UNIVERSITY OF CALIFORNIA,
IRVINE

Generative Problem Solving for Relational Data
Construction, Verification, and Discovery of Problems

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Electrical Engineering and Computer Science

by

Goli Vaisi

Dissertation Committee:
Professor Phillip Chen-Yu Sheu, Chair
Professor Nader Bagherzadeh
Assistant Professor Sitao Huang

2026

DEDICATION

To my husband, Reza Sayfoori, for his love and patience throughout this journey.

To my parents, for a lifetime of giving that made this moment possible.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor, Professor Phillip Chen-Yu Sheu, for his guidance and encouragement throughout my doctoral work. His insight shaped the direction of this dissertation, and his support helped me grow as a researcher.

I am also grateful to my committee members, Professor Nader Bagherzadeh and Professor Sitao Huang, for their time, valuable feedback, and willingness to serve on my committee.

I am thankful to the faculty and staff of the Department of Electrical Engineering and Computer Science at the University of California, Irvine, for their support throughout my doctoral studies. I would also like to thank Chengheng Lyu for his collaboration on GPS-SD.

Finally, I am grateful to my family for their constant support and encouragement throughout this journey.

Chapter 2 is adapted from material published in G. Vaisi and P. C.-Y. Sheu, "GPS-Relational: Generative Problem Solving with Relational Databases," International Journal of Semantic Computing (2026), used with permission from World Scientific Publishing Company.

ABSTRACT OF THE DISSERTATION

Generative Problem Solving for Relational Data
Construction, Verification, and Discovery of Problems

By

Goli Vaisi

Doctor of Philosophy in Electrical Engineering and Computer Science

University of California, Irvine, 2026

Professor Phillip Chen-Yu Sheu, Chair

Artificial intelligence systems are traditionally designed to solve problems that have already been defined. However, a more general form of intelligent problem solving should also support the ability to generate new problems, represent them in computational form, validate their meaning, and connect them to appropriate solution methods. This dissertation develops Generative Problem Solving (GPS) as a framework for moving from predefined problem solving toward the systematic generation, translation, validation, and solution of computational problems.

The first part of this dissertation introduces GPS-Relational, a grammar-guided framework for problem solving over relational databases. The framework translates natural-language problem descriptions into executable SQL by decomposing complex requests into relational components, mapping them to database schemas, and validating their semantic compatibility before constructing the final query. A formal grammar and Semantic Validity Model provide an explicit representation of relational operations and constraints, enabling complex problems to be constructed from validated components rather than generated as unconstrained text. The same grammar also provides a foundation for systematically generating relational problem instances with controlled structure and complexity.

The second part extends this generative capability by using the structured representation in the reverse direction to support the generation and verification of complex relational problems and their natural-language descriptions. Complex SQL structures are decomposed into explicit dependencies and translated into natural language, while round-trip translation and execution-based comparison are used to determine whether the generated descriptions preserve the intended relational semantics. Controlled query generation and database population further enable the creation of informative benchmark problems with varying structural complexity. Together, these mechanisms support not only problem generation, but also systematic validation and diagnosis of errors in generated problems and translations.

The final part extends GPS beyond problems that can be represented and solved directly through conventional SQL. An object-relational representation is introduced to describe more complex computational structures, including graph entities, relationships, objectives, and constraints. This representation enables the generation of graph-based problems such as constrained shortest-path and influence-based optimization problems and provides a bridge from natural-language problem descriptions to the structured inputs required by specialized solvers. In this way, GPS is extended from relational query problems toward broader classes of computational problems whose solutions require algorithms beyond traditional database query processing.

Together, these contributions establish a progression from translating and solving relational problems, to systematically generating and validating problems, and finally to extending problem generation beyond SQL to more general computational problem classes. The dissertation demonstrates how explicit grammars and structured representations can provide a foundation for AI systems that do more than answer predefined questions: they can construct, analyze, validate, and ultimately expand the space of problems that can be computationally explored.

Chapter 1

Introduction

Artificial intelligence systems are traditionally designed to solve problems that have already been defined. A user provides a task in an expected form, and the system attempts to compute an answer. Although this approach has enabled significant progress, it limits intelligent systems to the passive consumption of existing problems. A more general form of intelligence should also support the generation, representation, validation, and discovery of problems. Such a capability can produce new test cases, expose weaknesses in existing solvers, support controlled evaluation, and identify computational problems that have not yet been studied in a systematic form.

Large Language Models (LLMs) have made it possible to express computational tasks through natural language. In database systems, for example, users can describe an information need in ordinary language and ask a model to translate it into SQL. However, the fluency of an LLM does not guarantee that its output preserves the intended meaning. A generated query may be syntactically valid and executable while still containing incorrect joins, incompatible comparisons, misplaced aggregations, or altered dependencies between nested operations. Similar difficulties arise in the reverse direction, where a fluent natural-

language explanation of a SQL query may omit conditions or misrepresent its relational structure. These limitations become more significant as problems grow in complexity and contain multiple dependent operations.

This dissertation develops a grammar-guided approach to Generative Problem Solving (GPS). GPS extends the traditional objective of solving predefined tasks by enabling a system to construct, translate, validate, and, when an appropriate solver is available, solve computational problems. The grammar provides an explicit representation of the structural components of a problem and the rules by which those components may be composed. Unlike unconstrained generation, this representation makes it possible to inspect the reasoning process, enforce domain-specific validity conditions, and systematically control problem complexity.

The first part of this dissertation introduces GPS-Relational for natural-language-to-SQL translation and relational problem generation. The framework decomposes complex natural-language intent into smaller relational components, maps those components to the available database schema, and validates their compatibility before constructing executable SQL. A Semantic Validity Model prevents operations that are syntactically acceptable but semantically meaningless, such as aggregating incompatible attributes or connecting entities through invalid relationship paths. The same grammar also supports the controlled generation of natural-language and SQL pairs with different relational structures and nesting depths. These generated problems can be executed to produce reference answers and can therefore be used to evaluate language-model-generated SQL without requiring the generated SQL to match a single reference string or structure.

The second part considers the reverse direction: translating SQL into natural-language explanations. A readable explanation is not necessarily a faithful explanation, particularly when a query contains nested subqueries, aggregation dependencies, joins, or set operations. The proposed framework evaluates explanation faithfulness through round-trip translation. A natural-language explanation is translated back into SQL, and the reconstructed query is

compared with the original query through execution. When the results differ, the framework decomposes both queries into structural blocks and identifies the source of the mismatch, including dropped conditions, aggregation-scope errors, and incorrect dependencies between nested components. In this way, the grammar supports not only translation, but also verification and structural diagnosis.

The third part extends the GPS framework beyond conventional relational queries. The database representation is expanded from a relational schema to an object-relational representation capable of describing graph entities, attributes, relationships, objectives, and constraints. This extension supports the generation of natural-language graph problems such as constrained shortest-path problems and more complex graph-based tasks involving influence or interactions between computed structures. The system represents each generated problem in a structured form and translates it into the requirements expected by a compatible graph solver. When an existing solver supports the generated problem class, the framework can connect generation with problem solving. When no solver is available, the generated specification can instead serve as a formally described problem instance for future algorithm development.

The central thesis of this dissertation is that a grammar can provide a unified foundation for generative problem solving across multiple directions and domains. In the relational setting, it supports forward translation, reverse translation, executable validation, benchmark generation, and failure diagnosis. In the graph setting, it supports the generation and formal specification of optimization and problem-solving tasks beyond SQL. The grammar is therefore not limited to producing sentences or queries; it defines a structured space of computational problems and provides mechanisms for navigating, validating, and extending that space.

The remainder of this dissertation is organized as follows. Chapter 2 presents GPS-Relational for natural-language-to-SQL translation, semantic validation, recursive decomposition, SQL

construction, and controlled benchmark generation. Chapter 3 presents SQL-to-natural-language translation, round-trip verification, structural diagnosis, and execution-aware evaluation. Chapter 4 extends the framework to object-relational representations and graph-based problem generation, including shortest-path and higher-order graph problems. Finally, Chapter 5 summarizes the contributions, discusses the limitations of the current framework, and presents directions for future research.

Chapter 2

GPS-Relational: Generative Problem Solving with Relational Databases

This chapter presents GPS-Relational, a framework for evaluating large language model (LLM)-generated SQL by constructing an executable reference SQL query from the same natural-language intent and database schema. GPS-Relational constructs the reference query through grammar-guided decomposition, schema mapping, semantic validation, and SQL assembly. The reference SQL is executed to produce the expected result, which is compared with the execution result of the LLM-generated SQL, allowing equivalent SQL forms to be accepted without requiring textual or structural matching. In addition, GPS-Relational supports large-scale benchmark generation by systematically synthesizing natural-language and SQL pairs with controlled nesting depth and relational complexity. These generated problems enable systematic evaluation of LLM performance across varying levels of structural complexity. Experimental results demonstrate that GPS-Relational constructs executable reference SQL across existing benchmark and complex generated queries, while LLM execution accuracy decreases significantly as nesting depth increases.

2.1 Introduction

Large language models (LLMs) have made natural language to SQL translation more accessible. Given a user question expressed in natural language (NL) and a database schema, current models can often generate SQL that is syntactically valid and, in many cases, executable. This progress has shifted the main challenge from generating SQL to evaluating whether the generated SQL correctly represents the user's intent. This evaluation becomes especially difficult when queries are complex, deeply nested, or produced at scale.

Evaluation methods for text-to-SQL commonly rely on execution-based evaluation, where the generated SQL is executed and its result is compared with the result of a reference SQL query. However, this approach depends on having reliable reference SQL queries and benchmark test cases with sufficient structural coverage. Standard text-to-SQL benchmarks such as Spider [1] have been important for comparing models, but they provide limited coverage of nested and structurally complex queries. When such reference cases are limited, recent work has explored using LLMs to compare SQL queries and decide whether they have the same meaning [2, 3]. However, the reliability of this evaluation depends on the model making the judgment. If that model makes an incorrect or inconsistent decision, the evaluation result may also be unreliable. As a result, there is still a need for an evaluation framework that can construct executable reference queries and reference answers for complex natural-language intents, especially when the query requires nested reasoning, aggregation, comparison, or multi-step relational structure.

To address this need, we present GPS-Relational, a relational-database extension of Generative Problem Solving (GPS). GPS is a general problem-solving paradigm that supports both solving computational problems and generating new problem instances for testing, learning, and refinement. GPS-Relational adapts this paradigm to relational databases and applies it to the evaluation of language-model-generated SQL through grammar-guided refer-

ence construction. Given a natural-language query and a database schema, GPS-Relational recursively decomposes the query intent into smaller relational components, maps those components to schema-level operations, validates the mapped structure, and constructs an executable reference SQL query. The reference SQL is then executed to obtain the expected result. An LLM is evaluated by giving it the same natural-language query and schema, executing its generated SQL, and comparing the result with the reference result.

GPS-Relational also supports programmatic benchmark generation. Since existing benchmarks provide limited coverage of deeply nested and structurally complex queries, the same grammar can be used to generate natural language and SQL pairs with controlled levels of nesting and relational complexity. These generated cases can provide a benchmark for stress testing of LLM behavior beyond standard examples and for measuring how the performance changes as query structure becomes more complex.

One might ask why a dedicated framework is necessary for benchmark generation when modern Large Language Models can easily produce large datasets. To investigate this, we first prompted ChatGPT to generate 1,000 natural-language queries for a given retail database schema. At first glance, it appeared able to produce a large number of query examples. However, upon closer inspection, the outputs relied largely on superficial variation: most examples followed simple query patterns and differed mainly by replacing entities, attributes, keywords, or values, such as city names or product categories.

In a second test, we made the prompt more specific by requesting 10,000 queries with more than 15 levels of nesting. Rather than generating 10,000 structurally distinct complex queries, ChatGPT returned a master template for deep nesting and described how variable substitution within that template could produce 10,000 examples. The template achieved nesting by repeatedly stacking similar EXISTS-style clauses over the same few tables. Although such outputs may be syntactically valid, this mechanical expansion does not provide meaningful relational complexity or the logical diversity needed for real-world benchmark

evaluation.

This motivates the design of GPS-Relational. The framework is built on a foundational set of relational patterns that cover core SQL operations, including projection, filtering, aggregation, grouping, joins, comparisons, set operations, and nested subqueries. These patterns are not fixed sentence templates. They are grammar-defined building blocks, and explicit composition rules determine how they can be combined and expanded according to the requested level of complexity. The system then fills the required slots using schema-grounded entities, attributes, operators, and values, while checking that each combination is structurally and semantically valid. As a result, GPS-Relational can generate a large and diverse set of meaningful NL-SQL pairs, rather than simply producing surface-level variations of the same query form. This provides controlled coverage of relational logic across different levels of query complexity.

More importantly, benchmark construction requires more than generating many examples. Even with advanced prompting, LLM-generated outputs still require validation for schema compliance, structural constraints, semantic validity, and execution correctness. GPS-Relational builds these checks into the generation process, providing a more reliable and reproducible foundation for constructing controlled NL-SQL benchmarks.

This paper makes two main contributions. First, we introduce GPS-Relational, an evaluation framework that constructs executable reference SQL from natural-language intent and uses the resulting reference answers to evaluate LLM-generated SQL. Second, we provide a benchmark generation pipeline that creates natural language and SQL pairs with controlled levels of nesting and relational complexity, enabling systematic evaluation beyond standard benchmark examples.

2.2 Related Work

Research on natural-language interfaces to databases has followed several directions, including rule-based semantic grammars, schema-aware query interpretation, synthesis-guided construction, and more recent LLM-based text-to-SQL systems. Early systems such as LADDER [4], LUNAR [5], and CHAT-80 [6] relied on hand-crafted semantic grammars and domain-specific rules. These systems provided interpretable mappings from natural language to database operations, but they required substantial manual engineering and were difficult to transfer across schemas. Later systems such as PRECISE [7], NaLIR [8], SODA [9], Templar [10], and DBTagger [11] improved portability by incorporating schema linking, dependency parsing, metadata, query logs, or learned keyword mapping. These approaches addressed important parts of the grounding problem, but they still generally attempt to resolve the user request into a complete query interpretation as a whole. This becomes difficult when the intended SQL contains nested conditions, scoped aggregations, comparative subqueries, or coordinated relational branches.

Another line of work uses sketch-based, synthesis-guided, or alternative specification methods. SQLizer [12] generates query sketches and completes them through type-directed synthesis and repair, while DBPal [13] uses templates and synthetic paraphrases to reduce manual engineering. SCYTHE [14] and SQLSynthesizer [15] synthesize SQL from input-output examples rather than directly from natural language. Ontology-based systems such as ATHENA [16] and ATHENA++ [17] use domain ontologies to connect natural-language expressions to relational content, including more complex analytical queries. These methods improve different aspects of SQL construction, but many still rely on fixed templates, sketches, examples, or ontology engineering.

Recent LLM-based text-to-SQL systems have achieved strong performance on standard benchmarks. Methods such as DIN-SQL [18] use decomposition and self-correction, while

MAC-SQL [19] uses a multi-agent framework for collaborative SQL generation. Other approaches, such as PICARD [20] and TeCoD [21], constrain generation to improve syntactic or structural validity. These systems show the strength of LLMs for translating natural language into SQL, but their outputs may still be unreliable for deeply nested, schema-sensitive, or relationally constrained queries. In contrast, GPS-Relational is not proposed as another probabilistic SQL generator. Its role in this paper is to construct an executable reference SQL query from the same natural-language intent and schema, so that LLM-generated SQL can be evaluated by execution-result comparison.

Evaluation for text-to-SQL has also evolved. Exact string matching is too restrictive because different SQL queries can be semantically equivalent. Execution accuracy, used in benchmarks such as Spider [1], is more appropriate because it compares query results rather than SQL text. However, execution-based evaluation depends on having reliable reference SQL and sufficient benchmark coverage. Existing benchmarks have been essential for progress, but they provide limited control over nesting depth and relational complexity, especially for stress-testing model behavior on systematically deeper query structures. Recent model-based evaluation approaches, such as LLM-SQL-Solver [2] and FLEX [3], investigate whether LLMs or improved execution-based metrics can help assess SQL equivalence and reduce evaluation errors. These approaches are useful, but they still introduce dependence on learned or model-based judgment.

GPS-Relational provides a complementary evaluation strategy. Instead of asking another model to judge the LLM output, it constructs a reference SQL query through grammar-guided decomposition, schema mapping, semantic validation, and SQL assembly. The reference SQL is executed to produce the expected result, and the LLM output is evaluated by comparing execution results. In addition, because the same grammar can generate natural-language and SQL pairs with controlled nesting depth and relational complexity, GPS-Relational supports benchmark generation for cases that are difficult to cover system-

atically with static datasets alone.

2.3 System Overview

GPS-Relational is a grammar-guided framework for NL2SQL reference construction and evaluation. Given a natural-language problem and a database schema, the framework dynamically constructs an executable reference SQL query from the intended meaning of the input. This reference query is then executed on the database to produce the expected result used to evaluate SQL generated by a language model or another NL2SQL system.

We define the reference construction process as

$$f : (p, D) \rightarrow Q_{\text{ref}},$$

where p represents the natural-language problem, D denotes the database schema, and Q_{ref} is the reference SQL query constructed by GPS-Relational. The reference query is not used to require a language model to produce the same SQL text or the same SQL structure. Instead, it is used to produce the expected execution result:

$$R_{\text{ref}} = \text{Exec}(Q_{\text{ref}}, D).$$

For the same input p and schema D, a language model produces its own SQL query Q_{LLM}, whose execution result is

$$R_{\text{LLM}} = \text{Exec}(Q_{\text{LLM}}, D).$$

The generated SQL is evaluated by comparing R_{LLM} with R_{ref}. Thus, GPS-Relational provides the reference answer for evaluation while allowing the evaluated model to generate any SQL form that produces the same result.

The construction of Q_{ref} follows the NL2SQL pipeline used throughout the framework. The system architecture, illustrated in fig. 3.2, consists of three primary stages:

1. **Decomposition:** The query intent is extracted from p and recursively broken into smaller relational components, such as projections, filters, aggregations, comparisons, set operations, and nested conditions. The decomposition rules introduced in section 2.4 define how complex natural-language intent is represented before SQL construction.

2. **Schema Mapping:** The decomposed components are mapped to schema-specific tables, columns, and relationship paths in D. This stage grounds the natural-language intent in the available database structure and determines which schema elements are needed to construct the reference query.

3. **Semantic Validation:** The mapped components are checked using the Semantic Validity Model. This stage evaluates whether the selected schema elements are compatible with the required operations, semantic types, domains, and relationship paths. Components that pass this validation are used in the construction of Q_{ref}.

After decomposition, mapping, and semantic validation, the framework assembles the validated components into executable SQL. For complex queries, GPS-Relational organizes the reference query using Common Table Expressions (CTEs). The CTE structure is used to construct complex reference SQL in a modular way, preserving the dependency structure identified during decomposition.

The same reference construction process also supports benchmark generation. By controlling the grammar rules, nesting depth, and relational composition patterns, GPS-Relational can create natural-language and SQL pairs with specified levels of structural complexity. These generated pairs provide controlled test cases for evaluating how language models behave as query structure becomes more complex.

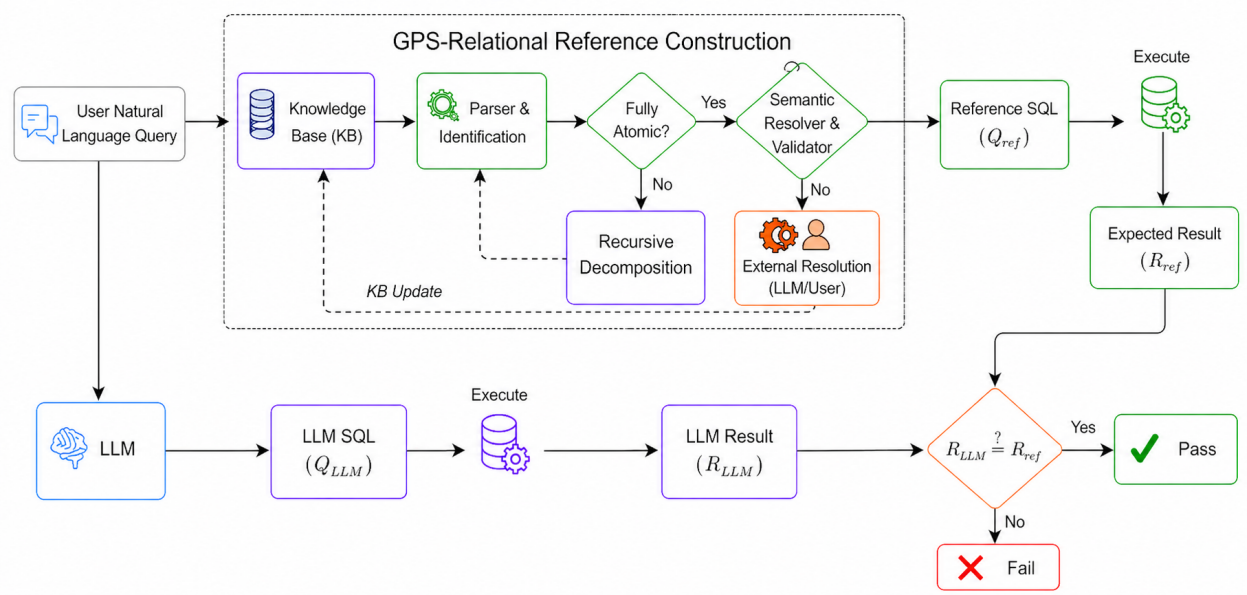


Figure 2.1: GPS-Relational evaluation framework. The upper pipeline constructs the reference SQL query Q_{ref} from the natural-language query and database schema, executes it to obtain the reference result R_{ref}, and compares it with the execution result R_{LLM} of the SQL generated by the LLM.

2.3.1 Semantic Validity Model

A fundamental challenge in query generation is distinguishing between queries that are executable and those that are logically meaningful. We define a query as semantically void if it satisfies the grammatical rules of SQL but violates the semantic constraints of the relational model.

To prevent the generation of such queries, we define a Semantic Validity Model that checks whether each mapped operation is compatible with the schema's semantic types, domains, and relationship paths. This model supports the construction of reference SQL queries whose operations remain grounded in the database structure.

Semantic Types

The model begins by partitioning the database schema into three disjoint semantic types: Measures (M), Identifiers (I), and Nominals (N). These categories map physical columns to the functional roles attributes play within a query and determine which SQL operations are semantically valid.

- **Measures (M):** Quantitative attributes representing magnitudes (e.g., `Balance`, `Loan_Amount`). These are the only attributes for which arithmetic comparisons and mathematical aggregations, such as `SUM` or `AVG`, produce statistically consistent results.
- **Identifiers (I):** Key-like attributes that denote entity identity (e.g., `Customer_ID`, `Branch_ID`). They support identity-based operations such as equality checks and relational joins. They are not valid targets for quantitative aggregations such as `SUM` or `AVG`, though order-based aggregations such as `MIN` and `MAX` may be meaningful when the identifier has an interpretable ordering.
- **Nominals (N):** Categorical attributes used for classification (e.g., `City`, `Transaction_Status`). They function as grouping keys or string-matching filters but are excluded by the model when used as inputs for arithmetic operations.

Semantic Domains

While semantic types establish broad relational boundaries, they alone cannot ensure the logical compatibility of attributes that share the same classification. For example, a schema may contain two distinct columns categorized as measures, such as `Annual_Salary` and `Account_Balance`. Although both support mathematical operations, comparing them is invalid because they refer to incompatible underlying units.

To address this, the framework introduces semantic domains, where we let Domain(c) represent the conceptual unit or dimension associated with attribute c. Under this model, an operation is considered valid only when the participating attributes share matching semantic domains. Formally, this requires that Domain(c_1) = Domain(c_2). This prevents semantic collisions, such as an attempted join between `Total_Balance` and `Product_Quantity`, even if both are stored as numeric types.

2.3.2 Semantic Validity Constraints

The foundational definitions of types and domains are operationalized through a set of logical constraints that determine whether a query's intent is meaningful. These constraints make semantic violations detectable during mapping and reference SQL construction. We categorize these rules into three levels: Local Constraints, Relational Constraints, and Global Constraints.

To bridge the gap between human intent and relational logic, we formalize these violations as semantic errors. Each rule defined in the following subsections corresponds to a specific error condition used during validation. If a violation is detected, the mapped component is rejected and is not used in the reference SQL.

Local Constraints

Local constraints govern the structural validity of individual expressions within a single clause. To operationalize these rules, the framework utilizes a Local Semantic Incompatibility Matrix that maps SQL operator categories to semantic types. This matrix serves as the first validation filter, identifying operations that are semantically invalid despite being syntactically valid in standard SQL. As shown in table 2.1, grouping operations are restricted to

identifiers and nominals, since grouping over continuous measures does not yield meaningful semantic partitions.

Table 2.1: Local Semantic Incompatibility Matrix.

SQL Operation Category	**Measure (M)**	**Identifier (I)**	**Nominal (N)**
Quantitative Aggregation (`SUM`, `AVG`)	Valid	Invalid	Invalid
Extreme Value Aggregation (`MIN`, `MAX`)	Valid	Valid	Invalid
Cardinality Aggregation (`COUNT`)	Valid	Valid	Valid
Grouping (`GROUP BY`)	Invalid	Valid	Valid

To formally define the validity of an operation over a column c, let $\tau(c) \in \{M, I, N\}$ denote the semantic type of c. We then define a validity function $\mathcal{V}(op, c)$. Let $O_{\text{quant}} = \{\texttt{SUM}, \texttt{AVG}\}$ represent the Quantitative Aggregation category and $O_{\text{range}} = \{\texttt{MIN}, \texttt{MAX}\}$ represent the Extreme Value Aggregation category. The formal validity rules are defined as:

$$\mathcal{V}(op, c) \iff \begin{cases} \tau(c) = M, & \text{if } op \in O_{\text{quant}} \\ \tau(c) \in \{M, I\}, & \text{if } op \in O_{\text{range}} \\ \text{True}, & \text{if } op = \texttt{COUNT} \end{cases}$$

These constraints are checked during the mapping and validation stages:

For grouping operations, the system restricts permissible semantic types to those representing discrete categories or unique entities (N, I), explicitly excluding continuous measures (M) to avoid semantically invalid results. This is formalized as:

$$\mathcal{V}(\texttt{GROUP BY}, c) \iff \tau(c) \in \{N, I\}$$

Finally, while equality $(=)$ is used in local filters (e.g., `City = 'Chicago'`), it is treated as an identity check within a domain rather than a relational comparison of magnitude. For

such point-filters, the rule is Value Admissibility, asserting that a constant value v must be a valid member of the set defined by the column c's semantic domain:

$$\mathcal{V}(=, c, v) \iff v \in \text{Domain}(c)$$

Relational Constraints

Relational constraints govern interactions between two distinct entities, such as column-to-column comparisons in joins or column-to-value comparisons involving magnitude. Unlike local constraints that focus on internal structure, relational constraints ensure that the relationship between entities is grounded in a shared conceptual space.

The first requirement for any relational interaction is domain equivalence. This ensures that two attributes are conceptually compatible, regardless of their underlying data types. For any binary operator *op*, validity is defined as:

$$\mathcal{V}(op, c_1, c_2) \iff \text{Domain}(c_1) = \text{Domain}(c_2)$$

This prevents semantic collisions, such as joining a `Department_ID` with an `Employee_Age`, even if both are represented as integers.

The second requirement applies specifically to Inequality Operators ($>, <, \geq, \leq$). As established, while equality ($=$) only requires identity within a domain, inequality requires a directional relationship based on magnitude. This imposes a stricter constraint involving both the semantic domain and the semantic type:

$$\mathcal{V}(op_{\text{ineq}}, x, y) \iff \big(\tau(x) = M \wedge \tau(y) = M\big) \wedge \big(\text{Domain}(x) = \text{Domain}(y)\big)$$

By enforcing this dual constraint, the framework ensures that inequality is applied only to measures (M) that share the same unit of analysis. This prevents logically flawed queries, such as comparing a geographical nominal to a numerical constant or comparing two measures from unrelated domains (e.g., `Revenue` > `Temperature`).

Global Constraints

Global constraints evaluate consistency across multiple clauses or query components. These rules identify cases in which SQL may execute successfully while still violating the intended relational structure.

Multi-Query Set Operations. For operations such as `UNION`, `INTERSECT`, and `EXCEPT`, the framework enforces domain consistency across the projected result sets. This ensures that columns merged from different queries share the same semantic purpose. Formally, for a set operator op_{set} and two queries Q_1, Q_2, validity is defined as:

$$\mathcal{V}(op_{\text{set}}, Q_1, Q_2) \iff \text{Domain}(\text{Proj}(Q_1)) = \text{Domain}(\text{Proj}(Q_2))$$

Degenerate Aggregation. This constraint identifies logical contradictions between a filtering predicate and an aggregation target. Let S represent the set of records satisfying the `WHERE` clause conditions. A degenerate aggregation error occurs if a statistical operator $op \in \{\texttt{AVG}, \texttt{SUM}\}$ is applied to a set with cardinality one. This typically occurs when a filter is applied to a unique identifier (I):

$$\mathcal{V}(op, S) = \text{False} \iff |S| = 1 \land op \in O_{\text{quant}}$$

Example: `SELECT AVG(Salary) FROM Employees WHERE Emp_ID = 101`. Since `Emp_ID` is a

unique identifier, the result set S contains only one record, making the "average" mathematically redundant and semantically void.

Projection Cardinality. This rule ensures that the grain of the query is logically compatible with the attributes in the `SELECT` clause. For a query to be valid, every projected attribute a must either be part of an aggregate function or be defined in the grouping set G. This prevents projection ambiguity, where the output cardinality is undefined:

$$\mathcal{V}(\text{Projection}) = \text{True} \iff \forall a \in \text{SelectClause} : (a \in \text{Aggregated}) \vee (a \in G)$$

Example: `SELECT City, SUM(Sales) FROM Transactions`. Without a `GROUP BY City` clause, the query is ambiguous because `City` has a higher cardinality than the single-row result produced by the `SUM` function.

2.4 Recursive Decomposition and Structural Mapping

The GPS-Relational framework constructs executable reference SQL from natural language (NL) intent by mapping linguistic structures to a set of grammar-defined patterns. This process is governed by a grammar expressed in Extended Backus–Naur Form (EBNF), which defines the correspondence between linguistic constructs and relational operations.

Rather than mapping queries to a fixed catalog of templates, the system dynamically constructs atomic units by combining a Base Pattern (B) with a compatible Filter Pattern (F). Formal definitions for these patterns are provided in fig. 2.2. An atomic query consists of a base pattern (B), optionally paired with a compatible filter pattern (F). When a filter is

present, the query is represented as the pair (B, F). The compatibility of (B, F) is determined by the Semantic Validity Model (section 2.3.2), which enforces the semantic type and relational constraints required for valid SQL operations.

Some linguistically valid query intents cannot be instantiated as a single atomic pattern. In such cases, the composition is not rejected; instead, it triggers recursive decomposition, which separates the operation into nested query components that can each be instantiated as valid atomic patterns.

2.4.1 Atomic Pattern Grammar

The production rules $\mathcal{R}$, presented in fig. 2.2, define the structural primitives used to instantiate atomic query components. Each rule specifies how a linguistic pattern is mapped to a canonical SQL fragment.

By combining a Base Pattern (B) with a compatible Filter Pattern (F), the resolver generates semantically grounded query components. In fig. 2.2, expressions of the form

$$\{\cdot\} \Rightarrow \text{SQL}$$

denote a translation from an abstract pattern specification to its corresponding SQL realization.

For generality, the placeholder $\{value\}$ represents either a literal constant (e.g., a string or number) or a scalar value produced by a nested query.

These patterns define the logical output of a query:

- B_1 **(Projection):** Produces a set of entities or attributes.

- B_2 **(Scalar Aggregation):** Computes a single quantitative value.
- B_3 **(Grouped Aggregation):** Computes aggregated values over entity groups.

These patterns define the scope of query constraints:

- F_1 **(Categorical Filter):** Applies conditions to individual records via the `WHERE` clause, triggering Value Admissibility (section 2.3.2) to verify operand compatibility.
- F_2 **(Numeric Filter):** Applies conditions to individual records via the `WHERE` clause, triggering Value Admissibility (section 2.3.2) to verify range compatibility.
- F_3 **(Aggregate Filter):** Applies conditions to aggregated results via the `HAVING` clause, triggering Global Constraints (section 2.3.2). When combined with a scalar base (B_2), the filter cannot be directly instantiated as a single atomic pattern and therefore requires recursive decomposition.

An atomic unit is accepted when the composition (B, F) satisfies the semantic admissibility conditions defined in section 2.3.2. Invalid compositions indicate the need for recursive decomposition strategies, as described in section 2.4.2. For semantic consistency during generation, we distinguish between `target_list` (nominal/ID attributes for retrieval) and `target` (numerical attributes for calculation). Furthermore, `group_list` defines the categorical boundaries for aggregated results, while the filter attribute (`cat_attr` or `num_attr`) specifies the row-level constraints. Finally, `agg` denotes the aggregation function, and `op` denotes the comparison operator used in filter predicates.

2.4.2 Vertical Decomposition

Vertical decomposition provides the recursive structure required to construct complex reference queries from natural-language intents that exceed the capacity of a single atomic pat-

tern. By breaking a query into vertical layers, the system checks that each sub-component satisfies specific semantic conditions before integration. We formalize vertical decomposition as a generalized structural mapping function:

$$f_C : (Q_{out}, Q_{in}) \Rightarrow Q_{out}(Q_{in})$$

Here, the inner query Q_{in} computes an intermediate structure, such as a relational set, an entity list, or a scalar value, that satisfies the operand requirements of the outer query Q_{out}. The proofs and algorithms for the rules in this section are presented in the author's PhD dissertation.

The system dynamically applies three primary vertical rules based on the linguistic context:

Vector Distribution (C_1)

This rule prevents analytical degeneracy when an outer aggregation targets an inner aggregation. Rather than allowing the inner intent to collapse into a single global scalar, C_1 maps the inner query into a grouped distribution (B_3) to preserve the population structure. As illustrated in example 2.1, C_1 isolates the inner average computation so that the outer maximum operation is applied across a mapped distribution of values rather than a single invalid point.

Example 1.

Find the maximum of average balance of customers who have an account in branches in Chicago.

Decomposition.

$Q_{\text{out}}(B_2)$: maximum of Q_{inner}

$Q_{\text{in}}(B_3 + F_1)$: average balance per customer who has an account in branches in Chicago

Instantiated SQL.

```
SELECT MAX(sub.v) FROM (
  SELECT a.customer_id, AVG(a.balance) AS v
  FROM Account a JOIN Branch b ON a.branch_id = b.branch_id
  WHERE b.city = 'Chicago'
  GROUP BY a.customer_id
) AS sub;
```

Set Membership (C_2)

This rule handles set-membership nesting, where the outer query depends on a set produced by an inner query. For this rule, the inner query must output a single-column list of entity keys, which corresponds to B_1 or B_3. This rule also resolves aggregate filtering conflicts. When the decomposition stage identifies an inner scalar base (B_2) constrained by an aggregate filter (F_3), the inner unit is converted into a grouped query rather than a scalar query. This inner unit then serves as the population used by the outer layer. As seen in example 2.2, C_2 evaluates the aggregate filter in the inner query to generate a valid subset of entities, allowing the outer query to compute its aggregation strictly for that isolated population.

Example 2.

Find the average balance of customers whose maximum balance is more than 1000.

Decomposition.

$Q_{\text{out}}(B_2)$	average balance of customers in Q_{in}
$Q_{\text{in}}(B_1 + F_3)$	customers whose maximum balance is greater than 1000

Instantiated SQL.

```
SELECT AVG(a.balance) FROM account a
WHERE a.customer_id IN (
  SELECT customer_id
  FROM account
  GROUP BY customer_id
  HAVING MAX(balance) > 1000
);
```

Comparative Subquery (C_3)

This rule handles dynamic operands where a comparison operator targets a computed expression rather than a literal constant. It isolates the computation into an inner scalar query (B_2), producing a single value to satisfy the outer operator's requirement. As shown in example 2.3, the comparative operand is dynamic. C_3 processes the targeted aggregate clause as the inner subquery, returning a computed scalar value that is subsequently used by the outer query's comparison operator.

Example 3.

Find customers whose balance is more than the average balance of customers who have an account in branches in Chicago.

Decomposition.

$Q_{\text{out}}(B_1 + F_2)$	customers whose balance is greater than Q_{in}
$Q_{\text{in}}(B_2 + F_1)$	average balance of customers who have an account in branches in Chicago

Instantiated SQL.

```
SELECT customer_id FROM account
WHERE balance > (
  SELECT AVG(balance)
  FROM account a JOIN branch b ON a.branch_id = b.branch_id
  WHERE b.city = 'Chicago'
);
```

2.4.3 Horizontal Decomposition

Horizontal decomposition addresses queries containing parallel logical units or distinct entity sets that require set-theoretic combination. This strategy handles lateral complexity, where a coordinating conjunction links two conditions over the same entity domain. To resolve this structure, the system identifies the Nominal Anchor N, which serves as the shared subject for both conditions, and decomposes the intent into two parallel branches:

$$Q_A = f(N, Cond_A), \qquad Q_B = f(N, Cond_B)$$

where $Cond_A$ and $Cond_B$ denote the specific linguistic constraints associated with each branch. The system then composes these branches using a generalized set-mapping function:

$$f_C : (Q_A, Q_B) \Rightarrow \text{SetOperation}(Q_A, Q_B)$$

By establishing this shared anchor, the system requires that both branches project the same key attribute, thereby satisfying the Projection Cardinality requirement.

example 2.4 illustrates how these base branches are extracted before any set operation is applied.

Example 4.

Find customers who have an account in Chicago [CONNECTOR] a loan in Chicago.

Decomposition.

N : all customers

Q_A : customers with an account in Chicago

Q_B : customers with a loan in Chicago

SQL Branches.

```
Q_A: SELECT customer_id FROM account a JOIN branch b ON a.branch_id = b.branch_id WHERE
b.city = 'Chicago'


Q_B: SELECT customer_id FROM loan l JOIN branch b ON l.branch_id = b.branch_id WHERE
b.city = 'Chicago'
```

Once the independent branches Q_A and Q_B are successfully instantiated, the system maps the specific linguistic connector to a generalized set-mapping function to compose the final query. The framework then applies the appropriate composition rule based on the linguistic context.

Set-Based Coordination (C_4, C_5, C_6)

These rules cover standard conjunctions such as *"and," "or,"* and *"but not."* Once the two branches are independently resolved, the system combines them using the relational operator that matches the intended connector. In this way, the decomposition remains the same across the three cases, while the final composition changes according to the logical relation expressed in the natural language query.

Symmetric Difference (C_7)

This rule handles exclusive conditions, typically triggered by expressions such as *"or . . . but not both."* In this case, the system still begins from the same independently computed branches Q_A and Q_B, but instead of directly intersecting, uniting, or subtracting them, it returns only those entities that belong to exactly one branch. This composition is represented as:

$$(Q_A \setminus Q_B) \ \cup \ (Q_B \setminus Q_A)$$

This construction ensures that entities satisfying both conditions are explicitly excluded from the final result.

Double Exclusion (C_8)

This rule handles joint negations, typically expressed by phrases such as *"neither . . . nor."* Here, the two decomposed branches again represent the entities satisfying each individual condition, but the final result is computed relative to the full anchor population N. The system first forms the union of the two exclusion branches and then subtracts that combined set from the shared baseline population:

$$N \setminus (Q_A \cup Q_B)$$

This allows the framework to return only those entities that satisfy neither condition.

table 2.2 summarizes how each horizontal rule combines the pre-computed branches into valid SQL structures.

Table 2.2: Summary of Horizontal Decomposition Composition Rules.

Rule	Linguistic Pattern	SQL Composition
C_4, C_5, C_6	*and*, *or*, *but not*	`Q_A [INTERSECT / UNION / EXCEPT] Q_B`
C_7	*or... but not both*	`(Q_A EXCEPT Q_B) UNION (Q_B EXCEPT Q_A)`
C_8	*neither... nor*	`N EXCEPT (Q_A UNION Q_B)`

2.5 Recursive Query Resolution and Assembly

Recursive query processing converts the hierarchical interpretation produced in section 2.4 into executable SQL through three steps: organizing the query into dependent levels, resolving these levels bottom-up into schema-grounded units, and assembling them using Common Table Expressions (CTEs).

2.5.1 Recursive Level Resolution

This stage organizes a complex query into dependent relational levels. example 2.5 provides the running example. fig. 2.3 shows how the query is split into a hierarchy: the construction *both ... and* creates a horizontal branch at Level 5 for accounts and loans in Chicago, while the remaining levels capture the vertical nesting of aggregations and comparisons.

Example 2.5.

Find the maximum of the average balance of customers whose minimum balance is more than the maximum balance of customers who have both an account and a loan in branches in Chicago.

The hierarchy is resolved bottom-up, from leaves to root, converting each level into a slot-filled relational unit. At each level, the main intent determines the base relational form, such as projection or aggregation, while the attached constraint defines the filter. If a value relies on a nested child, the parent stores a reference to that child rather than a literal. The resulting slot-filled hierarchy is shown in fig. 2.4.

2.5.2 Schema Mapping and Semantic Validation

The slot-filled units are grounded to physical database tables, columns, and join paths using lexical similarity [22] and relational context. When a term acts as a filter, the framework identifies candidate columns and searches through the schema graph G to connect the column back to the main entity of the level. In the running example, resolving `Chicago` at Level 5 identifies `branch.city` and the join path linking `Branch` to the `Account` or `Loan` branch.

After candidate mappings are found, the Semantic Validity Model from section 2.3.2 is applied to check whether the mapped components satisfy the required type, domain, and relationship constraints. Mappings that violate these constraints are rejected, while mappings that remain structurally valid but under-specified are treated as ambiguities and preserved for clarification.

Typical semantic errors include type mismatches, invalid relationships, attribute mismatches, invalid concepts, and domain inconsistencies in set operations. Typical ambiguities include schema-linking ambiguity, column or attribute ambiguity, and granularity ambiguity. Together, these checks ensure that the slot-filled levels are grounded to reachable and relationally coherent schema elements before SQL assembly.

2.5.3 Modular SQL Assembly with CTEs

Grounded levels are assembled into SQL from the innermost level outward using Common Table Expressions (CTEs). Each resolved level becomes a separate CTE definition, preserving the dependency structure established during decomposition. The final `SELECT` statement composes these CTEs according to the original recursive hierarchy.

For example 2.5, the final SQL in fig. 2.5 uses an `INTERSECT` for the coordinated Level 5 branches, followed by CTEs that resolve the nested maximum, minimum, average, and final

maximum operations. This modular structure provides a direct implementation path from recursive levels and supports localized inspection of intermediate query logic.

2.6 Benchmark Generation Pipeline

The same grammar used to construct reference SQL can be used to generate controlled benchmark cases. The pipeline creates a structural skeleton, binds its slots to schema elements, and synthesizes both the natural-language query and its reference SQL. Because nesting depth and relational composition are controlled during generation, and semantic constraints are enforced throughout the process, the resulting pairs can be used to evaluate model behavior beyond standard benchmark examples.

The generation process is governed by the grammar and dependency rules introduced earlier in the paper. The base and filter productions are defined in fig. 2.2, while the vertical nesting compatibility rules are summarized in table 2.3. Together, these rules define which atomic patterns can be combined and which nested extensions are valid during generation.

Table 2.3: Vertical Nesting Rules and Compatibility.

Rule	Operation	Valid Inner Base	Valid Outer Filter
C_1	Append aggregation	B_2	–
C_2	Set-membership nesting	B_1/B_3	$F_1/F2$
C_3	Replace {value} with subquery	B_2	F_2/F_3

Using these productions and compatibility rules, the generator constructs each benchmark query in three stages. First, it builds a structural skeleton by recursively selecting compatible base, filter, and compositional patterns. Second, it binds the abstract slots in the skeleton to schema elements that satisfy type, domain, and relationship constraints. Third, it resolves operators and values and synthesizes the natural-language query. The generated query is then passed through the GPS-Relational reference construction pipeline described in Secs. 2.3–2.5

to produce the corresponding reference SQL Q_{ref}.

Algorithm 1 formalizes the recursive natural-language generation process. The algorithm takes as input a target nesting depth L, a schema S, a grammar G, and a dependency matrix M. It first initializes an empty skeleton (Line 2). At the innermost level, $\ell = 0$, it selects a base pattern B, chooses a compatible filter F, and stores this atomic unit as the core of the skeleton (Lines 4–8). For each outer level, the algorithm retrieves the previous layer, selects a compatible compositional rule C, then chooses a valid filter and base pattern for the new layer according to the dependency matrix (Lines 10–17). This bottom-up loop continues until the target depth L is reached, producing a structural skeleton with controlled nesting complexity.

After the skeleton is generated, `BindSchema` grounds each layer to the database schema by selecting a filter attribute, identifying reachable columns, and choosing compatible target attributes (Lines 22–28). Finally, `GenerateNL` resolves operators and values and synthesizes the natural-language query from the grounded skeleton (Lines 29–33). The resulting Q_{NL} is then translated by GPS-Relational into Q_{ref}, which is executed to obtain the expected result used during evaluation.

Phase 1: Recursive Structural Generation

In the first phase, the system constructs a valid structural skeleton by recursively selecting compatible base, filter, and compositional patterns from the generation grammar. The process begins at the innermost level and expands outward, ensuring at each stage that the newly added layer satisfies the compatibility constraints defined by the grammar in Fig. 2.2 and the dependency matrix in Table 2.3.

To illustrate, consider a target complexity level $L = 2$:

```
Input  : Target nesting depth L, schema S, grammar G, dependency matrix M
Output: Structural skeleton Skeleton and natural-language query Q_NL
Function GenerateSkeleton(L, S, G, M)
    Skeleton ← [];
    for ℓ ← 0 to L do
        if ℓ = 0 then
            B ← Random({B_1, B_2, B_3});
            ValidFilters ← LookupFilters(B);
            F ← Random(ValidFilters);
            Skeleton.Append((ℓ, ∅, B, F));
        else
            (ℓ − 1, C_in, B_in, F_in) ← Skeleton[ℓ − 1];
            ValidRules ← LookupRules(B_in, M);
            C ← Random(ValidRules);
            ValidFilters ← LookupFilters(C);
            F ← Random(ValidFilters);
            ValidBases ← LookupBases(F);
            B ← Random(ValidBases);
            Skeleton.Append((ℓ, C, B, F));
        end
    end
    return Skeleton;
end
Function BindSchema(Skeleton, S)
    foreach layer in Skeleton do
        a ← SelectFilterAttribute(layer, S);
        R ← ReachableColumns(a, S);
        target ← SelectTarget(R);
    end
end
Function GenerateNL(Skeleton, S, G)
    Params ← ResolveOperatorsAndValues(Skeleton, S);
    Q_NL ← SynthesizeFromTemplates(Skeleton, Params, G);
    return Q_NL;
end
```

Algorithm 1: Recursive Generation of Natural-Language Benchmark Queries

- **Innermost Core:** The generator first selects an atomic unit $(\ell = 0, \emptyset, B_2, F_1)$, which serves as the anchor.
- **Outer Layer 1:** It wraps this core with a structurally compatible layer, yielding $(\ell = 1, C_3, B_1, F_2)$.
- **Outer Layer 2:** A final extension produces $(\ell = 2, C_2, B_1, F_1)$.

The resulting hierarchy is therefore:

$$(\ell = 2, C_2, B_1, F_1), \quad (\ell = 1, C_3, B_1, F_2), \quad (\ell = 0, \emptyset, B_2, F_1).$$

Phase 2: Semantic Schema Binding

Once the structural hierarchy has been generated, the system binds each abstract slot to schema elements that satisfy both semantic type constraints and relational reachability. For nested structures, the framework also checks that inner outputs remain compatible with the outer slots that consume them. Applying this process to the running example yields:

- **Outermost level** $(B_1 + F_1)$**:** The target is grounded to `customer.name`, and the categorical filter is grounded to `branch.city`, with the filter value supplied by the next inner level through rule C_2.
- **Middle level** $(B_1 + F_2)$**:** The numeric attribute is grounded to `account.balance`, with its comparison value supplied by the innermost level through rule C_3.
- **Innermost level** $(B_2 + F_1)$**:** The aggregation target is grounded to `account.balance`, and the categorical filter attribute is grounded to `branch.city`.

Phase 3: Parameter Resolution and NL Synthesis

In the final phase, the remaining functional parameters are resolved and the grounded structure is synthesized into natural language. Atomic value slots are filled by sampling literals from the corresponding schema domains, while nested value slots are replaced by the synthesized output of lower levels.

For the running example:

- The engine resolves the set-membership relation at the outer level, the scalar comparison operator at the middle level, and the aggregation at the innermost level.

- The literal value for the deepest categorical filter is sampled as *Chicago*, consistent with the domain of `branch.city`.

After parameter resolution, the final generated query is not assembled as free text, but synthesized directly from the validated structural skeleton. For the running example, the final synthesized query is:

> *Find customer names whose city is in the set of branch cities where account balance is greater than the average account balance of customers whose branch city is Chicago.*

The generated query is then processed by the decomposition, mapping, validation, and assembly pipeline described in Sections 2.3–2.5 to construct the corresponding reference SQL Q_{ref}.

2.7 Experiments and Results

The experimental evaluation is designed to validate GPS-Relational as an NL-to-SQL verifier and to demonstrate its use in benchmarking large language models. The evaluation is structured into two phases:

- We evaluate the translation reliability of GPS-Relational using existing ground-truth datasets such as Spider and a manually curated set of highly complex natural-language queries. This experiment tests whether the framework can translate NL questions into executable SQL and identify unsupported or invalid cases.
- We use the validated GPS-Relational pipeline to generate a benchmark of complex NL/SQL pairs with controlled nesting depth and relational complexity. These reference

outputs are then used to evaluate LLM-generated SQL by comparing execution results and analyzing how query complexity affects translation accuracy.

2.7.1 Reference Construction Accuracy

This experiment evaluates whether GPS-Relational can reliably translate natural-language queries into executable SQL that can be used as a reference for later LLM evaluation. The test set contains 500 natural-language queries. Of these, 400 were sampled from Spider across multiple database schemas and query types to test generality on existing benchmark examples. The remaining 100 were generated from the GPS grammar to test complex structures, including nested aggregation, set membership, comparative subqueries, conjunction-based set operations, and multi-level dependencies. Within this grammar-generated set, 86 cases were valid complex queries, and 14 cases were intentionally generated as unsupported or invalid inputs to test whether the framework would reject cases for which a reliable reference SQL should not be constructed.

For the valid complex cases, the grammar first produced candidate natural-language and SQL pairs. The researchers then manually checked the natural-language intent, SQL structure, and execution result to confirm that each retained pair was valid. For each test case, GPS-Relational received the natural-language question and the corresponding database schema. The framework either produced an executable SQL query Q_{GPS} or rejected the input when a required mapping, semantic constraint, or SQL construct was outside its coverage.

The generated query was executed and compared against the expected result: the Spider gold result for Spider cases and the manually checked result for the valid grammar-generated complex cases. A generated GPS query was counted as correct when its execution result matched the expected result. table 2.4 reports the number of total inputs, rejected inputs, generated SQL queries, and execution accuracy computed only over the generated SQL

queries.

Table 2.4: GPS-Relational reference construction results by query source.

Source	Total Samples	Rejected	Generated SQL	Exec. Accuracy
Spider	400	18	382	100%
Generated complex	100	14	86	100%
Total	500	32	468	100%

Table 2.5: Rejected cases during reference construction.

Rejection Reason	Cases
Missing lexical or operator mapping	10
Column or attribute ambiguity	8
Type mismatch	6
Domain inconsistency	5
Out-of-scope SQL structure	3
Total	32

The rejected cases were manually reviewed and grouped by rejection reason, as shown in table 2.5. Missing lexical or operator mappings occurred when key phrases or comparison expressions could not be grounded to known schema elements or operators. Column or attribute ambiguity occurred when a phrase could map to more than one plausible schema element. Type mismatch and domain inconsistency occurred when the requested operation violated the semantic constraints defined by the framework. Out-of-scope SQL structures were cases that required constructs not covered by the current grammar. No generated SQL failed execution-equivalence checking against the expected result. These results indicate that, within its supported grammar and mapping coverage, GPS-Relational constructs reliable executable SQL across existing benchmark queries and grammar-generated complex cases.

2.7.2 LLM Evaluation on Generated Benchmark

This experiment evaluates LLM-generated SQL using the benchmark generated by GPS-Relational. The goal is to measure how execution accuracy changes as query structure becomes more complex. For each benchmark case, the natural-language query and schema

were given to GPT-5, and the generated SQL was executed on the same database instance as the GPS-Relational reference SQL. The GPT-5 output was counted as correct when its execution result matched the reference result.

We evaluated GPT-5 on 1000 GPS-generated benchmark queries. The benchmark was stratified by nesting depth: 250 Level 0 queries, 300 low-complexity queries at Levels 1–3, 300 intermediate-complexity queries at Levels 4–9, and 150 high-complexity queries at Levels 10–15. This distribution tests both common flat queries and increasingly complex nested queries.

Performance was measured using execution accuracy. Because SQL queries can be written in multiple equivalent forms, the evaluation compares returned result tables rather than SQL text. Thus, a GPT-5 query is accepted if it produces the same execution result as the GPS-Relational reference query.

fig. 2.6 shows GPT-5 execution accuracy across the four nesting groups. Accuracy remains high for Level 0 queries, decreases for Levels 1–3, and drops sharply for deeper nesting levels. The lowest performance occurs at Levels 10–15, where the model must preserve multiple dependent subqueries and coordinated relational conditions.

A manual review of failed cases showed two common failure patterns. First, the model often lost track of dependencies between nested query levels, causing filters or aggregations to attach to the wrong scope. Second, in deeper multi-table queries, the model sometimes selected unsupported join paths or introduced unnecessary joins. These errors suggest that the main difficulty is preserving the intended relational structure as nesting depth increases.

Query Pattern (Q)**:**

$$Q ::= \{\textit{Intent Verb}\}\ \{B\}\ [\{\textit{Relational Trigger}\}\ \{F\}]$$

$\{\textit{Intent Verb}\} ::= \textit{Find} \mid \textit{List} \mid \textit{Show} \mid \textit{Return}$

$\{\textit{Relational Trigger}\} ::= \textit{whose} \mid \textit{where} \mid \textit{who} \mid \textit{with} \mid \textit{which}$

Base Patterns (B) $B ::= B_1 \mid B_2 \mid B_3$

- B_1 **(Projection):**

$$\{\textit{target\_list}\} \xrightarrow{f} \textsc{select}\ \{\textit{target\_list}\}\ \textsc{from}\ \{\textit{table}\}$$

- B_2 **(Scalar Aggregation):**

$$\{\textit{agg}\}\ \{\textit{target}\} \xrightarrow{f} \textsc{select}\ \{\textit{agg}\}(\{\textit{target}\})\ \textsc{from}\ \{\textit{table}\}$$

- B_3 **(Grouped Aggregation):**

$$\{\textit{agg}\}\ \{\textit{target}\}\ \textit{per}\ \{\textit{group}\} \xrightarrow{f} \begin{array}{l}\textsc{select}\ \{\textit{group\_attr}\},\ \{\textit{agg}\}(\{\textit{target}\}) \\ \textsc{from}\ \{\textit{table}\} \\ \textsc{group by}\ \{\textit{group\_attr}\}\end{array}$$

Filter Patterns (F) $F ::= F_1 \mid F_2 \mid F_3$

- F_1 **(Categorical Filter):**

$$\{\textit{cat\_attr}\}\{\textit{op}\}\{\textit{value}\} \xrightarrow{f} \textsc{where}\ \{\textit{cat\_attr}\}\{\textit{op}\}\{\textit{value}\}$$

- F_2 **(Numeric Filter):**

$$\{\textit{num\_attr}\}\{\textit{op}\}\{\textit{value}\} \xrightarrow{f} \textsc{where}\ \{\textit{num\_attr}\}\{\textit{op}\}\{\textit{value}\}$$

- F_3 **(Aggregate Filter):**

$$\{\textit{agg}\}\ \{\textit{num\_attr}\}\ \{\textit{op}\}\ \{\textit{value}\} \xrightarrow{f} \textsc{having}\ \{\textit{agg}\}(\{\textit{num\_attr}\})\{\textit{op}\}\{\textit{value}\}$$

Figure 2.2: Formal grammar definitions mapping abstract natural language slots (left of the arrow) to their corresponding SQL realizations (right). In the query skeleton, B and F denote the natural language forms of the base and filter productions, respectively.

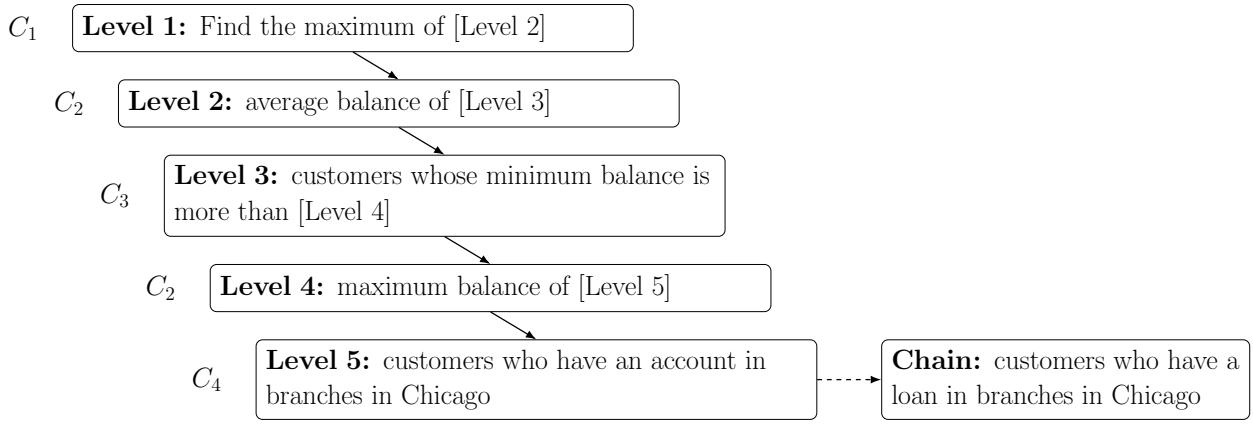


Figure 2.3: Hierarchical level map for the running example, including the conjunction-driven split at Level 5.

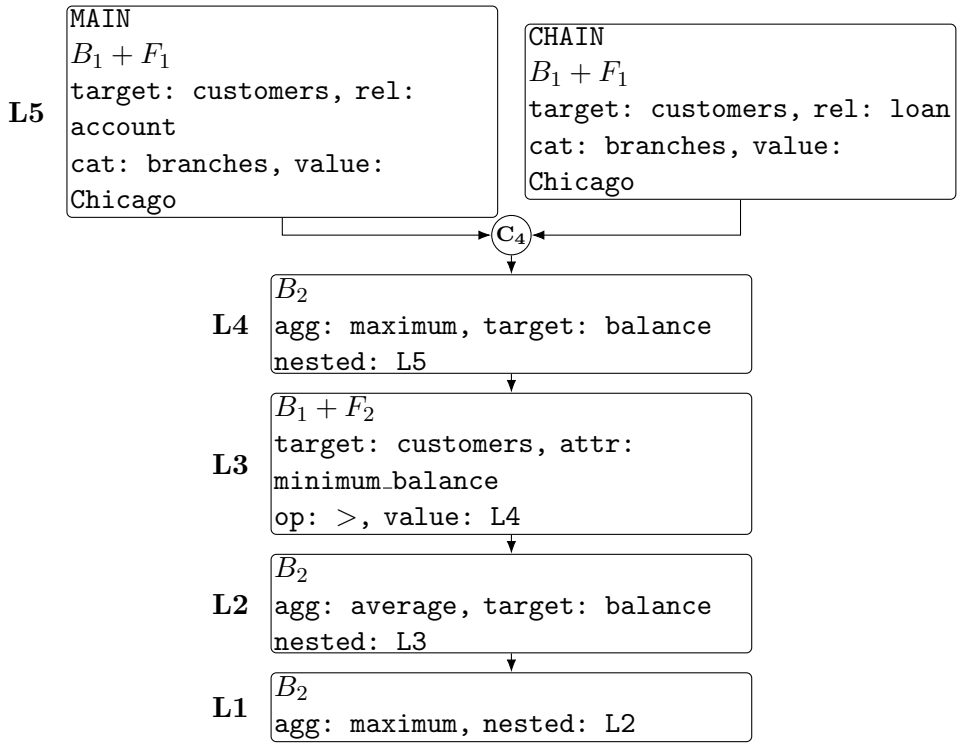


Figure 2.4: Slot-filled hierarchy for the running example, with parallel resolution at L5 and nested scalar composition from L4 to L1.

```
WITH
Level_5 AS (
  SELECT a.customer_id
  FROM Account a JOIN Branch b ON a.branch_id = b.branch_id
  WHERE b.city = 'Chicago'
  INTERSECT
  SELECT l.customer_id
  FROM Loan l JOIN Branch b ON l.branch_id = b.branch_id
  WHERE b.city = 'Chicago'
),
Level_4 AS (
  SELECT MAX(a.balance) AS max_bal
  FROM Account a
  WHERE a.customer_id IN (SELECT customer_id FROM Level_5)
),
Level_3 AS (
  SELECT a.customer_id
  FROM Account a
  GROUP BY a.customer_id
  HAVING MIN(a.balance) > (SELECT max_bal FROM Level_4)
),
Level_2 AS (
  SELECT AVG(a.balance) AS avg_bal
  FROM Account a
  WHERE a.customer_id IN (SELECT customer_id FROM Level_3)
  GROUP BY a.customer_id
)
SELECT MAX(avg_bal)
FROM Level_2;
```

Figure 2.5: Final SQL assembled from recursively resolved CTEs for the running example.

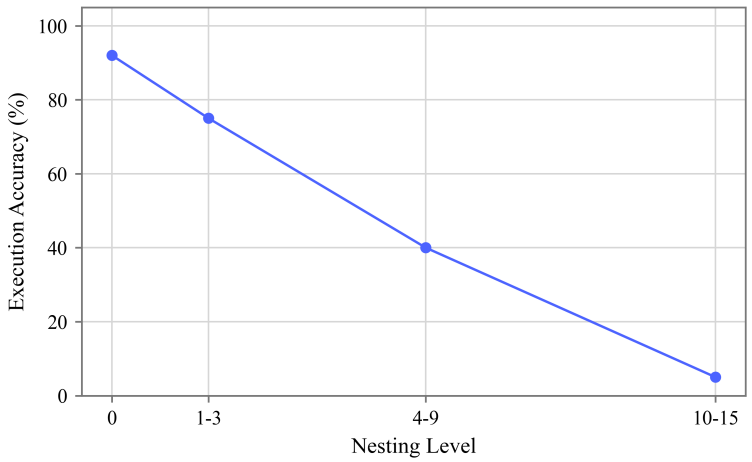


Figure 2.6: GPT-5 execution accuracy on 1000 GPS-generated benchmark queries stratified by nesting depth.

Chapter 3

GPS-CQ: Reasoning over Complex Relational Queries

In this chapter, we introduce GPS-CQ, an automated Generative Problem Solving framework for evaluating the faithfulness of natural-language explanations of complex SQL queries. The framework uses an SQL-to-natural-language-to-SQL round trip: an LLM-generated explanation is translated back into SQL, and the original and reconstructed queries are executed on dynamically generated database instances to test whether they produce equivalent results. When the results differ, GPS-CQ decomposes the queries into structural blocks and diagnoses errors involving aggregation scope, omitted conditions, nesting, join dependencies, and filtering logic. The framework also supports controlled benchmark generation. It first explores varied query structures to identify patterns that are difficult for a model and then generates targeted cases based on the diagnosed weaknesses. Canonical explanations constructed from these cases can provide structure-aware examples for later SQL-to-natural-language research. Model fine-tuning remains future work.

3.1 Introduction

SQL remains the standard language for accessing and analyzing relational databases, but complex SQL queries are notoriously difficult for non-expert users to interpret. The meaning of a complex query is rarely contained in a single clause. It is distributed across a sequence of interacting components, including nested subqueries, joins, aggregations, and set operations. To make these complex queries more accessible, Large Language Models (LLMs) are increasingly used to translate structured SQL into natural language explanations.

LLMs often produce fluent and readable summaries even when the explanation contains a semantic error. Some explanations combine operations that should remain dependent. Others omit a condition or describe the relationship between nested queries incorrectly.

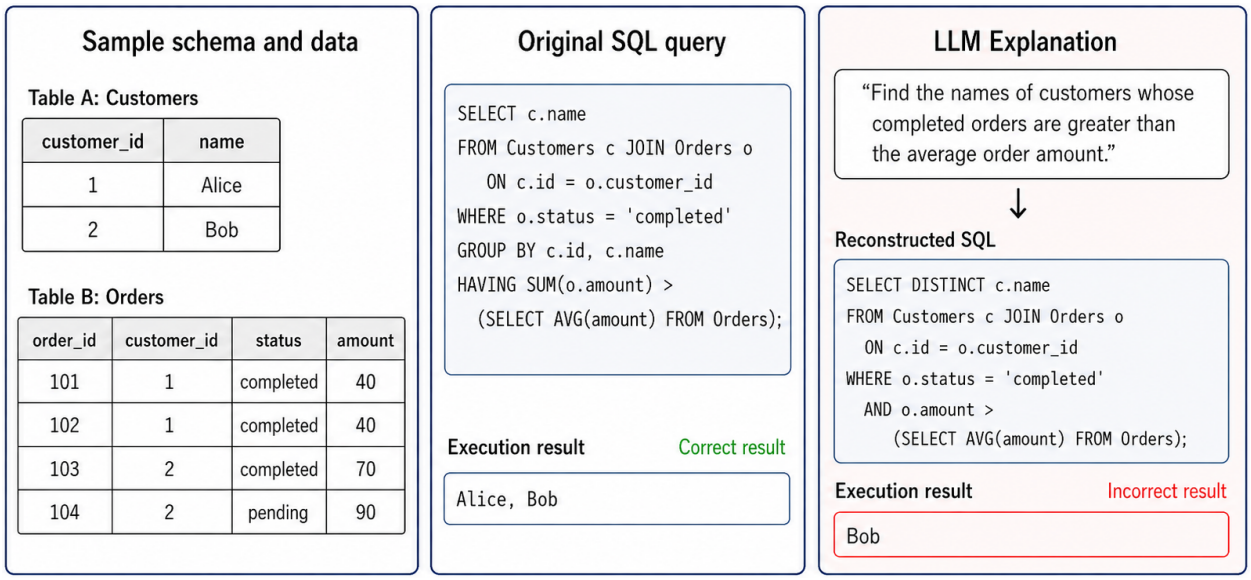


Figure 3.1: Aggregation-scope error caused by a fluent but unfaithful SQL-to-natural-language explanation.

fig. 3.1 shows an aggregation-scope error caused by a fluent but unfaithful explanation. The original SQL compares each customer's `SUM(o.amount)` over completed orders against the average order amount. However, the LLM explanation describes completed orders themselves as being greater than the average, which shifts the comparison from a customer-level

aggregate to individual order amounts. As a result, the reconstructed SQL returns a different result, even though the explanation appears natural and plausible.

Detecting these subtle logical failures is difficult, mainly because of limits in current evaluation methods. Existing benchmarks lack controlled examples with known reasoning structures. Execution-based validation is also hindered by the absence of informative database instances. Complex queries frequently return empty results on standard datasets, making it impossible to empirically prove that a translated explanation diverged from the source logic. Without controlled queries and meaningful database instances, it is difficult to determine whether an LLM is truly preserving SQL semantics or merely producing a plausible lexical approximation.

GPS-CQ is an automated Generative Problem Solving framework for complex relational queries. It evaluates an explanation through an SQL-to-natural-language-to-SQL round trip. The explanation is reconstructed as SQL, and both queries are executed on the same dynamically populated database instance. SMT-based population helps deeply constrained queries return non-empty results. Matching outputs mean that the explanation passes verification for that instance. A difference causes GPS-CQ to decompose the queries into atomic blocks and locate the structural source of the error.

GPS-CQ uses diagnosed structural weaknesses to guide benchmark generation. The initial cases cover varied query structures and nesting depths, including differences in aggregation and joins. Their failures identify structures associated with lower model accuracy, which become the focus of targeted cases. Canonical explanations from these cases can serve as structure-aware examples for later SQL-to-natural-language translation. The same cases may also contribute to future fine-tuning datasets. This paper focuses on verification, with structural diagnosis used to support inference-time guidance. Model training remains outside its scope.

3.2 Related Work

3.2.1 SQL Explanation and Query Decomposition

Research on SQL explanation has examined how to make structured queries more understandable to users. SQL-to-text systems have translated query structure into natural language using explicit structural representations, templates, or graph-based encodings [23, 24, 25, 26]. These works show that preserving structural information improves the faithfulness of generated explanations. But they generally aim to generate one fluent explanation of the SQL query as a whole. They do not expose the query logic as an explicit sequence of validated intermediate steps.

A related line of work studies decomposition as a way to improve interpretability of complex SQL. In particular, automated query decomposition has been used to break a SQL query into constituent subqueries and present them in a step-through form for debugging and inspection. This decomposition-based view is relevant to GPS-CQ because it exposes a complex query as a sequence of smaller units, even though SQL itself is declarative [27]. Earlier systems apply these units to execution debugging and the inspection of intermediate data. In GPS-CQ, decomposition is used to determine whether an LLM-generated explanation preserves the structure of the source SQL query.

3.2.2 Large Language Models for SQL Translation

Recent progress in database-language interfaces has been dominated by large language models. In Text-to-SQL, current methods improve generation through decomposition, schema linking, metadata augmentation, self-correction, and iterative refinement [28, 29, 30]. Benchmark-oriented work has further shown that model performance depends strongly on schema com-

plexity, domain shift, and benchmark design [1, 31].

These advances primarily address the generation of SQL from natural language. GPS-CQ examines whether a natural-language explanation of an existing SQL query preserves its relational semantics. The explanation is reconstructed as SQL and evaluated against the source query through execution agreement. When the results differ, structural diagnosis identifies changes in aggregation scope, filtering conditions, or dependencies between query blocks. In this setting, decomposition supports verification and diagnosis rather than generation. SQLucid uses intermediate explanations to support SQL generation [32], whereas GPS-CQ evaluates the executable faithfulness of an explanation associated with a given query.

3.2.3 Formal Verification, Evaluation, and Database Generation

A third thread of related work comes from formal verification and query evaluation. Research on SQL equivalence and symbolic reasoning has shown that SMT-based and symbolic methods can reason about SQL semantics beyond surface form [33, 34]. More recently, formal verification has been applied to Text-to-SQL evaluation by actively searching for counterexample databases that differentiate generated SQL from gold SQL, thereby exposing cases where static execution-based evaluation can be overly optimistic [35].

Verification methods often compare two SQL queries or search for database instances that expose differences between them. GPS-CQ uses execution equivalence for a different purpose. It tests whether a natural-language explanation preserves the meaning of its source query. To do this, the explanation is converted back into SQL. The reconstructed query and the source query are then executed on the same populated database. If the results differ, GPS-CQ examines the query structures to locate the mismatch.

Database-generation research also informs this evaluation. Previous systems generate data

that satisfy query constraints and help test database behavior [33]. They show why data should be created with the query in mind. This is important for deeply nested SQL because an ordinary database may cause the query to return an empty or uninformative result.

Most earlier data-generation systems were developed for database testing or workload generation. Some also support grading. GPS-CQ uses database population to produce informative outputs for round-trip evaluation. Values that affect query execution are generated using SMT constraints. The remaining attributes are filled with stochastic values. The resulting database allows GPS-CQ to compare the original and reconstructed queries and investigate any difference between their results.

3.3 Problem Definition

This section formally defines the SQL-to-natural-language explanation problem under the requirement of execution equivalence. We characterize the structural failures that disrupt this equivalence and define the execution-aware benchmark generation setting used to expose them.

3.3.1 Verified SQL-to-Natural-Language Explanation

Let S be a relational database schema, and let Q be a complex SQL query over S. The objective of the SQL-to-natural-language explanation problem is to generate an ordered natural-language explanation

$$E = \langle e_1, e_2, \ldots, e_m \rangle$$

that faithfully describes the relational logic of Q.

The input to the problem is the pair (Q, S), where Q specifies the relational computation

and S provides the schema information required to interpret tables, attributes, joins, and constraints. The output is the explanation E, where each statement e_i represents one logical step of the query.

The explanation must preserve the semantics of Q. It must retain all query conditions, preserve dependencies between nested subqueries, and represent aggregation scope correctly.

We do not rely on subjective judgments of fluency. Instead, we evaluate correctness through round-trip execution agreement on an informative database instance. Let $T(E)$ denote the SQL query reconstructed from the generated explanation E. We say that E passes round-trip verification on I if

$$\text{Exec}(Q, I) = \text{Exec}(T(E), I).$$

Under this definition, the reconstructed SQL must produce the same result as the original query on I.

3.3.2 Structural Reasoning Failures

When a language model produces a fluent but incorrect explanation, the reconstructed query $T(E)$ may no longer preserve the behavior of the original query Q. In such cases,

$$\text{Exec}(Q, I) \neq \text{Exec}(T(E), I).$$

We classify these deviations as structural reasoning failures. These failures occur when the explanation distorts the internal relational structure of the query. In our round-trip evaluation, the observed execution-equivalence failures fall into three dominant categories: aggregation scope errors, condition dropping, and dependency misalignment. For GPT-4o, these account for 35%, 41%, and 24% of failed cases, respectively. For Llama-3-70B, they

account for 28%, 34%, and 38%, respectively.

Representative failure types include:

- **Aggregation Scope Errors:** confusing row-level quantities with grouped or aggregated computations, such as explaining a grouped aggregate or HAVING condition as if it were a condition on individual rows.
- **Condition Dropping:** omitting intermediate predicates or filtering constraints during explanation compression, including WHERE predicates, nested filters, or join-related conditions required to preserve the original query logic.
- **Dependency Mismatch:** misrepresenting the relationship between nested inner queries and the outer queries that consume their results, including errors in IN, EXISTS, scalar subqueries, or parent-child query-block dependencies.

section 3.3.2 gives representative examples of how each failure category appears as a mismatch between the source and reconstructed SQL fragments.

Examples of structural failure categories.

Category	Q_{src}	Q_{pred}
Aggregation scope	`GROUP BY customer_id` `HAVING SUM(amount) > 1000`	`WHERE amount > 1000`
Condition dropping	`WHERE status = 'completed'` `AND city = 'Irvine'`	`WHERE status = 'completed'`
Dependency mismatch	`WHERE customer_id IN` `(SELECT customer_id ...)`	`WHERE customer_id IN` `(SELECT account_id ...)`

In the first row, the aggregate condition is reconstructed as a row-level filter; in the second, a required predicate is omitted; in the third, the inner query returns an incompatible identifier.

A secondary objective of the framework is to localize these failures by decomposing queries into atomic structural blocks. The finer-grained structural components used during diagnosis, such as filtering, aggregation attachment, grouping, projection, join dependency, and nesting relation, are treated as diagnostic evidence within these broader failure categories.

3.3.3 Execution-Aware Benchmark Generation

Evaluating explanation faithfulness requires an environment in which query behaviors are empirically distinguishable. We therefore define a benchmark record as a tuple

$$(Q_i, I_i, R_i),$$

where Q_i is a complex SQL query over schema S, I_i is a populated database instance over S, and

$$R_i = \text{Exec}(Q_i, I_i)$$

is the execution result of Q_i on I_i.

Let $\mathcal{G}$ denote a SQL generation grammar, and let Θ denote a set of structural generation constraints, including projection width, join structure, aggregation patterns, comparison predicates, and nesting depth.

Given $(S, \mathcal{G}, \Theta)$, the benchmark generation problem is to construct a finite collection

$$\mathcal{B} = \{(Q_i, I_i, R_i)\}_{i=1}^{N}$$

whose queries exhibit controlled structural complexity while preserving executable ground truth.

A valid benchmark record must satisfy three conditions: Q_i must be syntactically valid and consistent with schema S; I_i must satisfy the schema and query constraints required for meaningful execution; and R_i must be informative, so that semantic differences between structurally different queries can be observed through execution.

A central challenge is that deeply nested SQL queries frequently return empty or uninformative outputs on standard database instances. In such cases, incorrect explanations or

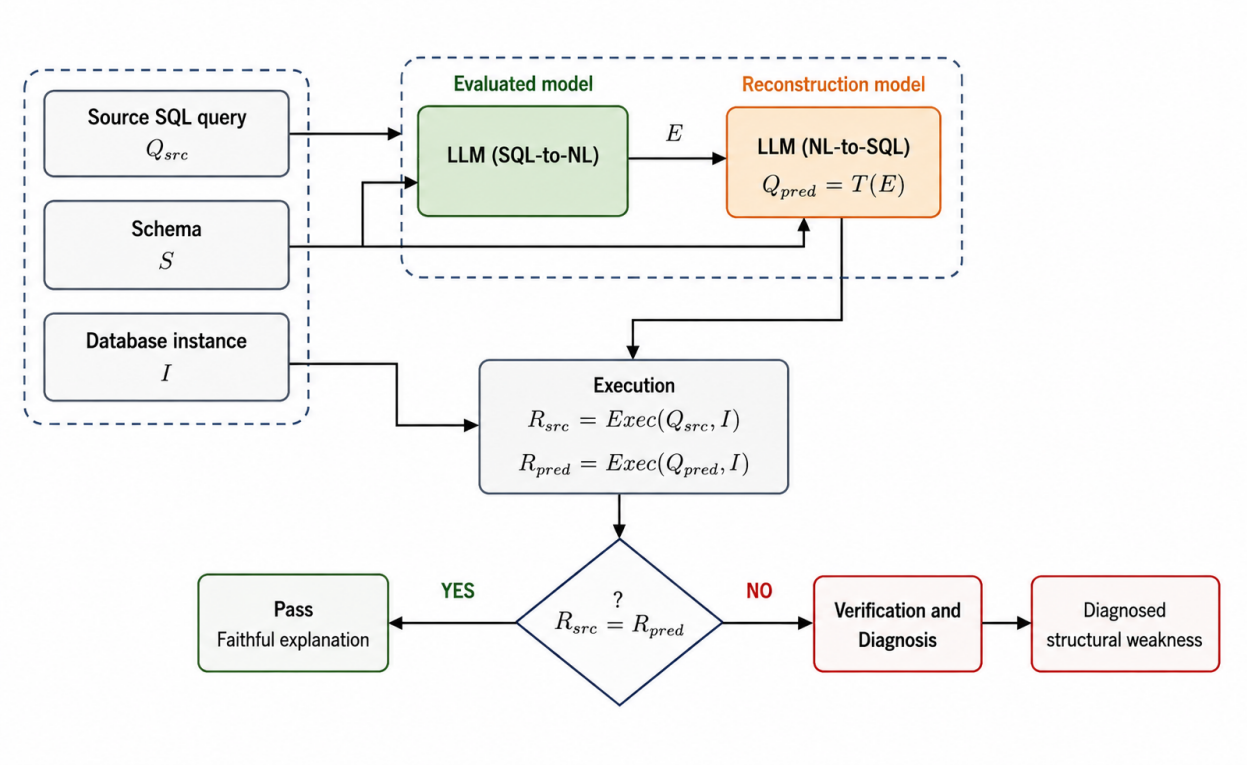


Figure 3.2: Overview of the GPS-CQ round-trip verification and diagnosis pipeline.

reconstructed queries may appear correct simply because the database does not expose the semantic difference. Therefore, benchmark construction requires structurally controlled SQL queries. It also requires database instances that make semantic differences observable.

The benchmark generation process supports two stages. The first stage performs broad structural exploration across diverse query patterns, nesting depths, aggregation structures, and join configurations. The second stage can generate additional targeted cases around diagnosed weak structures. The objective is to support execution-aware evaluation and failure diagnosis of LLM-generated SQL explanations.

3.4 GPS-CQ Verification and Diagnosis Framework

This section formalizes the verification and diagnosis framework used by GPS-CQ. The framework starts with a source SQL query Q_{src}, schema S, and database instance I. An LLM first translates Q_{src} into a natural-language explanation E. A second LLM then reconstructs SQL from E using the same schema S. We denote the reconstructed query as

$$Q_{\text{pred}} = T(E).$$

fig. 3.2 summarizes this round-trip verification process and the diagnosis path triggered when execution results diverge.

The framework executes Q_{src} and Q_{pred} on the same database instance I. If their results agree, E passes round-trip verification on I; otherwise, structural diagnosis begins. During diagnosis, both queries are decomposed into atomic SQL blocks and represented as dependency graphs. Redundant blocks are removed before the reduced structures are compared in dependency order. The first divergence may reveal an aggregation-scope error or a dropped condition. It may also expose an incorrect nesting or dependency relation. Differences in joins provide finer-grained evidence, as do errors in projection and grouping.

The reduced structure of Q_{src} also supports canonical explanation generation. After identifying the valid non-redundant structure of Q_{src}, the framework can realize each remaining block as a step in a canonical natural-language explanation. This explanation serves as the verified reference explanation for the source SQL query.

Definition 3.1 (Valid SQL)**.** *A SQL query Q is valid with respect to schema $\mathcal{S}$ if:*

1. *Q is syntactically parseable into atomic blocks $Q_0, Q_1, \ldots, Q_k$ under the decomposition rules;*

2. *each block Q_i satisfies the semantic type and domain validity constraints;*

3. *each block Q_i is constraint-feasible with respect to $\mathcal{S}$ under the SMT-based verification procedure.*

We denote the set of all valid SQL queries with respect to schema $\mathcal{S}$ as $\mathcal{Q}(\mathcal{S})$.

Definition 3.2 (Round-Trip Verification)**.** *Let $Q_{\text{src}} \in \mathcal{Q}(\mathcal{S})$ be a valid source SQL query, and let E be a natural-language explanation of Q_{src}. Define*

$$Q_{\text{pred}} = T(E)$$

as the SQL query reconstructed from E. For a valid and informative database instance I over schema $\mathcal{S}$, define

$$R_{\text{src}} = \text{Exec}(Q_{\text{src}}, I), \qquad R_{\text{pred}} = \text{Exec}(Q_{\text{pred}}, I).$$

The explanation E passes round-trip verification for Q_{src} on I if

$$R_{\text{src}} = R_{\text{pred}}.$$

Definition 3.3 (Structural Diagnosis)**.** *Let $Q_{\text{src}} \in \mathcal{Q}(\mathcal{S})$ be a valid source SQL query, let E be a natural-language explanation of Q_{src}, and let*

$$Q_{\text{pred}} = T(E)$$

denote the SQL query reconstructed from E. For a valid and informative database instance I over schema $\mathcal{S}$, let

$$R_{\text{src}} = \text{Exec}(Q_{\text{src}}, I), \qquad R_{\text{pred}} = \text{Exec}(Q_{\text{pred}}, I).$$

If

$$R_{\text{src}} \neq R_{\text{pred}},$$

then a structural diagnosis identifies the query block or dependency relation where the behavior of Q_{pred} *first diverges from the behavior of* Q_{src}*. The diagnosis is obtained by comparing the reduced dependency graphs of* Q_{src} *and* Q_{pred} *in dependency order and assigning the mismatch to the corresponding structural component, such as projection, filtering, aggregation, nesting, or join dependency.*

3.4.1 Validity Checking

Before performing structural verification or generating a canonical explanation, the framework checks whether the input SQL query belongs to the valid query class $\mathcal{Q}(\mathcal{S})$. This check establishes the conditions required by the subsequent decomposition and redundancy results. A query that fails any validity condition is rejected or flagged, and no canonical explanation is generated.

The semantic type and domain constraints used in this validity stage are adapted from the GPS-Relational reference-construction framework introduced in our earlier work [36]. Here, they ensure that the SQL blocks entering the verification and diagnosis pipeline are syntactically executable and semantically meaningful with respect to the schema.

The first condition is syntactic decomposability. The input query must be parseable into atomic SQL blocks

$$Q_0, Q_1, \ldots, Q_k$$

under the decomposition rules of the framework. This means that nested `SELECT` statements, scalar subqueries, membership subqueries, aggregate subqueries, and dependent predicates must be identifiable as separate blocks with well-defined input-output dependencies.

The second condition is semantic type and domain validity. Each block Q_i is checked to ensure that its relational operations are meaningful with respect to the schema $\mathcal{S}$. Let $\tau(c)$ denote the semantic type of a column c, and let $\text{Domain}(c)$ denote its conceptual domain. Aggregation, comparison, grouping, and set operations are valid only when their operands satisfy the required type and domain constraints. For example, a quantitative aggregation such as `AVG` or `SUM` is valid only over numerical measure attributes, while a magnitude comparison such as $>$ requires both operands to be measures from compatible domains.

Formally, for an operation op applied to column c, the local semantic validity condition is written as

$$V(\text{op}, c) = \text{true}.$$

For binary comparisons between two operands x and y, the framework requires compatible semantic types and domains. For inequality comparisons, this requires

$$V(\text{op}_{ineq}, x, y) \iff \big(\tau(x) = M \wedge \tau(y) = M\big) \wedge \big(\text{Domain}(x) = \text{Domain}(y)\big),$$

where M denotes a measure type. This prevents semantically invalid SQL structures from entering the verification and explanation pipeline for SQL expressions that compare incompatible quantities, such as age and balance, even if the SQL expression is syntactically executable.

The same compatibility requirement also applies across dependency edges between nested blocks. If a parent block consumes the output of a child block through a membership predicate such as `IN`, the child block must return a key- or domain-compatible set. If the parent block consumes the child output through a scalar comparison, the child block must return a scalar value whose semantic type and domain are compatible with the compared expression. Thus, validity is checked both locally within each block and across the parent-child dependency introduced by nesting.

The third condition is constraint feasibility. After semantic validity is checked, each block is encoded as a satisfiability problem. The formula combines schema constraints and query constraints:

$$\Phi = \Phi_{\text{schema}} \wedge \Phi_{\text{query}}.$$

Here, Φ_{schema} encodes structural constraints such as column types, nullability, primary keys, and foreign-key relationships, while Φ_{query} encodes the predicates that determine row membership. The block is considered feasible only if the resulting formula is satisfiable:

$$\text{SAT}(\Phi) = \text{true}.$$

If Φ is unsatisfiable, then the block is structurally or logically impossible under the schema constraints, and the input query is not treated as an element of $\mathcal{Q}(\mathcal{S})$.

Thus, validity checking serves as a gatekeeping stage:

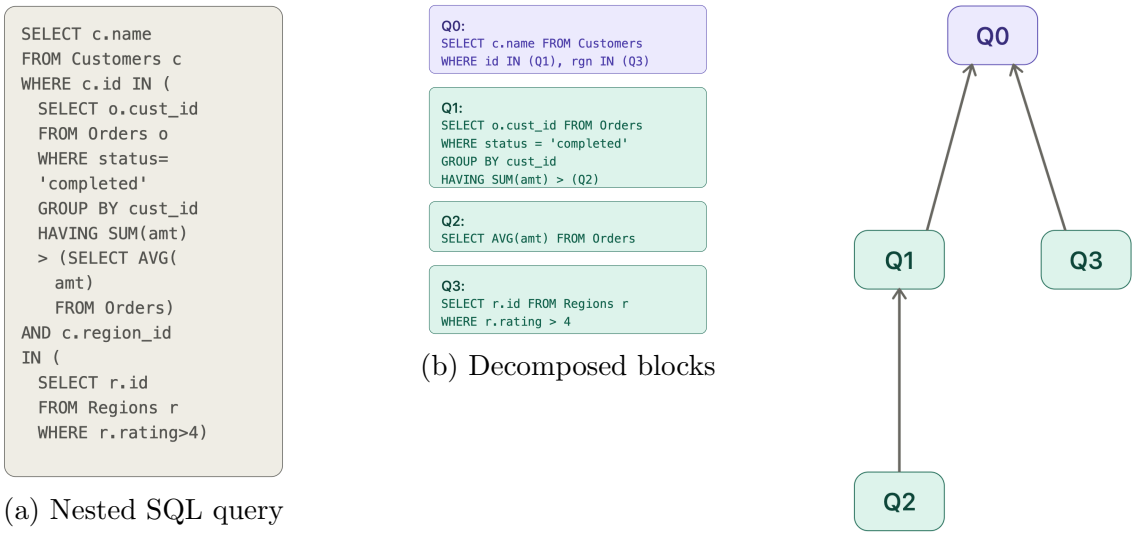
$$Q \in \mathcal{Q}(\mathcal{S}) \quad \Longleftrightarrow \quad C_{\text{syn}}(Q) \wedge C_{\text{sem}}(Q) \wedge C_{\text{sat}}(Q),$$

where $C_{\text{syn}}(Q)$ denotes syntactic decomposability, $C_{\text{sem}}(Q)$ denotes semantic validity, and $C_{\text{sat}}(Q)$ denotes constraint feasibility.

3.4.2 Query Decomposition

Given a valid SQL query $Q \in \mathcal{Q}(\mathcal{S})$, the first step of the framework is to expose its internal computation structure. Rather than treating Q as a single monolithic operation, the framework decomposes Q into atomic query blocks

$$Q_0, Q_1, \ldots, Q_k.$$



(a) Nested SQL query

(b) Decomposed blocks

(c) Dependency graph G_Q

Figure 3.3: Decomposition of a nested SQL query into atomic blocks and the corresponding dependency graph.

The representation of complex relational meaning through atomic grammar-defined units follows the GPS-Relational framework introduced in our earlier work [36]. GPS-CQ adapts this representation to existing SQL queries by extracting atomic blocks for local relational operations and their dependencies.

Each block Q_i represents one local relational operation that can be expressed independently in SQL while preserving its role in the full query. Depending on the structure of Q, an atomic block may define a filtered relation, compute a grouped result, produce an aggregate value, or return a scalar or set-valued result consumed by another block.

The decomposition begins from the outermost query block, as shown in Figure 3.3a. The input SQL is parsed into a structural representation so that nested `SELECT` clauses and dependent predicates can be identified explicitly. When an embedded subquery is found, it is extracted as a separate block Q_i, and its occurrence in the enclosing block is replaced by a placeholder reference to Q_i. This procedure is applied recursively until no nested subquery remains. The resulting atomic blocks are shown in Figure 3.3b.

After the blocks are extracted, the framework constructs a dependency graph

$$G_Q = (V, E),$$

where each node $v \in V$ corresponds to one atomic block and each directed edge

$$(Q_i, Q_j) \in E$$

indicates that the result of Q_i is consumed by Q_j. Thus, edges point from inner blocks to the enclosing blocks that depend on them. Figure 3.3c illustrates this structure for the running example.

This graph representation provides the structural reference used later for round-trip reconstruction, execution-aware verification, and localization of structural reasoning failures.

Lemma 3.1 (Decomposition Equivalence)**.** *Let $Q \in \mathcal{Q}(\mathcal{S})$ be a valid SQL query decomposed into atomic blocks $Q_0, Q_1, \ldots, Q_k$. Then for every valid database instance I over schema $\mathcal{S}$,*

$$\text{Exec}(Q, I) = \text{Exec}(Q_0[Q_1, \ldots, Q_k], I).$$

Proof. We prove the claim by structural induction on the nesting depth d of Q.

Base case. If $d = 0$, then Q contains no nested subquery. The decomposition procedure produces a single block $Q_0 = Q$. Therefore,

$$\text{Exec}(Q, I) = \text{Exec}(Q_0, I).$$

Inductive step. Assume the claim holds for all valid SQL queries of nesting depth less than d. Let Q be a valid query of nesting depth $d \geq 1$. By the decomposition procedure, each nested subquery q_i is extracted as a block Q_i, and its occurrence in the enclosing block

is replaced by a placeholder reference to Q_i.

Each extracted subquery q_i has nesting depth less than d. By the induction hypothesis,

$$\text{Exec}(q_i, I) = \text{Exec}(Q_i[\cdots], I).$$

Because relational query evaluation is compositional, replacing a subquery by an execution-equivalent query expression preserves the result of the enclosing query. Therefore, replacing every extracted subquery q_i by its corresponding reconstructed block Q_i preserves the execution result of Q.

Hence,

$$\text{Exec}(Q, I) = \text{Exec}(Q_0[Q_1, \ldots, Q_k], I).$$

□

3.4.3 Dependency Graph

The dependency graph $G_Q = (V, E)$ records how the decomposed blocks are composed into the full query. Its vertex set is

$$V = \{Q_0, Q_1, \ldots, Q_k\},$$

where each vertex represents one atomic query block. A directed edge

$$(Q_i, Q_j) \in E$$

is added when Q_j consumes the output of Q_i. Thus, Q_i must be evaluated before Q_j.

The graph therefore represents the execution dependency among atomic blocks. If a block

has no incoming edges, then it does not consume the output of any other block and can be evaluated independently. If a block has one or more incoming edges, then those predecessor blocks must be resolved first. The final output block is Q_0, which consumes the required intermediate results and appears after them in a topological ordering.

Because each edge is introduced only when a nested subquery is extracted from an enclosing query, dependencies always move from an inner block to the block that contains it. Thus, the dependency relation follows the nesting structure of the original SQL query and cannot create a cycle.

Lemma 3.2 (DAG Validity)**.** *Let Q be decomposed into atomic query blocks $Q_0, Q_1, \ldots, Q_k$, and let $G_Q = (V, E)$ be the dependency graph, where each node represents a query block and each edge $(Q_i, Q_j) \in E$ means that Q_j consumes the result of Q_i. Then G_Q is acyclic, and any topological ordering of G_Q evaluates every block before any block that consumes it.*

Proof. By construction, an edge (Q_i, Q_j) is added exactly when the result of Q_i appears as a placeholder inside Q_j. Therefore, Q_j depends on Q_i, and Q_i must be resolved before Q_j.

Since Q_i is extracted from a subquery nested inside Q_j, every dependency edge points from an inner block to an enclosing block. Hence, along every edge, the nesting depth strictly moves from a deeper subquery toward an outer query block. A cycle would require some block to eventually depend on itself, which would require the nesting relation to return to a deeper block after moving outward. This is impossible because SQL subquery containment is hierarchical.

Therefore, G_Q is acyclic. Since every acyclic directed graph admits a topological ordering, there exists an ordering of the blocks such that if $(Q_i, Q_j) \in E$, then Q_i appears before Q_j. Thus, every block is evaluated before any block that consumes its result. □

3.4.4 Redundancy Elimination

After constructing the dependency graph G_Q, the framework removes query blocks that do not change the key set consumed by downstream blocks. We consider two forms of redundancy. The first occurs when an intermediate block only reselects the same key set produced by a predecessor block without adding a new relational operation. The second occurs when two sibling blocks are combined by a set operator or a boolean connective, and one block is contained in the other.

Intermediate Redundant Blocks

We call a block Q_i an *intermediate redundant block* if it consumes the output of a predecessor block Q_{i-1} and returns the same key set without adding a new predicate, aggregation, grouping condition, projection change, or set operation. Suppose Q_i has the form

$$Q_i = \pi_K\big(\sigma_{K \in \pi_K(Q_{i-1})}(R)\big),$$

where R is the same base relation used by Q_{i-1} and K is the key column projected by Q_{i-1}. Since Q_i introduces no additional relational constraint or computation, it does not change the key set passed to downstream blocks.

In this case, the framework removes Q_i from the dependency graph and redirects its consumer to the block that produced the same key set. That is, the path

$$Q_{i-1} \to Q_i \to Q_j$$

is replaced by

$$Q_{i-1} \to Q_j.$$

Lemma 3.3 (Redundant Block Elimination)**.** *Let Q_{i-1}, Q_i, and Q_j be blocks in G_Q such that $(Q_{i-1}, Q_i) \in E$ and $(Q_i, Q_j) \in E$. Suppose Q_i is of the form*

$$Q_i = \pi_K\big(\sigma_{K \in \pi_K(Q_{i-1})}(R)\big),$$

where:

1. *R is the same base relation queried by Q_{i-1};*
2. *K is the key column projected by Q_{i-1};*
3. *Q_i adds no predicate, aggregation, grouping, projection change, or set operation;*
4. *Q_{i-1} is obtained from R using only selections and projections over attributes of R.*

Then

$$\pi_K(Q_i) = \pi_K(Q_{i-1}).$$

Proof. Let

$$A = \pi_K(Q_{i-1}).$$

By definition of Q_i,

$$\pi_K(Q_i) = \pi_K\big(\sigma_{K \in A}(R)\big).$$

Since Q_{i-1} is obtained from R using only selections and projections, every key value in A must already occur in R on column K. Therefore,

$$A \subseteq \pi_K(R).$$

Thus,

$$\pi_K\big(\sigma_{K \in A}(R)\big) = A \cap \pi_K(R) = A.$$

Since $A = \pi_K(Q_{i-1})$, we obtain

$$\pi_K(Q_i) = \pi_K(Q_{i-1}).$$

Therefore, Q_i returns the same key set as Q_{i-1} and can be removed without changing the input received by any downstream block Q_j. □

Example 3.1. *Consider the following nested SQL query:*

```
SELECT c.name
FROM Customers c
WHERE c.id IN (
    SELECT c2.id
    FROM Customers c2
    WHERE c2.id IN (
        SELECT o.customer_id
        FROM Orders o
        WHERE o.status = 'completed'
    )
);
```

The innermost subquery returns the identifiers of customers with completed orders. The middle subquery reselects the same customer identifiers from `Customers` *but adds no new predicate, aggregation, grouping condition, projection change, or set operation. By Lemma 3.3, the middle block can be eliminated, yielding the equivalent query:*

```
SELECT c.name
FROM Customers c
WHERE c.id IN (
    SELECT o.customer_id
    FROM Orders o
    WHERE o.status = 'completed'
);
```

Sibling Block Containment

A second form of redundancy arises when two sibling subqueries constrain the same parent key. Let Q_1 and Q_2 be sibling blocks consumed by the same parent block, and let $S_1(I)$ and $S_2(I)$ denote the sets of unique key values returned by these blocks on a valid database instance I. If

$$S_1(I) \subseteq S_2(I)$$

for every valid database instance I, then Q_1 is the stricter block and Q_2 is the weaker block.

This containment condition also covers duplicate sibling blocks as the special case $S_1(I) = S_2(I)$. Predicate subsumption is another special case: for example, if $c_1 \geq c_2$, then $x > c_1$ implies $x > c_2$, so the predicate $x > c_2$ is weaker and can be removed from a conjunction.

The following set identities justify the elimination:

$$S_1 \subseteq S_2 \quad \Longrightarrow \quad S_1 \cap S_2 = S_1,$$

and

$$S_1 \subseteq S_2 \quad \Longrightarrow \quad S_1 \cup S_2 = S_2.$$

Containment Checking. In the implementation, the containment condition $S_1(I) \subseteq S_2(I)$ is checked before applying the elimination rule. The framework represents each sibling block by a logical summary of the constraints that define its returned key set. Let C_1 and C_2 denote the summaries of Q_1 and Q_2, respectively. Containment is verified by checking whether the counterexample condition

$$C_1 \wedge \neg C_2$$

is satisfiable. If this formula is unsatisfiable, then no valid assignment satisfies C_1 while violating C_2; therefore $C_1 \Rightarrow C_2$, and the framework treats $S_1(I) \subseteq S_2(I)$ as verified for the

encoded block summaries.

Lemma 3.4 (Containment-Based Sibling Elimination)**.** *Let Q_{main} be a query block that filters an outer tuple x using two sibling subqueries Q_1 and Q_2 over the same key attribute. Let $S_1(I)$ and $S_2(I)$ denote the sets of unique key values returned by Q_1 and Q_2 on a valid database instance I. Suppose that, for every valid database instance I,*

$$S_1(I) \subseteq S_2(I).$$

Then the following eliminations preserve the selected key set of the parent block:

$$x \in Q_1 \wedge x \in Q_2 \equiv x \in Q_1,$$

and

$$x \in Q_1 \vee x \in Q_2 \equiv x \in Q_2.$$

Equivalently, under duplicate-eliminating set semantics,

$$Q_1 \ \texttt{INTERSECT}\ Q_2 \equiv Q_1$$

and

$$Q_1 \ \texttt{UNION}\ Q_2 \equiv Q_2.$$

Proof. The SQL predicate $x \in Q_i$ is evaluated as a boolean membership test. Its truth value depends only on whether x appears among the key values returned by Q_i. Therefore, for each valid database instance I, each subquery Q_i can be associated with a set of returned key values $S_i(I)$.

By assumption,

$$S_1(I) \subseteq S_2(I).$$

Hence, for any outer tuple value x,

$$x \in S_1(I) \implies x \in S_2(I).$$

For the conjunctive case, the parent condition requires

$$x \in S_1(I) \wedge x \in S_2(I),$$

which is equivalent to

$$x \in S_1(I) \cap S_2(I).$$

Since $S_1(I) \subseteq S_2(I)$,

$$S_1(I) \cap S_2(I) = S_1(I).$$

Thus,

$$x \in S_1(I) \wedge x \in S_2(I) \iff x \in S_1(I).$$

Therefore, under `AND` or `INTERSECT`, the weaker sibling Q_2 can be eliminated and the stricter sibling Q_1 is retained.

For the disjunctive case, the parent condition requires

$$x \in S_1(I) \vee x \in S_2(I),$$

which is equivalent to

$$x \in S_1(I) \cup S_2(I).$$

Since $S_1(I) \subseteq S_2(I)$,

$$S_1(I) \cup S_2(I) = S_2(I).$$

Thus,

$$x \in S_1(I) \vee x \in S_2(I) \iff x \in S_2(I).$$

Therefore, under `OR` or duplicate-eliminating `UNION`, the stricter sibling Q_1 can be eliminated and the weaker sibling Q_2 is retained.

These equivalences hold for every outer tuple x and every valid database instance I, so the selected key set of the parent block is preserved. □

Example 3.2. *Consider a parent query that filters customers using two sibling subqueries connected by `AND`:*

```
SELECT c.name
FROM Customers c
WHERE c.id IN (
    SELECT o.customer_id
    FROM Orders o
    JOIN Payments p ON o.order_id = p.order_id
    JOIN Branch b ON p.branch_id = b.branch_id
    WHERE o.status = 'completed'
      AND p.payment_type = 'credit'
      AND b.branch_city = 'Irvine'
      AND o.amount > (
          SELECT AVG(o2.amount)
          FROM Orders o2
      )
)
AND c.id IN (
    SELECT o3.customer_id
    FROM Orders o3
    JOIN Payments p3 ON o3.order_id = p3.order_id
    WHERE o3.status = 'completed'
      AND p3.payment_type = 'credit'
```

```
);
```

Let Q_1 be the first subquery and Q_2 be the second subquery. The first subquery contains all requirements of the second subquery: a completed order and a credit payment. It also adds additional restrictions, namely that the payment is associated with an Irvine branch and that the order amount is above the average order amount. Therefore, every customer identifier returned by Q_1 is also returned by Q_2, so

$$S_1(I) \subseteq S_2(I).$$

By Lemma 3.4,

$$c.id \in Q_1 \wedge c.id \in Q_2 \equiv c.id \in Q_1.$$

Thus, the second sibling subquery is eliminated.

Remark. The `UNION` case in Lemma 3.4 assumes standard duplicate-eliminating set semantics. It does not apply to `UNION ALL`, since `UNION ALL` preserves duplicate multiplicities. Eliminating a contained branch under `UNION ALL` may change the multiplicity of output tuples.

After all applicable elimination rules are applied exhaustively, the resulting reduced dependency graph is denoted by G_Q^*.

Theorem 3.1 (Reduced Structural Equivalence)**.** *Let G_Q^* be the reduced dependency graph obtained from G_Q after applying the redundancy elimination rules exhaustively. Let Q^* be the SQL expression represented by composing the remaining blocks of G_Q^* in topological order. Then for every valid database instance I over schema $\mathcal{S}$,*

$$\text{Exec}(Q_0[Q_1, \ldots, Q_k], I) = \text{Exec}(Q^*, I).$$

Proof. The reduced graph G_Q^* is obtained by a finite sequence of elimination steps. Each step is one of the two redundancy eliminations defined above.

First, if the step removes an intermediate redundant block, then by Lemma 3.3, the removed block returns the same key set as its predecessor:

$$\pi_K(Q_i) = \pi_K(Q_{i-1}).$$

Thus, any downstream block receives the same key set after the edge is redirected from Q_i to Q_{i-1}.

Second, if the step removes a containment-based sibling block, then by Lemma 3.4, the parent block's selected key set is unchanged under the corresponding `AND`, `OR`, `INTERSECT`, or duplicate-eliminating `UNION` composition.

Therefore, each individual elimination step preserves the execution result of the current composed query expression. By induction over the finite sequence of elimination steps, the final reduced expression Q^* represented by G_Q^* is execution-equivalent to the original decomposed expression $Q_0[Q_1, \ldots, Q_k]$. Hence,

$$\text{Exec}(Q_0[Q_1, \ldots, Q_k], I) = \text{Exec}(Q^*, I).$$

□

Exhaustive Redundancy Reduction

Theorem 3.1 shows that applying the redundancy elimination rules preserves the executable semantics of the decomposed SQL query. We now state the corresponding minimality property of the reduced graph. This minimality is with respect to the two elimination rules

defined in Section 3.4.4: intermediate redundant-block elimination and containment-based sibling elimination.

Theorem 3.2 (Redundancy Minimality)**.** *Let G_Q^* be the reduced dependency graph obtained from G_Q after exhaustively applying the redundancy elimination rules. Then G_Q^* is minimal with respect to these rules: no remaining block in G_Q^* satisfies either the intermediate redundant-block condition or the containment-based sibling elimination condition.*

Proof. We prove the claim by contradiction. Suppose there exists a remaining block or sibling condition in G_Q^* that satisfies one of the redundancy elimination conditions.

First, suppose there exists a block $Q_i \in V(G_Q^*)$ satisfying the intermediate redundant-block condition. Then Q_i consumes the output of some predecessor block Q_{i-1}, queries the same base relation, projects the same key column K, and adds no additional predicate, aggregation, grouping, projection change, or set operation. By Lemma 3.3,

$$\pi_K(Q_i) = \pi_K(Q_{i-1}).$$

Therefore, Q_i can be removed by redirecting its downstream consumer to Q_{i-1} directly.

Second, suppose there exist sibling blocks Q_1 and Q_2 in G_Q^* that satisfy the containment-based sibling elimination condition. Then, for every valid database instance I, their returned key sets satisfy either

$$S_1(I) \subseteq S_2(I)$$

or the symmetric containment relation. By Lemma 3.4, the appropriate sibling can be removed according to the corresponding `AND`, `OR`, `INTERSECT`, or duplicate-eliminating `UNION` composition without changing the selected key set of the parent block.

In either case, an additional elimination step would be possible. However, G_Q^* is defined as the graph obtained after applying the redundancy elimination rules exhaustively. Hence, any

block or sibling condition satisfying one of these elimination conditions should already have been removed. This contradicts the assumption that such a removable structure remains in G_Q^*.

Therefore, no remaining block in G_Q^* satisfies either redundancy elimination condition. Thus, G_Q^* is minimal with respect to the defined redundancy elimination rules. □

3.4.5 Structural Error Diagnosis

When full-query execution detects a mismatch between Q_{src} and Q_{pred}, the framework begins structural diagnosis. Execution equivalence has already established that the explanation is not faithful; diagnosis locates where the semantic divergence begins.

Require: Source SQL Q_{src}, reconstructed SQL Q_{pred}, database instance I
Ensure: Round-trip verification result or diagnosed failure category
1: $R_{\text{src}} \leftarrow \text{Exec}(Q_{\text{src}}, I)$
2: $R_{\text{pred}} \leftarrow \text{Exec}(Q_{\text{pred}}, I)$
3: **if** $R_{\text{src}} = R_{\text{pred}}$ **then**
4: **return** PASS
5: **end if**
6: $G^*_{\text{src}} \leftarrow \text{ReduceGraph}(\text{BuildGraph}(\text{Decompose}(Q_{\text{src}})))$
7: $G^*_{\text{pred}} \leftarrow \text{ReduceGraph}(\text{BuildGraph}(\text{Decompose}(Q_{\text{pred}})))$
8: $(\mathcal{P}, U_{\text{src}}, U_{\text{pred}}) \leftarrow \text{AlignBlocks}(G^*_{\text{src}}, G^*_{\text{pred}})$
9: **if** $U_{\text{src}} \neq \emptyset \vee U_{\text{pred}} \neq \emptyset$ **then**
10: **return** DEPENDENCYMISMATCH
11: **end if**
12: **for** each $(b_{\text{src}}, b_{\text{pred}}) \in \mathcal{P}$, in dependency order **do**
13: $R'_{\text{src}} \leftarrow \text{ExecSubgraph}(b_{\text{src}}, G^*_{\text{src}}, I)$
14: $R'_{\text{pred}} \leftarrow \text{ExecSubgraph}(b_{\text{pred}}, G^*_{\text{pred}}, I)$
15: **if** $R'_{\text{src}} \neq R'_{\text{pred}}$ **then**
16: **return** ClassifyMismatch$(b_{\text{src}}, b_{\text{pred}}, G^*_{\text{src}}, G^*_{\text{pred}})$
17: **end if**
18: **end for**
19: **return** OTHERSTRUCTURALMISMATCH

Algorithm 2: EXECUTIONDRIVENSTRUCTURALDIAGNOSIS

Algorithm 2 decomposes Q_{src} and Q_{pred} into atomic SQL blocks, constructs their dependency

graphs, and removes redundant blocks. Let G^*_{src} and G^*_{pred} denote the reduced dependency graphs of Q_{src} and Q_{pred}, respectively. The framework compares these graphs in dependency order, beginning with the innermost blocks and ending with the final output block.

For each corresponding pair, the framework compares the block-level execution results on the same database instance. Matching results advance the comparison to the next dependent block. A mismatch identifies the first divergent block and its dependency edges. This execution check excludes structural differences that do not affect query behavior.

The first divergent block or edge determines the diagnostic label. Aggregation-scope errors involve differences in an aggregation function, grouping attribute, or `HAVING` condition. Missing predicates, nested filters, and join-related conditions indicate condition dropping. Dependency mismatch is assigned when an inner block connects incorrectly to its consumer through `IN`, `EXISTS`, a scalar subquery, or a consumed output column. section 3.3.2 in section 3.3.2 provides examples of these failure categories.

Each localized failure is recorded as a structural pattern rather than as an isolated query error. This pattern includes the failed block, the filter or comparison structure attached to it, the parent-child dependency relation, the aggregation or grouping scope when present, and the depth of the block in the dependency graph. For example, a failure may correspond to a grouped aggregation block with a `HAVING` predicate being reconstructed as a row-level `WHERE` predicate.

The benchmark-generation component uses the extracted failure pattern to produce additional valid and executable examples with the same weak structure. These targeted cases show whether the failure is accidental or systematic and focus testing on structures that explanation models are most likely to misrepresent.

3.5 Benchmark Generation

We now describe our approach to the SQL benchmark generation problem defined in section 3.3.3. Because existing benchmarks lack controlled examples of deeply nested and structurally complex queries, our goal is to systematically generate a diverse benchmark of executable records in the form $(Q, I, \text{Exec}(Q, I))$ from a given schema. This generation process supports both broad structural exploration and targeted generation of additional cases in model-specific weak regions identified through evaluation.

The generation process begins with grammar-guided SQL synthesis, building bottom-up from atomic blocks to recursively composed queries. Next, database population ensures these queries yield informative execution results. We also enforce scalability control. This bounds the overall query space by limiting projection width, join paths, and nesting depth.

3.5.1 SQL Synthesis

We generate benchmark queries by extending the recursive construction of GPS-Relational [36]. The generator starts with an innermost atomic block and recursively adds dependent outer blocks. Base templates, filter patterns, and vertical-nesting rules determine the abstract SQL skeleton before schema elements are assigned.

Nesting Depth Bound. Nesting depth is a generation parameter that must be fixed before query construction begins. Without an upper bound, recursively wrapping blocks may produce queries that are either structurally redundant or not executable on the target database engine.

First, the schema itself limits how many meaningful dependencies can be introduced. Once the nesting depth exceeds the longest simple dependency path in the schema graph G_S,

the generator may revisit tables or roles that have already appeared in inner blocks. Such queries may still be syntactically valid, but they add little new relational complexity to the benchmark.

Second, database engines impose practical execution limits on nested subqueries. For example, Microsoft SQL Server lists *nested subqueries* with a maximum value of 32 in its maximum capacity specifications [37]. MySQL also defines an error condition for excessive SELECT nesting, `ER_TOO_HIGH_LEVEL_OF_NESTING_FOR_SELECT`, although the referenced documentation does not specify a fixed numerical bound [38]. Queries exceeding such practical limits may be syntactically valid but fail at runtime.

For these reasons, we define an initial structural upper bound on nesting depth as

$$L_{\max} = \min(L_S, L_{\text{dbms}}),$$

where

$$L_S = \max_{p \in \mathcal{P}(G_S)} |p|$$

is the length of the longest simple dependency path in the schema graph G_S, and L_{dbms} is the nesting limit of the target database engine. The schema-based bound L_S prevents structurally redundant nesting, while the execution-based bound L_{dbms} ensures that generated queries remain executable. Any selected nesting depth L must satisfy

$$L \leq L_{\max}.$$

In practice, the effective nesting depth used during benchmark construction is further evaluated empirically based on execution feasibility, generation scalability, and runtime behavior on the target database system. Section 4.7 reports this analysis.

Slot Type Constraints. Each generated block contains typed slots that restrict the schema elements allowed in its SQL template. Within a block, the SQL role determines the slot type. An aggregation slot such as `AVG({column})` requires a numerical measure column, whereas the value in `WHERE {column} = {value}` must come from the selected column's domain. Across nested blocks, the output of an inner query must be compatible with the consuming outer slot. Membership conditions such as `WHERE {column} IN (`Q_{inner}`)` require the outer and returned columns to have compatible keys or domains. Scalar comparisons such as `HAVING AVG({column}) >` Q_{inner} require both operands to belong to compatible numerical domains. These constraints limit the candidate columns, comparison operators, and aggregation functions considered during schema binding.

Column Assignment Search Space. A single atomic SQL block may contain column slots that must be assigned from the set of candidate attributes reachable in the schema graph. Let C denote the number of candidate columns available for a generated query block. If the generator were allowed to choose any non-empty subset of these columns, the number of possible column choices is

$$\sum_{i=1}^{C} \binom{C}{i} = 2^C - 1 = O(2^C),$$

which grows exponentially with C. As the schema grows and more attributes become reachable, even a single atomic block can have an intractably large column-assignment search space. To control this, the generator restricts each atomic block to at most K column placeholders.

Proposition 3.1 (Bounded Column Assignment Search Space)**.** *For fixed K, the number of possible column assignments for an atomic SQL block is polynomial in C.*

Proof. If each block contains at most K column placeholders, then the selection of columns is

Require: Block $b = (\mathbf{s}, \mathbf{c}, \mathbf{v})$, database d, used signatures H
Ensure: Column assignment $\mathbf{z}$, join set J, signature σ

repeat
  Sample z_1 from $\mathcal{C}_d(\tau_1)$
  $\mathbf{z} \leftarrow \{z_1\},\ J \leftarrow \emptyset$
  for $c_i \leftarrow c_2 : c_m$ **do**
    Run BFS from T_{z_1}
    At each frontier table T, compute $\mathcal{C}_i(T) = \{c \in T \mid \tau(c) = \tau_i\}$
    Let T^* be the first frontier table with $\mathcal{C}_i(T^*) \neq \emptyset$
    if no such T^* exists **then**
      Resample z_1; restart
    end if
    Sample z_i uniformly from $\mathcal{C}_i(T^*)$
    $\mathbf{z} \leftarrow \mathbf{z} \cup \{z_i\}$
    $J \leftarrow J \cup \mathrm{Path}(T_{z_1}, T^*)$
  end for
  Build signature $\sigma = \{(\mathrm{role}_i, T_{z_i}, z_i)\}_{i=1}^{m}$
until $\sigma \notin H$
$H \leftarrow H \cup \{\sigma\}$
for $v_j \leftarrow v_1 : v_n$ **do**
  Sample v_j from the domain of its associated column in $\mathbf{z}$
end for
return $\mathbf{z}, J, \sigma$

Algorithm 3: SchemaAwareColumnSamplingWithSignatureControl

restricted to subsets of size at most K. Therefore, the number of possible column assignments is bounded by

$$\sum_{i=1}^{K} \binom{C}{i} \leq KC^K = O(C^K).$$

Since K is fixed by the template grammar, the column-assignment search space is polynomial in C. □

Join-Path Control. During schema binding, type compatibility alone is not sufficient to produce relationally coherent SQL blocks. Two columns may satisfy the required semantic types but belong to tables that are far apart in the foreign-key graph, forcing the generated query to include long or unnecessary join paths. For example, if a block requires a numerical

target column and a text filter column, independent sampling may choose the target from an `account` table and the filter from a distant `branch_region` table, even when a closer compatible filter exists in a neighboring table. This produces syntactically valid SQL, but the join structure may be artificial and harder to interpret. Our schema binding procedure controls which column types are selected. It also controls how far the selected tables are from the anchor table of the block.

Schema-Aware Column Sampling. To implement this control, we anchor each atomic block at the table of the first selected column and search outward through the foreign-key graph until a type-compatible column is found.

For a given database d, we construct a foreign-key graph over the schema tables and traverse it as an undirected graph for join reachability. The distance between any two tables is the minimum number of hops required to connect them through foreign-key relationships. Define an atomic block b as $(\mathbf{s}, \mathbf{c}, \mathbf{v})$, where $\mathbf{s}$ is the structural pattern, $\mathbf{c} = [c_1, \ldots, c_m]$ is the ordered list of column placeholders, and $\mathbf{v} = [v_1, \ldots, v_n]$ is the list of value placeholders. Each placeholder c_i carries a slot type constraint τ_i that must be satisfied by any assigned column. Denote T_c as the table containing column c, and let $\mathcal{C}_d(\tau)$ be the set of all columns in d satisfying type constraint τ. The schema-aware column sampling procedure is described in Algorithm 3.

The intuition behind this sampling strategy is as follows: once the first column is selected, subsequent columns are drawn from tables that are as close as possible in the foreign-key graph, so as to minimize unnecessary joins and produce relationally coherent queries. We achieve this by performing a breadth-first traversal of the foreign-key graph from the table of the first selected column and taking the first frontier that contains a type-compatible candidate. To preserve benchmark diversity while still allowing column reuse, we maintain a set of previously generated column-role signatures H. A new assignment is accepted only

if its signature has not already appeared in H.

For example, suppose the sampled block produces

```
SELECT AVG(account.balance)
FROM account
JOIN customer ON account.customer_id = customer.id
WHERE customer.city = 'Irvine';
```

The corresponding column-role signature is

$$\sigma = \{(\text{target}, \texttt{account}, \texttt{balance}), (\text{filter}, \texttt{customer}, \texttt{city})\}.$$

Thus, the same column may still appear in future queries, but the same target–filter table-column pattern is not generated repeatedly.

3.5.2 Canonical Explanation Construction

After a valid SQL query is generated, the benchmark also constructs a canonical natural-language explanation from the same grammar structure. This explanation is not produced by an LLM. Instead, it is generated from the SQL skeleton, slot assignments, and dependency structure used during SQL synthesis.

Each generated SQL query is represented as a sequence of atomic blocks. For each block, the generator records the corresponding tables, columns, predicates, aggregation functions, comparison operators, constants, and references to child blocks. The natural-language explanation is then produced by rendering each block as one ordered explanation step.

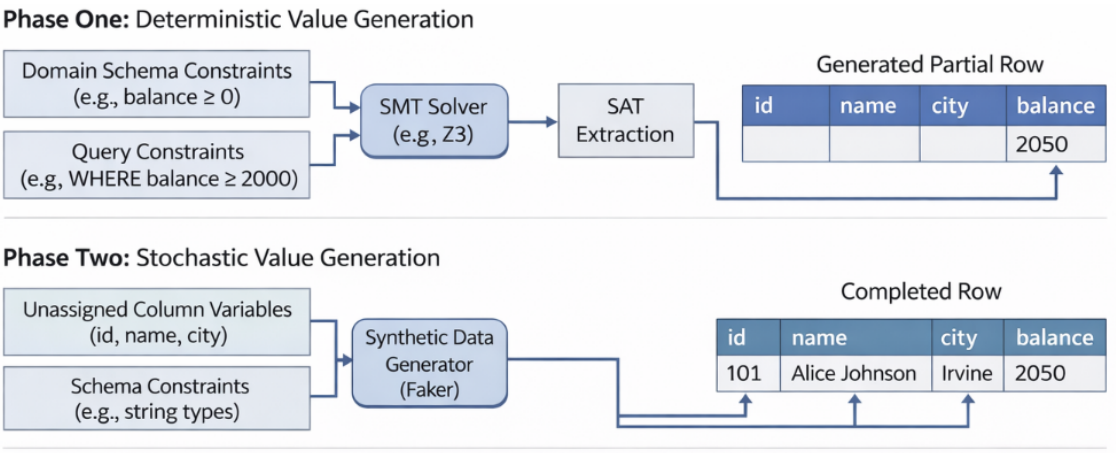


Figure 3.4: Two-phase database population workflow.

The blocks are realized in dependency order, so inner queries are described before the outer queries that consume them. This gives each benchmark record a structurally aligned reference explanation:

$$(Q_i, I_i, R_i, E_i),$$

where Q_i is the generated SQL query, I_i is the populated database instance, $R_i = \text{Exec}(Q_i, I_i)$ is the execution result, and E_i is the canonical explanation.

The benchmark therefore includes executable SQL queries and reference explanations aligned with the known query structure. These controlled references are used to test whether LLM-generated explanations preserve the same relational dependencies.

3.5.3 Database Population

The objective of database population is to pair each generated benchmark query Q with a database instance I such that executing Q on I produces an informative, non-empty result. This requirement is necessary for execution-based evaluation: if both the original query and

a reconstructed query return empty results, semantic differences between them may remain hidden.

We formulate population as a constraint-guided value generation problem. As summarized in fig. 3.4, the procedure has two phases. The first phase assigns values to the columns that directly affect query execution, including columns used in joins, filters, grouping, aggregation, and nested predicates. These values are generated by solving schema and query constraints. The second phase fills all remaining unconstrained attributes with type-appropriate synthetic values.

The population procedure must satisfy three global requirements:

- **Non-empty output:** the generated instance must satisfy

$$\text{Exec}(Q, I) \neq \emptyset.$$

- **Null avoidance:** values used in query predicates must be non-null, preventing null-rejecting predicates from invalidating the generated rows.
- **Positive numeric domain:** numeric values used in aggregation or comparison predicates are generated from a positive domain when required, so aggregate operations such as `SUM` and `AVG` produce meaningful non-zero results.

Phase One: Deterministic Value Generation

In the first phase, the system extracts the constrained columns of Q. These include columns that appear in join predicates, selection predicates, grouping clauses, aggregate expressions, and nested query conditions. The corresponding values are generated using an SMT solver

over a formula

$$\Phi = \Phi_{\text{schema}} \wedge \Phi_{\text{query}},$$

where Φ_{schema} encodes schema constraints such as primary-key uniqueness, non-nullability, foreign-key relationships, column types, and available value ranges, while Φ_{query} encodes the predicate constraints required for Q to return a non-empty result.

Rather than assigning arbitrary values, the solver translates SQL conditions into symbolic constraints and produces witness values that make the relevant predicates true. For equality constraints, such as `status = 'active'`, the symbolic attribute is restricted to the literal constant. For inequality and range constraints, such as `balance > 1000`, the solver selects a satisfying witness value from the valid domain. When no explicit numeric range is provided, unbounded numeric domains are constrained to a strictly positive base, such as $x \geq 1$, to avoid unintended zero-valued witnesses.

For aggregation predicates, the population step generates witness rows whose collective values satisfy the aggregate condition. For cardinality constraints of the form $\text{COUNT}(*) > k$, the generator instantiates at least $k + 1$ matching rows. For magnitude aggregates such as $\text{SUM}(x) > c$ or $\text{AVG}(x) > c$, the solver generates witness values $\{x_1, \ldots, x_n\}$, where x_i is the value of attribute x in the i-th inserted row, such that

$$\sum_{i=1}^{n} x_i > c \qquad \text{or} \qquad \frac{1}{n} \sum_{i=1}^{n} x_i > c,$$

respectively. For extremum aggregates, such as $\text{MIN}(x) > c$ or $\text{MAX}(x) < c$, the constraint is applied to every inserted witness row in the group: for all i, the generated value x_i must satisfy $x_i > c$ or $x_i < c$, respectively.

Values are inserted into the database in an order that preserves referential integrity. Parent rows are generated before child rows so that primary keys can be propagated into the corre-

sponding foreign-key columns. Columns participating in join predicates are unified during solving, ensuring that the exact same value is inserted on both sides of the join.

Phase Two: Stochastic Value Generation

After all query-constrained columns have been assigned values, the remaining unconstrained attributes are filled using stochastic synthetic generation. These columns do not affect the truth of the query predicates, but they are needed to complete valid tuples in the database instance. For such attributes, Faker is used to generate type-appropriate values, such as names, cities, dates, or categorical labels.

Finally, the generated query is executed on the populated instance:

$$R = \mathrm{Exec}(Q, I).$$

If $R = \emptyset$, the population step is repaired or regenerated. Otherwise, the benchmark stores the executable record

$$(Q, I, R).$$

3.6 Evaluation

The evaluation covers SQL-to-NL-to-SQL faithfulness measured by execution equivalence, query-generation robustness, and feasible nesting capacity.

Implementation details. SQL queries are executed using SQLite through Python's `sqlite3` interface. SQL parsing and decomposition are performed using SQLGlot, and satisfiability checking is performed with the Z3 SMT solver through SMT-LIB encodings. Unconstrained database fields are populated using Faker or schema-defined sample values.

3.6.1 Execution Equivalence and Failure Diagnosis

To assess whether relational semantics are preserved through the SQL-to-NL-to-SQL round trip, we evaluate GPT-4o and Llama-3-70B using standard zero-shot prompts across a benchmark of 1,000 generated queries, which are stratified by nesting depth. Both the source query and the reconstructed query are executed against the generated database instance, and a prediction is counted as correct only when the execution outputs match exactly. Because all source queries are guaranteed valid by the schema and generation grammar, any execution mismatch directly identifies semantic loss during translation rather than ambiguity in the original query.

As shown in table 3.1, execution equivalence decreases as nesting depth increases. Both models perform strongly on single-depth queries, but accuracy drops sharply on nested queries, especially when multiple dependency levels must be preserved.

Table 3.1: Execution equivalence by nesting depth.

Nesting Depth	**GPT-4o**	**Llama-3-70B**
Overall	**74.5%**	**58.2%**
Depth 1: simple joins and filters	91.2%	83.4%
Depth 2: nested subqueries	68.5%	49.1%
Depth 3+: deeply nested SQL	42.1%	21.6%

table 3.2 reports the distribution of error types among incorrect predictions only; each failed case is assigned to one dominant failure category, so the categories sum to 100% for each model.

Table 3.2: Failure diagnosis among queries that failed execution equivalence.

Failure Type	**GPT-4o**	**Llama-3-70B**
Aggregation scope error	35%	28%
Condition dropped	41%	34%
Dependency mismatch	24%	38%

3.6.2 GPS-CQ Query Generation Robustness

Generation of query-constrained values becomes deterministic after a SQL skeleton is fixed and the SMT solver returns *sat*. The returned witness values populate the constrained fields. Unconstrained attributes remain stochastic, as described in section 3.5.3. The robustness measurement therefore focuses on GPS-CQ query-structure generation. As reported in table 3.3, an attempt succeeds only when the sampled SQL skeleton can be bound to valid schema elements and its constraints are satisfiable.

Let N denote the total number of GPS-CQ query-generation attempts and x the number rejected. Of 1,000 attempts, 843 produced valid SQL queries, giving a success rate of 84.3%. Among the 157 rejected attempts, 61.8% lacked an eligible schema binding and 38.2% contained SMT-unsatisfiable constraints.

Table 3.3: GPS-CQ query-generation robustness over 1,000 generation attempts.

Metric	**Result**
Generation attempts (N)	1,000
Successfully generated SQLs ($N - x$)	843 (84.3%)
Rejected attempts (x)	157 (15.7%)

Rejected attempts arose from limited schema coverage or contradictory constraints. Some schemas lacked reachable measure, ID, or attribute columns for the generated query skeleton. Other attempts produced nested filters or comparisons that made the SMT constraints unsatisfiable.

3.6.3 GPS-CQ Feasible Nesting Capacity

Because deeper nesting requires more successful schema bindings across multiple query levels, we evaluate the framework's *feasible nesting capacity*. We compare two schema settings: a compact schema with fewer role-compatible measure, ID, and attribute columns, and a broad

schema with more reachable columns for each required relational role.

For each schema and target depth, we requested 1,000 valid SQL queries. For each requested query, the generator was allowed up to 10 attempts. If no valid schema-bound SQL query was produced within 10 attempts, that requested query was counted as failed. The success rate reports the percentage of requested queries that were successfully generated within this attempt limit. We report the success rate as:

$$\text{Success Rate} = \frac{\text{Number of valid queries generated}}{\text{Number of requested queries}}.$$

This experiment measures the nesting depth at which the framework can still generate meaningful, schema-bound SQL queries before schema coverage or constraint feasibility becomes limiting.

Table 3.4: GPS-CQ feasible nesting capacity by schema setting and target depth.

Target Depth	**Compact Schema**	**Broad Schema**
1	99.2%	99.8%
3	94.5%	96.1%
5	62.1%	88.4%
7	18.3%	74.2%
10	0.0%	41.5%
15	0.0%	12.1%

As shown in Table 3.4, the compact schema begins to degrade after Depth 3 and becomes infeasible by Depth 10 due to binding failures. The broad schema provides greater coverage, maintaining feasible generation through Depth 7 before deeper nesting substantially reduces robustness.

Chapter 4

GPS-SD: Set-level Problem Discovery in Object-Relational Domains

This chapter presents GPS-SD, an extension of the grammar-based GPS framework for discovering set-level relational problems in object-relational domains. Unlike conventional natural-language-to-SQL approaches that operate primarily on individual records and attributes, GPS-SD treats sets of entities as first-class objects and combines them with domain-defined functions, comparisons, aggregation, quantification, and nested structures. Given an object-relational schema and a vocabulary of set-level functions, the framework systematically generates structurally and semantically valid problem formulations and represents them using a formal SQL-like language that supports corresponding natural-language descriptions. The framework is evaluated in the influence and shortest-distance domains, demonstrating two complementary forms of problem discovery. In the influence domain, the generated problem space recovers formulations supported by an existing influence-maximization solver while also revealing additional structures beyond its coverage. In the shortest-distance domain, extending a first-order shortest-path operation to sets exposes a new family of set-level optimization problems. Solver methods are developed for this discovered family, together

with complexity and approximation guarantees. These results demonstrate how GPS-SD can connect generated problems to existing solution methods while also identifying new problem classes that require the development of additional solvers.

4.1 Introduction

Natural language querying has become a central goal for relational databases, since many users who need answers from structured data cannot write SQL themselves. This has motivated a large body of work on translating natural language questions into executable queries. Early systems relied on rule-based and grammar-based methods; more recent work uses learned semantic parsers and large language models. Most of these methods define queries over individual tuples and attributes in the relational schema, even when joins or aggregations are involved. Many real problems, in contrast, are naturally defined over sets of entities, not individual records.

Consider the question, "Which group of three genes has the strongest combined influence on this set of diseases?" The object being sought is a set of genes, and its quality is determined by an influence function evaluated between that set and a set of diseases. Problems with this structure arise in biomedical influence analysis, where sets of influential genes are identified through graph-based influence maximization [39]. Similar set-selection problems arise in social networks, where a fixed-size set of seed nodes is selected to maximize influence spread [40], and in facility location, where a set of facilities is selected to minimize distance-based service objectives over a set of clients [41]. In each case, the answer is a set whose quality depends on a function evaluated over or between sets, rather than a single record.

Standard SQL can encode some such problems through complex combinations of subqueries and aggregation. But conventional query frameworks do not explicitly support arbitrary

sets as query objects together with domain-defined functions evaluated over those sets. As a result, existing grammars and benchmarks largely omit this problem space. This omission is also hard to notice, since these problems do not appear as failed queries. They simply do not occur among the problems that the systems are designed to represent.

GPS-SD extends our earlier GPS-Relational framework [36] to discover set-level relational problems from a schema and a vocabulary of domain-defined functions. The schema supplies the objects and relationships in an application. The function vocabulary specifies the relationships evaluated over sets of those objects. The grammar uses this information to build set selections and conditions. It supports comparisons over aggregated or quantified values, including nested expressions. Semantic rules remove formulations that are not structurally or semantically well formed. Accepted problems are stored in a formal SQL-like representation. The framework describes these problems in natural language and maps supported natural-language problems back into the same formal form.

GPS-SD uses problem generation to expose formulations beyond those already handled by an application or solver. We examine this process in the influence and shortest-distance domains. The influence cases include formulations handled by an existing maximization solver and additional cases outside its scope. Extending shortest-path operations from individual objects to sets reveals a set-level optimization family not supported by the original solver. We develop solver support for this family and analyze its computational complexity and approximation guarantees. This example connects problem generation with the development of a new solution method.

The remainder of this paper is organized as follows. Section 4.2 reviews related work in natural language to SQL translation, problem formalization, set level querying, and solver integration. Section 4.3 introduces the set level relational framework and formal grammar. Section 4.4 presents grammar guided problem generation and semantic filtering. Section 4.5 describes the mapping between the formal representation and natural language. Section 4.6

develops solver support for constrained set to set optimization and establishes its complexity and approximation properties. Section 4.7 presents the experimental evaluation.

4.2 Related Work

Research on formalizing natural-language problems has focused mainly on converting existing problem descriptions into structured forms suitable for computation or optimization. NL4Opt [42] extracts semantic entities from linear-programming word problems and generates a meaning representation for solver input. Holy Grail 2.0 [43] uses large language models to derive constraint models from natural-language descriptions of combinatorial problems. Both approaches assume that the problem description is given in advance. Neither generates possible problem formulations from a domain schema and a vocabulary of domain-defined functions.

Grammar-based generation has mainly been used to create instances or queries within a predefined problem space. GraPPa [44] uses a synchronous context-free grammar to generate synthetic natural-language and SQL pairs for table semantic parsing, while SING-SQL [45] and OmniSQL [46] extend schema-conditioned synthesis to larger-scale query generation. Related work in combinatorial optimization has also used grammar-based generation to construct heuristics or algorithms for an established problem class, such as variable-selection heuristics for constraint satisfaction [47] or perturbative heuristics for routing and timetabling [48]. In all of these studies, the problem class is known in advance. The grammar is used within that class, not to identify new problem formulations.

Set-oriented query frameworks provide another closely related direction. Subset-query approaches allow queries whose results are subsets of tuples satisfying collective conditions [49], while the Package Query Language (PaQL) [50] supports the selection of groups of tuples

subject to global constraints and optimization objectives. More recent higher-order relational formulations provide abstractions in which candidate relations or sets can be manipulated as query objects [51]. These approaches show that set-level problems can be modeled without a fixed query structure. However, they start from a known set-level query or optimization objective. They do not explore the space of problems a schema and domain-defined functions could construct.

A separate body of work is directly relevant to the constrained set-to-set optimization family developed in Section 4.6. Facility-location problems select a set of facilities to serve a set of clients according to distance-based objectives. Classical formulations such as k-median, k-center, k-supplier, and max-min dispersion are NP-hard and admit well-studied approximation algorithms [52, 53, 54, 55, 41]. Another line of research studies shortest paths under constraints such as resource budgets, forbidden elements, and label restrictions using specialized shortest-path algorithms [56, 57]. These two lines of work generally address different settings. Facility-location methods typically assume an unconstrained distance metric, while constrained shortest-path methods focus on individual source-target pairs. The constrained set-to-set optimization family developed in Section 4.6 combines these settings by evaluating set-selection objectives over constrained path distances and determining when classical approximation guarantees continue to apply.

GPS-SD uses an object-relational schema and a vocabulary of domain-defined set-level functions to generate problem formulations for a domain. The resulting formulations can match established problem classes or reveal cases for which solver support has not yet been developed.

4.3 Set-Level Relational Problem Framework

The proposed framework represents relational problems in which the unknowns are sets of objects and the relationships of interest are evaluated between those sets. It takes an object-relational schema and a vocabulary of domain-defined functions as input. These elements are used to construct a formal SQL-like representation that identifies the sets to be found, their domains and constraints, and the functions used to evaluate them. The framework also supports the recognition of set-level problems whose results require computation over graph structure.

4.3.1 Object-Relational and Graph Setting

A set-level problem can be expressed over an object-relational schema containing object types, relationships, and attributes. When the schema admits a graph interpretation, object instances correspond to vertices and relationships correspond to edges. Relationship attributes may provide labels, weights, or other values used by the problem. We denote the resulting graph by

$$G = (V, E),$$

where V contains the object instances and E represents the relationships among them.

Following the set-variable formulation in [58], we define a set variable as a subset of V whose members belong to a declared object type. A set literal denotes an explicitly specified subset. Problems may search for one or more sets or compare a searched set with a fixed or independently defined set.

A domain-defined function evaluates a relationship, cost, or measure over sets of objects. A function has the general form $f(A, B)$, where $A, B \subseteq V$. When its dependence on the graph

must be explicit, we write $f(A, B; G)$.

Definition 4.1 (Computational Graph Problem)**.** *Let $P = (G, f)$, where $G = (V, E)$ and f is evaluated over one or more subsets of V. A necessary condition is that changing the relationship structure can change the value of f:*

$$\exists\, G_1 = (V, E_1),\ G_2 = (V, E_2),\ A, B \subseteq V :$$
$$E_1 \neq E_2, \qquad f(A, B; G_1) \neq f(A, B; G_2).$$

This condition alone is not sufficient. A direct relationship lookup may change when one edge changes, even though its evaluation does not require graph topology. The sufficient requirement is that the result of f depends on the topology connecting or relating its input sets. There must exist a function h such that

$$f(A, B; G) = h(A, B, \mathrm{Topo}_G(A, B))\,,$$

where $\mathrm{Topo}_G(A, B)$ denotes the path and connectivity structure within or between A and B. The function h cannot be evaluated from vertex attributes or one fixed relationship alone.

The next subsection describes how GPS-SD recognizes this structure using the graph interpretation and function definitions supplied for an application domain.

4.3.2 Framework Recognition of Computational Graph Problems

The preceding definition characterizes a computational graph problem independently of how it is recognized. GPS-SD performs this recognition using domain information supplied in advance. It does not infer the graph interpretation directly from the natural-language description.

A domain expert identifies which object types correspond to vertices and which relationships correspond to edges. The expert also declares the functions whose evaluation requires the topology described in Section 4.3.1. These functions form the graph-function library $\mathcal{F}_{\text{graph}}$. Each entry specifies the function name, its valid set arguments, and its domain meaning.

Given a natural-language problem, the parser matches its object references to the declared vertex types and its relationship references to the declared edge types. Function expressions are matched against the supplied function vocabulary. If these elements cannot be grounded, the graph-based status of the problem remains undetermined.

Let $P = (G, f)$ be a grounded problem. After confirming the graph interpretation of its objects and relationships, GPS-SD applies the recognition rule

$$f \in \mathcal{F}_{\text{graph}} \quad \Longrightarrow \quad \text{RecognizedGraph}(P).$$

Membership in $\mathcal{F}_{\text{graph}}$ is sufficient for recognition because the domain expert has already verified that each included function requires graph topology. A function that only reads an object attribute or one fixed relationship is not included. If a formulation contains several functions, the function defining its graph-dependent condition or objective must belong to $\mathcal{F}_{\text{graph}}$.

Once the problem has been grounded and recognized, its sets, functions, and conditions are assembled using the formal grammar presented in the next subsection.

4.3.3 Formal Grammar

Building on the object-relational setting described above, we define a restricted SQL-like grammar for representing set-level problems. Figure 4.1 presents its abstract structure. Each

Grammar Rules

terminals in quotes · nonterminals defined with ::= · comment = meaning

```
Query                ::=  StandardQuery | ExistsQuery
StandardQuery        ::=  "SELECT" SelectClause "FROM" FromClause [ "WHERE" WhereClause ] [ LimitClause ]
ExistsQuery          ::=  "SELECT" "EXISTS" "(" StandardQuery ")"

# SELECT — what the query returns
SelectClause         ::=  SelectItem { "," SelectItem }                  # one or more returned expressions
SelectItem           ::=  SetName | Func                                 # a set or a function value

# FROM — declare the set variables and their domains
FromClause           ::=  SetDecl { "," SetDecl }                        # one or more set declarations
SetDecl              ::=  "set-of-" Domain SetName                       # e.g., set-of-gene A

# WHERE — conditions
WhereClause          ::=  Condition { "AND" Condition }                   # one or more conjunctive conditions
Condition            ::=  SizeCond | SetCond | FuncCond                  # supported condition forms
SizeCond             ::=  "size(" SetName ")" CompOp IntValue            # compare a set cardinality
SetCond              ::=  SetName "=" SetLiteral | SetName "<>" SetName  # bind a set or require distinct sets
FuncCond             ::=  Func CompOp FuncRight                          # compare a function value
FuncRight            ::=  NumValue | Func | "(" StandardQuery ")"
                        |  "ANY" "(" StandardQuery ")"
                        |  "ALL" "(" StandardQuery ")"
                        |  Aggregate "(" StandardQuery ")"               # scalar, function, nested, quantified, or aggregated result

# Functions and operators
Func                 ::=  FunctionName "(" SetArg "," SetArg
                            [ "," ConstraintSpec ] ")"                  # two sets with optional constraint
SetArg               ::=  SetName | SetLiteral                           # symbolic or explicit set
ConstraintSpec       ::=  any domain-defined constraint expression       # optional function constraint
CompOp               ::=  "=" | ">" | "<" | ">=" | "<="                  # comparison operator
Aggregate            ::=  "MIN" | "MAX" | "AVG" | "SUM"                  # scalar aggregation operators

# LIMIT — restrict the number of returned results
LimitClause          ::=  "LIMIT" IntValue                               # maximum number of results to return

# Value tokens
SetLiteral           ::=  "{" NameList "}"                               # explicitly listed set
NameList             ::=  Name { "," Name }                              # one or more schema object names
SetName              ::=  Identifier                                     # label for a set variable
IntValue             ::=  positive integer literal (1, 2, 3, ...)        # must be positive
NumValue             ::=  any numeric literal                            # e.g., 3, 0.5, 10.2
Name                 ::=  any schema object name                         # e.g., BRCA1, Breast Cancer
```

Figure 4.1: Grammar rules for set-level relational queries.

problem uses the familiar `SELECT`, `FROM`, and optional `WHERE` clauses, extended to support set variables, set literals, and functions evaluated between sets.

The `Query` production has two forms: a `StandardQuery` and an `ExistsQuery`. A `StandardQuery` contains `SELECT` and `FROM` clauses, followed by optional `WHERE` and `LIMIT` clauses. An `ExistsQuery` wraps a standard query in `SELECT EXISTS(...)` and returns a Boolean value indicating whether the inner query produces at least one result.

The `SelectClause` of a standard query contains one or more `SelectItem` expressions. Each item may be a symbolic set variable or a function expression. A standard query may return sets, function values, or both. For example, `SELECT A`, `SELECT A,B`, and `SELECT`

A,influence(A,B) are permitted forms. The optional LIMIT clause bounds the number of returned results. LIMIT 1 is used when only one satisfying result is requested.

The FromClause declares the symbolic set variables available within the query. Each SetDecl associates a SetName with an entity domain supplied by the input schema, using the form set-of-Domain SetName. The clause may contain one or more declarations, including multiple variables associated with the same domain. For example, set-of-gene A, set-of-gene B, and set-of-disease C declare two gene-set variables and one disease-set variable.

The optional WhereClause expresses restrictions and comparisons over the declared sets. It contains one or more conditions connected by AND. The grammar distinguishes three forms of condition: cardinality conditions, set conditions, and function conditions.

A SizeCond restricts the cardinality of a symbolic set by comparing size(SetName) with an integer value. For example, size(A)=3 restricts A to sets containing three members.

A SetCond either binds a symbolic variable to an explicitly listed set or requires two symbolic sets to be distinct. The first form has the structure SetName = SetLiteral, such as B=D1,D2. The second has the structure SetName <> SetName, such as A<>C.

A FuncCond compares the value of a set-level function with another permitted expression. A function expression contains two required set arguments. It may also include a ConstraintSpec when an optional constraint argument is declared in the function vocabulary. Each set argument may be a symbolic set variable or an explicit set literal. For example, influence(A,B) evaluates the influence associated with the ordered pair of sets A and B. The optional argument allows a function such as shortest_distance(A,B,C) to include a domain-defined path constraint C.

The comparison operator in a function condition may be =, >, <, >=, or <=. Its right-hand side may be a numeric value, another function expression, or a nested standard query. Thus, the

grammar permits forms such as `influence(A,B)>0.5` and `influence(A,B)>influence(C,B)`.

A nested `StandardQuery` that returns a single value may be used directly. If it returns multiple values, `ANY` or `ALL` can quantify the result. An aggregate such as `MIN`, `MAX`, `AVG`, or `SUM` can instead reduce it to one scalar value. Because `StandardQuery` appears recursively within `FuncRight`, a nested query may contain another nested query. This permits multiple levels of nesting without requiring a separate rule for each depth.

4.4 Problem Generation

The formal grammar defines the space of structures that may be used to represent set-level relational problems. Problem generation instantiates these structures by selecting schema domains, declaring set variables, choosing returned expressions and conditions, and combining them into complete formal SQL-like representations. This section first presents the grammar-guided construction process and then introduces the semantic rules used to constrain and validate the generated combinations.

4.4.1 Grammar-Guided Generation

The grammar in Figure 4.1 defines the alternatives available within each clause, while Figure 4.2 illustrates how these alternatives are selected and combined. The generator does not rely on a single fixed problem template. Instead, it follows the grammar productions to determine the number and types of set variables, the expressions returned by the problem, the conditions imposed on the sets, and the form of each function comparison.

Step 1: Construct the `FROM` clause. The generator begins with the object types provided by the input schema. It selects one or more types and declares one or more symbolic set

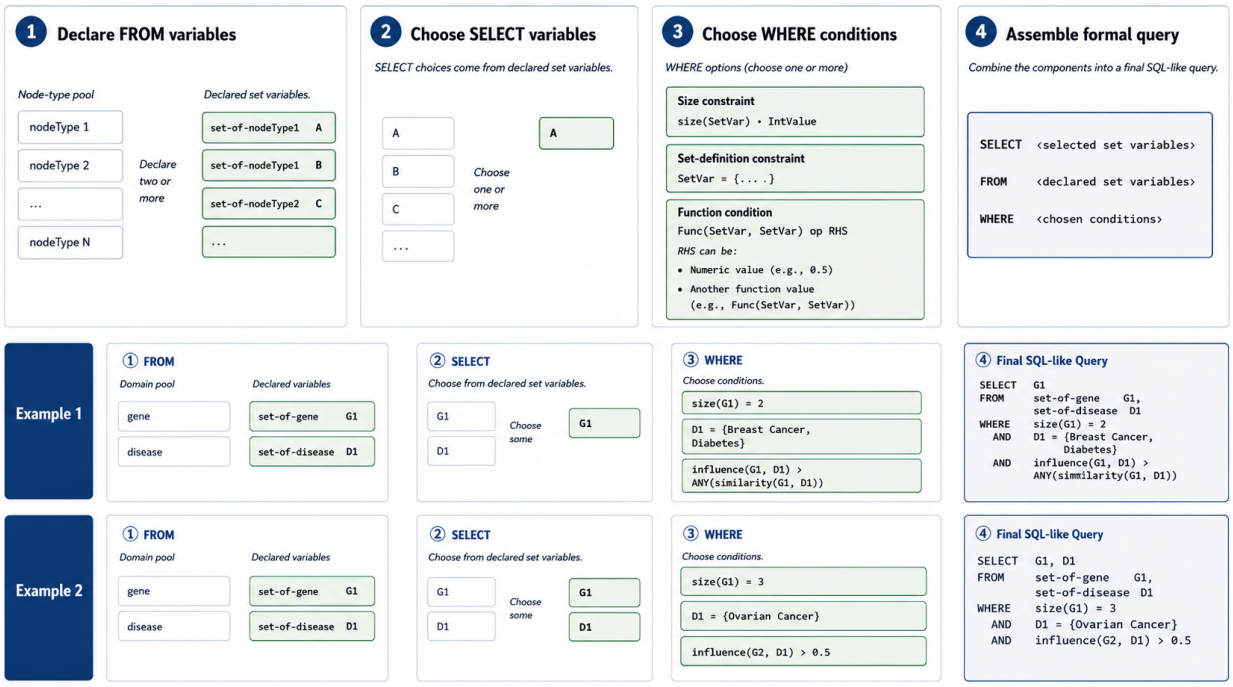


Figure 4.2: Grammar-guided construction of formal set-level problem representations.

variables for each selected type. For example, a schema containing gene and disease objects may produce the declarations `set-of-gene G1`, `set-of-gene G2`, and `set-of-disease D1`. Together, these declarations form the variable pool used to construct the remaining clauses.

Step 2: Construct the `SELECT` clause. The generator chooses one or more expressions permitted by the `SelectItem` production. An outer problem may return one or more declared set variables, such as `SELECT G1` or `SELECT G1,G2`. The grammar also permits function expressions in this clause. This form is used particularly within nested problems, where the selected function values become the input to a comparison in the enclosing problem.

Step 3: Construct the `WHERE` clause. The generator selects one or more condition forms defined by the grammar. A cardinality condition may restrict the number of objects contained in a set, as in `size(G1)=3`. A set condition may bind a symbolic variable to an explicitly specified collection, as in `D1={Diabetes,LungCancer}`. A function condition is

constructed by selecting a function from the function vocabulary, choosing its two set arguments, selecting a comparison operator, and constructing its right-hand side. The right-hand side may be a numeric value, another function expression, or a nested problem. Examples include `influence(G1,D1)>0.5` and `influence(G1,D1)>influence(G2,D1)`.

Step 4: Assemble the formal representation. The selected components are assembled as a `StandardQuery`. The generator may append a `LIMIT` clause when a bounded number of results is requested. For a decision problem, it wraps the standard query in `SELECT EXISTS(...)`. Choices made in the preceding steps produce different formal problem structures. For example, the following representation searches for a set of three genes whose influence on a fixed disease set is greater than 0.5:

```
SELECT G1
FROM set-of-gene G1, set-of-disease D1
WHERE size(G1) = 3
AND D1 = {Diabetes, LungCancer}
AND influence(G1,D1) > 0.5
```

Recursive construction of nested problems. When the right-hand side of a function condition is selected as a nested problem, the same construction procedure is applied recursively. The nested structure contains its own `SELECT`, `FROM`, and optional `WHERE` clauses. Its `SELECT` clause returns the function values required by the enclosing comparison, and the result may be used directly or through `ANY`, `ALL`, or an aggregate.

For example, the following representation searches for a three-gene set $G1$ whose influence on $D1$ is at least as large as that of every other three-gene candidate set $G2$:

Semantic rule categories applied during problem generation.

Category	Requirement
Variable and scope	Variables must be declared and referenced within a valid scope.
Function and type	Function arguments must match their required set types.
Comparison and nesting	Compared expressions must have compatible types and cardinalities.
Logical consistency	Conditions must be mutually consistent.
Other or redundant structures	Variables and outputs must contribute to the defined problem.

```
SELECT G1
FROM set-of-gene G1, set-of-disease D1
WHERE size(G1) = 3
AND D1 = {Diabetes, LungCancer}
AND influence(G1,D1) >= ALL (
  SELECT influence(G2,D1)
  FROM set-of-gene G2
  WHERE size(G2) = 3
)
```

The nested problem is generated using the same grammar as the enclosing problem. The variable $G2$ is declared in the nested `FROM` clause, its cardinality is specified in the nested `WHERE` clause, and `influence(G2,D1)` is returned for comparison with the outer function value. Because the grammar is recursive, the same process may be repeated to construct additional levels of nesting.

4.4.2 Semantic Filtering

The grammar guarantees that a generated representation follows the permitted `SELECT--FROM--WHERE` structure, but structural correctness alone does not guarantee that the represented problem is meaningful. Because the grammar productions may be combined independently, a gen-

erated representation may contain properly formed clauses while failing to define the sets returned by the problem.

For example, the following representation is structurally valid according to the grammar:

```
SELECT A
FROM set-of-gene A, set-of-gene C, set-of-disease B
WHERE size(C) = 3
AND B = {D1,D2}
AND influence(C,B) > 0.5
```

However, the selected set A does not participate in any condition and is not constrained by either its size or a function relationship. The representation therefore places conditions on C and B while returning an unrelated set A. Although every clause conforms to the grammar, the generated structure does not define a meaningful problem for A.

To prevent such combinations, we introduce a semantic filter that applies additional rules beyond the syntactic grammar. These rules were derived from recurring failure patterns observed when independently combining grammar productions. They target problems involving variable scope, disconnected outputs, incompatible function arguments, unbounded candidate sets, inconsistent constraints, and improperly formed nested comparisons. Their effectiveness in reducing semantically invalid generated problems is evaluated in section 4.7.

The semantic filter may be applied incrementally during generation by restricting each selection to semantically compatible alternatives and again after construction as a final validation step. A generated representation is retained only when it satisfies all applicable rules listed in Section 4.4.2.

Nested structures are checked using the same rules. Their locally declared variables belong to the nested scope, while references to variables declared by the enclosing problem are

permitted. The expression returned by the nested `SELECT` clause must also provide the value expected by the enclosing comparison. For example, a comparison with `ALL` requires the nested structure to return compatible function values for the candidate sets being compared.

4.5 Natural-Language Description and Parsing

The formal SQL-like representation provides a common structure for connecting generated problems with natural language. We use this connection in two directions. Each generated formal problem is mapped to a canonical natural-language description, with components that correspond directly to elements of the formal representation. A natural-language problem that follows the supported description pattern can also be parsed back into the same formal form. Both directions use a shared vocabulary that associates formal functions, comparison operators, quantifiers, and aggregates with their natural-language expressions.

4.5.1 Generating Natural-Language Descriptions

A natural-language description is constructed from the components of the formal representation rather than generated as an unrestricted sentence. The framework uses a vocabulary that associates each formal element with a corresponding linguistic expression. For example, the operator `>` may be expressed as "greater than," while a comparison of the form `>= ALL` over the candidate sets may be expressed as "maximum among all candidate sets." Domain-defined functions are treated in the same way: the function vocabulary supplies the phrase used to describe the relationship computed by each function.

The descriptions follow a canonical pattern that exposes the main components of a set-level problem:

Mapping from the formal representation to the canonical natural-language pattern.

Formal component	**NL component**	**Role**
`SELECT` $A_1,\ldots,A_m$	`<m>`	Number of sets requested
`set-of-Domain` A_i	`<object type>`	Type of the requested sets
`size(`A_i`)=k`	`<k>`	Cardinality of each requested set
`B={...}`	`<target set>`	Set on which the function is evaluated
`f(`A_i`,B)`	`<function>`	Set-level relationship being evaluated
Operator and quantifier	`<comparison>`	Relationship between the two function values
Numeric, function, or nested expression	`<RHS>`	Value or candidate expression used for comparison

```
Find <m> set(s) of <k> <object type> whose <function> on <target set> is <comparison>
                                        <RHS>.
```

Each slot in this pattern is obtained directly from a component of the formal representation, as summarized in Section 4.5.1.

For example, the formal components `SELECT A`, `size(A)=3`, `B={D1,D2}`, and `influence(A,B)>0.5` instantiate the pattern as:

Find one set of three genes whose influence on {D1,D2} is greater than 0.5.

Nested representations are translated compositionally. They do not require a separate template for every possible nesting structure. The nested representation is first translated using the same mapping, and its description is then incorporated into the right-hand side of the enclosing condition. For example, a condition comparing `influence(A,B)` with `ALL` influence values produced by candidate sets of the same size can be described as "whose influence on B is maximum among all sets of three genes." This recursive construction allows the same description mechanism to be applied at arbitrary nesting depths supported by the grammar.

The purpose of the canonical description is to preserve the information represented formally:

the requested sets, their cardinalities and types, the function being evaluated, the target of that function, and the comparison that defines the problem. Because this direct composition may produce language that is correct but less natural stylistically, an LLM can optionally be used after the mapping step to paraphrase the canonical description. In this case, the LLM changes only the wording; the meaning and formal components of the problem remain fixed.

4.5.2 Parsing Natural-Language Problems

The reverse direction begins with a natural-language problem, not a generated formal representation. The parser grounds the terms in the question to the predefined objects and relationships in the schema; if these cannot be matched, the problem is not treated as a supported graph-based problem.

The parser recognizes the canonical pattern defined in Section 4.5.1 and identifies the values occupying its semantic slots. It also determines whether the problem requires existence testing or a bound on the number of returned results. These elements are then mapped to the corresponding grammar components and assembled into a formal SQL-like representation.

Consider the canonical structure:

```
Find <m> set(s) of <k> <object type> whose <function> on <target set> is <comparison>
                                        <RHS>.
```

The parser first extracts `<m>`, `<k>`, and `<object type>`. The value of m determines the searched set variables $A_1, \ldots, A_m$ placed in the `SELECT` clause, while the object type determines their `FROM` declarations and k produces the corresponding cardinality conditions. The `<target set>` produces a target-set variable and its set binding. The function and comparison phrases are matched against the shared vocabulary to obtain the corresponding formal

function and operator, and the right-hand side determines the final comparison expression.

For example, the parser decomposes

> *Find one set of three genes whose influence on {D1,D2} is greater than 0.5.*

into the requested-set count $m = 1$, cardinality $k = 3$, object type *gene*, function *influence*, target set $\{D1, D2\}$, comparison *greater than*, and numeric right-hand side 0.5. These components are mapped to:

```
SELECT A
FROM set-of-gene A, set-of-disease B
WHERE size(A) = 3
AND B = {D1,D2}
AND influence(A,B) > 0.5
```

When the right-hand side expresses another candidate-set problem, the parser applies the same procedure recursively and constructs the corresponding nested representation. Thus, the parsing process mirrors the compositional description process. It does not require a distinct parsing rule for each nesting depth.

The deterministic parser expects the semantic components of the problem to occur in a recognizable form. A natural-language problem may express the same meaning using wording that does not directly match the canonical vocabulary or pattern. In such cases, an LLM can be used as a normalization step to rephrase the input into the supported canonical form. The normalized sentence is then processed by the parser in the same manner as a directly recognized sentence. The LLM does not construct the formal representation itself. It only bridges linguistic variation between unrestricted natural language and the pattern the parser recognizes.

4.6 Constrained Set-to-Set Optimization

The Shortest-Distance Schema in the following section illustrates a case in which the grammar of Section 4.3 and the semantic rules of Section 4.4 generate second-order problems involving set-level functions such as `shortest_distance(A,B)`, while the reference solver provides only shortest-path computation between individual nodes. Here, we consider a class of such problems in which a subset of one entity set is selected with respect to another using shortest-path distances subject to path constraints. We characterize the computational properties of these problems and identify conditions under which approximation guarantees can be obtained.

4.6.1 Problem Definition

Following the object-relational setting of Section 4.3.1, object instances correspond to nodes and relationships to weighted edges; attributes may further assign categorical labels to both.

Definition 4.2 (Colored Weighted Graph)**.** *A* ***colored weighted graph*** *is a tuple* $G = (V, E, w, c_V, c_E, \Sigma_V, \Sigma_E)$*, where* V *is a set of vertices,* $E \subseteq V \times V$ *a set of edges,* $w : E \to \mathbb{R}^+$ *assigns non-negative edge weights, and* $c_V : V \to \Sigma_V$*,* $c_E : E \to \Sigma_E$ *assign labels from finite alphabets. In an object-relational schema this arises from an object table (nodes with type attributes) and a relationship table (edges with weight and type attributes), e.g., road segments with length and road-class columns.*

A **path** $\pi = \langle v_0, \ldots, v_\ell \rangle$ is a sequence of vertices connected by edges, with weight $w(\pi) = \sum_{i=0}^{\ell-1} w(v_i, v_{i+1})$.

Constraint categories. We classify path constraints into five categories by their computational structure. **Category A (hard filters)** forbids certain nodes or edges, or restricts

paths to edges of allowed types; validity is memoryless. **Category B (finite memory)** contains constraints expressible by finite automata over the path, such as the requirement to visit at least one node of a designated waypoint set. **Category C (budgets)** bounds accumulated resources, such as the number of hops or the total cost. **Category D (global precomputed)** filters by precomputed global properties (centrality, community membership), reducing to a Category-A-style subgraph restriction after preprocessing. **Category E (path-dependent)** contains constraints whose validity depends on the full path history, such as simple-path (no-revisit).

Definition 4.3 (Constrained Distance)**.** *Given a constraint set $\mathcal{C}$, let $\Pi_{\mathcal{C}}(u, v)$ denote the set of u–v paths satisfying $\mathcal{C}$. The* ***constrained distance*** *is $d_{\mathcal{C}}(u, v) = \min_{\pi \in \Pi_{\mathcal{C}}(u,v)} w(\pi)$, with $d_{\mathcal{C}}(u, v) = \infty$ if $\Pi_{\mathcal{C}}(u, v) = \emptyset$, and $d_{\mathcal{C}}(u, u) = 0$.*

Definition 4.4 (Constrained Set-to-Set Optimization)**.** *Given a colored weighted graph G, a candidate set $S \subseteq V$, a target set $T \subseteq V$, a budget k, and a constraint set $\mathcal{C}$, select $X \subseteq S$ with $|X| = k$ optimizing one of:*

$$\textbf{\textit{k-median:}} \min_{X} \frac{1}{|T|} \sum_{v \in T} \min_{u \in X} d_{\mathcal{C}}(u, v) \quad \textit{(average service distance);}$$

$$\textbf{\textit{k-center:}} \min_{X} \max_{v \in T} \min_{u \in X} d_{\mathcal{C}}(u, v) \quad \textit{(worst-case service distance);}$$

$$\textbf{\textit{k-dispersion:}} \max_{X} \min_{u,v \in X,\, u \neq v} d_{\mathcal{C}}(u, v) \quad \textit{(diversity; no target set).}$$

Because S and T are distinct sets, the min–max objective is the k-supplier *problem; classical k-center results apply only to the special case $S = T$.*

In the vocabulary of Section 4.3, these objectives can be represented by set-level functions built on the pairwise operation $d_{\mathcal{C}}$. Average distance corresponds to k-median, worst-case distance to k-center, and minimum pairwise distance $f(A, A)$ to k-dispersion. Different choices of $\mathcal{C}$ define constrained versions of these objectives that remain expressible by the

grammar. Combining the three objectives with the five constraint categories, together with structural variants such as within-set diameter bounds, produces the second-order problem classes considered here.

4.6.2 An Example Problem

Consider a municipal schema with location objects (candidate shelter sites and communities), road relationships carrying length and road-class attributes, and a flood-zone attribute marking hazardous locations. A planner asks, in natural language:

> *"Choose 3 emergency-shelter sites from the candidate sites so that the average evacuation distance from the communities is minimized, where evacuation routes must not pass through flood-zone areas."*

The problem is second-order because the decision variable is a set of shelter locations and the objective is defined by constrained shortest-path distances between the selected sites and the communities. In the function vocabulary, `shortest_distance(A, B, avoid` F_V`)` returns the constrained distances from members of B to their nearest member of A, with paths through flood-zone nodes excluded. The corresponding minimization can be expressed using `<= ALL` over candidate sets:

```
SELECT A1
FROM set-of-location A1, set-of-location B
WHERE size(A1) = 3
AND AVG(shortest_distance(A1, B, avoid F_V)) <= ALL (
  SELECT AVG(shortest_distance(A2, B, avoid F_V))
  FROM set-of-location A2
  WHERE size(A2) = 3
)
```

Formally, this is the k-median objective of Definition 4.4 with the Category A constraint "avoid F_V", where S is the candidate site set, T is the community set, $k = 3$, and F_V the flood-zone nodes:

$$\min_{X \subseteq S \setminus F_V,\ |X|=3} \ \frac{1}{|T|} \sum_{v \in T} \min_{u \in X} d_{\mathcal{C}}(u, v), \qquad \mathcal{C} = \{\text{avoid } F_V\}.$$

Other instances covered by the formulation include:

- Content-delivery server placement minimizing worst-case latency over routes with at most h hops (k-center with a Category C hop limit);
- Diverse backup-server selection avoiding compromised links (k-dispersion with Category A forbidden edges);
- Warehouse placement restricted to highway routing (k-median with a Category D edge-type filter);
- Service-center placement where all routes must pass a security checkpoint (k-center with a Category B waypoint).

4.6.3 Complexity and Approximation Guarantees

We now characterize the computational complexity and approximation guarantees of these problems. Throughout, G is undirected and edge weights are non-negative.

Theorem 4.1 (NP-Hardness by Restriction)**.** *Constrained k-median, constrained k-center (k-supplier), and constrained k-dispersion are NP-hard for every constraint class that contains the trivial constraint set (e.g., no forbidden elements, hop limit $h \geq n$, all edge types allowed).*

Proof. With trivial constraints, $d_{\mathcal{C}}$ coincides with the ordinary shortest-path metric and the problems specialize to their classical unconstrained versions, each NP-hard: k-median [52, 59], k-center and k-supplier [60, 61], and max–min dispersion [55]. A polynomial algorithm for the constrained generalization would therefore solve an NP-hard problem. □

Theorem 4.2 (Constrained Distance Computation)**.** *Suppose each constraint in $\mathcal{C}$ is* state-Markovian*: representable by a finite state space Q with an initial state, a per-edge transition function, and accepting states, such that a path is valid iff its state trajectory is defined and ends in an accepting state. Then all constrained distances from a source are computable by a label-setting (Dijkstra) algorithm on the product space $V \times Q$ in $O(|Q|\, m + |Q|\, n \log(|Q|\, n))$ time. In particular, Categories A and D have $|Q| = 1$; a Category B automaton contributes its state count; a hop limit h gives $|Q| = h+1$; an integer budget B gives $|Q| = B+1$ (pseudo-polynomial; real-valued budgets require Pareto-dominance label-correcting methods [56, 57]). For Category E the induced state space may be exponential (e.g., 2^n for simple paths), and no polynomial bound is claimed.*

Proof. Under the state-Markovian assumption, the feasibility and valid extensions of a partial path depend only on its endpoint and constraint state, so Dijkstra's invariant applies verbatim to the product graph with $|Q|\, n$ vertices and at most $|Q|\, m$ arcs; the bound is Dijk-

stra's with a binary heap. The search must continue past a target reached in a non-accepting state (e.g., a waypoint not yet visited) and report the first extraction of the target in an accepting state. □

The Structural Dichotomy: Concatenation-Closure

Whether classical approximation guarantees extend to constrained distances depends on concatenation-closure of valid paths rather than subpath-closure.

Definition 4.5 (Subpath-Closure and Concatenation-Closure)**.** *$\mathcal{C}$ is **subpath-closed** if every contiguous subpath of a valid path is valid. $\mathcal{C}$ is **concatenation-closed** if for all u, v, w, every $\pi_1 \in \Pi_{\mathcal{C}}(u, v)$ and $\pi_2 \in \Pi_{\mathcal{C}}(v, w)$ satisfy $\pi_1 \circ \pi_2 \in \Pi_{\mathcal{C}}(u, w)$.*

The two properties are incomparable: hop limits are subpath-closed but not concatenation-closed, while waypoint requirements are concatenation-closed but not subpath-closed.

Lemma 4.1 (Triangle Inequality)**.** *If $\mathcal{C}$ is concatenation-closed, then $d_{\mathcal{C}}(u, w) \leq d_{\mathcal{C}}(u, v) + d_{\mathcal{C}}(v, w)$ for all $u, v, w \in V$. If moreover G is undirected and validity is invariant under path reversal (as for all categories considered here), $d_{\mathcal{C}}$ is a symmetric extended pseudometric.*

Proof. If either right-hand term is infinite the claim is trivial. Otherwise take shortest valid paths π_{uv} and π_{vw} (minimizers exist for state-Markovian constraints, since the search space of Theorem 4.2 is finite with non-negative weights). By concatenation-closure $\pi_{uv} \circ \pi_{vw} \in \Pi_{\mathcal{C}}(u, w)$, so $d_{\mathcal{C}}(u, w) \leq w(\pi_{uv}) + w(\pi_{vw}) = d_{\mathcal{C}}(u, v) + d_{\mathcal{C}}(v, w)$. Symmetry follows by reversing shortest valid paths. □

Proposition 4.1 (Classification)**.** *(i) Categories A and D are concatenation-closed: they restrict valid paths to a fixed subgraph $G' \subseteq G$, so $d_{\mathcal{C}}$ is the shortest-path metric of G'. (ii) The must-visit waypoint constraint of Category B is concatenation-closed but not subpath-closed; general regular-language constraints need not be. (iii) Category C is subpath-closed*

but not *concatenation-closed, and may violate the triangle inequality (Example 4.1). (iv) Category E is in general neither.*

Proof. (i) Validity is a per-element condition, preserved by both gluing and cutting. (ii) If π_1 visits a required node, so does $\pi_1 \circ \pi_2$; a subpath avoiding it is invalid. (iii) Subpaths of an h-hop path have at most h hops, but concatenating two h-hop paths may use $2h$. (iv) Concatenating two simple paths may repeat vertices. □

Example 4.1 (Category C Violates the Triangle Inequality)**.** *Let G be the path $v_0 - v_1 - v_2 - v_3 - v_4$ with unit weights plus the edge $\{v_0, v_4\}$ of weight $W > 4$, under hop limit $h = 2$. Then $d_{\mathcal{C}}(v_0, v_2) = d_{\mathcal{C}}(v_2, v_4) = 2$ but $d_{\mathcal{C}}(v_0, v_4) = W > 4$ (and $= \infty$ without the extra edge). Hop-constrained distances are therefore not metrics, and the guarantees below do* not *apply to Category C. The experiments in Section 4.7.4 further show that algorithms relying on the metric property can perform poorly for Category C constraints.*

Approximation Guarantees

For the remainder of this subsection, $\mathcal{C}$ is concatenation-closed and reversal-invariant, and instances are feasible (all distances appearing in the objective are finite for at least one size-k solution).

Theorem 4.3 (Constrained k-Supplier: Greedy 3-Approximation)**.** *Consider the algorithm: pick an arbitrary client $t_1 \in T$ and set $X_1 = \{x_1\}$ with $x_1 = \arg\min_{x \in S} d_{\mathcal{C}}(x, t_1)$; for $i = 2, \ldots, k$, let t_i be the client farthest from X_{i-1} and add $x_i = \arg\min_{x \in S} d_{\mathcal{C}}(x, t_i)$. The output $X = X_k$ satisfies $\max_{t \in T} \min_{x \in X} d_{\mathcal{C}}(x, t) \leq 3\, r^*$, where r^* is the optimal radius over all $X^* \subseteq S$, $|X^*| = k$.*

Proof. Let r be the greedy radius, attained by client $\hat{v}$, and for each client t let $f(t) \in X^*$ satisfy $d_{\mathcal{C}}(f(t), t) \leq r^*$. Since $f(t_i) \in S$ is available to the $\arg\min$ defining x_i, we have

$d_{\mathcal{C}}(x_i, t_i) \le r^*$ for every i; in particular, if $\hat{v} = t_i$ for some i then $r \le r^*$ and we are done, so assume otherwise. For $i < j$, when t_j was selected the facility x_i was already present, and by the farthest-client rule and monotonicity of coverage, $d_{\mathcal{C}}(x_i, t_j) \ge \min_{x \in X_{j-1}} d_{\mathcal{C}}(x, t_j) \ge \min_{x \in X_{j-1}} d_{\mathcal{C}}(x, \hat{v}) \ge r$; the triangle inequality then gives $d_{\mathcal{C}}(t_i, t_j) \ge r - r^*$, and likewise $d_{\mathcal{C}}(t_i, \hat{v}) \ge r - r^*$. Thus the $k+1$ clients $t_1, \ldots, t_k, \hat{v}$ are pairwise at distance $\ge r - r^*$, yet they are served by only k optimal facilities, so two of them lie within r^* of the same facility and hence within $2r^*$ of each other. Therefore $r - r^* \le 2r^*$, i.e., $r \le 3r^*$. □

The client-based initialization is essential: seeding instead with the candidate minimizing the maximum distance to all clients (a 1-supplier heuristic) admits instances with unbounded approximation ratio. When $S = T = V$, placing each new center at the farthest client itself recovers the classical factor 2 [53]; in the set-to-set setting, factor 3 is best possible unless P=NP [54].

Theorem 4.4 (Constrained k-Median: Local Search)**.** *A solution X that is locally optimal under single swaps satisfies $\phi(X) \le 5\,\phi(X^*)$, where ϕ is the k-median objective; local optimality under p-swaps yields $\phi(X) \le (3+2/p)\,\phi(X^*)$. With the standard ε-improvement stopping rule, polynomial running time is guaranteed at the cost of $(5+\varepsilon)$, resp. $(3+2/p+\varepsilon)$ [41].*

Proof. The charging argument of Arya et al. [41] uses only the availability of the optimal facilities in S, symmetry, and the triangle inequality, and is otherwise oblivious to how distances arise; by Lemma 4.1, $d_{\mathcal{C}}$ provides all three. The $1/|T|$ normalization cancels in the ratio. □

Theorem 4.5 (Constrained k-Dispersion: Greedy 2-Approximation)**.** *Farthest-first greedy (arbitrary first node; then repeatedly add the candidate maximizing the minimum distance to the selected set) returns X whose dispersion is at least $\delta^*/2$, where δ^* is the optimal dispersion.*

Proof. We claim that at every iteration $i < k$, some $w^* \in X^*$ has $\min_{v \in X_i} d_{\mathcal{C}}(w^*, v) \geq \delta^*/2$. Otherwise every $w \in X^*$ has a witness $v(w) \in X_i$ with $d_{\mathcal{C}}(w, v(w)) < \delta^*/2$; since $|X^*| = k > i$, two distinct w, w' share a witness v, and the triangle inequality gives $d_{\mathcal{C}}(w, w') < \delta^*$, contradicting the optimal dispersion. By the greedy rule, each inserted node is at distance $\geq \delta^*/2$ from all previously selected nodes; hence for any pair in X, the later-inserted member was at distance $\geq \delta^*/2$ from the earlier one, and the dispersion of X is at least $\delta^*/2$. □

[Optimality of the Guarantees] Unless P=NP, constrained k-supplier admits no $(3 - \varepsilon)$-approximation [54] and constrained k-dispersion no $(2 - \varepsilon)$-approximation [55]; constrained k-median admits no $(1 + 2/e - \varepsilon)$-approximation unless NP $\subseteq$ DTIME$(n^{O(\log \log n)})$ [62, 63]. All bounds follow by restriction as in Theorem 4.1. Thus Theorems 4.3 and 4.5 are optimal and Theorem 4.4 is within a constant factor of optimal.

Consequently, constant-factor approximation guarantees apply when the constrained distance is concatenation-closed, including Categories A and D and the waypoint variant of Category B. Constrained distances for Categories A–D are polynomial-time computable under the assumptions of Theorem 4.2. For Category C, the loss of the triangle inequality prevents the approximation guarantees above from carrying over, while Category E may require exponential state.

4.7 Experimental Evaluation

We evaluate the proposed framework on two different object-relational application domains: a Gene–Disease Influence Domain and a Location–Road Network Domain. The experiments address two questions. First, we examine whether the semantic rules are necessary for restricting grammar-generated structures to meaningful problems and whether their effect is consistent across different domains. Second, after semantic validation, we examine whether

the generated problem space corresponds to known problem classes with existing solution methods or exposes problem classes that require new solver development.

4.7.1 Experimental Setup

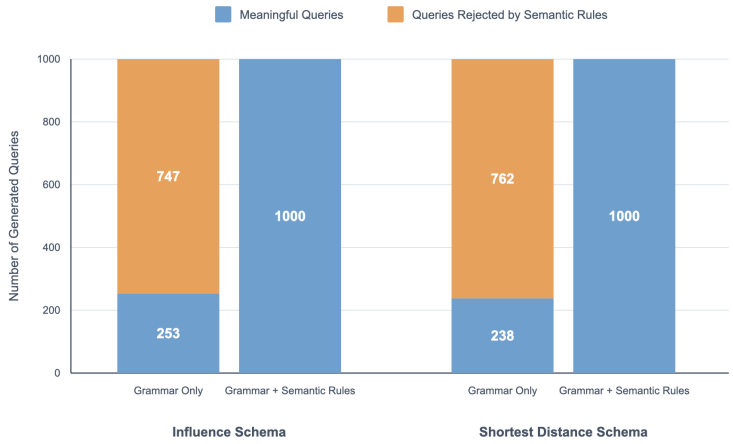
The first experiment uses a Gene–Disease Influence Domain in which genes and diseases are represented as objects connected through weighted relationships. The function vocabulary includes `influence(A,B)`, where A is a set of genes and B is a set of diseases. The function returns a numeric value representing the collective influence of the genes in A on the diseases in B. Using this schema and function vocabulary, the framework generates different set-level influence problem formulations.

The second experiment uses a Location–Road Network Domain in which locations are represented as nodes and road connections as weighted relationships. The function vocabulary includes `shortest_distance(A,B)`, where A and B are sets of locations. The function returns a numeric value representing a distance relationship between the two sets based on shortest-path distances between their members. Using this schema and function vocabulary, the framework generates different set-level shortest-distance problem formulations.

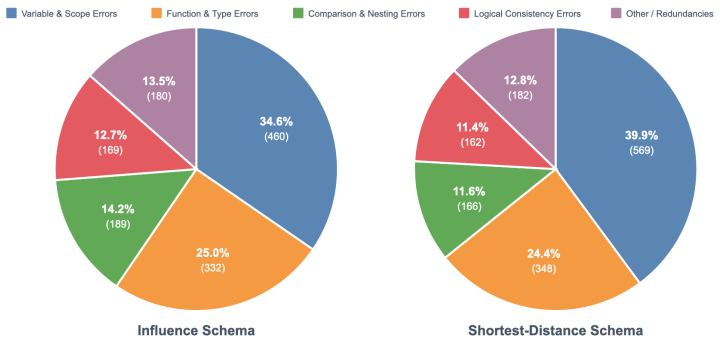
For each domain, we generated 1,000 problems under two configurations. In the *grammar-only* configuration, problems are generated directly from the grammar without enforcing the semantic rules. In the *grammar + semantic rules* configuration, generation is restricted by the semantic requirements described in Section 4.4.2. This comparison isolates the contribution of the semantic component while keeping the underlying grammar and domain vocabulary unchanged.

4.7.2 Effect of Semantic Rules

Figure 4.3a shows the overall effect of the semantic rules. For the Gene–Disease Influence Domain, 253 of the 1,000 grammar-only problems are meaningful, while 747 contain at least one semantic issue. The Location–Road Network Domain produces a similar result: 238 problems are meaningful and 762 are rejected by the semantic rules. In contrast, generation using the grammar together with the semantic rules produces 1,000 meaningful problems for each domain.



(a) Effect of semantic rules on generated problems.



(b) Distribution of semantic error occurrences in grammar-only generation.

Figure 4.3: Semantic evaluation for the Gene–Disease Influence and Location–Road Distance domains. The two domains exhibit similar grammar-only validity rates and similar distributions of semantic error categories.

The similarity between the two grammar-only results is notable. Although the two domains use different schemas and set-level functions, only about one quarter of the unrestricted grammar-generated problems are meaningful in either experiment. This suggests that the need for semantic restrictions is not specific to a particular domain or function, but arises more generally from the possible combinations of set variables, functions, comparisons, constraints, and nested structures permitted by the grammar. To examine the rejected problems in more detail, we group the detected semantic issues into five categories: variable and scope errors, function and type errors, comparison and nesting errors, logical consistency errors, and other or redundant structures. Their distributions are shown in Figure 4.3b. Because a generated problem may contain more than one semantic issue, the category counts represent occurrences of semantic errors rather than mutually exclusive groups of problems.

The distributions are again similar across the two domains. Variable and scope errors form the largest category in both experiments, accounting for 34.6% of the detected issues in the Gene–Disease Influence Domain and 39.9% in the Location–Road Network Domain. Function and type errors form the second largest category in both cases, accounting for approximately one quarter of the detected issues. The remaining categories also occur at broadly comparable rates. The two domains also show similar numbers of rejected grammar-only problems. Together, these results indicate that the semantic rules address recurring structural issues across different application domains, not issues specific to just one.

4.7.3 Problem Discovery Screening

We next examine whether the generated problem formulations correspond to computational problems studied in existing research and whether available solution methods can be reused or extended to support them. Because the generation process can produce a large problem space containing many combinations of functions, constraints, comparisons, quantifiers, and

nesting structures, we first organize this space before comparing it with existing research. The screening process consists of four steps: normalization, structural matching, grouping, and problem-type assignment.

Step 1: Normalization. Normalization is applied directly to the generated SQL-like representation. The purpose is to remove differences that correspond only to particular instances while preserving the structure that may determine the computational problem. Numerical constants are replaced by scalar constants, explicit set literals are replaced by defined sets, and set-variable names are normalized while their roles within the query are preserved. Functions, comparison operators, quantifiers such as `ALL` and `ANY`, the number and relationship of selected sets, and the nesting structure are retained.

For example, the expressions

```
size(A)=3,    B={D1,D2},    influence(A,B)>0.5
```

are normalized as

```
size(role1)=scalar constant,
role2=defined set,
influence(role1,role2)>scalar constant,
```

respectively. Thus, different cardinalities, target sets, or threshold values do not by themselves create different normalized structures.

Step 2: Structural Matching. The normalized representations are then compared using structural matching criteria. Two problems are considered structurally equivalent when their corresponding `SELECT` and `WHERE` structures match. The `SELECT` comparison considers the number and structure of the returned sets. The `WHERE` comparison considers the number

of conditions, the role or function involved in each condition, the comparison operator, and whether the comparison target is a scalar, defined set, function value, or nested query. Nested queries are compared recursively using the same criteria.

Step 3: Grouping. Problems that satisfy the structural matching criteria are placed in the same group. For example, two queries may use different set cardinalities and different target sets but reduce to the same normalized form:

```
SELECT role1
FROM set-of-X role1, set-of-Y role2
WHERE size(role1) = scalar constant
AND role2 = defined set
AND influence(role1,role2) >= ALL (
  SELECT influence(role3,role2)
  FROM set-of-X role3
  WHERE size(role3) = scalar constant
)
```

Such queries represent different instances of the same structural problem and are therefore analyzed as one group. This step reduces the generated problem space to its structurally distinct forms rather than treating every generated instance independently.

Step 4: Problem-Type Assignment. Each structural group is then assigned a computational problem type according to its overall semantics. The primary types considered in this work are optimization, search or feasibility, decision or existence, and enumeration. Optimization problems compare candidate sets to determine a maximum or minimum. Search or feasibility problems request a set satisfying specified conditions. Decision or existence problems ask whether such a set exists, while enumeration problems return all sets satisfying the conditions. A structural group may also combine these types, for example by placing

an optimization condition inside an existence query.

LLM-Assisted Research Screening and Expert Verification. Research screening is performed after normalization, grouping, and problem-type assignment. Each structural group is screened once instead of reviewing every generated instance. The LLM receives the representative normalized SQL-like structure together with its abstract natural-language form.

The natural-language form removes instance-specific values while retaining the function semantics and computational relationships. For example, the normalized influence-maximization structure is described as

> Find a set of k objects whose influence on a target set is maximum among all comparable candidate sets.

The LLM uses both representations to retrieve relevant terminology and potentially related publications. It does not determine whether a formulation is known or novel. Each suggested match is checked against the corresponding publication. A structural group is classified as known only when the published problem definition covers the normalized structure. Groups without a verified match are retained as candidate discoveries for further literature review. The absence of a match is not treated as proof of novelty.

Among the 1,000 semantically valid problems generated for the Gene–Disease Influence Domain, 238 were classified as optimization problems. The screening process found verified matches for 15 of them: 10 maximization problems and 5 minimization problems. The other 223 optimization problems had no verified match. The remaining generated problems consisted of 244 search or feasibility problems, 245 decision or existence problems, and 273 enumeration problems. No verified matches were found for these groups.

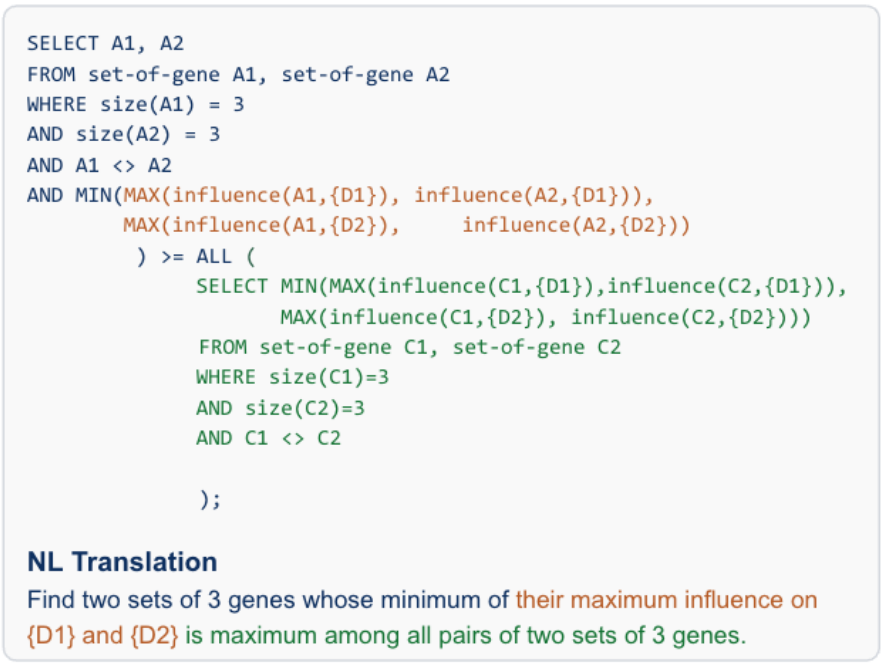


Figure 4.4: Candidate discovery: a generated two-set problem with nested min–max influence optimization.

The maximization matches include the influence-maximization structure studied by Wang et al. [39]. A fixed-size set is selected to maximize its influence on a target set. The same structure also appears in social-network influence maximization [40].

Figure 4.4 presents a generated two-set problem with nested min–max influence optimization. The formulation combines several structural operations within one problem and did not have a verified direct match after LLM-assisted screening and manual review. It is therefore retained as a candidate discovery.

The Location–Road Network Domain produced 256 optimization problems, 254 search or feasibility problems, 234 decision or existence problems, and 256 enumeration problems. Among the optimization problems, 79 had verified matches with existing work: 3 maximization problems and 76 minimization problems. The remaining 177 optimization problems had no verified match. No matches were verified for the other problem types.

Screening the unmatched shortest-distance groups exposed the constrained set-to-set opti-

mization family developed in Section 4.6. This family was selected for further study, and its solver support is evaluated in section 4.7.4. These results show that GPS-SD can recover established computational problems while isolating structures that may require new solution methods.

4.7.4 Solving Constrained Set-to-Set Optimization Problems

The analysis in Section 4.7.3 identified a family of constrained set-to-set optimization problems among the generated shortest-distance problems. We evaluate the solver of Section 4.6 on this family using the constrained-distance procedure of Theorem 4.2 and the approximation algorithms of Theorems 4.3 to 4.5.

We conduct three experiments: approximation quality relative to exhaustive optima, runtime scaling of exact and approximate solution methods, and the cost of constrained-distance computation across constraint categories. All experiments use random colored weighted graphs (about three edges per node, weights uniform in $[1, 10]$) whose node sets are partitioned into candidates S, targets T, and free routing nodes on which routing constraints act.

Optimality across instance size and constraint type. The first study runs 675 instances: five instance sizes ($n = 12$ to 28, with $|S|$ from 4 to 11 and k from 2 to 4) $\times$ nine constraint configurations $\times$ three objectives $\times$ five seeds. Exact optima are obtained by enumerating all $\binom{|S|}{k}$ facility subsets over the same precomputed constrained distances used by the approximation algorithms; 669 of the 675 instances are feasible (the six exceptions arise when a constraint disconnects the network, affecting all solvers equally). Table 4.1 reports the mean and worst approximation ratio for each configuration and objective, pooled over sizes and seeds.

For concatenation-closed configurations, all observed approximation ratios remain within

Table 4.1: Approximation ratio vs. exact optimum by constraint configuration (mean / worst over up to 25 runs per cell, $n = 12$–28). Guarantees 5 for k-median, 3 for k-supplier, ≥ 0.5 for k-dispersion—apply only to concatenation-closed (CC) configurations. †Additionally, in 3 of 25 runs greedy dispersion returned an infeasible set although a feasible optimum existed.

Configuration	Cat.	CC	k-median	k-center	k-dispersion
Unconstrained	–	Yes	1.00 / 1.00	1.00 / 1.02	0.97 / 0.85
Forbidden nodes	A	Yes	1.00 / 1.00	1.02 / 1.20	0.95 / 0.64
Forbidden edges (20%)	A	Yes	1.00 / 1.02	1.02 / 1.33	0.96 / 0.74
Edge types (3 of 4)	D	Yes	1.00 / 1.03	1.03 / 1.42	0.96 / 0.65
Waypoint (must-visit)	B	Yes	1.00 / 1.00	1.00 / 1.00	1.00 / 1.00
Hop limit $h = 3$	C	No	1.00 / 1.00	1.00 / 1.02	0.96 / 0.64
Hop limit $h = 2$	C	No	1.00 / 1.00	1.01 / 1.20	0.88 / 0.50
Cost budget (75th pct.)	C	No	1.00 / 1.00	1.00 / 1.02	0.87 / **0.19**†
Forbidden + hop $h = 4$	A+C	No	1.00 / 1.00	1.02 / 1.20	0.96 / 0.74

their theoretical guarantee: single-swap local search stays within 1.03 of the k-median optimum (bound: 5), the client-seeded greedy within 1.42 of the k-supplier optimum (bound: 3), and greedy dispersion within 0.64 of the dispersion optimum (bound: 0.5). The quality does not systematically degrade with instance size; the worst ratios occur at $n = 20$–24, rather than at the largest tested size. The Category C experiments also illustrate the failure mode described in Example 4.1. Under the cost-budget constraint, greedy dispersion attains a ratio of 0.19, below the classical 1/2 guarantee, and returns an infeasible set in three additional runs despite the existence of a feasible optimum. For the hop-limit configuration with $h = 2$, the minimum observed ratio is 0.50. These failures are observed only in the tested non-concatenation-closed configurations. The edge-type case gives the largest k-supplier ratio, 1.42, below the factor-3 bound in Theorem 4.3.

The speedup of approximation. The second study fixes a Category A configuration (forbidden routing nodes, where guarantees apply) and grows the candidate pool and budget from $(|S|, k) = (10, 3)$ to $(26, 6)$ on graphs of $n = |S| + 25$ nodes, with $|T| = 15$, comparing the solve-phase runtime of exact enumeration and the approximation algorithms over the same precomputed distances (six runs per tier: three objectives $\times$ two seeds).

Table 4.2: Exact vs. approximate solve-phase runtime and quality as the candidate pool grows (means over 6 runs per tier; ratio range spans all objectives).

$\|S\|$	k	Subsets	Exact (s)	Approx. (ms)	Speedup	Ratio range
10	3	120	0.003	0.55	6×	[1.00, 1.22]
14	4	1 001	0.024	1.15	21×	[0.94, 1.32]
18	5	8 568	0.244	2.83	86×	[0.96, 1.00]
22	5	26 334	0.769	5.22	147×	[1.00, 1.00]
26	6	230 230	7.683	6.71	1144×	[0.91, 1.10]

Exact enumeration grows rapidly with $\binom{|S|}{k}$: runtime increases from 0.003 seconds for 120 subsets to 7.683 seconds for 230,230 subsets. At the measured enumeration throughput of approximately 3×10^4 subsets/s, enumerating $\binom{50}{10} \approx 1.03 \times 10^{10}$ subsets would take approximately four days, excluding distance computation. The approximation algorithms remain below 20 ms throughout while every measured ratio stays within the theoretical bounds and most stay within 10% of the optimum. At these growth rates, exhaustive enumeration is practical mainly for validating small instances. For larger candidate sets, the approximation algorithms require much less solving time while retaining the guarantees established in Section 4.6.3.

Constrained computation by category. The third study isolates the other runtime component: the constrained-distance phase of Theorem 4.2, whose cost is predicted to scale with the constraint state space $|Q|$. We fix the source–target workload (up to 30×30 pairs) and measure the total distance-computation time on graphs of $n = 50$ to 200 nodes for one configuration per category, reporting the slowdown relative to unconstrained Dijkstra on the same pairs.

The measured factors in Table 4.3 are stable across graph sizes and broadly consistent with the state-space analysis, although the slowdown is not proportional to $|Q|$. Categories A and D add little overhead in these experiments, with slowdowns of at most 1.18×. The hop limit costs a small constant factor ($\approx$ 2.2×, well below its $|Q| = h{+}1 = 6$ worst

Table 4.3: Distance-phase slowdown relative to unconstrained Dijkstra (same pairs; unconstrained absolute time 0.29 s at $n = 50$ rising to 1.81 s at $n = 200$). The cost-budget rows certify feasibility for the 75% of pairs within budget by construction.

Configuration	Cat.	$\|Q\|$	n=50	n=100	n=150	n=200
Forbidden nodes	A	1	1.04×	0.99×	1.13×	1.18×
Edge types	D	1	1.05×	1.13×	1.04×	1.09×
Hop limit $h = 5$	C	6	2.37×	2.37×	2.20×	2.10×
Waypoint	B	2	3.87×	4.27×	4.02×	3.80×
Cost budget	C	pseudo-poly.	8.3×	11.6×	9.9×	10.4×

case, since most shortest paths use few hops). The waypoint constraint costs $\approx 4\times$: its automaton has only two states, but the search must also continue past targets reached in the non-accepting state (Theorem 4.2). The real-valued cost budget is the most expensive tested configuration, with overheads around 8–12×. Unlike the finite-state constraints above, accumulated cost cannot generally be represented by a fixed-size state space, consistent with the pseudo-polynomial complexity described in Theorem 4.2. For the tested Category E no-revisit constraint, runtime differs between feasible and infeasible queries.

These experiments evaluate the solver introduced for the constrained set-to-set problems identified in Section 4.7.3. On the tested concatenation-closed configurations, the approximation algorithms satisfy their theoretical bounds and remain close to the exact optimum on small instances. Their solve-phase runtime scales much better than exhaustive enumeration as $|S|$ and k increase. The constrained-distance experiments further show that computational overhead depends strongly on the constraint categories: filtering constraints add little overhead, finite-state constraints incur moderate costs, and resource- or history-dependent constraints can be significantly more expensive. The measured runtime differences are consistent with the computational distinctions established in Section 4.6.3.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

This dissertation developed Generative Problem Solving as a structured approach for constructing, validating, and discovering computational problems. The central idea is that problem formulation should be part of the reasoning process rather than treated only as an input supplied to a solver. Across the three frameworks, grammars and semantic representations define how problems are constructed, which combinations are meaningful, and how a generated formulation can be evaluated or connected to a solution method.

GPS-Relational established the foundation for this approach in relational databases. It constructs executable reference SQL from natural-language intent through decomposition, schema mapping, semantic validation, and query assembly. Execution results provide the basis for evaluating LLM-generated SQL without requiring it to match a single reference expression. The grammar also supports benchmark generation with controlled relational structures and nesting depths.

GPS-CQ extended the framework to natural-language explanations of complex SQL. It evaluates an explanation through an SQL-to-natural-language-to-SQL round trip and compares the original and reconstructed queries through execution. When their results differ, the framework identifies errors involving aggregation scope, omitted conditions, or dependencies between query blocks. The same structural information supports the generation of targeted benchmark cases around observed weaknesses.

GPS-SD moved beyond conventional SQL by treating sets of entities as first-class objects. It generates set-level problem formulations from an object-relational schema and a vocabulary of domain functions. Evaluation in the influence and shortest-distance domains recovered known problem structures and exposed formulations outside the scope of existing solvers. In the shortest-distance domain, this process led to a constrained set-to-set optimization family for which solver methods, complexity results, and approximation guarantees were developed.

Together, these frameworks show how an explicit representation of problem structure can support translation, validation, diagnosis, benchmark generation, and problem discovery. The progression from GPS-Relational to GPS-CQ and GPS-SD extends Generative Problem Solving from constructing executable relational queries to examining their explanations and discovering computational problems that require additional solution methods.

5.2 Future Work

The GPS-Relational experiments identified inputs that could not be processed because of missing lexical or operator mappings, ambiguous schema references, or unsupported relationship paths. Future work can expand this coverage while preserving the semantic restrictions that prevent invalid queries. Improved schema grounding and ambiguity resolution would allow the framework to support a wider range of natural-language formulations and database

structures.

GPS-CQ currently uses diagnosed structural weaknesses to guide benchmark generation and provide structure-aware examples. Model training remains outside the scope of the present work. A direct extension is to use these examples for fine-tuning and then evaluate whether targeted training reduces aggregation-scope errors, condition dropping, and dependency mismatches. The diagnosed patterns could also guide an explanation model during generation rather than being used only after an incorrect explanation is detected.

Future work on GPS-SD will broaden the vocabulary of set-level functions and reduce the amount of domain information that must be supplied in advance. Unsupported computational functions could be identified and presented to domain experts for implementation. Schemas that do not include a declared graph interpretation could also be mapped to suitable graph representations. The framework could then combine functions from different abstract models over the same schema, including relationships such as influence and similarity.

Greater automation can also support category discovery and literature screening for generated problem structures. Such automation would help organize a larger problem space and identify possible connections to existing work. Expert review should remain part of this process, particularly before an unmatched formulation is treated as a new problem discovery.

These directions preserve the main principle of the dissertation: generated problems should remain connected to explicit structures that can be inspected, validated, and related to available solution methods.

Bibliography

[1] T. Yu, R. Zhang, K. Yang, M. Yasunaga, D. Wang, Z. Li, J. Ma, I. Li, Q. Yao, S. Roman, Z. Zhang and D. Radev, Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task, in *Proc. 2018 Conf. Empirical Methods in Natural Language Processing*, 2018, pp. 3911–3921.

[2] F. Zhao, L. Lim, I. Ahmad, D. Agrawal and A. El Abbadi, LLM-SQL-Solver: Can LLMs determine SQL equivalence? preprint (2023), arXiv:2312.10321.

[3] H. Kim, T. Jeon, S. Choi, S. Choi and H. Cho, FLEX: Expert-level false-less execution metric for reliable text-to-SQL benchmark, in *Proc. 2025 Conf. Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, 2025, pp. 4448–4475.

[4] G. G. Hendrix, E. D. Sacerdoti, D. Sagalowicz and J. Slocum, Developing a natural language interface to complex data, *ACM Trans. Database Syst.* 3(2) (1978) 105–147.

[5] W. A. Woods, R. M. Kaplan and B. Nash-Webber, *The lunar sciences natural language information system: Final report*, BBN Report No. 2378, Bolt Beranek and Newman, Cambridge, Massachusetts (1972).

[6] D. H. D. Warren and F. C. N. Pereira, An efficient and easily adaptable system for interpreting natural language queries, *Am. J. Comput. Linguist.* 8(3–4) (1982) 110–122.

[7] A. Popescu, O. Etzioni and H. Kautz, Towards a theory of natural language interfaces to databases, in *Proc. 8th Int. Conf. Intelligent User Interfaces*, 2003, pp. 149–157.

[8] F. Li and H. V. Jagadish, NaLIR: An interactive natural language interface for querying relational databases, in *Proc. ACM SIGMOD Int. Conf. Management of Data*, 2014, pp. 709–712.

[9] L. Blunschi, C. Jossen, D. Kossmann, M. Mori and K. Stockinger, SODA: Generating SQL for business users, *Proc. VLDB Endow.* 5(10) (2012) 932–943.

[10] C. Baik, H. V. Jagadish and Y. Li, Bridging the semantic gap with SQL query logs in natural language interfaces to databases, in *Proc. IEEE Int. Conf. Data Engineering*, 2019, pp. 374–385.

[11] A. Usta, A. Karakayali and O. Ulusoy, DBTagger: Multi-task learning for keyword mapping in NLIDBs using bi-directional recurrent neural networks, *Proc. VLDB Endow.* 14(5) (2021) 813–821.

[12] N. Yaghmazadeh, Y. Wang, I. Dillig and T. Dillig, SQLizer: Query synthesis from natural language, *Proc. ACM Program. Lang.* 1(OOPSLA) (2017) 63.

[13] F. Basik, B. Hättasch, A. Ilkhechi, A. Usta, S. Ramaswamy, P. Utama, N. Weir, C. Binnig and U. Cetintemel, DBPal: A learned NL-interface for databases, in *Proc. 2018 Int. Conf. Management of Data*, 2018, pp. 1765–1768.

[14] C. Wang, A. Cheung and R. Bodík, Synthesizing highly expressive SQL queries from input-output examples, in *Proc. 38th ACM SIGPLAN Conf. Programming Language Design and Implementation*, 2017, pp. 452–466.

[15] S. Zhang and Y. Sun, Automatically synthesizing SQL queries from input-output examples, in *Proc. 28th IEEE/ACM Int. Conf. Automated Software Engineering*, 2013, pp. 224–234.

[16] D. Saha, A. Floratou, K. Sankaranarayanan, U. F. Minhas, A. R. Mittal and F. Ozcan, ATHENA: An ontology-driven system for natural language querying over relational data stores, *Proc. VLDB Endow.* 9(12) (2016) 1209–1220.

[17] J. Sen, C. Lei, A. Quamar, F. Ozcan, V. Efthymiou, A. Dalmia, G. Stager, A. R. Mittal, D. Saha and K. Sankaranarayanan, ATHENA++: Natural language querying for complex nested SQL queries, *Proc. VLDB Endow.* 13(11) (2020) 2747–2759.

[18] M. Pourreza and D. Rafiei, DIN-SQL: Decomposed in-context learning of text-to-SQL with self-correction, in *Proc. 37th Int. Conf. Neural Information Processing Systems*, 2023, pp. 36339–36348.

[19] B. Wang, C. Ren, J. Yang, X. Liang, J. Bai, L. Chai, Z. Yan, Q.-W. Zhang, D. Yin, X. Sun and Z. Li, MAC-SQL: A multi-agent collaborative framework for text-to-SQL, in *Proc. 31st Int. Conf. Computational Linguistics*, 2025, pp. 540–557.

[20] T. Scholak, N. Schucher and D. Bahdanau, PICARD: Parsing incrementally for constrained auto-regressive decoding from language models, in *Proc. 2021 Conf. Empirical Methods in Natural Language Processing*, 2021, pp. 9895–9901.

[21] S. Jivani, S. Maheshwari and S. Sarawagi, Reliable answers for recurring questions: Boosting text-to-SQL accuracy with template constrained decoding, *Proc. ACM Manag. Data* 3(6) (2025) 357.

[22] V. I. Levenshtein, Binary codes capable of correcting deletions, insertions, and reversals, *Sov. Phys. Dokl.* 10(8) (1966) 707–710.

[23] G. Koutrika, A. Simitsis, and Y. E. Ioannidis, "Explaining Structured Queries in Natural Language," in *2010 IEEE 26th International Conference on Data Engineering (ICDE)* (2010), pp. 333–344.

[24] K. Xu, L. Wu, Z. Wang, Y. Feng, and V. Sheinin, "SQL-to-Text Generation with Graph-to-Sequence Model," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP)* (2018), pp. 931–936.

[25] D. Ma, X. Chen, R. Cao, Z. Chen, L. Chen, and K. Yu, "Relation-Aware Graph Transformer for SQL-to-Text Generation," *Applied Sciences*, vol. 12, no. 1, article 369, 2022.

[26] A. Al Lawati, J. Lucas, and P. Mitra, "Semantic Captioning: Benchmark Dataset and Graph-Aware Few-Shot In-Context Learning for SQL2Text," in *Proceedings of the 31st International Conference on Computational Linguistics (COLING)* (2025), pp. 8026–8042.

[27] S. Haroon, C. Brown, and M. A. Gulzar, "DeSQL: Interactive Debugging of SQL in Data-Intensive Scalable Computing," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, article 35, 2024.

[28] M. Pourreza and D. Rafiei, "DIN-SQL: Decomposed In-Context Learning of Text-to-SQL with Self-Correction," in *Advances in Neural Information Processing Systems 36 (NeurIPS)* (2023).

[29] B. Wang, C. Ren, J. Yang, X. Liang, J. Bai, L. Chai, Z. Yan, Q.-W. Zhang, D. Yin, X. Sun, and Z. Li, "MAC-SQL: A Multi-Agent Collaborative Framework for Text-to-SQL," in *Proceedings of the 31st International Conference on Computational Linguistics (COLING)* (2025), pp. 540–557.

[30] H. Kobayashi, W. Lan, P. Shi, S. Chang, J. Guo, H. Zhu, Z. Wang, and P. Ng, "You Only Read Once (YORO): Learning to Internalize Database Knowledge for Text-to-SQL," in *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT), Volume 1: Long Papers* (2025), pp. 1889–1901.

[31] J. Li, B. Hui, G. Qu, J. Yang, B. Li, B. Li, B. Wang, B. Qin, R. Geng, N. Huo, X. Zhou, C. Ma, G. Li, K. C. C. Chang, F. Huang, R. Cheng, and Y. Li, "Can LLM Already Serve as a Database Interface? A Big Bench for Large-Scale Database Grounded Text-to-SQLs," in *Advances in Neural Information Processing Systems 36 (NeurIPS)* (2023), pp. 42330–42357.

[32] Y. Tian, J. K. Kummerfeld, T. Jia-Jun Li, and T. Zhang, "SQLucid: Grounding Natural Language Database Queries with Interactive Explanations," in *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology (UIST)* (2024), article 12, 20 pages.

[33] M. Veanes, N. Tillmann, and J. de Halleux, "Qex: Symbolic SQL Query Explorer," in *Proceedings of the 17th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR-17)* (2010), pp. 425–446.

[34] Y. He, P. Zhao, X. Wang, and Y. Wang, "VeriEQL: Bounded Equivalence Verification for Complex SQL Queries with Integrity Constraints," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA1, article 110, 2024.

[35] R. Klopfenstein, Y. He, A. Tremante, Y. Wang, N. Narodytska, and H. Wu, "SpotIt: Evaluating Text-to-SQL Evaluation with Formal Verification," in *International Conference on Learning Representations (ICLR)* (2026).

[36] G. Vaisi and P. C.-Y. Sheu, "GPS-Relational: Generative Problem Solving with Relational Databases," *International Journal of Semantic Computing*, vol. 20, no. 3, pp. 395–419 (2026).

[37] Microsoft Corporation, "Maximum Capacity Specifications for SQL Server," Microsoft Learn, online documentation. Available: `https://learn.microsoft.com/en-us/sql/sql-server/maximum-capacity-specifications-for-sql-server?view=sql-server-ver17`.

[38] Oracle Corporation, "Server Error Message Reference," *MySQL Reference Manual*, online documentation. Available: `https://dev.mysql.com/doc/mysql-errors/en/server-error-reference.html`.

[39] C. C. N. Wang, J. Jin, J.-G. Chang, M. Hayakawa, A. Kitazawa, J. J. P. Tsai, and P. C.-Y. Sheu, "Identification of Most Influential Co-occurring Gene Suites for Gastrointestinal Cancer Using Biomedical Literature Mining and Graph-Based Influence Maximization," *BMC Medical Informatics and Decision Making*, vol. 20, Art. no. 208 (2020).

[40] D. Kempe, J. Kleinberg, and É. Tardos, "Maximizing the Spread of Influence through a Social Network," *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 137–146 (2003).

[41] V. Arya, N. Garg, R. Khandekar, A. Meyerson, K. Munagala, and V. Pandit, "Local Search Heuristics for k-Median and Facility Location Problems," *SIAM Journal on Computing*, vol. 33, no. 3, pp. 544–562 (2004).

[42] R. Ramamonjison, T. Yu, R. Li, H. Li, G. Carenini, B. Ghaddar, S. He, M. Mostajabdaveh, A. Banitalebi-Dehkordi, Z. Zhou, and Y. Zhang, "NL4Opt Competition: Formulating Optimization Problems Based on Their Natural Language Descriptions," *Proceedings of the NeurIPS 2022 Competitions Track*, vol. 220, pp. 189–203 (2022).

[43] D. Tsouros, H. Verhaeghe, S. Kadıoğlu, and T. Guns, "Holy Grail 2.0: From Natural Language to Constraint Models," *arXiv preprint arXiv:2308.01589* (2023).

[44] T. Yu, C.-S. Wu, X. V. Lin, B. Wang, Y. C. Tan, X. Yang, D. R. Radev, R. Socher, and C. Xiong, "GraPPa: Grammar-Augmented Pre-Training for Table Semantic Parsing," *International Conference on Learning Representations (ICLR)* (2021).

[45] H. A. Caferoğlu, M. S. Çelik, and Ö. Ulusoy, "SING-SQL: A Synthetic Data Generation Framework for In-Domain Text-to-SQL Translation," *arXiv preprint arXiv:2509.25672* (2025).

[46] H. Li, S. Wu, X. Zhang, X. Huang, J. Zhang, F. Jiang, S. Wang, T. Zhang, J. Chen, R. Shi, H. Chen, and C. Li, "OmniSQL: Synthesizing High-Quality Text-to-SQL Data at Scale," *Proceedings of the VLDB Endowment*, vol. 18, no. 11, pp. 4695–4709 (2025).

[47] A. Sosa-Ascencio, G. Ochoa, H. Terashima-Marín, and S. E. Conant-Pablos, "Grammar-Based Generation of Variable-Selection Heuristics for Constraint Satisfaction Problems," *Genetic Programming and Evolvable Machines*, vol. 17, no. 2, pp. 119–144 (2016).

[48] G. Mweshi and N. Pillay, "An Improved Grammatical Evolution Approach for Generating Perturbative Heuristics to Solve Combinatorial Optimization Problems," *Expert Systems with Applications*, vol. 165, Art. no. 113853 (2021).

[49] S. R. Valluri and K. Karlapalem, "Subset Queries in Relational Databases," *CoRR*, cs.DB/0406029 (2004).

[50] M. Brucato, J. F. Beltran, A. Abouzied, and A. Meliou, "Scalable Package Queries in Relational Database Systems," *Proceedings of the VLDB Endowment*, vol. 9, no. 7, pp. 576–587 (2016).

[51] D. R. Pratten, L. Mathieson, and F. Ramezani, "Relational Algebras for Subset Selection and Optimisation," *arXiv preprint arXiv:2509.06439* (2025).

[52] O. Kariv and S. L. Hakimi, "An Algorithmic Approach to Network Location Problems. II: The p-Medians," *SIAM Journal on Applied Mathematics*, vol. 37, no. 3, pp. 539–560 (1979).

[53] D. S. Hochbaum and D. B. Shmoys, "A Best Possible Heuristic for the k-Center Problem," *Mathematics of Operations Research*, vol. 10, no. 2, pp. 180–184 (1985).

[54] D. S. Hochbaum and D. B. Shmoys, "A Unified Approach to Approximation Algorithms for Bottleneck Problems," *Journal of the ACM*, vol. 33, no. 3, pp. 533–550 (1986).

[55] S. S. Ravi, D. J. Rosenkrantz, and G. K. Tayi, "Heuristic and Special Case Algorithms for Dispersion Problems," *Operations Research*, vol. 42, no. 2, pp. 299–310 (1994).

[56] S. Irnich and G. Desaulniers, "Shortest Path Problems with Resource Constraints," in *Column Generation*, G. Desaulniers, J. Desrosiers, and M. M. Solomon, Eds., Springer, pp. 33–65 (2005).

[57] I. Dumitrescu and N. Boland, "Improved Preprocessing, Labeling and Scaling Algorithms for the Weight-Constrained Shortest Path Problem," *Networks*, vol. 42, no. 3, pp. 135–153 (2003).

[58] W. Ying, Y. Li, and P. C.-Y. Sheu, "A GA-Based Approach to Optimizing Combinational Queries in SCDL," *International Journal of Semantic Computing*, vol. 2, no. 2, pp. 273–289 (2008).

[59] G. Cornuejols, M. L. Fisher, and G. L. Nemhauser, "Location of Bank Accounts to Optimize Float: An Analytic Study of Exact and Approximate Algorithms," *Management Science*, vol. 23, no. 8, pp. 789–810 (1977).

[60] W.-L. Hsu and G. L. Nemhauser, "Easy and Hard Bottleneck Location Problems," *Discrete Applied Mathematics*, vol. 1, no. 3, pp. 209–215 (1979).

[61] T. F. Gonzalez, "Clustering to Minimize the Maximum Intercluster Distance," *Theoretical Computer Science*, vol. 38, pp. 293–306 (1985).

[62] S. Guha and S. Khuller, "Greedy Strikes Back: Improved Facility Location Algorithms," *Journal of Algorithms*, vol. 31, no. 1, pp. 228–248 (1999).

[63] U. Feige, "A Threshold of ln n for Approximating Set Cover," *Journal of the ACM*, vol. 45, no. 4, pp. 634–652 (1998).