Structural SQL Similarity via Hybrid Graph Matching and AST-Guided Tree Edit Metrics

Anonymous ACL submission

Abstract

Assessing semantic similarity between SQL queries is vital for Text-to-SQL evaluation, query clustering, deduplication, and code auditing. Existing metrics—such as Execution Accuracy and CodeBERTScore—either require database access or rely on token-level similarity, overlooking deeper structural and logical equivalence. These limitations hinder their use in schema-less, privacy-sensitive, or real-time settings. In this paper, we propose structure-aware, execution-free evaluation methodologies, AST-TE and Hybrid-GMN, combining symbolic and neural methods. AST-TE computes Zhang–Shasha-Style Tree-Edit distance over normalized SQL Abstract Syntax Trees to capture structural and semantic differences. Hybrid GMN encodes ASTs and Relational Operator Trees (ROTs) into a heterogeneous graph, enabling fine-grained semantic alignment via cross-graph attention. Experiments on Spider, BIRD[1], and internal subquery dataset generated via a fine-tuned SQLCoder-7B model[2] demonstrates significant performance improvements over existing symbolic and neural baselines. Specifically, on the Spider dataset, our methods surpass the state-of-the-art CrystalBLEU metric by approximately 23% in ROC AUC and more than 95% in Spearman correlation. Our findings underscore critical shortcomings in traditional execution-based and token-level similarity metrics, establishing AST-TE and Hybrid GMN as robust, scalable, and schema-agnostic alternatives for evaluating SQL query equivalence.

1 Introduction

Evaluating the semantic similarity between structured SQL queries is crucial in a range of applications, including Text-to-SQL evaluation, query deduplication, semantic search, and retrieval-based question answering. Traditional evaluation metrics, such as the accuracy of the exact match, the precision of the execution and neural similarity scores such as CodeBERTScore, exhibit substantial limitations. Exact Match is overly brittle, failing on semantically equivalent queries due to minor formatting or aliasing differences. Execution Accuracy depends on access to a live database and cannot distinguish between logically distinct queries that coincidentally produce the same result ((Kim et al., 2025); (Renggli et al., 2025)). Neural metrics like CodeBERTScore only capture token-level embeddings and struggle with structural variations and logic-based intent ((Renggli et al., 2025)).

Recent work has further underscored these shortcomings. (Kim et al., 2025) proposed FLEX to expose high false positive and negative rates in execution-based metrics, while (Renggli et al., 2025) argued for more principled evaluation methods that go beyond surface-level and embedding-based comparisons. These insights highlight the need for execution-free, structure-aware evaluation metrics that assess query similarity based solely on syntactic structure and semantic intent—particularly vital in privacy-constrained, schema-less, or real-time environments.

While recent graph-based evaluation techniques (e.g., FuncEvalGMN, (Zhan et al., 2024)) advance execution-free logical plan alignment, they still rely solely on Relational Operator Trees (ROT) and lack symbolic normalization.

In this work, we propose a structure-aware, execution-free framework for evaluating SQL similarity by integrating symbolic and neural perspectives. Our approach compares both the syntactic and semantic content of queries using two complementary techniques:

- **AST-TE (Abstract Syntax Tree Tree-Edit):** A symbolic tree-edit distance metric operating on normalized Abstract Syntax Trees

[1] https://github.com/Leon0-0/FuncEvalGMN/tree/main/GMN/database

[2] https://huggingface.co/defog/sqlcoder-7b

(ASTs) of SQL queries. AST-TE captures fine-grained syntactic and semi-semantic differences by computing the minimal sequence of tree transformations needed to convert one query to another.

- **Hybrid GMN (Graph Matching Network):** A neural similarity model that jointly encodes both the SQL AST and its Rotational Operator Tree (ROT)—a proxy for logical query execution—as a heterogeneous graph. By applying cross-graph attention mechanisms, Hybrid GMN captures deeper structural and functional alignment between queries.

By unifying symbolic edit-distance reasoning with graph-based neural alignment, our framework provides a scalable, interpretable, and execution-free alternative to existing evaluation metrics. We validate our approach on both standard datasets (Spider, BIRD) (Yu et al., 2018) and internal subquery logs generated via a fine-tuned SQLCoder-7B model[3], demonstrating consistent improvements over prior symbolic and neural baselines.

2 Related Work

Evaluating the similarity or correctness of SQL queries has mainly depended on metrics that compare the surface form or output of the queries. Common metrics include Exact Match Accuracy, Execution Accuracy, and neural similarity scores like CodeBERTScore.

Exact Match Accuracy (El Maalouly, 2022) measures whether the predicted query string matches the ground truth exactly. While it's easy to compute, it is quite fragile. Minor changes in formatting, aliasing, or join order can create false negatives, even if the queries are semantically the same. On the other hand, it may produce false positives when the predicted and reference queries have the same structure but yield different results due to differences in logic or hidden assumptions.

Execution Accuracy (Chaudhuri et al., 2004) checks if the predicted and gold queries return the same results when run on the target database. It is widely used in Text-to-SQL benchmarks like Spider. However, execution-based metrics need access to the database instance, which may not be practical in many real-world situations, such as privacy-restricted, time-sensitive, or schema-less environments. Also, execution equivalence alone doesn't indicate semantic similarity because two distinct queries might accidentally yield the same result on a specific database instance.

CodeBERTScore (Zhou et al., 2023) is a neural similarity metric based on BERTScore, which measures semantic similarity between code sequences using contextual embeddings from CodeBERT. It combines the cosine similarity of token-level embeddings with F1-based alignment between predicted and reference queries. While CodeBERTScore is better than string-based metrics by capturing some semantic overlap, it still has major drawbacks: (i) it is sensitive to tokenization and vocabulary differences, (ii) it misses deeper structural equivalence like join paths and logical operator reordering, and (iii) its F1-based aggregation doesn't distinguish between critical and minor parts of the query (see Appendix B for an example and configuration of CodeBERTScore) .

CodeBLEU (Ren et al., 2020) is an evaluation metric created specifically for code generation tasks. Unlike standard BLEU, In addition to n-gram overlap, it takes into account code-specific structural details like data-flow matching, syntax-aware heuristics, and abstract syntax tree (AST) similarity. As a result, it is more pertinent to the generated code's semantic and structural accuracy. including SQL queries. In this work, (see Appendix C for implementation details and configuration of CodeBLEU) we consider CodeBLEU a suitable baseline for assessing the functional similarity of SQL generation outputs.

Limitations of Prior Graph-Based Evaluation Recent work by Wang et al. (Wang et al., 2025) introduced a database-free evaluation framework using Relational Operator Tree (ROT) graphs to approximate the logical semantics of SQL queries. While this graph-based comparison mitigates certain drawbacks of execution-dependent metrics, it presents several limitations.

First, **ROT-based** methods (Passing et al., 2017) assume structural rigidity, often failing to recognize semantic equivalence across restructured but logically identical queries—such as join reordering or subquery unfolding. Second, they lack symbolic normalization mechanisms (e.g., alias unification, syntactic canonicalization), making them sensitive to superficial variations. Third, these methods rely on a single modality (ROT), ignoring complementary syntactic information embedded in Abstract Syntax Trees (ASTs), which are crucial for under-

[3] https://huggingface.co/defog/sqlcoder-7b

standing clause nesting, scope, and ordering.

These limitations motivate the need for a richer, multimodal approach that captures both structural and semantic query intent without execution or schema dependence.

Our contribution addresses these gaps by introducing symbolic AST-based edit metrics (AST-TE and ETM) and proposing a Hybrid Graph Matching Network that fuses AST and ROT features for joint structural and semantic alignment. This dual-modality approach bridges the symbolic-neural divide and provides a scalable, interpretable alternative to execution-based or token-level metrics.

3 From SQL to AST: JSON Trees via `SQLGlot`

To structurally compare SQL queries, we first convert each query into an Abstract Syntax Tree (AST) (Dai et al., 2025). We use the `SQLGlot` library for this task, which parses SQL into a nested expression tree composed of rich Python objects. These trees capture the full syntactic and semantic structure of the query—including functions, operators, literals, subqueries, and joins—and can be introspected, traversed, and serialized into JSON-like dictionaries.

The structured output enables us to extract nodes and edges for graph-based similarity models. This AST-based representation is significantly more robust than comparing raw SQL strings or token sequences, as it encodes the actual computation and data flow semantics of the query.

For example, the SQL query:

```
SELECT TRIM(b + c, ' ') FROM jobs
```

can be parsed in `sqlglot` using:

```
from sqlglot import parse_one
ast = parse_one("SELECT TRIM(b + c,
' ')
FROM jobs")
print(ast.to_dict())
```

This representation helps capture the hierarchical and logical structure of SQL programs, enabling effective functional similarity modeling. It also facilitates programmatic transformations, such as normalization, template substitution, or equivalence checking during training and evaluation.

These JSON ASTs are used both in our symbolic similarity metric (AST Tree Edit Distance) and as input to the graph construction module in our Hybrid Graph Matching Network (GMN). Their structural consistency enables us to perform alignment and similarity scoring across semantically equivalent SQL queries without requiring access to a database.

4 AST Tree Edit Similarity (AST-TE)

To evaluate SQL similarity structurally, we use a tree edit distance approach over abstract syntax trees (ASTs) derived from SQL queries. We parse SQL strings using the `sqlglot` library[4], which generates grammar-aware expression trees.As shown in Figure 1

These expression trees are converted into tree structures compatible with the Zhang–Shasha Tree Edit Distance (ZSS) algorithm[5], allowing us to compute structural differences between two SQL queries. (see Appendix A for an example of the ZSS algorithm)Each node in the ZSS tree contains not only the SQL grammar token (e.g., `Select`, `Where`) but also relevant values like identifiers and literals when available (Zhang and Shasha, 1989).

The conversion process includes:

- Extracting the `key` of each `sqlglot` node as the node label.
- Appending string representations of meaningful fields such as `this`, `expression`, and `expressions` to enhance value sensitivity.
- Recursively processing child nodes to build the full tree.

We define cost functions for ZSS as follows:

- **Insert cost:** 1.0
- **Remove cost:** 1.0
- **Update cost:** 2.0 if labels differ, 0.0 if they match

The final similarity score is computed as:

$$Similarity = 1 - \frac{TreeEditDistance(T_1, T_2)}{\max(|T_1|, |T_2|)}$$

where T_1 and T_2 are the converted ASTs of the two SQL queries. If both trees are empty, similarity defaults to 1.0.

[4] https://github.com/tobymao/sqlglot
[5] https://github.com/timtadh/zhang-shasha

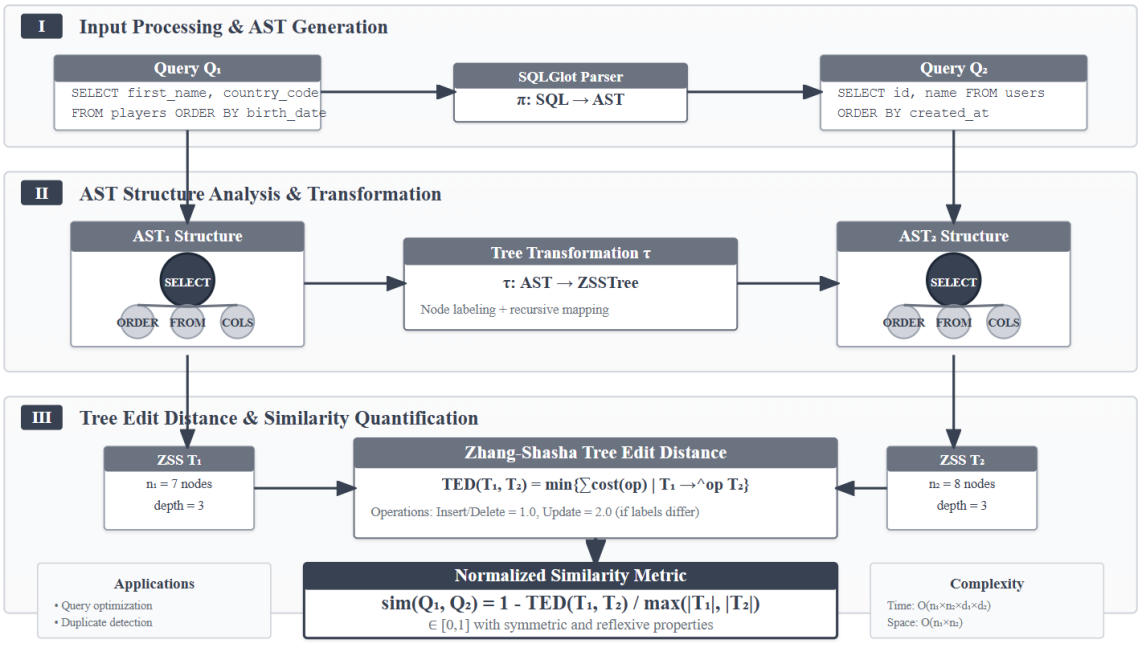

Figure 1: Overview of the AST-TE model architecture.

This AST-based comparison captures structural and literal-level variations, making it more robust than string-based or token-based approaches in scenarios where SQL queries are semantically similar but syntactically different.

5 Hybrid Graph Matching with AST and ROT Features

While AST-based similarity captures the syntactic and structural elements of SQL queries, it may miss deeper semantic equivalences present at the logical plan level. To address this, we introduce a hybrid model that combines Abstract Syntax Trees (ASTs) and Relational Operator Trees (ROTs) using a Graph Matching Network (GMN) As shown in Figure 2 for fine-grained SQL similarity evaluation.

5.1 Motivation and Representation

ASTs reflect the surface structure of SQL queries, while ROTs describe their logical execution plans. By jointly leveraging both, we aim to model both syntactic and semantic similarity between SQL queries. Each SQL query is converted into:

- A tree-structured AST (via `sqlglot`) capturing tokens and literals.
- A DAG-style ROT capturing relational algebra operations (e.g., `Project`, `Join`, `Filter`, `Aggregate`).

These two views are merged into a heterogeneous graph with two node types:

- **Computation nodes:** Represent operators (e.g., `JOIN`, `SELECT`).
- **Content nodes:** Represent identifiers, column names, and literals.

5.2 Graph Construction and Matching

The graph is constructed by linking content nodes to computation nodes according to AST and ROT structure. For instance, a `JOIN` operator node connects to its left and right tables, as well as the join condition. This yields a unified representation of the SQL query.

We use a Graph Matching Network (GMN) to compare two such graphs:

- Each node is embedded using a type-specific encoder: CNN for content nodes and a learned linear layer for computation nodes.
- Random Walk Positional Encoding (RWPE) is applied to capture relative structural positions.
- A multi-layer GNN (GraphSAGE or GAT) encodes local and global features.
- Cross-graph attention is applied to align substructures and compute similarity.

5.3 Contrastive Pretraining and Inference

The GMN is trained using a contrastive loss over labeled SQL pairs:

$$\mathcal{L} = BCE(sim(G_1, G_2),\, y)$$

where $sim(G_1, G_2)$ is the predicted similarity score and $y \in \{0, 1\}$ is the ground-truth similarity label. During inference, the GMN outputs a similarity score between 0 and 1, indicating structural and functional closeness of SQL queries.

This hybrid approach improves generalization to both surface-form and logical variations, offering a robust measure of SQL similarity without requiring database execution.

6 Dataset

To evaluate SQL query similarity without executing against a database, we use publicly available labeled SQL pair datasets from the FuncEvalGMN (Zhan et al., 2024) benchmark[6]. Each dataset contains pairs of SQL queries annotated with binary similarity labels indicating whether the queries are functionally equivalent.

6.1 Datasets Used

- **spider_pair_train:** Contains 17,665 SQL query pairs sampled from the Spider dataset. Used for training our models.
- **spider_pair_dev:** Contains 1,744 SQL query pairs used for validation and evaluation. Queries target multiple databases and represent diverse schemas and question intents.
- **bird_pair_dev:** Contains 2,978 SQL query pairs curated from the BIRD benchmark, used to test generalization to queries from unseen distributions and schemas.

Each sample in these datasets consists of:

- `gt` (ground truth SQL query),
- `answer` (predicted or alternative SQL query), and
- `label` $\in \{0, 1\}$ indicating functional equivalence.

[6] https://github.com/Leon0-0/FuncEvalGMN/tree/main/GMN/database

These datasets allow us to assess the ability of our models (AST-TE and Hybrid-GMN) to measure semantic similarity between SQL queries based solely on their structure and content, without requiring database access or execution.

7 Evaluation and Baselines

We evaluate the performance of our proposed models—AST Tree Edit (AST-TE) similarity and Hybrid GMN—on two benchmark datasets: `spider_pair_dev` and `bird_pair_dev`. These models assess SQL query similarity without requiring execution access to the database. Evaluation is performed using standard metrics: ROC AUC and Spearman's rank correlation coefficient (ρ).

We compare our approach with the following baselines widely used for semantic SQL evaluation:

CodeBERTScore uses a pretrained CodeBERT model to compute similarity based on cosine similarity and token-level F1 scores. However, it is sensitive to surface-level variation and struggles to capture structural and semantic equivalence. Reported performance:

- **Spider**: ROC AUC = 0.7044, Spearman $\rho = 0.2966$
- **BIRD**: ROC AUC = 0.7405, Spearman $\rho = 0.3521$

CrystalBLEU is a BLEU-style metric that uses SQL templates to assess structural similarity. Although more structure-aware than CodeBERTScore, it still lacks functional understanding. Reported scores:

- **Spider**: ROC AUC = 0.6521, Spearman $\rho = 0.2627$
- **BIRD**: ROC AUC = 0.7660, Spearman $\rho = 0.3895$

8 Results and Discussion

Our models outperform both baselines across all metrics. AST-TE is particularly effective, leveraging tree-edit distance over rich, semantically grounded SQL ASTs. Hybrid GMN, which integrates both AST and Relational Operator Tree (ROT) representations, performs competitively on Spider. The detailed evaluation results are presented in Table 1.

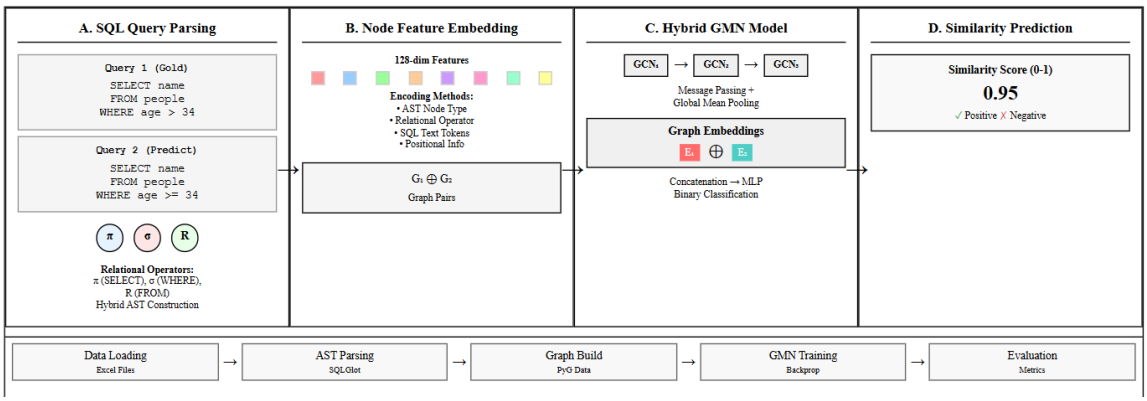

Figure 2: Overview of the hybrid AST+ROT model architecture.

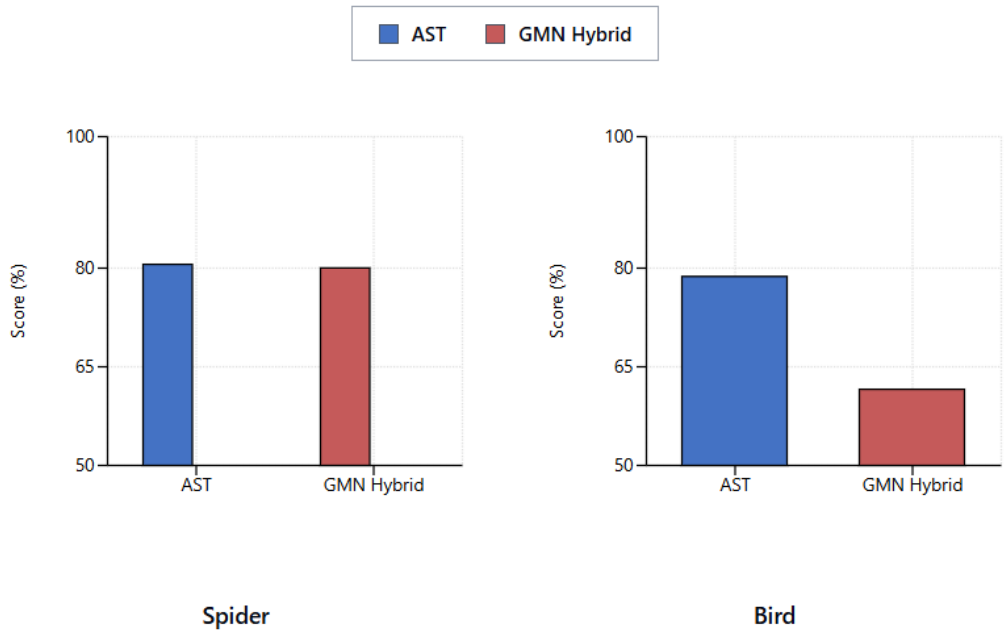

Figure 3: performance of AST and GMN Hybrid.

AST-TE consistently outperforms all other methods in both, as shown in Figure 4. Its success comes from its ability to compare SQL queries using tree edit distance on detailed, semantically labeled abstract syntax trees (ASTs). This approach captures deeper semantic similarities beyond just matching tokens. On the Spider dataset, AST-TE achieves a 23.57% higher ROC AUC and 96.15% higher Spearman ρ compared to CrystalBLEU, and also outperforms CodeBERTScore by 14.39% in ROC AUC and 46.31% in Spearman ρ.

Hybrid GMN also performs well, especially on the Spider dataset, thanks to its use of both AST and relational operator tree features. It shows a 22.73% gain in ROC AUC and 95.89% in Spearman ρ over CrystalBLEU, and performs 13.61% and 46.11% better than CodeBERTScore on the same metrics, respectively. The GMN model was trained on the `spider_pair_train` dataset, which explains its strong performance on Spider-related queries.

However, it is less effective on the BIRD dataset due to differences in schema variety and relational complexity. On BIRD, AST-TE still maintains an edge, performing 2.79% better than CrystalBLEU in ROC AUC and 12.45% better in Spearman ρ. It also outperforms CodeBERTScore by 6.33% and 24.40% respectively. While the Hybrid GMN does not outperform CodeBERTScore on BIRD, this can be attributed to its training solely on the Spider dataset.

By fine-tuning and training on

Method	Dataset	ROC AUC	Spearman ρ
AST-TE	Spider	**0.8058**	**0.5153**
AST-TE	BIRD	**0.7874**	**0.4380**
Hybrid GMN	Spider	0.8003	0.5146
Hybrid GMN	BIRD	0.6158	0.1610
CrystalBLEU	Spider	0.6521	0.2627
CrystalBLEU	BIRD	0.7660	0.3895
CodeBERTScore	Spider	0.7044	0.3522
CodeBERTScore	BIRD	0.7405	0.3521

Table 1: Evaluation results on the Spider and BIRD datasets using ROC AUC and Spearman's ρ correlation metrics. AST-TE consistently outperforms Hybrid GMN, with strong generalization to BIRD. CodeBLEU and CodeBERTScore provide lower correlation, highlighting their limitations in capturing SQL semantics.

Method	Dataset	ROC AUC (Ours)	ROC AUC (Paper)	Spearman ρ (Ours)	Spearman ρ (Paper)
RelPM	Spider	0.8103	0.8442	0.5578	0.5967
RelPM	BIRD	0.7936	0.8357	0.4406	0.4927
ASTPM	Spider	0.7965	0.8281	0.5110	0.5718
ASTPM	BIRD	0.7501	0.8038	0.3661	0.4457
CodeBERTScore	Spider	0.7044	0.7044	0.3522	0.3522
CodeBERTScore	BIRD	0.7405	0.7405	0.3521	0.3521
FuncEvalGMN	Spider	0.8860	**0.9750**	0.6663	**0.8529**

Table 2: Benchmarking results on Spider and BIRD datasets. We compare our reproduced implementations of RelPM, ASTPM, and FuncEvalGMN against reported results in the original paper (Zhan et al., 2024). While our reimplementations exhibit slight drops in performance, they closely follow the trends established in the original benchmarks, validating correctness and reproducibility. CodeBERTScore is included as a strong pretrained baseline.

`bird_pair_train` or similar datasets rich in schemas, the Hybrid GMN model could improve and perform better on BIRD as well. This shows the need for dataset-specific adjustments in neural SQL similarity models. The detailed evaluation results are presented in Table 1. Overall, our results show that structural and semantic representations of SQL queries, especially ASTs, are more effective for query similarity than pretrained language models or surface-level metrics like BLEU or token-level F1. These findings highlight the value of compositional and logic-aware methods for assessing the accuracy of SQL generation systems.

9 Conclusion and Future Work

In this work, we introduced two novel methods for evaluating SQL query similarity without relying on execution or database access: AST Tree Edit (AST-TE) and a Hybrid Graph Matching Network (GMN). By leveraging the structural and semantic properties of Abstract Syntax Trees and Relational Operator Trees, our models provide a robust and database-agnostic assessment of SQL equivalence.

Experimental results on the `Spider` and `BIRD` benchmarks demonstrate that our methods outperform existing baselines such as CodeBERTScore and CrystalBLEU in both ROC-AUC and Spearman correlation. In particular, AST-TE achieves state-of-the-art performance across both datasets, showcasing the effectiveness of structure-aware comparison.

The Hybrid GMN also performs competitively, especially on the Spider dataset, which it was trained on. Its relatively lower performance on the BIRD dataset highlights a key opportunity for improvement—domain adaptation and further fine-tuning. With dedicated training on BIRD-like queries or domain-specific relational structures, we expect the Hybrid GMN to generalize more effectively.

Future Work: We plan to extend our models in several directions:

- Incorporating schema linking and question-context alignment to improve similarity understanding in real-world, multi-turn scenarios.
- Adapting the Hybrid GMN for multilingual SQL generation and cross-domain evaluation.
- Exploring fine-grained error categorization (e.g., operator mismatch, column misalignment) to provide interpretable feedback in SQL learning and debugging tools.

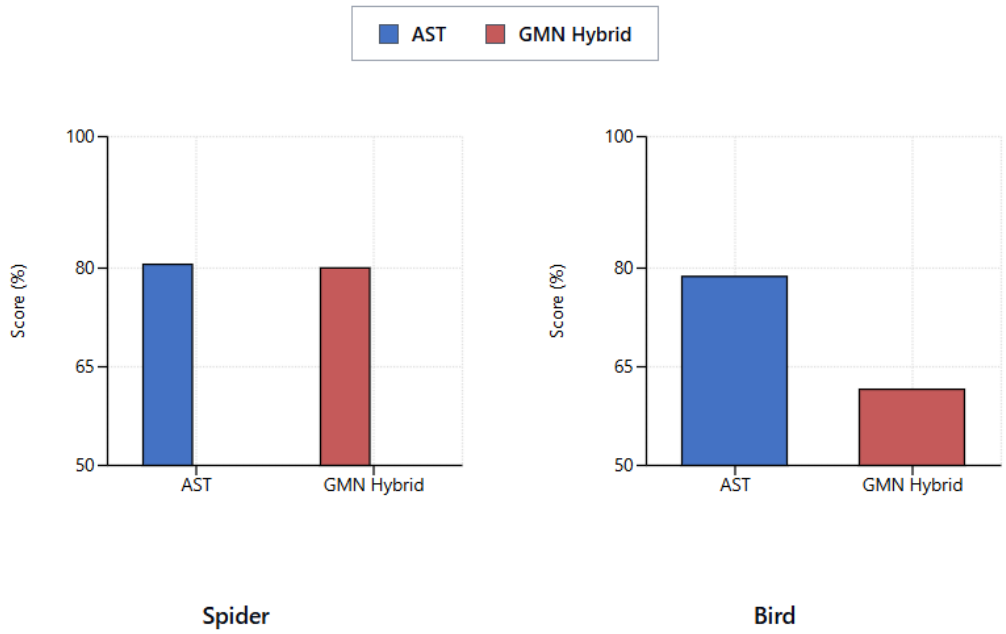

Figure 4: Performance of AST and GMN Hybrid.

Overall, our approach provides a promising step toward more reliable and execution-independent metrics for SQL query evaluation.

10 Acknowledgments

We gratefully acknowledge Varahe Analytics for providing the computational resources and infrastructure that enabled training and large-scale evaluation.

11 Limitations

While our proposed methods demonstrate strong performance on both the Spider and BIRD datasets, several limitations remain:

Dataset Size and Diversity: The BIRD dataset, although valuable for evaluating functional SQL similarity, is relatively small and lacks the schema diversity present in Spider. This limits the model's generalizability and may cause overfitting. Future work could explore expanding the BIRD dataset to include more domains, schemas, and query structures to better train and evaluate similarity models in diverse settings.

Training Time and Computation: The Hybrid GMN model requires non-trivial computational resources. Due to limited GPU access, we were constrained to relatively low epoch counts and batch sizes. Increasing the number of training epochs and conducting larger-scale hyperparameter tuning could improve final performance, especially for the Hybrid GMN which combines multiple graph modalities.

Graph Construction Bottlenecks: Generating structured ASTs and ROTs from SQL queries is currently sequential and parser-dependent. This preprocessing stage could become a bottleneck in large-scale or real-time applications. Optimizing or parallelizing this stage may enable faster training and inference pipelines.

References

Surajit Chaudhuri, Vivek Narasayya, and Ravishankar Ramamurthy. 2004. Estimating progress of execution for sql queries. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 803–814. ACM.

Yujie Dai, Haoyu Yang, Mengnan Hao, and Peixiang Chao. 2025. PARSQL: Enhancing text-to-SQL through SQL parsing and reasoning. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 661–681.

Nicolas El Maalouly. 2022. Exact matching: Algorithms and related problems. *arXiv preprint arXiv:2203.13899*.

Ji-Hyun Kim, R. Rajkumar, X. Lin, and R. Das. 2025. Flex: A false-less execution metric for evaluating text-to-sql. In *Proceedings of the 2025 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*.

Lukas Passing, Mirko Then, Nico C. Hubig, Hendrik Lang, Michael Schreier, Stephan Günnemann, and Thomas Neumann. 2017. SQL- and operator-centric data analytics in relational main-memory databases. In *Proceedings of the 20th International Conference on Extending Database Technology (EDBT)*, pages 84–95, Venice, Italy.

Shuo Ren, Daya Guo, Shujie Lu, Long Zhou, Shuo Liu, Duyu Tang, ..., and Shuming Ma. 2020. Codebleu: A method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*.

Cedric Renggli, Matt Gardner, and Mirella Lapata. 2025. Beyond match scores: Towards a principled evaluation of semantic parsing. *Transactions of the Association for Computational Linguistics (TACL)*.

Yicheng Wang, Ling Zhao, and Ruiqi Xu. 2025. Execution-free evaluation of text-to-SQL via relational operator tree alignment. In *Proceedings of the 30th International Conference on Computational Linguistics (COLING 2025)*, pages 3511–3524, Barcelona, Spain. International Committee on Computational Linguistics.

Tao Yu, Rui Zhang, Kaiyang Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, and Dragomir Radev. 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*.

Yi Zhan, Yang Sun, Han Weng, Longjie Cui, Guifeng Wang, Jiajun Xie, Yu Tian, Xiaoming Yin, Boyi Liu, and Dongchi Huang. 2024. Funcevalgmn: Evaluating functional correctness of sql via graph matching network. *arXiv preprint arXiv:2407.14530*.

Kaizhong Zhang and Dennis Shasha. 1989. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal on Computing*, 18(6):1245–1262.

Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q Weinberger, and Yoav Artzi. 2019. Bertscore: Evaluating text generation with bert. *arXiv preprint arXiv:1904.09675*.

Shuyan Zhou, Uri Alon, Sumit Agarwal, and Graham Neubig. 2023. Codebertscore: Evaluating code generation with pretrained models of code. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 13921–13937, Singapore. Association for Computational Linguistics.

Appendices

A AST-Based Tree Edit Distance (AST-TE)

Our AST similarity model computes tree edit distance between SQL Abstract Syntax Trees (ASTs). We use the `mo-sql-parsing` library to convert SQL queries into structured, JSON-based ASTs. These trees represent the syntactic structure of the SQL query in a hierarchical format that captures nesting, logical operators, joins, and clause ordering.

We use a rule-based transformation pipeline to normalize the ASTs in order to increase robustness against structural variations. This entails rearranging commutative operations (such as conjunctions in the `WHERE` clause), normalizing predicate structures, flattening nested expressions, and standardizing join orderings. Logically equivalent queries are guaranteed to have comparable tree structures thanks to these transformations.

The Zhang–Shasha (ZSS) tree edit distance algorithm (Zhang and Shasha, 1989) is used to calculate the distance between normalized ASTs. The least expensive set of edit operations (insertions, deletions, and substitutions) needed to change one tree into another is determined by this traditional dynamic programming algorithm.

Zhang–Shasha Tree Edit Distance The ZSS algorithm operates on rooted, ordered trees and defines three basic edit operations:

- **Insert**: Add a node to the tree.
- **Delete**: Remove a node from the tree.
- **Substitute**: Replace one node label with another.

Each operation is assigned a cost, which we adapt to the SQL domain. For instance, substituting a `SELECT` clause node with a `WHERE` node incurs a high penalty, while swapping column names within the same clause incurs a lower penalty. These cost functions are designed to reflect the syntactic role and logical importance of each node.

The ZSS algorithm builds a dynamic programming table to efficiently compute the optimal sequence of operations over all subtrees. Its time complexity is $O(n^3)$ in the worst case, where n is the number of nodes, but optimizations and pruning heuristics are applied in practice to accelerate computation.

After computing the raw edit distance, we apply min-max normalization over the dataset:

where $d(T_1, T_2)$ is the ZSS distance between trees T_1 and T_2, and $\min d$, $\max d$ are the minimum and maximum distances observed across the test dataset.

This metric is particularly effective in capturing structural alignment between SQL queries, and is used in conjunction with other metrics such as CodeBLEU and RelPM to assess semantic equivalence.

B CodeBERTScore

We evaluate SQL similarity using **CodeBERTScore**, a transformer-based automatic evaluation metric adapted from BERTScore(Zhang et al., 2019) Unlike traditional n-gram-based metrics such as BLEU, which often fail to capture semantic equivalence between syntactically divergent queries, CodeBERTScore leverages contextual embeddings from large pretrained language models specialized for source code, such as CodeBERT (Zhang et al., 2019). This enables CodeBERTScore to assess deeper functional and semantic similarity between SQL queries.

CodeBERTScore computes cosine similarity between token embeddings of the candidate and reference queries, with each token's contribution weighted by its Inverse Document Frequency (IDF). This approach is more robust to variations in variable names, keyword ordering, and surface syntax, making it well-suited for evaluating functionally equivalent SQL queries written in different styles.

The metric demonstrates a strong correlation with human judgments of functional correctness. For example, in the original study, when comparing two candidate code snippets against a reference, BLEU incorrectly rated a syntactically similar but incorrect candidate higher, whereas CodeBERTScore correctly favored the functionally accurate alternative. This illustrates its utility for assessing approximate equivalence in code and SQL generation.

B.1 Implementation and Usage

The official implementation is publicly available via PyPI and supports evaluation in multiple programming and query languages. Models fine-tuned for different languages (e.g., `neulab/codebert-python`, `neulab/codebert-java`) are automatically

selected using the `lang` parameter.

A typical usage example is as follows:

```
import code_bert_score
pred_results = code_bert_score.score
(cands=predictions, refs=references,
lang='sql')
```

This returns a 4-tuple of scores: precision, recall, F1, and F3. The F3 score is a variant of F1 that weights recall three times more heavily than precision, aligning with the intuition that failing to match relevant code tokens is more critical than overpredicting extra ones.

B.2 Advanced Features

The following features are also supported by CodeBERTScore:

item **Source-aware encoding**: The embeddings can be contextualized by passing natural language descriptions (such as questions) via the `sources=` argument, but similarity is only calculated on the code tokens.**IDF weighting**: Reduces the weight of common tokens to improve performance. The tools offered can be used to generate or supply precomputed IDF dictionaries. **Layer tuning**: The `num_layers` parameter allows you to customize the embedding layer that is used to calculate similarity; this layer is usually between layers 5 and 10.

CodeBERTScore has been shown to align well with human evaluations in natural language to code generation tasks, and is particularly suited for evaluating SQL query similarity in scenarios where surface-level matching is insufficient to judge correctness.

C CodeBLEU

CodeBLEU is a composite metric originally designed for evaluating code generation by combining multiple dimensions of similarity: lexical, syntactic, and semantic. It extends the classic BLEU metric by integrating weighted n-gram match, syntax match via AST (Abstract Syntax Tree) alignment, and data-flow match to capture logic-level equivalence.

For the SQL domain, we modified the CodeBLEU scoring script[7]. We discovered CodeBLEU to be a helpful stand-in for evaluating both structural and functional similarity in SQL queries, despite the fact that it was initially developed for general-purpose programming languages like Python, Java, and C++. This makes it useful not only for lexical token overlap but also as a secondary metric to validate functional similarity judgments.

Four components are weighted averaged by CodeBLEU:

- **BLEU**: standard n-gram precision match;
- **Weighted n-gram match**: modifies the contribution of rare or informative tokens;
- **Syntax match**: compares the AST structures of reference and predicted code.
- **Data-flow match**: evaluates the coherence of variable usage and dependencies.

The official implementation is accessible via `pip` and parses code in multiple languages using `tree-sitter`. We make use of the most recent version from the XLCoST/CodeBLEU repository, which offers precompiled tree-sitter grammars for a number of languages and supports platform-independent builds.

All sub-metrics and the final CodeBLEU score are calculated by the `calc_codebleu` function. To handle SQL-specific constructs like joins, subqueries, and group-by clauses, we extend the AST and data-flow parsers and employ a custom tokenizer in our SQL adaptation.

Compared to more conventional metrics like BLEU and accuracy, CodeBLEU has demonstrated a stronger correlation with human assessment of semantic and logical correctness, making it a promising option for code and query evaluation tasks.

[7] https://github.com/sola-st/crystalbleu