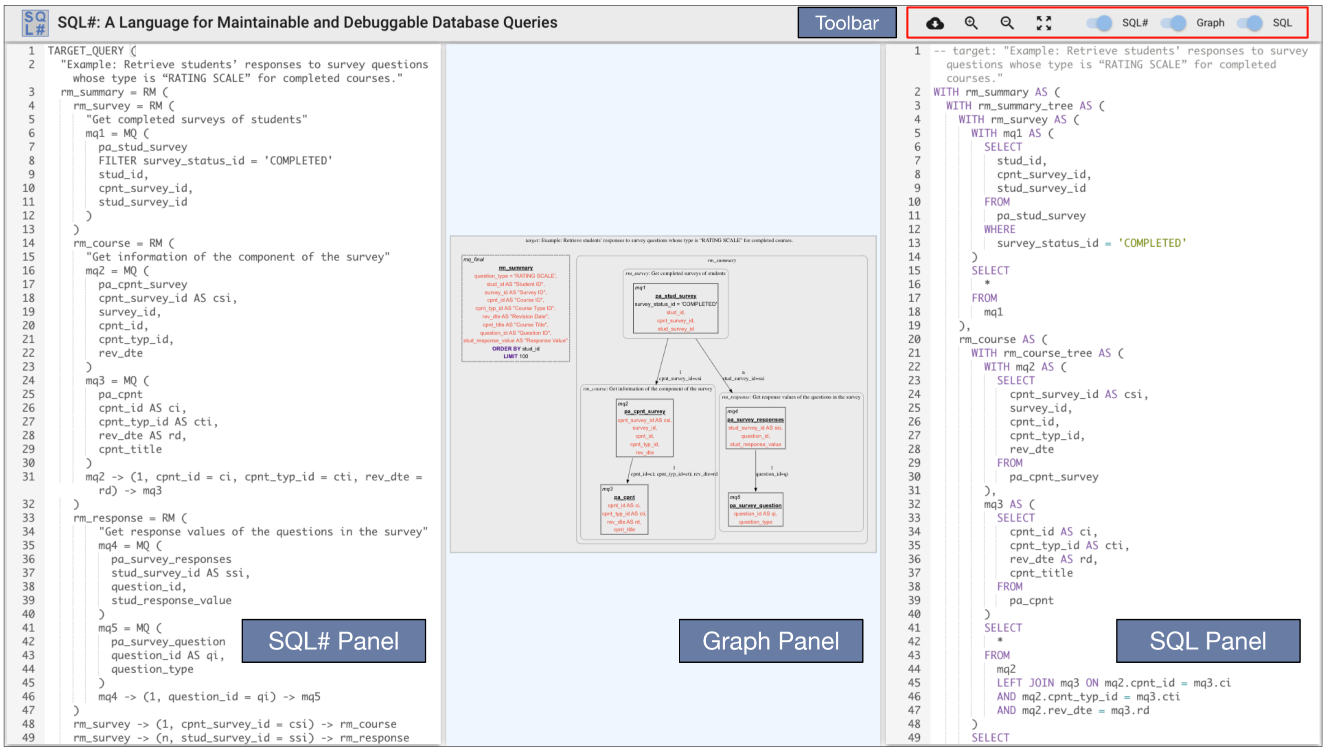
User Instruction for the SQL# Language and the SQL#-based System


SQL#: A Language for Maintainable and Debuggable Database Queries
Toolbar
SQL#
Graph
SQL
SQL# Panel
Graph Panel
SQL Panel

```
TARGET_QUERY (
  "Example: Retrieve students' responses to survey questions
    whose type is “RATING SCALE” for completed courses."
  rm_summary = RM (
    rm_survey = RM (
      "Get completed surveys of students"
      mq1 = MQ (
        pa_stud_survey
        FILTER survey_status_id = 'COMPLETED'
        stud_id,
        cpnt_survey_id,
        stud_survey_id
      )
    )
    rm_course = RM (
      "Get information of the component of the survey"
      mq2 = MQ (
        pa_cpnt_survey
        cpnt_survey_id AS csi,
        survey_id,
        cpnt_id,
        cpnt_typ_id,
        rev_dte
      )
      mq3 = MQ (
        pa_cpnt
        cpnt_id AS ci,
        cpnt_typ_id AS cti,
        rev_dte AS rd,
        cpnt_title
      )
      mq2 -> (1, cpnt_id = ci, cpnt_typ_id = cti, rev_dte =
        rd) -> mq3
    )
    rm_response = RM (
        "Get response values of the questions in the survey"
        mq4 = MQ (
          pa_survey_responses
          stud_survey_id AS ssi,
          question_id,
          stud_response_value
        )
        mq5 = MQ (
          pa_survey_question
          question_id AS qi,
          question_type
        )
        mq4 -> (1, question_id = qi) -> mq5
    )
    rm_survey -> (1, cpnt_survey_id = csi) -> rm_course
    rm_survey -> (n, stud_survey_id = ssi) -> rm_response
```

```
-- target: "Example: Retrieve students' responses to survey
  questions whose type is “RATING SCALE” for completed
  courses."
WITH rm_summary AS (
  WITH rm_summary_tree AS (
    WITH rm_survey AS (
      WITH mq1 AS (
        SELECT
          stud_id,
          cpnt_survey_id,
          stud_survey_id
        FROM
          pa_stud_survey
        WHERE
          survey_status_id = 'COMPLETED'
      )
      SELECT
        *
      FROM
        mq1
    ),
    rm_course AS (
      WITH rm_course_tree AS (
        WITH mq2 AS (
          SELECT
            cpnt_survey_id AS csi,
            survey_id,
            cpnt_id,
            cpnt_typ_id,
            rev_dte
          FROM
            pa_cpnt_survey
        ),
        mq3 AS (
          SELECT
            cpnt_id AS ci,
            cpnt_typ_id AS cti,
            rev_dte AS rd,
            cpnt_title
          FROM
            pa_cpnt
        )
        SELECT
          *
        FROM
          mq2
          LEFT JOIN mq3 ON mq2.cpnt_id = mq3.ci
          AND mq2.cpnt_typ_id = mq3.cti
          AND mq2.rev_dte = mq3.rd
      )
      SELECT
```

Figure 1 The SQL#-based System

SQL is the dominant language for managing relational databases. However, complex SQL queries are hard to write and understand because of the intricate inter-table and inter-column relations. Consequently, we design a system to facilitate the development of maintainable and debuggable database queries. We first design a novel query language called SQL#, through which programmers could construct complex database queries module by module and step by step according to the logical steps of constructing queries. We then leverage visualization techniques and debugging techniques to facilitate the understanding of SQL# queries.

A demo video is at https://github.com/QueryDebugger/QuerySystem/releases/download/1.0/ExplanatoryVideo.

The SQL#-based system is deployed at https://querydebugger.github.io/.

1 SQL# Language

The SQL# language consists of four components (mini-query, requirements module, expanding edge, and union edge) and a feature (reference).

1.1 Mini-query

Mini-queries are the basic units in SQL#, which are similar to the SELECT statements in SQL. Its syntax is presented below, where the clauses that are in gray color are optional. *<single-table>* can be a base table in databases or a table represented by a mini-query or requirements module.

```
<identifier> = MQ (
    <single-table>
    <filter-clause>
    <group-by-clause>
    <having-clause>
    <select-item-list>
    <order-clause>
    <limit-clause>
)
```

1.2 Requirements module

Using requirements modules, programmers can divide a complex query into independent modules, and finally combine these smaller modules into a big complex query. Its syntax is presented below.

```
<identifier> = RM (
        <description>
        one or more <mini-query> / <requirements-module>
        one or more <expanding-edge> / one <union-edge>
)
```

In the body of a requirements module, programmers can write:

- A table. The table is represented by a mini-query or a requirements module. The table may refer to other mini-queries and requirements modules.
- An expanding query. Multiple tables are connected by one or more expanding edges. The expanded table is the table of the outside requirements module.
- A union query. Multiple tables are connected by a union edge. The resulting table is the table of the outside requirements module.

1.3 Expanding edge

An expanding edge points a source table to a target table, and expands the source table using the rows in the target table to constitute a bigger table. Its syntax is presented below.

```
<identifier> --> ( 1 or n , one or more <identifier> = <identifier> separated by ',' ) -> <identifier>
```

For example, the expanding edge “table1 -> (*1*, *col1* = *col2*) -> *table2*” means that “SELECT * FROM *table1* LEFT JOIN *table2* on *table1.col1* = *table2.col2*”, and the first element in the brackets (i.e., “1” or “*n*”) means that one record in *table1* corresponds to one or *n* (or zero) records in *table2*, which is a helpful indication for debugging query errors on the number of entries.

1.3 Union edge

A union edge unions multiple tables that are represented by mini-queries or requirements modules into one table. Its syntax is presented below.

```
UNION ( DISTINCT / ALL ) two or more <identifier>
```

1.4 Reference

Mini-queries can refer to the declared tables (requirements modules and mini-queries) through their identifiers. Thus, mini-queries can further process the referred table. The multiple references constitute a process chain, which corresponds to the logical steps of constructing queries.

Not all declared tables can be referred to. The declaration of a requirements module results in a new scope. Thus, the nesting of requirements modules results in the nesting of scope. The mini-queries in an inner scope can only refer to the tables in the current scope and the enclosing scopes.

2. System Interface

The interface of the SQL#-based system is presented in Figure 1, which contains three panels: SQL# Panel, Graph Panel, and SQL Panel. Each panel can be switched on or off independently.

When writing SQL# programs, programmers could switch off SQL Panel to make good use of the space. When the SQL# program in the SQL# Panel is changed, the program will be compiled into a DOT program. If the compilation is successful, the graph in the Graph Panel will be re-rendered, thus programmers could observe the structure of the written query in real time and obtain information such as expanding conditions from the graph to write SQL# query.

When debugging a SQL# program, programmers could switch off the SQL# panel. Programmers can click the component on the graph in Graph Panel, and then obtain the corresponding SQL program (then execute the SQL code on their database systems to obtain the corresponding table). The graph in Graph Panel could be zoomed in, zoomed out, and centered through the functionality in the toolbar. Programmers can also use the mouse wheel and the touchpad to zoom the graph.