QUERY GENERATION FOR QUERY OPTIMIZER TESTING VIA MACHINE LEARNING

YONGTAI YANG

A THESIS SUBMITTED TO
THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF ARTS

GRADUATE PROGRAM IN INFORMATION SYSTEM AND TECHNOLOGY
YORK UNIVERSITY
TORONTO, ONTARIO
DECEMBER 2025

Abstract

Modern database management systems (DBMSs) are important to data-driven applications. However, testing DBMS bugs is still a challenging task as the DBMS is a very complex system. Bugs in the DBMS often appear only under specific execution plan patterns, such as nested-loop joins combined with aggregation. Reproducing such bugs requires generating SQL queries whose execution plans contain the pattern that triggers the bug. Existing rule-based query generators and learning-based approaches both fail to generate queries under the execution plan pattern constraint. To overcome this limitation, we propose QueryMorpher, a plan-driven query generation framework that generates SQL queries from the problematic execution plan that triggers the bug. QueryMorpher begins with a problematic execution plan and a plan pattern that triggers the bug, and implements a sequence of learned plan mutation operations guided by a sequence-to-sequence model. The mutated plan is then translated back into SQL by using a plan-to-query translation module, which guarantees that the resulting query reproduces the desired execution

plan while remaining syntactically and semantically valid.

Experimental results demonstrate that QueryMorpher can generate diverse and valid queries whose execution plans contain the user-defined patterns. On TPC-H, QueryMorpher achieves a target-pattern rate of 0.6 vs 0.4 for the best baseline, while maintaining 10% higher plan diversity under the same budget. On TPC-DS, QueryMorpher achieves similar improvements, indicating that QueryMorpher is stable on different database schemas. By bridging the gap between query generation and query execution plan control, QueryMorpher enables automated and controllable DBMS testing.

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Professor Xiaohui Yu, for his invaluable guidance, insightful feedback through this research journey. His profound knowledge and patience have been instrumental in shaping this thesis.

I am also thankful to my supervisory committee members, Professor Ling Jiang, and Professor Aijun An, for their constructive comments and encouragement.

Special thanks go to the members of the IBM CAS project for their technical advice, particularly Dr. Calisto Zuzarte. Also, I am deeply grateful to Dr. Yueting Chen for his valuable advice and help.

My last tribute is to those who know I am not perfect but still love me.

Contents

List of Tables

List of Figures

1 Introduction

Modern database management systems (DBMSs) are the fundamental component of data-driven applications based on the strong capability of providing data storage, query execution, and transactional guarantees. Despite their success, supporting the correctness and the robustness of the DBMS remains a challenge. A mainstream DBMS integrates many modules, including the optimizer, executor, storage engine, etc. The complex interactions between the modules make DBMS testing and debugging a very challenging problem. When a query is executed, the optimizer transforms the SQL statement into a physical execution plan, which specifies the operators and their connections that determine how data is actually processed during execution. The physical execution plan is then input to the executor, and the executor performs the execution according to the plan. As such, many runtime bugs (e.g., incorrect result, performance regression) in the executor are plan-dependent, which means the bugs exist only under specific combinations of operators or plan structures, such as a hash join within a particular aggregation context, as shown

in Figure 1.2. These bugs are particularly hard to reproduce as the triggering patterns arise only when the optimizer happens to produce the corresponding plan. To systematically reproduce such issues, the ideal approach is to generate SQL queries whose execution plans contain the plan patterns that trigger the same bugs. However, generating such queries is non-trivial.

```
select
   sum(l_extendedprice* (1 - l_discount))
as revenue
from
   lineitem, part
where
(
   p_partkey = l_partkey and
   p_brand = 'Brand#54' and
   l_quantity >= 8 and
   l_quantity <= 8 + 10
) or
(
   p_partkey = l_partkey and
   p_brand = 'Brand#41' and
   l_quantity >= 20 and
   l_quantity <= 20 + 10
) or
(
   p_partkey = l_partkey and
   p_brand = 'Brand#44' and
   l_quantity >= 28 and
   l_quantity <= 28 + 10
);
```

Figure 1.1: Example query

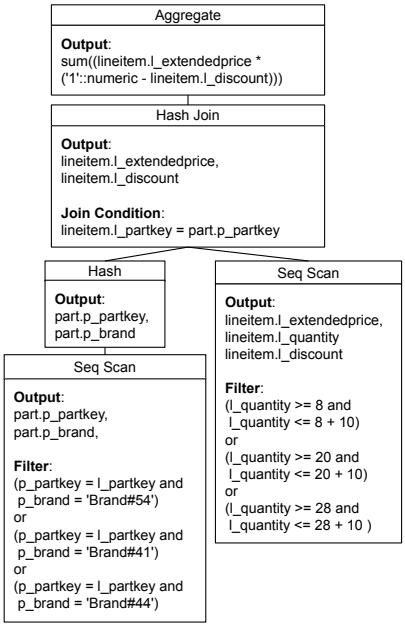

Figure 1.2: Example execution plan

Existing SQL query generation approaches can be broadly categorized into rule-based and learning-based approaches, each exhibiting significant limitations when applied to plan-level DBMS testing.

Rule-based approaches, such as SQLSmith [18], generate syntactically valid SQL queries by randomly combining tables, attributes, and predicates according to the database schema. SQLSmith is designed for discovering parser and executor crashes by exploring a large number of queries without human interaction. However, such rule-based approaches work on the SQL syntax level with a randomized generation procedure, and thus they have no additional knowledge to associate SQL queries with the actual physical plan executed by DBMS, providing no explicit ways to control the direction of generated queries. As a result, the execution plans obtained from such generated queries are diverse in nature and are usually used to perform extensive testing instead of producing queries that specifically target certain patterns in the execution plans.

To further provide guidance during SQL generation towards the desired direction, machine learning-based approaches are explored with certain user-defined constraints. For example, LearnedSQLGen [27] utilizes Reinforcement Learning (RL) models to generate query distributions that align with real workloads or satisfy statistical constraints, such as estimated cardinalities or runtime cost. While the RL model is able to generate more promising queries as it interacts with the DBMS environment through feedback, it still relies on SQL grammar-based automata to produce syntactically valid queries. Consequently, the model lacks explicit awareness of the underlying execution plan structure and cannot directly reason about

operator choices or join strategies that impact performance, limiting its ability to accurately capture the physical behavior of generated queries. Moreover, training RL models can also be time-consuming and computationally expensive, especially when the exploration space is large.

In this thesis, we explore the approach of generating SQL queries with explicit plan structure control. Specifically, we aim to produce SQL queries whose execution plans contain user-specified patterns. Such patterns represent a common combination of operator types in the physical execution plan, and thus can be used to expose bugs under certain execution paths. Unlike existing approaches that attempt to generate SQL queries directly to capture underlying characteristics, our key intuition is to operate at the plan level directly to generate execution plans and then translate them back into valid SQL queries that yield the desired plans. This approach ensures that the target pattern is always included in the generated queries while also maintaining diversity to capture a broad range of potential query forms.

Following the above intuition, we propose QueryMorpher, a framework that generates SQL queries for a given plan pattern. The framework takes as input a plan pattern specified by users, which captures only the essential structure of certain operations and the relationships among them. In practice, such plan patterns can be constructed or extracted from suspicious queries collected and identified from

the query logs. The procedure of our framework involves three main steps: (1) Based Plan construction: We first obtain a base execution plan that conforms to the user-specified plan pattern. We obtain the base plans by filtering a large number of execution plans and collecting the plans with the plan pattern. (2) Plan mutation: Starting from the base plan, we apply a sequence of mutation operations to transform it into new variants. Each mutation operation modifies the current plan by inserting, extending, or replacing a node, thereby introducing structural diversity while preserving the target pattern. (3) For each mutated plan, we convert it back into a SQL query using Common Table Expressions (CTEs) to accurately capture the hierarchical structure of the plan.

However, due to the vast search space of possible execution plans and the intrinsic constraints of certain plan structures, some execution plans obtained in Step 2 cannot be translated back into valid SQL queries during Step 3, resulting in additional generation overhead without producing valid SQL outputs. To address this issue, we model the plan mutation step as a sequence generation problem and leverage the BART model to learn transformation patterns directly from execution plans collected from real databases. By operating at the plan level rather than the query level, our framework ensures that the generated SQL query produces an execution plan consistent with the desired structure.

Through the experiments, the thesis answers the following research questions:

- RQ1: Can we reliably generate queries whose execution plans contain a specified operator pattern?
- RQ2: Can a learned plan-mutation policy outperform rule-based baselines in pattern rate and diversity under a fixed budget?
- RQ3: How sensitive is the method to pattern complexity, mutation rate, and model size?

In summary, we make the following contributions.

- **Formalizing the Matching Plan Generation Problem.** We clearly define the Matching Plan Generation problem, which focuses on creating SQL queries whose execution plans contain specific operator structures, called plan patterns.
- **Introducing the QueryMorpher Framework.** We introduce QueryMorpher, a framework that generates SQL queries by working directly at the plan level. The QueryMorpher includes a plan-level-mutation module (node-to-sequence, BART-based policy, compiler) and a plan-to-query module. To our knowledge, no prior work directly controls plan structure while generating queries; we fill this gap.
- **Experimental Evaluation on TPC-H and TPC-DS.** We evaluate QueryMorpher against rule-based baselines on both TPC-H and TPC-DS. The re-

sults show that QueryMorpher achieves higher pattern-matching accuracy and generates more diverse execution plans.

2 Related Work

We introduce the related works on the problem of query generation. And we also discuss the limitations of solving this problem or its ties with our proposed solution.

2.1 Query Generation

SQL query generation aims to automatically generate a large volume of SQL queries with semantic meaningfulness and syntactic correctness. The generated queries that tailor to a specific database schema or system will play an essential role in the downstream applications. The downstream applications include database system benchmarking, database robustness testing, or database-related data-driven model training. The existing query generation methods can be classified into three groups: template-driven generation, random generation, and machine learning generation. In the following chapters, they will all be introduced.

2.1.1 Template-Driven Generation

The earliest query generation methods are primarily rule-based or template-based. Poess et al. proposed qgen in their paper TPC-H [16], which is a template-driven query generation for industry-standard benchmarks. The qgen is responsible for generating parameterized queries using 22 SQL templates defined with TPC-H specifications. Each pre-defined template consists of different placeholders such as [:DATE], [:REGION], [:NATION]. Besides the templates, qgen also requires the TPC-H schema information and parameter substitution rules that are stored in separate files. In the query generation, the qgen first chooses a template, and replaces the placeholder in the chosen template with the parameters based on the pre-defined rules. The output of the qgen is the executable query with variations determined by replacing parameters.

The template-driven query generation methods have become popular because the templates are designed to reflect real-world workloads, such as business logic like revenue calculations, regional summaries, and customer filtering. And due to the placeholder is randomly replaced, generated queries are reproducible by using the random seed. Also, randomly replacing the placeholders with predefined parameters diversifies the generated queries. However, since the template is fixed, all of the generated queries must follow one of the 22 templates, which allows no structural

variation. Also, because the templates are manually crafted and hardcoded, they are not general for all the database schemas and all use cases. These limitations mean that only a few datasets have the customized query generation tool.

2.1.2 Random Generation

To solve the diversity problem that the template-driven query generation methods suffer from, random query generation methods are proposed [17][25][1]. Rigger et al. proposed a pivoted query synthesis method, SQLancer [17], to generate queries for bug testing. It extends the concept of SQLSmith by introducing metamorphic testing to identify logic bugs, not just crashes. SQLancer first randomly generates logical expressions that are used in the WHERE or JOIN clause by randomly generating tables and rows, and then randomly selects a row from each table. And then, generate the query by selecting all the columns from all the tables generated at the beginning, and concatenate the logical expression generated before to the WHERE clause. In comparison with the template-driven query generation, SQLancer is more compatible with more database systems, as it builds the database schema automatically during the query generation. Also, the generated queries from SQLancer are more diverse. However, because it builds the schema and logical expressions in a purely random way, the queries it generates are meaningless and do not align with the real-world usage pattern.

The SQLFuzz proposed by Yuan et al. proposed a novel idea of generating SQL queries and also a solution to the problem of meaningless generation that the traditional random query generation method shared. SQLFuzz is a mutation-based query generation tool focused on discovering vulnerabilities and robustness flaws in database engines. Unlike grammar-based generators, it starts from seed queries and applies mutations. SQLFuzz takes real or handcrafted seed queries as input and applies syntactic or semantic mutations, such as changing the constant in the logical operation, or replacing the operators, such as replacing "=" with ">". Also, SQLFuzz can mutate the structure of the query, such as inserting or removing clauses, and swapping subtrees in parse trees. All the mutation actions are guided by the predefined grammar rules to avoid syntax errors. SQLFuzz successfully mimics the real-world workload dataset patterns in the generated query workload. And also, it guides the generation towards specific query types or features. Although the handcrafted seed queries harm the diversity, the method proposed in SQLFuzz finds a good balance between validity and diversity of the generated query workload. However, due to the nature of the mutation generation, SQLFuzz may generate redundant queries, and this method is not suitable for query generation from scratch.

In summary, in comparison with the template-driven query generation, the random query generation methods are better in terms of generation diversity, because

the template-driven methods use hardcoded templates, and only change the parameters to make variation in the generated query. However, the template-driven query generation methods are better at generating a query workload that mimics the real-world workload, while the random generation methods often generate meaningless queries. Switching from one method to another has a tradeoff, and SQLFuzz attempts to mitigate this tradeoff by randomly mutating the handcrafted query to generate new queries. By using SQLFuzz, the generated queries are more like real-world queries, while the diversity of the generated queries is not limited by the placeholders.

2.1.3 Machine Learning Generation

Recent research has demonstrated the application of machine learning models and large language models for SQL query generation. For example, Liu et al. proposed a GAN-based framework, TreeGAN [8], that generates SQL queries by simulating the structure of the given workload patterns. The objective of TreeGAN is to generate syntactically valid sequences, e.g., SQL queries, that conform to a given context-free grammar, by generating a parse tree using Generative Adversarial Networks. TreeGAN consists of two parts, the generator and the discriminator. The generator is a grammar-constrained tree generator that constructs abstract syntax trees (ASTs) in a top-down manner using valid production rules from the CFG, and a

Tree-LSTM-based discriminator evaluates whether the tree is generated by the generator or is from the real dataset. The training objective is that the ASTs generated by the generator have the same pattern as the real dataset. In the sampling stage, the generated ASTs are converted to structured sequences (e.g., SQL query). Since the generator builds ASTs based on the context-free grammar, the syntax of the output is guaranteed. The use of the GAN model increases the similarity between the distribution of the generated queries and the real dataset. However, TreeGAN requires a well-defined context-free grammar, which limits its performance in cases where the generation rules are less formal or ambiguous. Also, while TreeGAN ensures syntactic correctness, it does not guarantee semantic correctness. More than that, TreeGAN requires large volumes of representative SQL query data for training and struggles to generalize unknown patterns or schemas.

As the improvement of the TreeGAN, Zhang et al. proposed LearnedSQLGen [27], a constraint-aware SQL generation method using a Reinforcement Learning model. LearnedSQLGen solves the problem that TreeGAN requires a large size of dataset, and lacks control over query characteristics like cardinality or execution cost. The core of LearnedSQLGen is a Finite State Machine (FSM), which is derived from the SQL query grammar and plays a role in a rule-based controller to guide the query generation process. As the generator walks through the FSM, it builds a SQL query token by token, which supports the syntactic correctness.

More than the FSM, LearnedSQLGen uses an Actor-Critic reinforcement learning framework. Under this framework, the generator, which is the agent, learns a policy to select valid transitions (actions) through the FSM given its current context (state). The policy network is trained to maximize expected reward, which is computed based on whether a generated query satisfies the target cardinality and cost constraint by executing the query in the database management system. By using the reinforcement learning model, LearnedSQLGen can generate syntactically valid queries that satisfy the user constraint. However, the training of the Reinforcement Learning model is unstable and slow, and it often lacks structural diversity, as the model tends to exploit query patterns seen before that satisfy the constraints rather than exploring unfamiliar structures.

In summary, Machine Learning generation methods generally have better performance than the template-driven methods and random generation methods in the diversity and syntactic correctness. However, the training of the machine learning models is complicated. Those models either need a large high quality dataset or the training is unstable.

2.2 Query Optimization

Before end-to-end Learned Query Optimizations appeared, significant progress had been made in the area of using modern Machine Learning approaches for query

optimization [7][10][11][4]. For example, Krishnan et al. proposed DQ[7], a reinforcement learning based join order optimization model. They pointed out that join order can be reframed as a sequential decision-making problem, aligning closely with reinforcement learning paradigms. The DQ model optimizes join order as a Markov Decision Process, where the state is the one-hot representation of the currently built partial plan, e.g., which tables are joined so far. The action is the selection of the next join operations, i.e., which table to join next. In this case, the execution plan is presented as a compact binary state, e.g., a one-hot vector indicating which relations have been joined. This lightweight plan representation has a fatal problem, which the one-hot vector can not hold much information.

With a similar idea of using a Reinforcement Learning model, ReJOIN[10] reframes join ordering as an episodic decision-making process, where each query is an "episode" in reinforcement learning. It allows the optimizer to learn from execution feedback and adjust its strategy, overcoming the brittleness of static cost models. As described by ReJOIN, the state is captured as a list of partial join subtrees, each encoded as a vector representing which relations are present and at what height of the join tree. With the help of additional input features, including join predicates and selection predicates, the model can distinguish joinable relations. The action of the model is to select one pair of subtrees to join next. Thus, the ReJOIN progressively generates a binary join tree until all relations are joined. Although DQ

and ReJOIN are all using a reinforcement learning model as the backbone, they are different in several aspects. First, DQ uses a value-based Deep Q-Network, which leverages both cost-model bootstrapping and experience replay to achieve higher sample efficiency. While ReJOIN uses a policy gradient network, which can learn an effective join order directly from the feedback, and does not rely on the cost models.

Modern systems like NEO[12], RTOS[24], and Balsa[23] build on these foundations by adding richer state encodings like Tree-LSTM or GNN, safe exploration, curriculum learning, and full optimizer pipelines beyond join ordering. As the first end-to-end Learned Query Optimization method, NEO reshapes queries by learning from actual query executions rather than depending on handcrafted heuristics and cost models. As the method proposed by NEO, it starts from observing decisions made by a real database optimizer. Plans and execution statistics are collected, and NEO learns to mimic the expert using supervised learning to achieve a reliable starting point. Then, NEO trains a value model, a neural network that takes as input a partial query plan along with query features such as joins, predicates, and schemas, and predicts the expected execution time of that plan. The value network integrates the tree convolution network to reflect the query execution plan. Once the query features have been encoded, Neo uses the value model to search the query execution plans and find the plan with the minimum predicted execution

time. As new queries are generated, Neo executes those queries and collects real execution statistics to refine the value model. Thus, enabling long-term adaptation to shifting workloads. However, NEO suffers from heavy training overhead, especially initial bootstrapping. It also suffers from inference latency overhead due to the value-based search.

Balsa[23] is based on the same architecture as NEO, proposes a method that does not depend on the expert optimizer output, which speeds up the training step and achieves up to 2.8x performance improvement over expert optimizers, and achieves robust generalization to unseen join structures. Balsa introduces a three-phase learning framework, which involves bootstrapping a value network in a minimal cost model, fine-tuning the value network in real execution, and using a tree search algorithm to build query plans. Balsa starts by training with a lightweight simulator using only a simple logical cost model (e.g., PostgreSQL cardinality estimates, not actual runtime). In the sampling, instead of random exploration, Balsa samples among multiple top candidate plans predicted by its value network and picks the best unseen one, which balances the exploration and safety. In comparison with NEO, Balsa replaces the expert optimizer with a lightweight cost estimation model. Also, the three-phase framework leads to a strong generalization to new queries and efficient and safe trial-and-error learning via simulation and controlled exploration.

RTOS[24] is also based on NEO, but RTOS addresses shortcomings in previous

DRL-based join order optimizers, like ReJOIN and DQ, by leveraging Tree-LSTM to model the hierarchical structure of join plans instead of feature vectors. By using a Tree-LSTM, the model computes embeddings from leaves to the root, capturing subtree structure, join predicates, and selection filters. This approach better captures hierarchical dependencies and also generalizes robustly across schema changes or aliasing scenarios.

In summary, the popular works demonstrate that query execution plans are better represented as a structured, hierarchical object. In this case, the plan manipulation can be designed as a sequential decision process. These insights inspire our design of the QueryMorpher. The execution plan is designed as a tree, and we use a learned model to mutate the plan incrementally.

2.3 Graph Generation

Since the execution plans are trees, the graph generative models are conceptually related. Sampling graphs from a target distribution is widely studied in many subjects. Generative machine learning models have recently shown their competitiveness in this problem. Different from random graph models [3], those models use learning models as their backbone, including VAE [19][21] and GAN [14][13]. These models are trained to capture nested graph structural patterns and generate high-quality graphs with desired attributes.

Recently, the invention of the diffusion model has provided more ways of thinking in graph generation. Graph diffusion models have gained prominence by decomposing the graph generation task into multiple denoising steps, resulting in superior generative performance compared to methods that predict the entire adjacency matrix simultaneously. Based on the different task demands, the diffusion model has different variations, including score-based approaches [15] and discrete approaches [22]. However, the traditional diffusion models suffer from two drawbacks: (1) Low efficiency. Due to the nature of the diffusion model, the sampling process involves a long denoising process before the final output is generated, and it is time-consuming. (2) Hard to impose constraint. Different from the auto-regressive model, the diffusion model is one-shot generation; hence, it is hard to implement the generation rules during sampling.

GraphARM [6] proposed an autoregressive graph generative model using an autoregressive diffusion model [5] to generate graphs. Instead of diffusing a graph in adjacency matrix space, this model directly generates a graph in the discrete graph space. Due to its autoregressive generation nature, GraphARM allows for constraint incorporation. However, this method is built on a prerequisite, such that the diffusion model needs to know the node ordering in advance. In this setting, GraphARM is inflexible for complex tasks.

Inspired by GraphARM, ConStruct [9] proposed a discrete diffusion framework

that performs constrained graph generation with specific structural properties. The work focuses on the structural generation of the graph and improves the efficiency by utilizing incremental constraint satisfaction algorithms and a blocking edge hash table to reduce the computational redundancy during the reverse process. However, ConStruct is not suitable for the task of execution plan generation as it is not scalable. Since the input of the model sampling is a batch of isolated nodes, and the task of the ConStruct is to generate edges to connect nodes to a graph, the size of the generated graph is limited by the initial input.

In summary, the graph generation methods mentioned above are not suitable for generating execution plans, as most of them utilize adjacency matrices, and adjacency matrices scale poorly for typical plan sizes. Also, it is difficult for adjacency matrices to encode operator semantics and attributes such as operator type, selection predicates, and join conditions. Most importantly, the existing approaches lack plan pattern constraints during the generation.

During the literature review, we are unaware of the works about generating queries whose execution plans contain the user-defined plan pattern, and this is a gap.

3 Problem Definition

Given a database D and a target execution plan pattern P, our goal is to generate SQL queries whose execution plans match P in structure and operator composition.

Specifically, we model an execution plan T as a tree. Each node $t \in T$ corresponds to a physical operator (e.g., `Scan`, `Join`, etc), and edges represent parent-child relationships in the operator hierarchy. We focus on physical operators because they directly determine the runtime behavior of the DBMS. While logical operators define query semantics, the physical plan captures the concrete execution decisions of the optimizer, making it the appropriate level of abstraction for testing and performance analysis.

For simplicity, we consider a node t as a triplet, $t = (o_t, A_t, C_t)$, where $o_t \in O$ denotes the operator type, which is fixed for a given RDBMS and could vary depending on different RDBMS, A_t denotes the attributes related to the node t, including join keys, filter predicates, etc, and C_t is a set of child nodes.

In practice, we wish to generate queries that could lead to an execution plan

with a certain structure. Such structured plan targets arise in settings, including optimizer testing and differential analysis, regression detection for plan stability, and the reproduction of specific operator pipelines for performance diagnosis. In such cases, we are primarily concerned with the operator types that appear in a given plan structure, since they usually correspond to a specific execution path in the underlying code base. Consequently, even when operator attributes (e.g., predicates, join keys) differ, plans that share the same operator skeleton are likely to exercise similar optimizer rules and execution logic, making the skeleton a robust proxy for targeted testing and analysis.

To express such requirements, we introduce the concept of a *plan pattern*, which consists only of the plan structure and the operator type assigned to each node, while excluding operator-specific attributes. Formally, we use P to denote such a target pattern, where each node $p \in P$ is represented as $p = (o_p, V_p)$, with $o_p \in O$ denoting the operator type, and V_p denoting the child nodes.

We define the matching plan as follows.

Definition 1 (Matching Plan)**.** *For a target pattern P, a plan T is a matching plan if there exists an injective mapping $\varphi : P \mapsto T$ such that for every node $p = (o_p, V_p) \in P$, with $\varphi(p) = (o_t, A_t, C_t) \in T$, we have*

1. *$o_p = o_t$ (operator types match);*

2. *for every child $p' \in V_p$, it holds that $\varphi(p') \in C_t$ and the correspondence is recursively preserved (parent–child relations are preserved).*

Intuitively, a plan T matches a pattern P if P can be found as a contiguous subtree of T with the same operator types and parent–child relationships, ignoring attributes. In other words, T matches P if the target pattern P is isomorphic to the subtree of T. Attributes A_t in T are ignored in the matching.

Figure 3.1 shows a simple pattern P (3.1a) and a matching plan T (3.1c); the mapping φ highlights how each node is mapped.

We therefore focus on generating a set of queries that could produce a set of matching plans for a given target pattern P. We refer to this problem as Matching Plan Generation, formally:

Definition 2 (Matching Plan Generation (MPG))**.** *For a given target pattern P, generate a set of plans $\mathcal{T}$, such that for each plan $T \in \mathcal{T}$, T is a matching plan of P. Practically, we aim for a set of matching plans with high structural diversity.*

The Matching Plan refers to a single plan, and the MPG is the problem of producing Matching Plans. We therefore focus on generating a set of queries that (1) yield the same plan structure as a given target plan, and (2) exhibit diversity in their attributes, query conditions, and the operator types present in other parts of the plan. In the next chapter, we present a generative modeling framework to

produce queries whose execution plans satisfy these matching constraints.

3.1 Assumptions and scope

The problem is based on the following assumptions. We only consider problems on the single DBMS, PostgreSQL 15.3, with the deterministic optimizer. We do not consider the cost and latency in the MPG problem. The pattern matching is purely structural and attribute-agnostic.

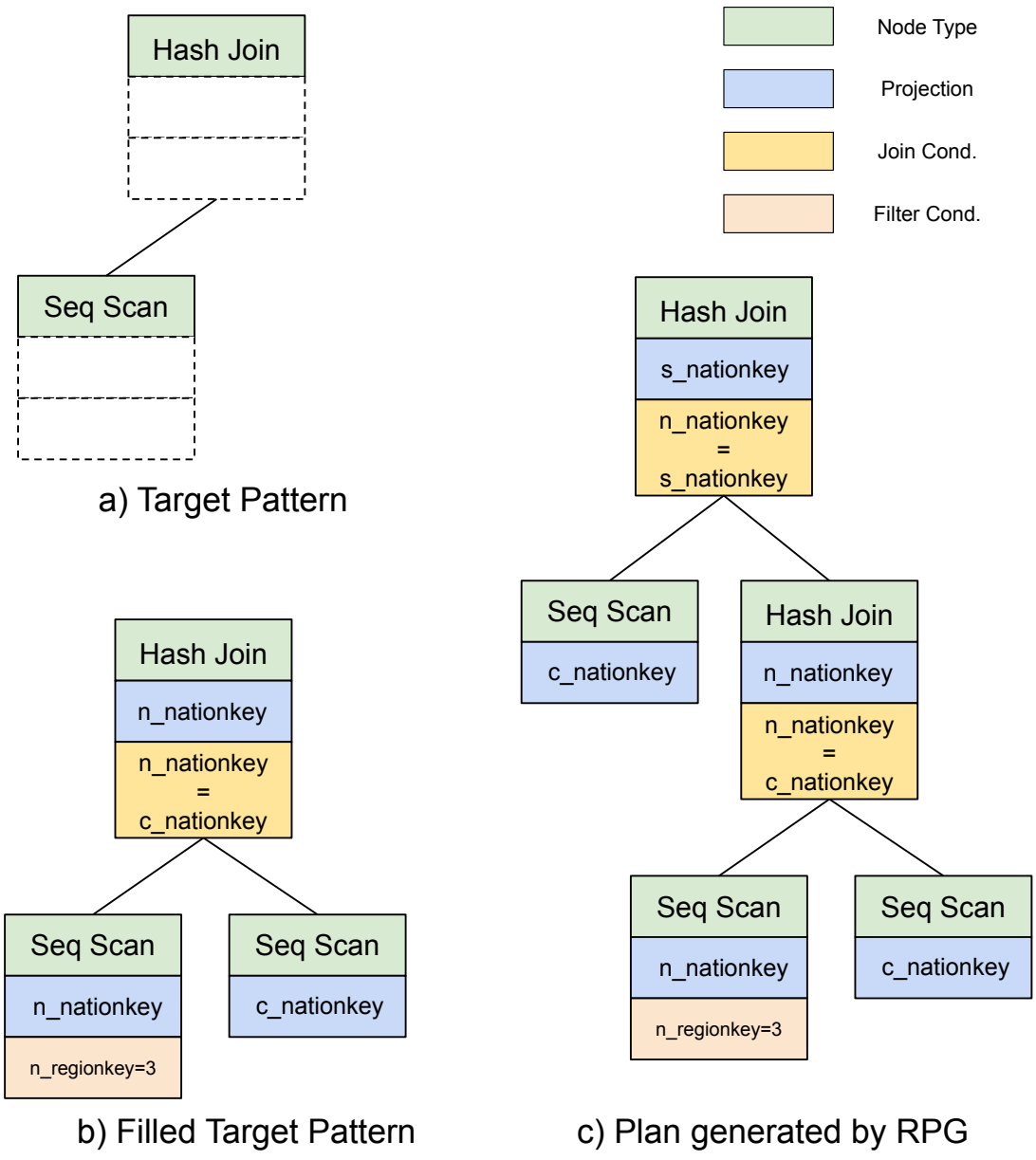

Figure 3.1: Target Structure

4 Baseline

Following Chapter 2, we introduce two complementary baselines. **RPG** constructs a valid matching plan directly from the plan pattern using deterministic schema or grammar rules (construction baseline). **RAG** assumes a sample matching plan and applies rule-based mutations outside the plan pattern (mutation baseline). In Chapter 4, we retain the same mutation interface but replace RAG's heuristic policy with a learned policy.

4.1 Rule-Based Plan Generation (RPG)

As per our definition in Chapter 3, the plan pattern is not a valid execution plan, due to (a) the absence of attributes for each node in the plan pattern, and (b) the absence of other possible operators above or below the plan pattern. To bridge the gap, a straightforward approach is to first generate valid attributes for each node in the plan pattern to obtain the **filled plan pattern**, and then grow the filled plan pattern upward with additional operators into a valid execution plan. This

design choice is not intrinsic, and the reason for choosing this design is that adding operators on top of the filled plan pattern is easier to implement in comparison with adding operators below the filled plan pattern.

4.1.1 Generation Rule

Generating a filled plan pattern needs rules to guide it. By using the given database D, we get the schema information, and we set schema rules based on the schema information. During the plan generation, the attribute should follow the schema information, i.e., for the scan node, the column name in the projection information should come from the table that aligns with the relation name of the node. Also, two keys in the join condition should exist in a foreign key-primary key relationship.

Besides the schema rules, there are grammar rules. The attribute A_n of a node n should be in the union of the attributes from its child nodes. i.e. $A_n \in A_u \cup A_v$, where $u \in C_n$ and $v \in C_n$. In other words, the information used to fill the attribute of the node n should come from the attributes of its children.

4.1.2 Plan Pattern Filling

With the generation rule, the filling procedure is defined as two steps. First, if the plan pattern does not have a valid plan structure, i.e., the leaf node is not a scan or join node and has less than two children, the scan node is used as the child of the

non-scan leaf node or the join node to make the plan pattern a valid plan structure. Then, for each non-scan node in the plan pattern with the valid plan structure, there are three cases. 1) If the child is a scan node, then randomly use one table to fill the scan node, and the attribute of the non-scan node is determined by its child. 2) If the non-scan node contains one scan node and one non-scan node as its children, the attribute of the scan node is determined by the projection of the non-scan node in order to make them joinable. 3) If the non-scan node contains two scan nodes as its children, then the attribute of the two scan nodes is filled with two related tables.

This procedure continues recursively upward until reaching the root node. Upon completion, the resulting structure forms a valid base plan, denoted by T_b.

4.1.3 Rule-based Plan Generation

The Rule-based Plan Generation method starts with filling the plan pattern to get the base plan T_b. Then, a new node is randomly added on top of the valid base plan to generate a new plan T_n. The attribute information of the newly added node is then filled following the same rule set, ensuring that T_n remains a valid plan. By iteratively repeating this procedure, a matching plan that satisfies the structural and semantic constraints can be obtained.

In conclusion, the rule-based plan generation method can solve the matching

plan problem, but it has a problem. The plan pattern is always located at the leaf, and the plan generation procedure grows the plan tree upward. For the plan pattern in the generated plan, its input diversity is harmed, as the input of the plan pattern is always the scan node.

4.2 Rule-Based Action Generation (RAG)

To address the problem of the Rule-based Plan Generation approach, it is important to generate a plan that grows the plan both upward and downward, starting from the plan pattern. One approach to achieve this is to mutate each node of a sample plan that contains the plan pattern. Specifically, the approach takes the plan pattern and a sample plan that contains the plan pattern as the input, and the approach edits every node of the sample plan that is not in the plan pattern. The edit operation includes insertion, extension, and replacement. By doing so, the sub-plans that are above and below the plan pattern will be extended by the insertion and extension operation, and the diversity of the generated plan is boosted by the replacement operation.

4.2.1 Plan Mutation

The sample plans and a plan pattern P are given by the user. Each sample plan T is a matching plan for the plan pattern P. Plan mutation refers to the application

of mutation operations—namely insertion, replacement, deletion, and skip—on the nodes within a specified plan pattern. Nodes outside the designated plan pattern remain unaffected by these operations. The following definitions formally characterize each mutation type:

1. Insertion. Given a sample plan $T = \{t\}$, where $t = (o_t, A_t, C_t)$, inserting a new node $t_{new} = (o_{new}, A_{new}, C_{new})$ in between the node v and its parent u yields a new plan T', where $T' = T \cup (t_{new}), C_{new} = C_{new} \cup v, C_u = C_u \cup t$.

2. Replacement. Given a sample plan $T = \{t\}$, where $t = (o_t, A_t, C_t)$, replacing the node $v \in C_u$ with a new node t_{new} yields a new plan $C_u = (C_u \setminus \{v\}) \cup \{t_{new}\}, C_{new} = C_v$, where u is parent of v.

3. Skip. Given a sample plan $T = (V, E)$, where V denotes the set of nodes and E denotes the set of parent-child relationships, the skip operation means keeping the current node untouched.

Using the mutation operations defined above, a given plan can be transformed into an alternative plan through a sequence of mutation actions. It should be noted, however, that the application of a mutation action may have non-local effects, potentially propagating to parent or descendant nodes of the mutated node. In the following, we provide a detailed analysis of each mutation action.

The impact of insertion depends on the type of node introduced into the plan. For instance, 1) inserting a hash join node between the current node v and its parent

node u introduces a new table into the output of v. Consequently, the output of the hash join node becomes the input to u, thereby altering the input cardinality of u. 2) If the inserted node has a single child, such as a sort operator, placing it between u and v results in the input of u being sorted. Formally, the insertion procedure is performed as follows: the current node v is assigned as a child of the new node v_{new}; the new node v_{new} is then inserted as a child of the parent node u; finally, v is removed from the list of children of u.

When replacing nodes, each plan node can be regarded as having specific rules, such as the logical result it must produce and the physical properties it must provide to its parent. During replacement, these rules must be preserved. For example: (1) a sequential scan may be replaced by an index scan; (2) join operators, such as hash join, merge join, and nested loop join, may be interchanged. If a node is replaced by a hash join operator and none of its children are hashed, a hash node is inserted between the first child and the hash join node. Similarly, if a node is replaced by a merge join operator and its children are not sorted, a sort node is introduced between the merge join operator and the child that lacks a sort property.

To execute the replacement operation in practice, the procedure is as follows: assign the children of the current node v to the children list of the new node v_{new}; insert v_{new} into the children list of the parent node u; finally, remove the current node v from the children list of u.

When discussing the deletion operation, a node may be removed only if the resulting plan continues to return the same multiset of rows and all ancestor nodes still receive any physical properties upon which they depend. Formally, given a parent node $u = (o_u, A_u, C_u)$, deleting a node $v \in C_u$ is valid if and only if the following conditions hold:

$$\mathcal{R}(C_u) = \mathcal{R}(C_u \setminus \{v\}) \quad \text{and} \quad \mathcal{P}(u) \subseteq \mathcal{P}'(u),$$

Where $\mathcal{R}(T)$ denotes the output multiset of rows of plan T, $\mathcal{P}(u)$ is the set of physical properties required by ancestor node u, and $\mathcal{P}'(u)$ is the set of properties still satisfied after the deletion.

If the deleted node is the sole provider of such a property, the corresponding ancestors must be modified so that they no longer depend on it. In this case, the operation is no longer a pure deletion. Since applying a deletion to the current node has a significant probability of altering the attributes of all ancestor nodes, the operation is difficult to implement in practice. Therefore, the deletion operation will not be further considered in this thesis.

4.2.2 Rule-based Action Generation

One of the solutions for the plan mutation problem is a rule-based plan mutation algorithm, which we call this approach as Rule-based Action Generation. Given the sample plan T, the algorithm traverses each node in the pattern. For the node that does not belong to the plan pattern P, the algorithm randomly applies one of the three mutation actions to the node. After the traversal, the algorithm outputs the mutated plan T'.

The functions in the pseudocode, for example $Schema(v)$, $AddToChild(c, p)$, $getReplaceableNT(v)$, $Random(NT)$, and $Scan(joinTable)$, are auxiliary functions that help with the algorithm. $Schema(v)$ takes a plan node as input, and returns a list of tables that can be joined with the given node. $AddToChild(c, p)$ takes a child node and a parent node as input, and adds the node c as the child of the node p. $getReplaceableNT(v)$ takes a plan node as the input, and returns a node whose node type is the same as the input plan node type. $Random(NT)$ takes a list of node types as input, and randomly returns a plan node with one of the node types. $Scan(joinTable)$ takes a database table name as input, and returns a scan node of the input table.

The algorithm takes the sample plan T and plan pattern P as input. There are hard-coded mutation operation list Op and node type list NT provided by the

Algorithm 1 Rule-based Plan Mutation Algorithm

Input: Sample Plan T, Plan Pattern P
Output: Return the mutated sample plan that consists of the plan pattern

function RULE-BASED PLAN MUTATION ALGORITHM(P, T)
  Op = {Insertion, Replacement, Skip}
  NT = {Hash Join, Merge Join, Nested Loop Join, Hash, Sort, Aggregate}
  for all $v \in T$ **do**
    $op \leftarrow$ Random(Op)
    if $v \notin P$ **then**
      if $op = Insertion$ **then**
        $nt \leftarrow$ Random(NT)
        if isJOIN(nt) **then**
          $joinTable \leftarrow$ SCHEMA(v)
          $scanNode \leftarrow Scan(joinTable)$
          addToChild(v, nt)
          addToChild($scanNodem, nt$)
          $nt.output = v.output \cup scanNode.output$
          addToChild(nt, u)
        else
          addToChild(v, nt)
          $nt.output = v.output$
          addToChild(nt, u)
        end if
      else if $op = Replacement$ **then**
        $nt \leftarrow$ getReplaceableNT(v)
        $nt.output = v.output$
        $nt.childList = v.childList$
        addToChild(nt, u)
      else if $op = Skip$ **then**
        skip
      end if
    end if
  end for
end function

algorithm. For each node v in the sample plan, the algorithm randomly picks a mutation operation *op*. If the node v is in the pattern P, the algorithm ignores the node. Otherwise, if the mutation operation is insertion, the algorithm randomly chooses a node type nt from the node type list NT. If nt is a join, implement the join insertion we discussed earlier; otherwise, implement as the situation when the new node has one child. If the operation *op* is replacement, then the node type nt is randomly picked from the replaceable node type list, and then the algorithm implements the replacement operation. If the operation is the skip operation, then the algorithm does nothing to the node.

A limitation of the rule-based mutation algorithm is that the node relationship distribution of the mutated sample plan T' does not necessarily align with the node relationship distribution of the plan patterns generated by the optimizer. Consequently, when T' is translated into a SQL query and executed within the DBMS, the actual execution plan may diverge from T'.

5 QueryMorpher

5.1 Basic Idea

To address *MPG*, we reformulate it as a sequence-to-sequence problem by employing a node-to-sequence transformation, which simplifies the task. In this formulation, a learned sequence-to-sequence model is required to determine which mutation action should be applied to each node. Since existing sequence-to-sequence models are not capable of generating a complete action instruction string for mutating the entire plan in a single inference step, we instead design the model to output an action instruction string for mutating one node at a time during each inference step.

To implement plan mutation based on the action instruction string, a compiler is required. Since the learned model may still produce erroneous outputs, the compiler incorporates an *Auto-Correction Algorithm* to validate the action instruction string prior to applying the mutation action. Once validated, the action instruction string is executed on the given sample plan T, yielding the mutated sample plan T'.

To translate mutated sample plans into executable queries, a plan-to-query

translation algorithm is employed. This ensures that the generated queries can trigger the optimizer to reproduce the corresponding mutated plans.

The proposed framework directly addresses the *MPG Problem*. The compiler guarantees that the produced plans preserve the desired structural characteristics while maintaining generation diversity. In parallel, the plan-to-query translation algorithm enables the corresponding queries to faithfully reproduce the targeted execution plans. Consequently, the queries generated by QueryMorpher achieve structural consistency while simultaneously exhibiting diversity.

5.2 Node-to-Sequence

The Node-to-Sequence algorithm reformulates the *MPG Problem* as a sequence-to-sequence learning task. Specifically, it provides the sequence-to-sequence model with contextual information about the current node by translating the node into a structured input string representation. Figure 5.3 demonstrates an example of converting a hash join node to a sequence.

To enable the sequence-to-sequence model to capture both the local context of the current node and its hierarchical relationships, the input representation incorporates information from the current node, its parent node, and its child nodes. For each node, the representation includes the following components: (1) the operator type of the SQL query execution plan (e.g., `Hash Join`); (2) the projected

attributes; (3) the join condition, if the operator type corresponds to a join operator; and (4) the filter condition, if applicable. The operator type specifies the database operation executed by the DBMS, guiding the model in learning which operator types are most suitable for newly inserted nodes during mutation. The projected attributes specify which attributes are propagated upward in the plan, assisting the model in determining which attributes should be carried by a new node. The join condition is essential when the node operator type is a join, as it defines the join behavior, while the filter condition captures the logical constraints applied at the node level.

Given a node $t = (o_t, A_t, C_t)$ from the plan pattern, Node-to-Sequence algorithm extract o_t, A_t of the node, and also extract o_c, A_c where $c \in C_t$. The sequence s is built with the extracted element, i.e. $s = \{o_t, A_t, o_c, A_C\}$. The sequence-to-sequence model takes s as input and outputs the action sequence $a = (m, t, j, l)$. The m denotes the mutation action type, t denotes the node type, j represents the set of projection columns of the node, and l denotes the condition logic (if required by the node type).

5.3 Sequence-to-Sequence Model

We explored multiple sequence-to-sequence models for the action instruction string generation task. We first tried to use the GAN model to generate the SQL queries

as the work proposed by Sun et al. [20]. However, the queries generated by the GAN model are hard to validate in syntax and semantics. We also tried to use the discrete diffusion model to generate SQL queries. The discrete diffusion model performs better than the GAN model in the query syntax, but the fatal issue for the diffusion model is that the generated action instruction strings are limited to a certain length. Since the objective of the sequence-to-sequence model is to understand the context of the input sequence and output the sequence with no limitation on the length, we finally chose the BART model.

The BART (Bidirectional and Auto-Regressive Transformer) model is a powerful sequence-to-sequence generative model that utilizes the strengths of both a bidirectional encoder and an autoregressive decoder, making it particularly well-suited for structured generation tasks, such as generating action sequences for execution plan mutation.

In QueryMorpher, BART serves as the core that learns to map node sequences into action sequences. Given an node sequence s as the input of the BART model, $\mathbf{s} = (nodeType, output, cond, plans)$, where *output* represents output column names c of the node, i.e. $\mathbf{output} = (c_1, c_2, \ldots, c_n)$. The *cond* represents a join condition or filter condition, and the *plans* represents the input plans of the node. *plans* and *cond* are optional depending on the node type. Then, the BART model learns a

mapping:

$$\mathbf{a} = M(\mathbf{s}) = \text{Dec}(\text{Enc}(\mathbf{s})),$$

Where $a = (m, t, j, l)$ is the action instruction sequence that describes how to mutate the input plan.

5.4 Compiler

Since the output of the BART model only has a validation rate of about 10%, it is necessary to validate the action instruction string before applying it to the plan mutation. The common error of the action instruction string includes the mismatch of the mutation action type and the operator type, and the invalid join condition.

Preliminary results shows that implementing the rule-based validation during the sequence generation does not work as we expected, we therefore chose to validate the action instruction string after the sequence generation. Then, an *Auto-Correction Algorithm* is employed to minimally modify the predicted sequence while preserving its original intent. The validated action instruction sequence produced by this algorithm is then passed to the plan mutation procedure, which applies the corresponding mutation to the given sample plan.

5.4.1 Auto-Correct Algorithm

Since the action instruction sequence generated by the BART model may contain errors that cause failures in subsequent plan mutation or plan-to-query translation, an auto-correction mechanism is required to ensure a higher rate of valid generation. The objective of the *Auto-Correct Algorithm* is to apply the heuristic rule-based correction to transform an invalid action instruction sequence into a valid one, while preserving the semantic intent of the original sequence.

The Auto-Correct Algorithm takes the current traversed node $n = (o_n, A_n, C_n)$ and the action instruction sequence a as input. The node used in the mutation $u = (o_u, A_u, C_u)$ is then built based on a. If o_u is not a join, then the attribute of u is filled based on a. However, if o_u is a join, the algorithm adds a scan node v under u as its child node. The attribute of v follows the rules: 1) The projection columns comes from the same table, and 2) at least one column in the projection column set of v should have the primary key-foreign key relationship with at least one column in the projection column set of n, i.e. n and v are joinable.

When u is built, the algorithm first checks the mutation action type m. If m is a replacement action but the node type of n and u are not matched, i.e. $o_n \neq o_u$, the mutation action will be changed based on the node type of o_u. If o_u is join, i.e. $|C_u| = 2$, the action type is changed to extension action; otherwise, the action type

is changed to insertion action. Then, the algorithm checks the projection columns. If m is not a join, the columns in A_u should also in A_n, i.e. $A_u \in A_n$, otherwise the columns in A_u but not in A_n is deleted. If m is a join, then the columns in A_u should be in the union of A_n and A_v, i.e. $A_u \in A_n \cup A_v$. Finally, the algorithm checks the join condition when m is join. The join condition should follow the rule described in the Chapter 4.1.1. Otherwise, the join keys will be replaced by two columns selected from its two children. The selected columns should have a primary key-foreign key relation.

Algorithm 2 Auto-Correct Algorithm

Input: Traversed node $n = (o_n, A_n, C_n)$; instruction sequence a

$u \leftarrow \textsc{BuildNode}(a)$ ▷ parses a to obtain o_u, A_u, C_u

$m \leftarrow \textsc{ActionType}(a)$

if $m = \text{replace} \ \wedge \ o_n \neq o_u$ **then**

$$m \leftarrow \begin{cases} \text{extend}, & |C_u| = 2 \\ \text{insert}, & |C_u| \neq 2 \end{cases}$$

end if

if $m \neq \text{join}$ **then**

$A_u \leftarrow A_u \cap A_n$ ▷ enforce $A_u \subseteq A_n$

else

$A_u \leftarrow A_u \cap (A_n \cup A_v)$ ▷ enforce $A_u \subseteq A_n \cup A_v$

$J_u \leftarrow \textsc{JoinCond}(u)$

require $J_u \in \mathcal{J}_{4.1.1}$ ▷ rule-compliant join

if $J_u \notin \mathcal{J}_{4.1.1}$ **then**

$(x^\star, y^\star) \in \arg \max_{x \in A_v,\, y \in A_n} \mathbf{1}\{\text{PKFK}(x, y) = 1\}$

$J_u \leftarrow \{(x^\star, y^\star)\}$

end if

end if

5.4.2 Plan Mutation Algorithm

The *Plan Mutation Algorithm* is designed to mutate a given sample execution plan according to the provided action instruction sequence. Given a validated action instruction sequence as input, the algorithm first extracts the node type of the new node to be introduced by the mutation action and retrieves the corresponding node template. Next, the template is populated using the node-specific information specified in the action instruction sequence, such as the projected output attributes and join conditions. The completed template is then applied in the mutation step, where the mutation action (e.g., node insertion or node replacement) is executed in accordance with the instruction sequence. The output of the algorithm is a mutated execution plan that conforms to the desired structural pattern.

5.5 BART Training

5.5.1 Training Dataset Construction

Given the dataset, first use their built-in query generation methods to generate queries. Then, execute queries in the DBMS to get their query execution plans, and then traverse each node in the plans. To be specific, given a plan T, the algorithm mutates the node $v_1 \in T$ based on the randomly generated action instruction string a_1 to produce a mutated plan T_1. Translating T_1 into the query and analyzing the

query with the DBMS, we can get the execution plan T_1'. T_1 and T_1' may not be exactly the same, as the randomly generated action instruction string may not be the most efficient, which will be optimized by the optimizer in the DBMS. Intersecting the sub-plan sets of T_1 and T_1', we can get a set of sub-plans that exists in both T_1 and T_1'. Then, the score s_1 of action instruction string a_1 is calculated as the size of the intersection set divided by the size of the sub-plan set of T_1, i.e.

$$s_1 = \frac{\left|\mathcal{S}(T_1') \cap \mathcal{S}(T_1)\right|}{\left|\mathcal{S}(T_1)\right|}.$$

Where $\mathcal{S}(T_1)$ is the sub-plan set of T_1 and $\mathcal{S}(T_1')$ is the sub-plan of T_1'. Based on T_1, the algorithm mutates the node v_2 based on the randomly generated action instruction string a_2 we can get T_2. By doing the same process described above, we can get the score s_2 of the action instruction string a_2. If the score of an action string is greater than the score of its previous action string, i.e., $s_2 > s_1$, we first translate the current node to the node string using the node-to-sequence algorithm, and concatenate the node string with the current action instruction string a_2 as the data of the training dataset.

5.5.2 Training and Inferencing

In the BART model training, the input is the tokenized node string, and the label is the tokenized action instruction string corresponding to the node string.

In inferencing, the input is the tokenized node string, and the BART model will output an action instruction string.

5.6 Plan-to-Query Translation Algorithm

Plan-to-Query Translation Algorithm takes the fully mutated sample plan as the input, then uses the Node-to-CTE module to split the plan into blocks and translates each block into CTEs. Finally, the Plan-to-Query concatenates the CTEs to build the query. This process is presented in Figure 5.2.

5.6.1 Node-to-CTE

To regulate the database optimizer in generating the query execution plan with the same structure as the mutated plan, we use the Common Table Expression (CTE).

Specifically, given the mutated plan tree, the algorithm partitions it into discrete blocks of simple sub-plans. Each block contains one, two, or three operators depending on the currently traversed node type. If the node type is a hash join, the block contains the hash join and its child hash node. If the node type is a merge

join, the block contains the merge join and two sort nodes; otherwise, the block contains one node only. We determine block boundaries via a post-order traversal. If the node currently traversed is a hash or sort, while its parent is the hash join or the merge join, we skip this node; otherwise, we consider this node as a distinct CTE with a distinct alias.

Because many database optimisers may inline or reorder a plain query differently, using CTEs helps enforce a stronger correspondence between the mutated structure and the final executed plan. In the case of PostgreSQL versions later than 12, if a CTE is used only once, PostgreSQL may inline it. To solve this issue, we built a dummy CTE for each CTE block that references the CTE block. In other words, the dummy CTEs act as the "optimization fence".

Finally, the generated CTEs are concatenated and ordered by their corresponding block location in the plan.

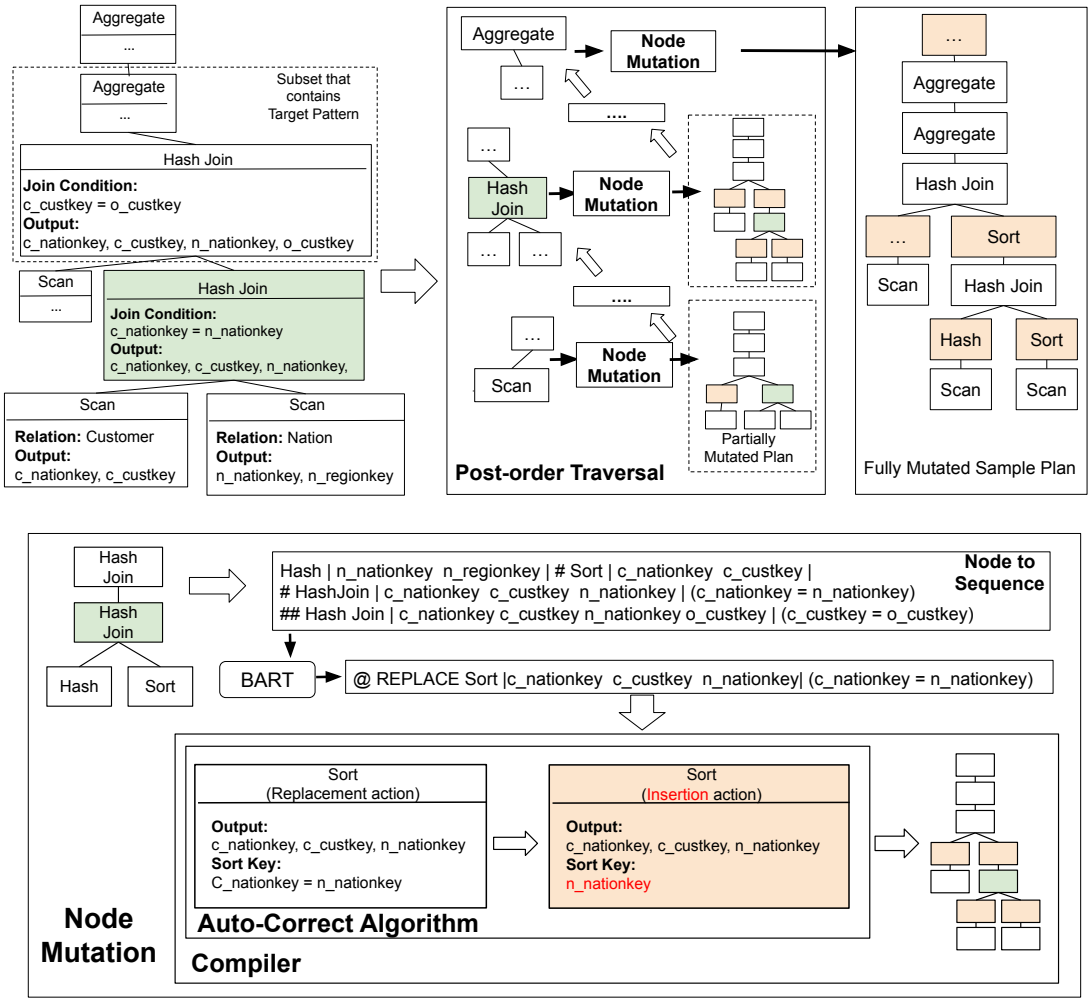

Figure 5.1: QueryMorpher Structure.

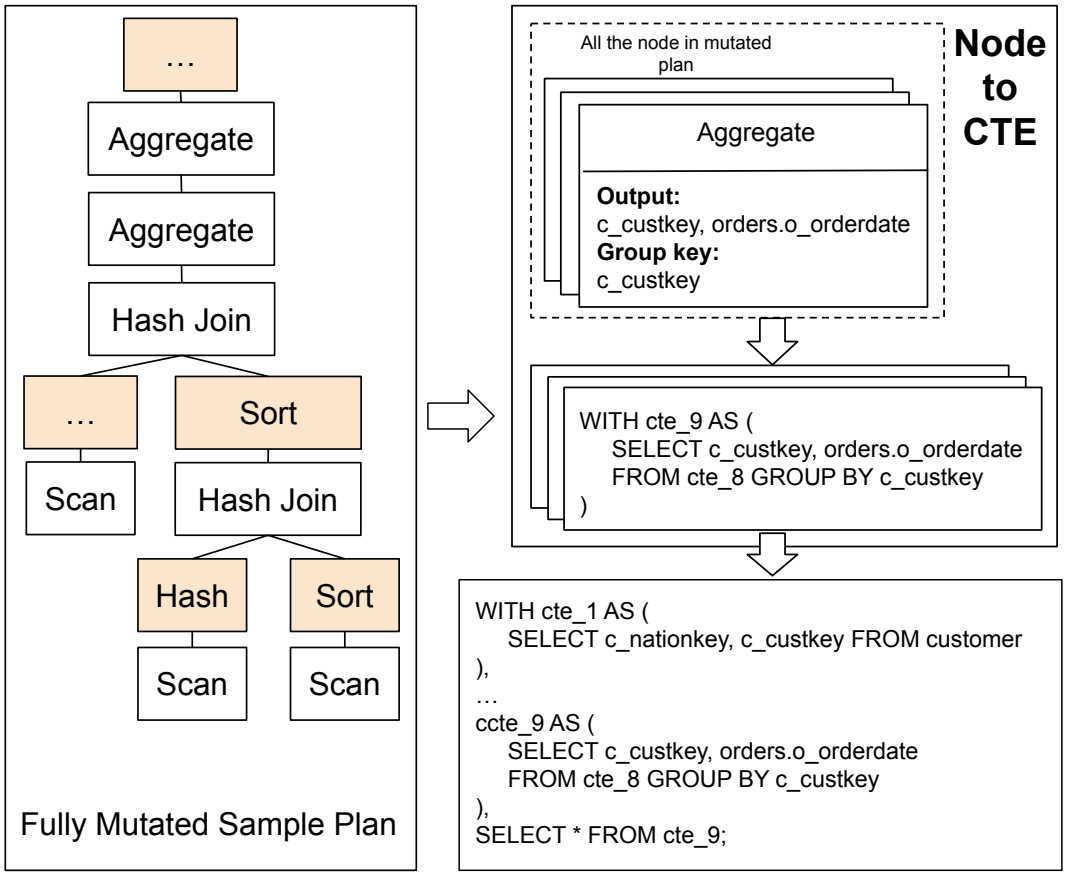

Figure 5.2: Plan to Query

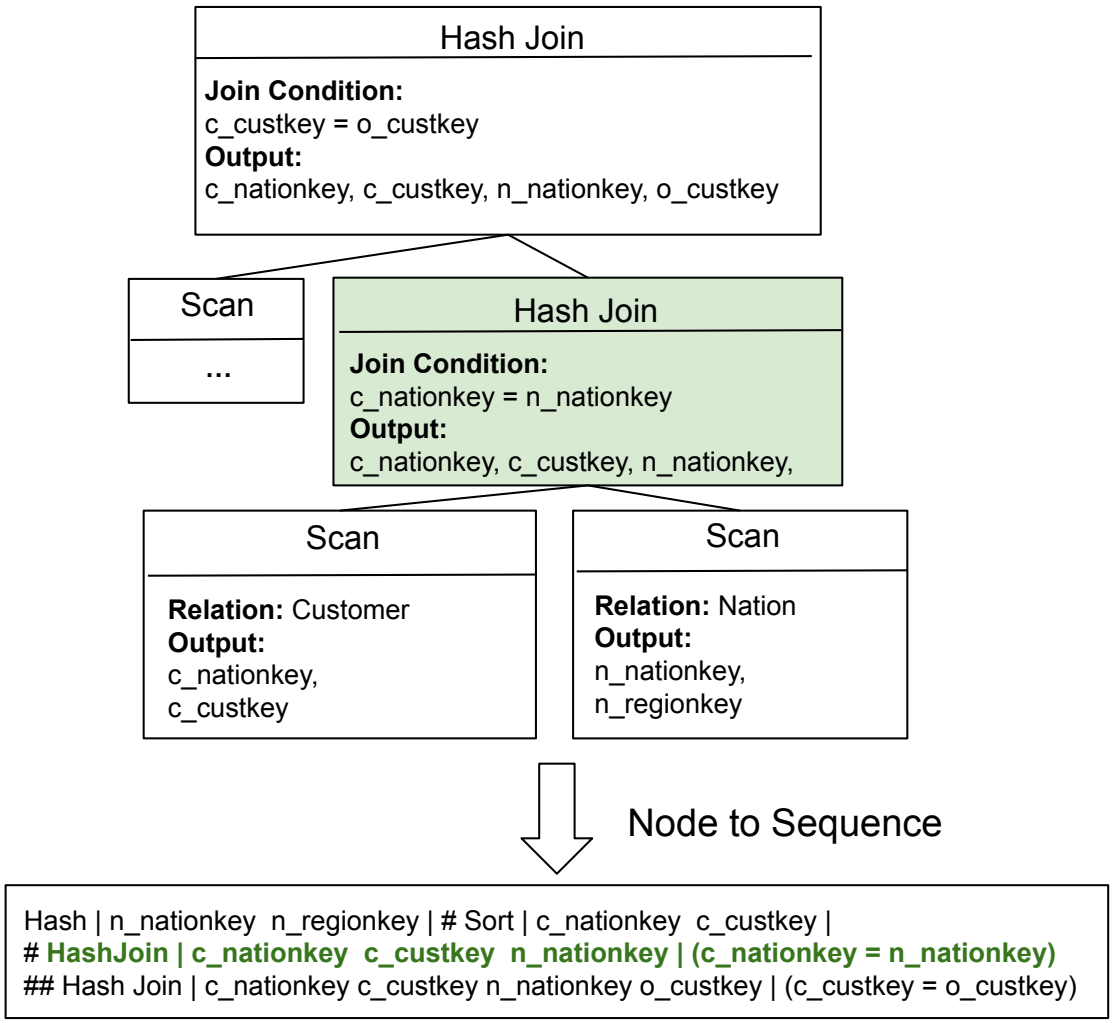

Figure 5.3: Node to Sequence

6 Experiments

6.1 Experiment Setting

6.1.1 Environment

Our experiment uses AMD Ryzen Threadripper 3960X 24-Core Processor, 128G RAM, and 3.5 TB storage. The DBMS we use is PostgreSQL 15.3 running on the Ubuntu operating system.

6.1.2 Dataset

We utilize two datasets from TPC benchmarks: TPC-H [2] and TPC-DS.

TPC-H. TPC-H is commonly used in the field of database performance evaluation, execution plan analysis, and query optimization research. In the experiment, we use a synthetic dataset generated by DBGEN with the scale factor of 10. We generate 5,000 queries using the built-in query generation tool, QGEN, and also generate their execution plan by analyzing those queries in the DBMS.

TPC-DS. TPC-DS is the dataset to test the proposed framework under a more complex use case. The TPC-DS dataset includes 7 fact tables and 17 dimensions, mirroring a multichannel retailer. In the experiment, we use a synthetic dataset generated by DBGEN with the scale factor of 10, and use QGEN to generate 5,000 queries.

6.1.3 Target Pattern

The target pattern 3 is the subplan pattern that we desire in the generated plans.

In the experiment, we group the target patterns by their height. We provide 5 groups of target patterns from a height of 2 to a height of 6. The target pattern rate and the diversity of methods for each group are the average value of each group. Each target pattern is schema-feasible sampling from the operators of hash join, merge join, hash, and sort.

6.2 Evaluation Metrics

Based on our objective of the method, we use weighted structure diversity and target pattern rate to measure the performance of QueryMorpher.

6.2.1 Plan Diversity

In the MPG problem, an important requirement is to measure diversity across a set of plans. Diversity measurement ensures that the set of generated plans explores a wide range of structural alternatives rather than collapsing onto a small number of nearly identical plans. The plan diversity measures the diversity of generated plans from each method. We use Tree Edit Distance (TED) [26] to calculate the diversity. Tree Edit Distance [26] is the minimal-cost sequence of node edit operations (insert, delete, modify) to transform one tree to another.

Let T_i and T_j be two SQL query execution plan trees. An *edit script* $s \in \mathcal{S}(T_i \rightarrow T_j)$ is a finite sequence of node operations. Each operation s_i is one of:

- **Insertion:** insert a node v' with label $l(v')$, with cost $c_{\text{ins}}(v')$;
- **Deletion:** delete a node v, with cost $c_{\text{del}}(v)$;
- **Substitution:** relabel a node v_1 to v_2, with cost $c_{\text{sub}}(v_1, v_2)$.

We define $s_{min} = \underset{s \in \mathcal{S}}{min}\ s$ as the shortest sequence in the set of $\mathcal{S}(T_i \rightarrow T_j)$. Then *Tree Edit Distance* (TED) between T_1 and T_2 is calculated as the minimal total cost among all valid edit scripts transforming T_1 into T_2:

$$\text{TED}(T_i, T_j) = \min_{S \in \mathcal{S}(T_i, T_j)} \sum_{s_i \in S} c(s_i) \tag{6.1}$$

To better compare the diversity among different methods, we normalized the final output. We first do the pairwise normalization:

$$\mathrm{TED_N}(T_i, T_j) = \frac{TED(T_i, T_j)}{|T_i| + |T_j|} \tag{6.2}$$

Where $|T_i|$ denotes the number of nodes in T_i and $|T_j|$ denotes the number of nodes in $|T_j|$. After the pairwise normalization, we also normalize the weighted TED value of the batch.

$$D_{\text{plan}} = \frac{1}{N_{\text{total}}(N_{\text{total}} - 1)} \sum_{i=1}^{N_{\text{total}}} \sum_{j=i+1}^{N_{\text{total}}} TED_N(T_i, T_j) \tag{6.3}$$

where:

- D_{plan} denotes the *plan diversity*,
- $TED_N(T_i, T_j)$ is the *pairwise normalized tree edit distance* between the generated plans T_i and T_j,
- N_{total} is the total number of generated plans.

6.2.2 Target Pattern Rate

The target pattern rate is calculated as the number of generated plan that contains the target pattern divided by the total number of generated plans.

$$R_{\text{target}} = \frac{N_{\text{target}}}{N_{\text{total}}} \tag{6.4}$$

where:

- R_{target} denotes the *target pattern rate*,
- N_{target} is the number of generated plans that contain the target pattern,
- N_{total} is the total number of generated plans.

6.2.3 Plan Fidelity

The fidelity denotes how similar the raw plan and the final plan are after the raw plan is converted to the final plan. This chapter compares the fidelity of RPG, RAG, and QueryMorpher. In order to demonstrate how good each method is, we set the best case and label it as DBMS. The raw plans used by DBMS are the plans output by the optimizer, which are the optimized plans and also the most similar plans. In the following chapter, we will first introduce the fidelity calculation method and then analyze the result.

In the fidelity calculation, we define the plan T as a plan set S_T of all possible sub-plans, i.e. $S_T = \{t_i | t_i \in T\}$. For each pair of raw plan set S_r and the final plan set S_f, the fidelity is calculated as $\frac{|S_r \cap S_f|}{min(|S_r|,|S_f|)}$. The formula calculates the ratio of the common sub-plan set over the plan set of the smaller plan, and the result is between 0 and 1. When two plan sets share nothing in common, the fidelity is 0, and when the smaller set is entirely contained in the larger set, the fidelity is 1. Because the plan-to-query algorithm translates all the nodes in the raw plan to CTEs, the final plan can only be larger than the raw plan, and this prevents the fidelity calculation in the situation where the final plan is the subplan of the raw plan.

6.3 BART Model Setting

6.3.1 Dataset for BART

The dataset used for the BART model is generated as follows. We first randomly select 1,000 distinct query execution plans, and then use RAG to mutate the selected plans. For those mutation instruction sequences generated by RAG that successfully guide the optimizer to reproduce the mutation actions, we collect both the mutation instruction sequences and the corresponding node sequences. The dataset consists of 12,114 rows of sequences in total. We split the dataset as 70%

Model Variant	model dim	enc layer	dec layer	enc head	dec head	dropout
BASE	512	4	4	8	8	0.3
LARGE	768	6	6	12	12	0.3

Table 6.1: Model Hyperparameter

Model Variant	Learning Rate	Batch Size	Epochs
BASE	1e-4	8	20
LARGE	1e-4	8	20

Table 6.2: Training settings

for training, 15% for validation, and 15% for testing.

6.3.2 Training Setup

The Table 6.1 demonstrates the hyperparameters of the BART model used in the experiment, and the Table 6.2 demonstrates the training setting for the BART model.

6.4 Baselines

Since there is no existing proposed method for query generation with execution plan structure constraint, we compare the QueryMorpher with three other methods proposed by us. The methods are Rule-based Plan Generation, CTE Generation, and Rule-based Action Generation.

Rule-Based Plan Generation (RPG). We use the baseline method discussed

in Chapter 4.1, which consists of two main phases: target pattern filling and node generation.

Rule-Based Action Generation (RAG). We also use the baseline method discussed in Chapter 4.2.2, which mutates sample plan nodes based on rules.

CTE Generation (CTEG) CTE Generation method first takes the target pattern as input, and fills the target pattern as described in the Chapter 4.1.2. Then, translate the filled base plan to the CTE as introduced in the Chapter 5.6.1. Finally, randomly generate a SQL query that involves the CTE. The output of the CTEG is a SQL query with the CTE.

Because QueryMorpher and RAG generate plans by mutating the sample plan, their generation diversity is sensitive to the number of the given sample plan. In this case, for RAG and QueryMorpher, we use 10 sample plans in the generation for baseline fairness. For the RPG and CTEG, as they benefit from the stronger randomization, they are limited to using one filled plan pattern to generate plans.

6.5 Experiment Result

Our experimental objective is to assess (i) the ability of each method to realize the desired target pattern (Target Pattern Rate), (ii) the structural faithfulness of realized plans to intended plans (Raw→Final Plan Fidelity), and (iii) the diversity of generated plans. We study sensitivity to target pattern height, node mutation

rate, and model capacity, and we evaluate generalization to unseen patterns and a second DBMS. For each method, we fix the budget to 500 attempts. Also, we set the random seed to 0.

In the following, we will demonstrate the result through line plots and bar plots. The shade in the line plot and the error bar in the bar plot are the 95% confidence interval of the data.

Before the experiment, we define the plans directly generated by the method before inputting them to the Plan-to-Query Algorithm as the raw plans. Translating the raw plan to the query using the Plan-to-Query algorithm and analyzing the query with the DBMS can get the execution plan. We define this plan as the final plan. Also, in the experiment, we only consider the raw plans whose corresponding queries can be successfully executed by the optimizer.

6.5.1 Target Pattern height

We first show the impact of the height of the target pattern on different methods. For simplicity, we vary the number of levels in our target pattern P (l_P) from 2 to 6. Specifically, we generate 9 distinct target patterns for l_P, and we use l_P to generate 100 plans for each method. For RPG and CTE, l_P is directly used in the generation, while for the RAG and QueryMorpher, we first filter the sample plans that contain l_P, and use them in the plan generation. The generation settings are:

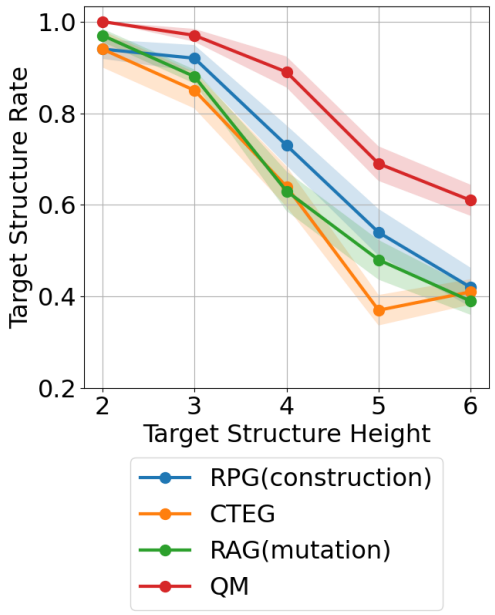

Figure 6.1: Final Plan Target Pattern Rate with different Target Pattern Height using TPC-H dataset

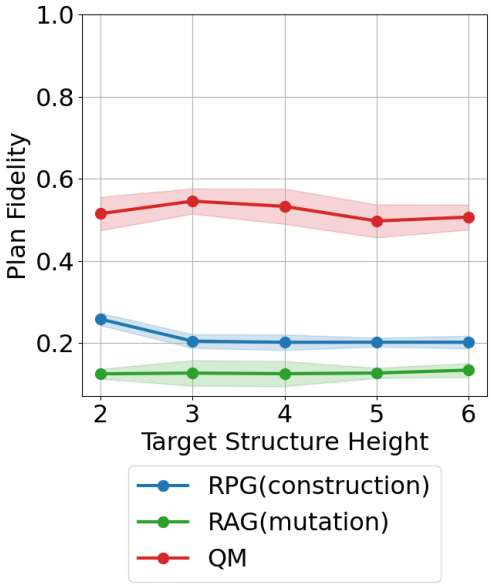

Figure 6.2: Plan Fidelity with different Target Pattern Height using TPC-H dataset

1) node mutation number to be 6, 2) the number of sample plans used for the RAG and QueryMorpher is 10. The evaluation has three parts: the target pattern rate, plan fidelity, and diversity. We calculate the target pattern rate for the final plan, and the diversity includes 2 parts: 1) The diversity of raw plans, labeled as "Raw Plan Diversity" in the figure. 2) The diversity of final plans, labeled as "Final Plan Diversity".

Begin with the plan fidelity score. Figure 6.2 represents the plan fidelity generated by each method across target pattern heights from 2 to 6. The x-axis is the

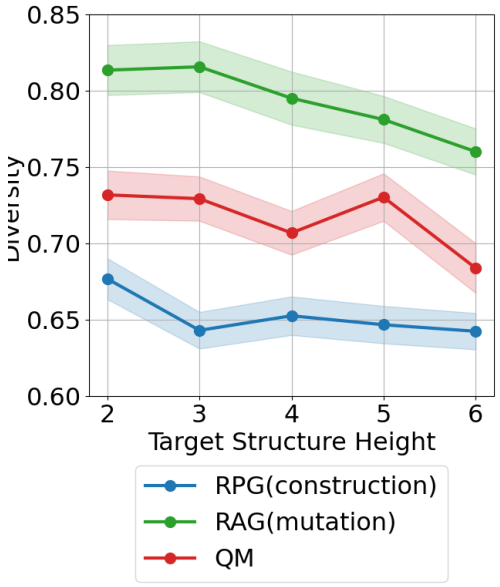

Figure 6.3: Raw Plan Diversity with different Target Pattern Height using TPC-H dataset

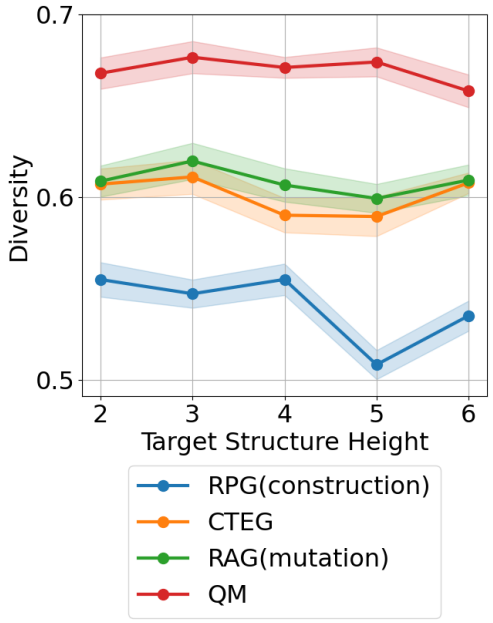

Figure 6.4: Final Plan Diversity with different Target Pattern Height using TPC-H dataset

target pattern height, and the y-axis is the fidelity score. The trend shows that plan fidelity for each method does not change as the target pattern height varies. QueryMorpher outperforms RPG and RAG, as QueryMorpher utilizes the learned model to assist the plan mutation, and the model is trained to give the mutation instruction that is reproducible by the optimizer.

As shown in Figure 6.1, the target pattern rate (TPR) of the final plans generated by each method is evaluated across target pattern heights ranging from 2 to 6. The x-axis represents the target pattern height, and the y-axis denotes the TSR.

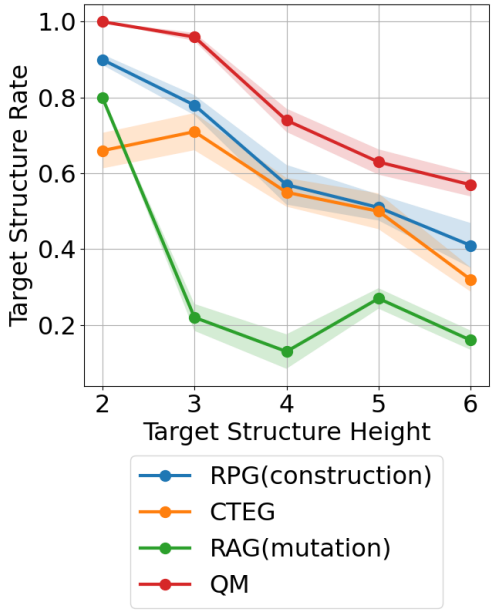

Figure 6.5: Final Plan Target Pattern Rate with different Target Pattern Height using TPC-DS dataset

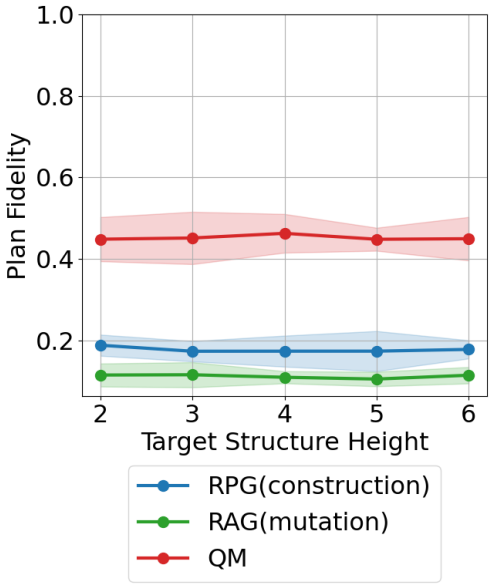

Figure 6.6: Plan Fidelity with different Target Pattern Height using TPC-DS dataset

Overall, the TSR decreases as the target pattern becomes more complex. Among all methods, CTEG performs the worst because it defines only the target pattern, while the remaining parts of the plan are determined by the DBMS optimizer, which introduces variations. In contrast, RPG achieves a higher TSR than RAG because its target pattern is positioned at the bottom of the plan tree—meaning only the upper structures influence the final pattern. Meanwhile, RAG places the target pattern in the middle of the plan, making it vulnerable to interference from both upper and lower substructures. In comparison, QueryMorpher achieves the

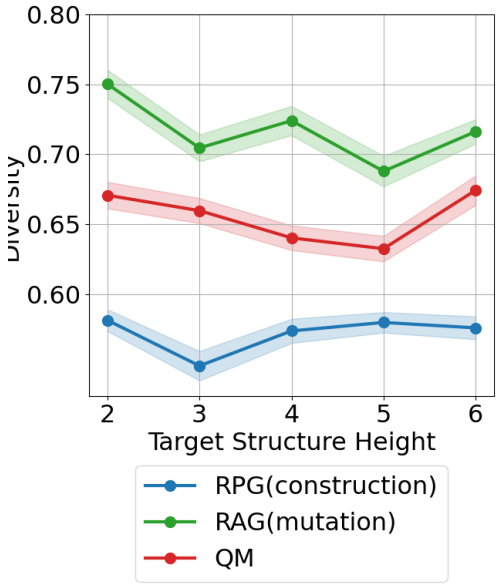

Figure 6.7: Raw Plan Diversity with different Target Pattern Height using TPC-DS dataset

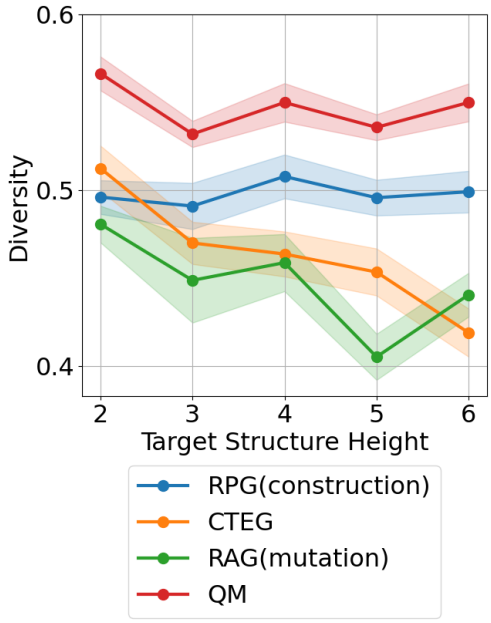

Figure 6.8: Final Plan Diversity with different Target Pattern Height using TPC-DS dataset

highest TSR because it is trained to mutate execution plans while preserving the intended structure, thereby mitigating the optimizer's effect.

Moving from the target pattern rate to the diversity, Figure 6.3 demonstrates the diversity of the raw plans generated by RPG, RAG, and QueryMorpher using target patterns of different height. The trend is relatively flat for the RPG and QueryMorpher, and decreasing for the RAG. This indicates that the increasing target pattern height has little influence on RPG and QueryMorpher, but hurts the RAG. RAG and QueryMorpher have higher diversity than the RPG, as their

generated plans are mutated from 10 sample plans; their diversity score is on top of the diversity of the sample plans. The diversity of RAG is higher than the diversity of QueryMorpher, as RAG randomly mutates nodes while QueryMorpher mutates nodes that follow a learned distribution.

Figure 6.4 shows the diversity of the final plans. Similar to the Figure 6.3, the trend in this figure is also approaching a flat line, except for the CTEG, as we can only control the target pattern in CTEG; the rest of the plan generated by the CTEG is unstable, leading to its unstable final diversity. The diversity of each method drops in comparison with the Figure 6.3, while the diversity of QueryMorpher drops less, as the learned model is better at giving mutation instructions that are reproducible by the optimizer, and thus its diversity score is higher than the rest of the methods.

We also experimented on the TPC-DS dataset. In this experiment, we use sample plans that have a similar height to the sample plans used in the experiment using the TPC-H dataset.

The plan fidelity score, as shown in the Figure 6.6, shows the same trend in comparison with the Figure 6.2. This proves the robustness of the QueryMorpher in terms of the generated plan fidelity. However, the QueryMorpher's plan fidelity score drops in the Figure 6.6 as the database schema gets more complex.

Figure 6.5 demonstrates the target pattern rate of final plans generated using

the TPC-DS, varying the height from 2 to 6. In dealing with the more complex database schema, the advantage of the QueryMorpher drops. One interesting thing is that the RAG method performs the worst when the target pattern gets complex. This indicates that rule-based node mutation on the large sample plan with the complex schema significantly reduces the plan's reproducibility.

Furthermore, Figure 6.7 shows the diversity of raw plans generated by each method across the target pattern height from 2 to 6, using the TPC-DS dataset. This figure indicates that the target pattern height does not affect the diversity of the final plans generated by each method, whereas the more complex database schema decreases the overall diversity generated for each method.

In addition, the diversity of the final plans generated by each method with the dataset of TPC-DS is reported in Figure 6.8. From this figure, we can find that QueryMorpher still has the best performance, but the diversity difference between QueryMorpher and the method in the second place is less than the diversity difference in Figure 6.4.

In summary, all methods perform worse on the TPC-DS, but QueryMorpher still outperforms the other methods, indicating its database compatibility.

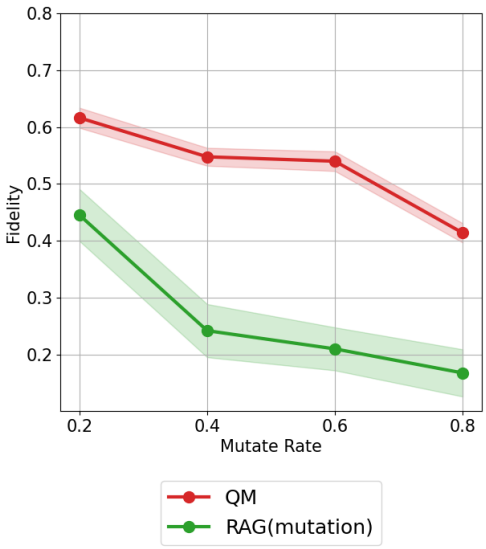

Figure 6.9: Plan Fidelity with different Node Mutation Rate using TPC-H dataset

6.5.2 Node Mutation Rate

This chapter introduces how the portion of nodes mutated in the sample plan affects the plan generation of QueryMorpher and the other baselines. We use the node mutation rate to represent the portion, and it is calculated as the number of nodes mutated divided by the total number of nodes in the sample plan. In this chapter, we fix the number of sample plans used for the RAG and QueryMorpher is 10, the target pattern height used in the plan generation is 5, and the size of the generated plan batch used for evaluation is 100.

Start with the plan fidelity score. Figure 6.9 demonstrates the generated plan

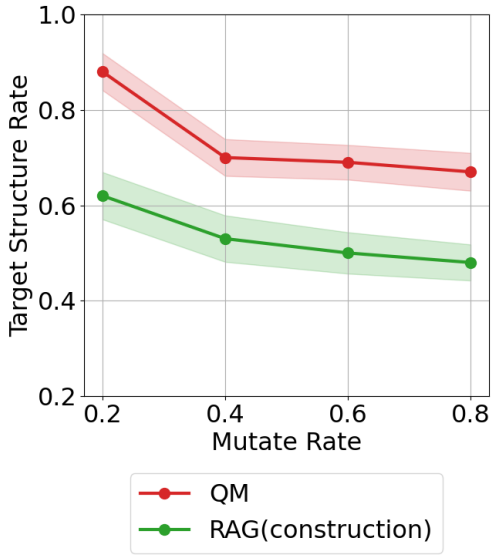

Figure 6.10: Final Plan Target Pattern Rate with different Node Mutation Rate using TPC-H dataset

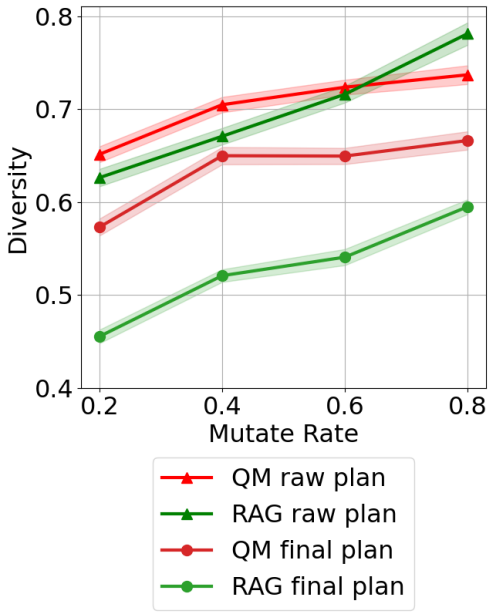

Figure 6.11: Raw Plan and Final Plan Diversity with different Node Mutation Rate using TPC-H dataset

fidelity of RAG and QueryMorpher across the different plan mutation rates. In the figure, we can find that as more nodes are mutated, the plan fidelity decreases. Benefit from the learned seq2seq model, QueryMorpher is better in the generated plan fidelity.

Figure 6.10 is the target pattern rate with different node mutation rates. The x-axis is the mutation rate, and the y-axis is the target pattern rate. Here we can find that as more node is mutated, the target pattern rate decreases because more mutated node affects the DBMS to reproduce the target pattern.

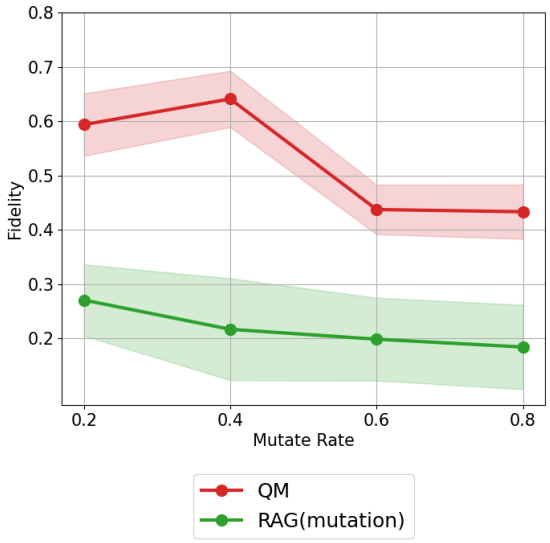

Figure 6.12: Plan Fidelity with different Node Mutation Rate using TPC-DS dataset

In addition, Figure 6.11 demonstrates the diversity of the raw plan and the diversity of the final plan. Here, we focus more on the diversity of the final plans generated by RAG and QueryMorpher. When the mutation rate of the sample plan is low, the diversity of the raw plan and the final plan generated by the 2 methods is similar. When the node mutation rate increases, the diversity of the raw plan and the final plan also increases, but the diversity of the raw plan increases faster. In comparison with the raw plan, the diversity of the final plan is harmed by the DBMS, but since the QueryMorpher is better at mutating nodes that can be reproducible by the DBMS, its diversity of the final plan is higher than the RAG.

We also did the experiment with the TPC-DS dataset.

From the Figure 6.12, the trend of the generated fidelity is decreasing as more

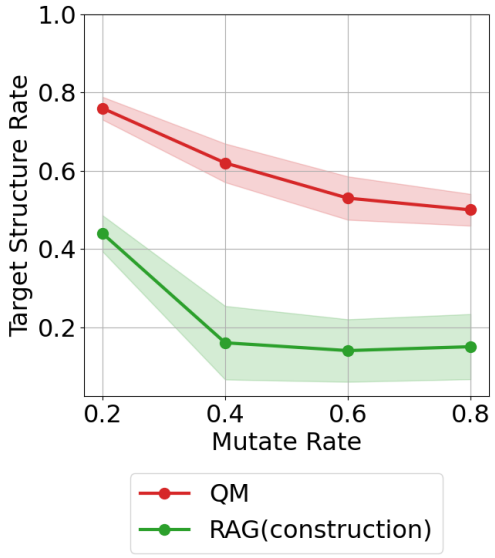

Figure 6.13: Final Plan Target Pattern Rate with different Node Mutation Rate using TPC-DS dataset

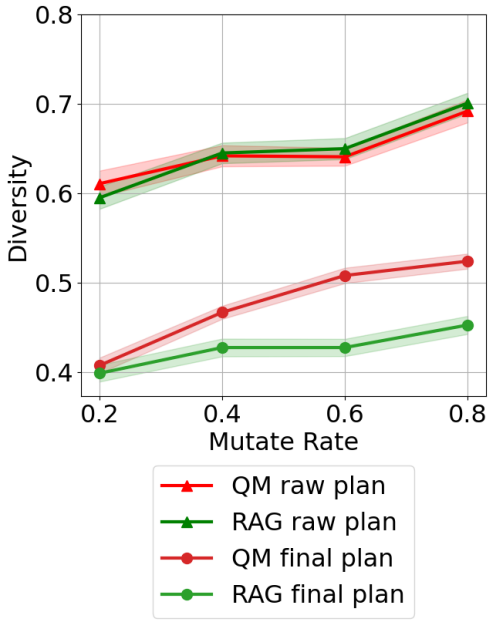

Figure 6.14: Raw Plan and Final Plan Diversity with different Node Mutation Rate using TPC-DS dataset

nodes are mutated in the plan, same as the trend in the Figure 6.9. This also proves the robustness of the QueryMorpher in dealing with different database schemas.

Also, consistent with the trend observed in Figure 6.10, Figure 6.13 is the target pattern rate generated by the QueryMorpher with different node mutation rates using the TPC-DS dataset. Similarly to the results from the TPC-H experiment, as more nodes are mutated in the sample plan, the target pattern rate decreases. This suggests that higher mutation rates introduce greater structural variation, thereby reducing the likelihood that the DBMS can reproduce the intended target

pattern.

The Figure 6.14 shows the diversity of the raw plan and the diversity of the final plan generated by the QueryMorpher and RAG with different node mutation rates using the TPC-DS dataset. In comparison with the results from the TPC-H dataset, as the sample plan gets complex, the overall diversity decreases. This indicates that the increased schema complexity of TPC-DS limits the extent of plan variation that can be effectively reproduced by the DBMS, thereby reducing both raw and diversity across methods.

6.5.3 BART model size

This chapter introduces the effect of the BART model size on the performance of the target pattern rate, plan fidelity, and diversity. In this chapter, we fix the node mutation number to be 6, the number of sample plans used for the RAG and QueryMorpher is 10, the target pattern height used in generation is 5, and the generated plan batch used for evaluation is 100.

A clear difference can be found in the Figure 6.15 that the framework with a learned model outperforms the rule-based methods. Also, we can see that using a larger model size helps the BART model to output the mutation instruction that is better reproducible by the optimizer.

In addition, the Figure 6.16 demonstrates the target pattern rate of plans gen-

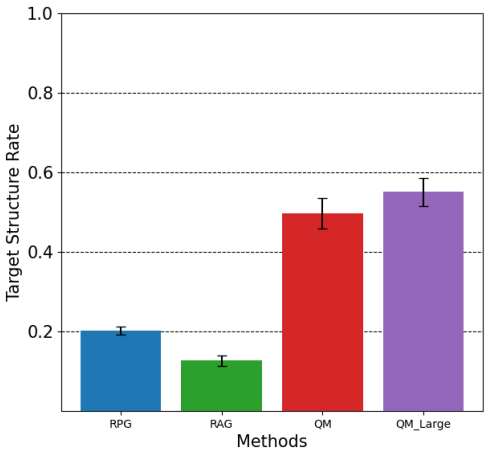

Figure 6.15: Plan Fidelity comparison with QM Large using TPC-H dataset

erated by each method and the QueryMorpher with the larger size of the BART model. As shown in the figure, the larger BART model increases the performance of the QueryMorpher on the target pattern rate. This improvement suggests that the increased model capacity allows QueryMorpher to capture more complex structural relationships, thereby guiding the optimizer to reproduce the intended target patterns more accurately.

Furthermore, the Figure 6.17 demonstrates the diversity of the raw plan and the final plan. By comparing the diversity gap between the raw and final plans, we can infer the degree of plan reproducibility for each approach. As shown in

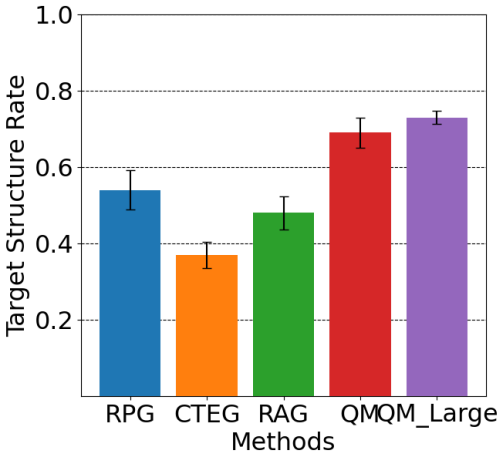

Figure 6.16: Final Plan Target Pattern Rate comparison with QM Large using TPC-H dataset

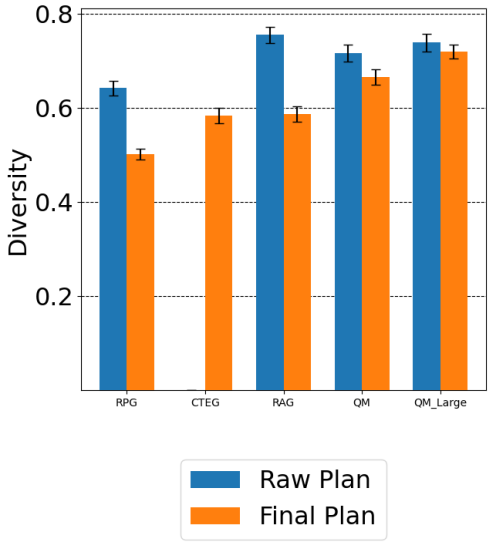

Figure 6.17: Raw Plan and Final Plan Diversity comparison with QM Large using TPC-H dataset

the figure, QueryMorpher exhibits a smaller diversity difference than the other methods, indicating that its generated raw plans are more likely to be reproduced by the optimizer.

We also did this experiment with the TPC-DS dataset. As shown in the Figure 6.18, the overall generated plan fidelity drops slightly as the database schema is more complex. Figure 6.19 indicates that when dealing with a dataset that has complex schema information, the rule-based node mutation method, RAG, has a very high performance drop. In comparison with the Figure 6.16, Figure 6.19 demon-

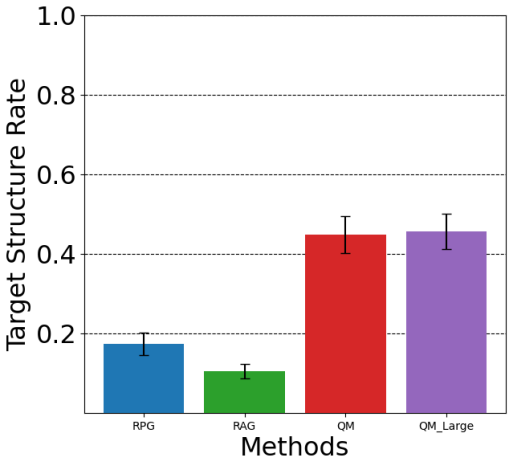

Figure 6.18: Plan Fidelity comparison with QM Large using TPC-DS dataset

strates that the raw plan reproducibility of each method has a performance drop, but the QueryMorpher still has the best raw plan reproducibility in comparison with the RPG and RAG. This result suggests that the learned mutation strategy of QueryMorpher generalizes more effectively to complex database schemas than the rule-based approaches.

The experiments evaluate QueryMorpher, a framework for generating SQL queries whose execution plans contain user-defined plan patterns. The experiments compare QueryMorpher against three baselines (RPG, RAG, and CTEG) across two benchmark datasets (TPC-H and TPC-DS) and examine three major evaluation

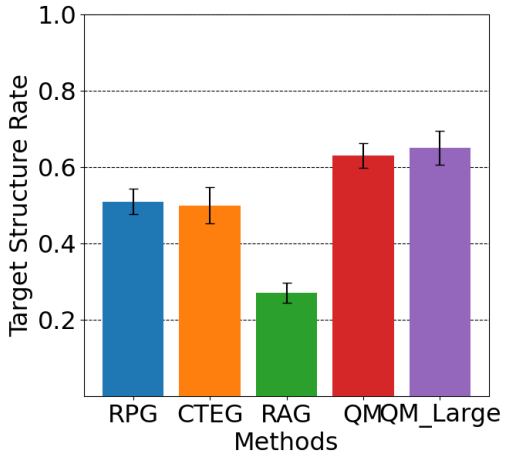

Figure 6.19: Final Plan Target Pattern Rate comparison with QM Large using TPC-DS dataset

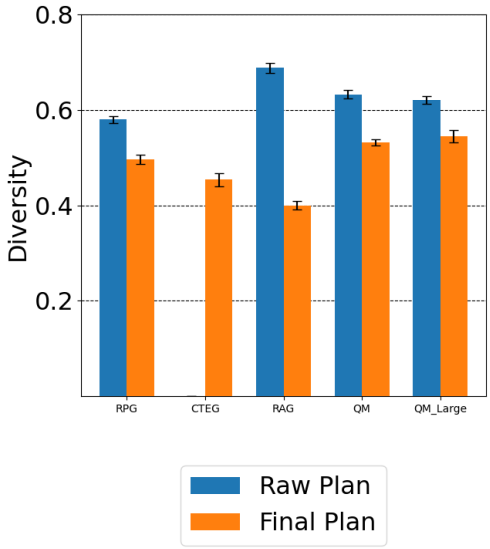

Figure 6.20: Raw Plan and Final Plan Diversity comparison with QM Large using TPC-DS dataset

metrics (TPR, diversity, and plan fidelity). Throughout the experiments, QueryMorpher demonstrates an outstanding and stable performance. Also, the experiments answer the three questions listed in the Chapter 1. In the experiments, we found that although QueryMorpher cannot generate queries whose execution plans contain the desired plan pattern at all times, QueryMorpher is the best choice among the four methods. Also, the experiments show that a learned plan-mutation policy outperforms rule-based baselines under a fixed budget.

7 Conclusion

In this thesis, we developed an effective and general framework, QueryMorpher, for the MPG problem.

In the workflow of the framework, we take the execution plan pattern and the sample plan that contains the plan pattern as the input. For each node in the plan traversal, we first convert the node to a sequence to help the Seq2Seq model understand the context of the node. Then we input the node sequence to the Seq2Seq model, and use the learned model to generate the mutation instruction sequence for a better mutated plan fidelity. Then, we translate the mutated plan to the SQL query for end-to-end query generation.

Our experimental findings, using two mainstream synthetic datasets, showed that QueryMorpher outperformed baseline methods, supporting stable and scalable performance for diverse query generation tasks, especially when the database schema information or the user-defined plan pattern is simple.

However, this proposed framework has limitations: First, QueryMorpher lacks

real-bug evaluation. The QueryMorpher focuses on the MPG problem, while the MPG problem is just a simulation of the bug in the database. Second, the experiment only focuses on the PostgreSQL database. As QueryMorpher is built on the PostgreSQL database, it is not general enough for other DBMSs. Third, the proposed framework heavily relies on the BART model. Thus, the input and the output sequence of the BART model are complex, making the model training difficult. Fourth, the QueryMorpher does not consider the cost and cardinality of the generated query.

In the future, we will study the problem involving the real DBMS bug evaluation, making the framework practically used in the industry. Also, we will study the problem on multiple DBMS patterns to make the QueryMorpher compatible with more DBMSs, and we will consider using learned plan embeddings, e.g., GNNs, instead of using node-to-sequence. Moreover, we will explore the query generation with cost and cardinality control. Furthermore, we would like to study how the QueryMorpher can be adapted to cases where new structures of the physical operators are introduced, and queries whose execution plan contains the operator structure are required to test the performance or ensure the validity of the introduced structure.

Bibliography

[1] Guillaume Bagan et al. "gMark: Schema-driven generation of graphs and queries". In: *IEEE Transactions on Knowledge and Data Engineering* 29.4 (2016), pp. 856–869.

[2] Melyssa Barata, Jorge Bernardino, and Pedro Furtado. "An overview of decision support benchmarks: TPC-DS, TPC-H and SSB". In: *New Contributions in Information Systems and Technologies: Volume 1* (2015), pp. 619–628.

[3] P. Erdos and A. Rényi. "On the evolution of random graphs". In: *Publ. Math. Inst. Hung. Acad. Sci* 5.1 (1960), pp. 17–60.

[4] Jonas Heitz and Kurt Stockinger. "Join query optimization with deep reinforcement learning algorithms". In: *arXiv preprint arXiv:1911.11689* (2019).

[5] E. Hoogeboom et al. "Autoregressive diffusion models". In: *International Conference on Learning Representations (ICLR)*. 2022.

[6] Lingkai Kong et al. "Autoregressive diffusion model for graph generation". In: *International Conference on Machine Learning (ICML)*. 2023.

[7] Sanjay Krishnan et al. “Learning to optimize join queries with deep reinforcement learning”. In: *arXiv preprint arXiv:1808.03196* (2018).

[8] Xinyue Liu et al. “TreeGAN: syntax-aware sequence generation with generative adversarial networks”. In: *2018 IEEE International Conference on Data Mining (ICDM)*. IEEE. 2018, pp. 1140–1145.

[9] Manuel Madeira et al. “Generative Modelling of Structurally Constrained Graphs”. In: *International Conference on Neural Information Processing Systems (NeurIPS)*. 2024.

[10] Ryan Marcus and Olga Papaemmanouil. “Deep reinforcement learning for join order enumeration”. In: *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*. 2018, pp. 1–4.

[11] Ryan Marcus and Olga Papaemmanouil. “Towards a hands-free query optimizer through deep learning”. In: *arXiv preprint arXiv:1809.10212* (2018).

[12] Ryan Marcus et al. “Neo: A learned query optimizer”. In: *arXiv preprint arXiv:1904.03711* (2019).

[13] Karolis Martinkus et al. “SPECTRE: Spectral conditioning helps to overcome the expressivity limits of one-shot graph generators”. In: *International Conference on Machine Learning (ICML)*. 2022.

[14] l. Maziarka et al. “Mol-CycleGAN: A generative model for molecular optimization”. In: *Journal of Cheminformatics* 12.1 (2020), pp. 1–18.

[15] Chenhao Niu et al. “Permutation invariant graph generation via score-based generative modeling”. In: *International Conference on Artificial Intelligence and Statistics (AISTATS)*. 2020.

[16] Meikel Poess and Chris Floyd. “New TPC benchmarks for decision support and web commerce”. In: *ACM Sigmod Record* 29.4 (2000), pp. 64–71.

[17] Manuel Rigger and Zhendong Su. “Testing database engines via pivoted query synthesis”. In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 2020, pp. 667–682.

[18] A. Seltenreich. *sqlsmith*. 2020. URL: `https://github.com/anse1/sqlsmith`.

[19] M. Simonovsky and N. Komodakis. “GraphVAE: Towards generation of small graphs using variational autoencoders”. In: *International Conference on Artificial Neural Networks*. Springer, 2018, pp. 412–422.

[20] Weihua Sun, Run-An Wang, and Zhaonian Zou. “Query Generation based on Generative Adversarial Networks”. In: *arXiv preprint arXiv:2303.14777* (2023).

[21] Clement Vignac and Pascal Frossard. “Top-N: Equivariant set and graph generation without exchangeability”. In: *International Conference on Learning Representations (ICLR)*. 2021.

[22] Clement Vignac et al. “DiGress: Discrete denoising diffusion for graph generation”. In: *International Conference on Machine Learning (ICML)*. 2022.

[23] Zongheng Yang et al. “Balsa: Learning a query optimizer without expert demonstrations”. In: *Proceedings of the 2022 International Conference on Management of Data*. 2022, pp. 931–944.

[24] Xiang Yu et al. “Reinforcement learning with tree-lstm for join order selection”. In: *2020 IEEE 36th international conference on data engineering (ICDE)*. IEEE. 2020, pp. 1297–1308.

[25] Ye Yuan et al. “sqlFuzz: directed fuzzing for SQL injection vulnerability”. In: *Electronics* 13.15 (2024), p. 2946.

[26] Kaizhong Zhang and Dennis Shasha. “Simple fast algorithms for the editing distance between trees and related problems”. In: *SIAM journal on computing* 18.6 (1989), pp. 1245–1262.

[27] Lixi Zhang et al. “Learnedsqlgen: Constraint-aware sql generation using reinforcement learning”. In: *Proceedings of the 2022 international conference on management of data*. 2022, pp. 945–958.