ling, 2005) refers to the area under the Receiver Operating Characteristic (ROC) curve. The ROC curve is a graphical representation that illustrates the diagnostic ability of a binary classifier system as its discrimination threshold is varied.

$$AUC = \int \mathrm{TPR}(\xi)\mathrm{FPR}'(\xi)\, d\xi$$

where $TPR(\xi)$ is the true positive rate and $FPR(\xi)$ is the false positive rate at threshold ξ, and $FPR'(\xi)$ takes the differentiation with respect to the threshold.

Spearman R (r_s) (Pranklin, 1974) is a nonparametric measure of rank correlation, which assesses the statistical dependence between the rankings of two variables or data sets:

$$r_s = \frac{\mathrm{cov}(\mathrm{R}(Y^1), \mathrm{R}(Y^2))}{\sigma_{\mathrm{R}(Y^1)}\sigma_{\mathrm{R}(Y^2)}},$$

where $\mathrm{cov}(\mathrm{R}(Y^1), \mathrm{R}(Y^2))$ expresses the covariance between the rankings of Y^1 and Y^2, represented by $\mathrm{R}(Y^1)$ and $\mathrm{R}(Y^2)$ and σ refers to the standard deviation.

Kendall-Tau (τ) (Kendall, 1938) assesses the relationship between two rankings by measuring the ordinal or rank correlation between a given variable and a reference measurement. The formula is:

$$\tau = \frac{Concordant - Discordant}{Concordant + Discordant},$$

where $Concordant$ is the number of pairs for which the two measurements agree on their relative rank. Conversely, $Discordant$ counts the pairs in which the two measurements demonstrate conflicting ranking orders.

J Test Set AUC Trends During Training

AST Graph Matching (ASTGM). As shown in Figure 18, the green dashed line represents the method using Abstract Syntax Trees (AST) for SQL parsing and graph partial matching. The initial AUC is 87%, with slow growth, eventually stabilizing around 90%. This curve shows significantly weaker performance compared to other experiments, indicating that AST struggles to capture SQL's rich semantic information.

Relnode Graph Matching (RelGM). Compared to Experiment (0), the blue solid line uses RelNode for SQL parsing, starting with a higher AUC than ASTGM and stabilizing around 94% after 100 epochs. RelNode, based on SQL's logical execution plan, reduces discrepancies caused by different syntactic structures expressing the same functionality.

Logic and Data Flow in RelNode. The red dashed line with triangles represents the experiment where control and data flow are added to RelNode to capture richer semantic information. It performs similarly to Experiment (1) in the early epochs (50-100) but achieves a slightly higher AUC, stabilizing at 94.5%.

Positional Encoding (PE). Incorporating separate PE led to a significant fluctuation early in training, but performance gradually declined, showing signs of overfitting. The final result was 0.05% lower than Experiment (3), indicating that focusing on structural differences in SQL pairs with similar syntactic structures fails to provide a clear positional inductive bias. In contrast, Experiment (4), which uses global graph PE by calculating node positional encodings across connected graphs, achieved steady improvement throughout the training process.

RelGM with GCL. The orange curve represents the model with Graph Contrastive Learning (GCL). It enhances unsupervised representation learning for graph-structured data, maximizing feature consistency across augmented views. This improves the robustness of graph representations, resulting in 0.43% performance boost for FuncEvalGMN.

K Performance on join queries, nested queries and group by queries in different datasets

In Table 5, we show a comparision of the performance for FuncEvalGMN and *test suite* on queries with important keywords. On queries related to *Join*, our FuncEvalGMN approach is superior to *test suite* on all test datasets, which reveals weakness of *test suite*. When there are foreign key relationships in certain tables, *test suite* cannot effectively identify these redundancies. For more information, please refer to Appendix F. However, our approach shows performance deterioration in difficult cases in BIRD-pair dev dataset, which was not observed in Spider-pair dev. This suggests further finetuning data is needed.

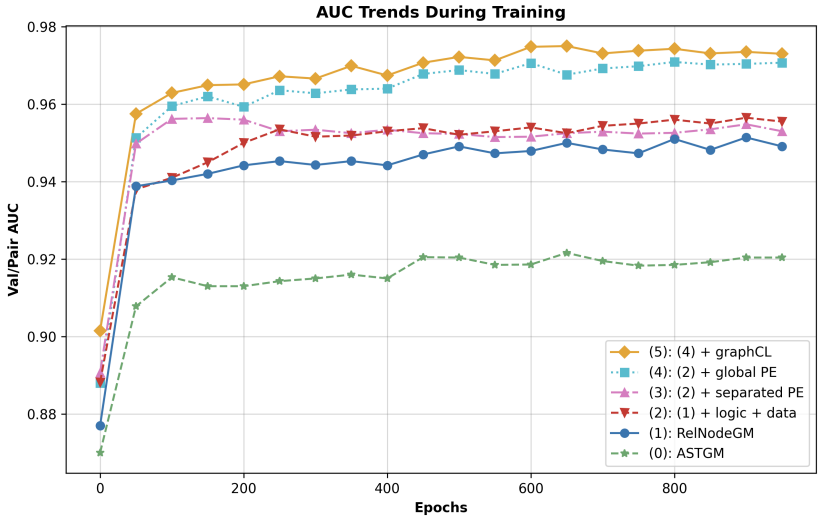


Figure 18: AUC Trends in test dataset

Keywords	Methods	Dataset								
		Spider-pair dev			BIRD-pair dev			Spider-DK-pair dev		
		AUC	τ	r_s	AUC	τ	r_s	AUC	τ	r_s
Join Queries	FuncEvalGMN	**0.9580**	0.7782	0.7782	**0.9289**	0.7568	0.7568	**0.9589**	0.8259	0.8259
	Test Suite	0.9414	**0.9114**	**0.9114**	0.9139	**0.8824**	**0.8824**	0.9270	**0.9027**	**0.9027**
Nested Queries	FuncEvalGMN	**0.9662**	0.8169	0.8169	0.8912	0.6564	0.6564	0.9493	0.7366	0.7366
	Test Suite	0.9586	**0.9299**	**0.9299**	**0.9138**	**0.8946**	**0.8946**	**0.9500**	**0.9318**	**0.9318**
Group By Queries	FuncEvalGMN	**0.9740**	0.8427	0.8427	0.6463	0.4549	0.4549	**0.9791**	0.8647	0.8647
	Test Suite	0.9640	**0.9365**	**0.9365**	**0.8333**	**0.7943**	**0.7943**	0.9407	**0.9044**	**0.9044**

Table 5: Performance comparison of GMN and *test suite* across different datasets for Join Queries, Nested Queries, and Group By Queries. The best performance results for different keyword queries across each dataset are highlighted in bold. It can be observed that, in the case of join queries, the AUC scores of GMN consistently outperform those of *test suite*.

L Evaluation of Code LMs

Our training and testing datasets are sourced from GPT3.5, GPT-4, and CodeLlama. To validate the effectiveness of our FuncEvalGMN against other models, we conducted inference using the DeepSeek model on the Spider dataset. The obtained AUC, r_s, and r_p are 91.64%, 51.55%, and 59.94%, respectively, demonstrating that our approach also exhibits strong evaluation capabilities on other large models.

Finally, we evaluated four Code Large Models (LMs) on the Spider dataset using three evaluation metrics: FuncEvalGMN, *Test Suite* (Zhong et al., 2020), and Execution Accuracy (Yu et al., 2018). The original output range of FuncEvalGMN, denoted as y, ranged from negative infinity to zero. We normalized the output results to the range of 0 to 1 using the formula $y = \max\left[\frac{y+3}{3}, 0\right]$. As shown in Figure 19, our FuncEvalGMN can also serve as a good metric for evaluating SQL generation. Compared to Execution Accuracy and *Test Suite*, we do not incur the cost of maintaining and executing databases.

M Keywords Distribution in BIRD-pair dev, WikiSQL dev, and Spider-pair Datasets

In our study, we analyzed five datasets: Spider-pair Train, Spider-pair dev, BIRD-pair dev, Spider-DK-pair dev, and WikiSQL dev, to understand the distribution of SQL keywords. This analysis provides insight into the complexity and nature of queries present in each dataset.

The **Spider-pair Train** dataset shows a notable usage of the `WHERE` keyword, present in 61.57% of the queries, indicating a strong focus on filtering data. The `JOIN` keyword appears in 48.02% of the queries, suggesting a moderate level of combining data from multiple tables. Additionally, `Aggregation` operations (e.g., `COUNT`, `SUM`, `AVG`) are found in 45.63% of the queries, showing a

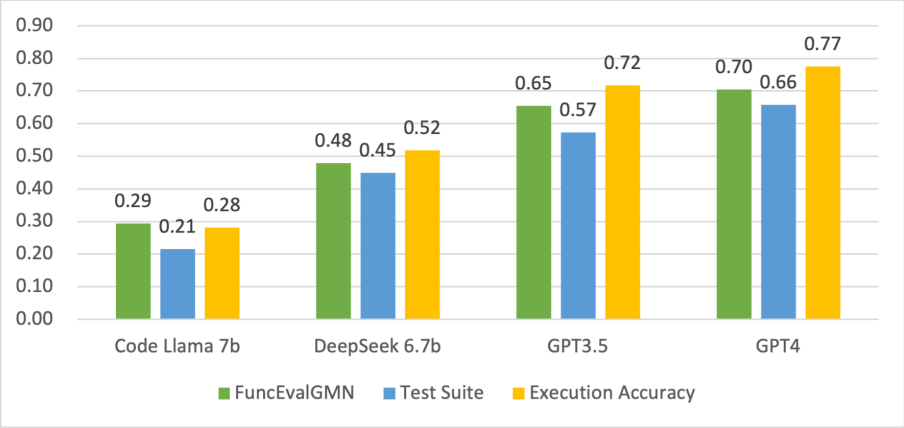


Figure 19: Evaluation of Code LMs

Keyword	Spider-pair train (%)	Spider-pair dev (%)	BIRD-pair dev (%)	Spider-DK-pair dev (%)	WikiSQL-pair dev (%)
Where	61.57	47.45	90.63	48.81	100.00
Join	48.02	44.65	74.87	31.46	N/A
Group By	32.37	35.95	6.25	22.96	N/A
Order By	20.76	22.75	15.82	6.80	N/A
Limit	13.80	19.71	15.69	4.59	N/A
Subquery	62.08	67.09	3.69	6.63	N/A
Union	6.45	9.18	0.07	5.10	N/A
Aggregation	45.63	58.76	37.15	57.31	31.35
Count	31.20	44.28	28.22	39.63	9.96
Average	5.86	6.63	2.96	8.16	4.01
MinMax	6.61	6.51	1.28	10.54	12.88
Sum	3.53	3.89	2.65	0.34	4.50
Distinct	5.11	4.26	16.66	10.71	N/A
StrFunc	N/A	13.49	4.33	N/A	N/A
Regex Filter	1.78	0.97	2.49	4.76	N/A
Cast	3.30	5.66	7.96	N/A	N/A

Table 6: Distribution of Keywords in Various Datasets

significant emphasis on data summarization. The dataset also includes a relatively high frequency of `Subquery` usage at 62.08%, reflecting complex query structures.

The **Spider-pair dev** dataset presents a different distribution. The `WHERE` keyword appears in 47.45% of the queries, which is less frequent than in Spider-pair Train, indicating fewer conditions are applied to filter data. The `JOIN` keyword is used in 44.65% of the queries, showing a similar pattern to Spider-pair Train but slightly lower. Notably, `Aggregation` operations occur in 58.76% of the queries, and `Subquery` is used in 67.09%, suggesting a higher focus on complex query patterns.

The **BIRD-pair dev** dataset exhibits a high frequency of the `WHERE` keyword, present in 90.63% of the queries, which indicates a strong emphasis on filtering conditions. The `JOIN` keyword is also prevalent, appearing in 74.87% of the queries, suggesting that many queries involve combining data from multiple tables. Additionally, `Aggregation` operations are found in 37.15% of the queries, reflecting a moderate level of data summarization.

The **Spider-DK-pair dev** dataset has a balanced usage of the `WHERE` keyword, present in 48.81% of queries, and the `JOIN` keyword in 31.46%. These values indicate moderate filtering and table-combining operations. Moreover, there is considerable emphasis on `Aggregation` operations (57.31%) and the use of `Order By` (6.80%) and `LIMIT` (4.59%), showing diverse query types.

The **WikiSQL-pair dev** dataset stands out for its high reliance on the `WHERE` keyword, used in 100% of the queries, which highlights its focus on filtering. However, it shows no occurrence of `JOIN`, `GROUP BY`, or `DISTINCT`, indicating a simpler query structure with minimal need for combining tables or unique data selection. The dataset does include `Aggregation` operations in 31.35% of the queries, indicating a moderate amount of summarization tasks.

Overall, the analysis reveals distinct patterns in

SQL keyword usage across these datasets. The Spider-pair Train dataset has a strong focus on filtering, joining, and complex queries involving subqueries and aggregations. The BIRD-pair dev dataset emphasizes filtering and joining, while the Spider-pair dev and Spider-DK-pair dev datasets show diverse query patterns with significant usage of aggregations and subqueries. The WikiSQL-pair dev dataset, in contrast, is characterized by simpler query structures, relying heavily on filtering without complex joins or aggregations. These differences highlight the varied query complexities and use cases catered to by each dataset.

N Prompt Design for LLM-Based Tasks

N.1 LLMs-based Text2SQL Task

The Text2SQL task requires generating SQL queries that accurately correspond to a user's question, utilizing the given database schema. Below is an illustrative example of a prompt designed for this task using a Large Language Model (LLM):

Text2SQL Prompt

```
Table schema: There are two tables: stadium, singer.

the structure of table stadium is as follows:
    | column name | column type |
    | ----------- | ----------- |
    | stadium_id | number |
    | location | text |
    | name | text |
    | capacity | number |
    | highest | number |
    | lowest | number |
    | average | number |
    stadium_id is the primary key.

the structure of table singer is as follows:
    | column name | column type |
    | ----------- | ----------- |
    | singer_id | number |
    | name | text |
    | country | text |
    | song_name | text |
    | song_release_year | text |
    | age | number |
    | is_male | others |
    singer_id is the primary key.

Question: How many singers do we have?

Please write a sql based on the table schema above to answer
    question.
```

N.2 GPT-4-based SQL Evaluation

As discussed in Section 5.1, GPT-4 is utilized as a judge to evaluate the consistency between the ground truth SQL and the predicted SQL. The following example demonstrates a detailed evaluation prompt:

GPT4-Evaluation Prompt

```
<Instruction>
Given two SQLs, please return the score of these two SQLs in
    functional and logical consistency scope, directly output
    the score as 0 or 1 (0 means inconsistent, 1 means
    consistent).
</Instruction>

<Input>
SQL_1:
SELECT NAME FROM People WHERE AGE > 34
SQL_2:
SELECT NAME FROM People WHERE AGE >= 34
</Input>

<Response>
score:
</Response>
```