Towards Automatic Grading of SQL Queries

by

Vijay Kumar Venkatamuniyappa

B.E., S.J.C Institution of Technology, India, 2012

A REPORT

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Computer Science
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2018

Approved by:

Major Professor
Dr. Doina Caragea

Copyright

Abstract

An *Introduction to Databases* course involves learning the concepts of data storage, manipulation, and retrieval. Relational databases provide an ideal learning path for understanding database concepts. The Structured Query Language (SQL) is a standard language for interacting with relational database. Each database vendor implements a variation of the SQL standard. Furthermore, a particular question that asks for some data can be written in many ways, using somewhat similar or structurally different SQL queries. Evaluation of SQL queries for correctness involves the verification of the SQL syntax and semantics, as well as verification of the output of queries and the usage of correct clauses. An evaluation tool should be independent of the specific database queried, and of the nature of the queries, and should allow multiple ways of providing input and retrieving the output. In this report, we have developed an evaluation tool for SQL queries, which checks for correctness of *MySQL* and *PostgreSQL* queries with the help of a parser that can identify SQL clauses. The tool developed will act as a portal for students to test and improve their queries, and finally to submit the queries for grading. The tool minimizes the manual effort required while grading, by taking advantage of the SQL parser to check queries for correctness, provide feedback, and allow submission.

Table of Contents

List of Figures vi

List of Tables vii

List of Algorithms viii

List of Listings ix

Acknowledgements ix

Dedication x

1 Introduction 1

1.1 Automatic Evaluation of Computer Programs 1

1.2 Previous Work in SQL Query Evaluation 3

1.3 Proposed Approach 4

2 Tool Implementation 6

2.1 Architecture 6

2.2 Project Components 7

2.3 Parsing SQL Queries 8

2.3.1 Pre-parser Operations 11

2.3.2 Post-parser Operations 14

2.4 Query Evaluation Phase 15

3 Tool Evaluation 17

3.1 Testing Life Cycle 17
3.2 Evaluating An SQL Assignment 18
3.2.1 Quantitative analysis 18
3.2.2 Qualitative analysis 19

4 Tool Readability And Reusability 23
4.1 Project Setup and Execution 23
4.2 Style Guide 24
4.3 Additional Application 25

5 Future Work and Conclusion 26

Bibliography 28

List of Figures

2.1 Architecture and Components of the Grading Tool 7
2.2 User Login Page 8
2.3 MySQL Query Response Page 9

3.1 Tool response on a query with syntax error 19
3.2 Tool response on a valid query that failed to parse 20
3.3 Identifying a correct query result and its output 21
3.4 Query with an incorrect result 22
3.5 Valid query interpreted as wrong 22

List of Tables

2.1 Regular expressions to identify SQL clauses 11

2.2 JSON form of SQL clause 14

3.1 Assignment evaluation: quantitative report 18

List of Algorithms

1 Pre-parsing algorithm 10

2 Post-parsing algorithm 16

List of Listings

1 Sample query for parsing 11

2 Intermediate result of query and JSON in the pre-parser step 12

3 Dictionary for storing clauses for later processing in the pre-parser step . . . 13

4 Output of the post-parser step 15

5 Steps on setting up the project 24

6 Coding conventions followed 24

Acknowledgments

I would first like to thank Dr. Doina Caragea, for guiding and mentoring me in master's report. I express my gratitude to Dr. Caragea for the time and effort spent on providing positive critique and suggestions that molded the outcome of project and report. I credit Dr. Torben Amtoft for teaching programming language course which has equipped me with the knowledge in the implementation of the project. I am thankful to Dr. Daniel Andresen for his contribution.

Dedication

For the Lord giveth wisdom; out of His mouth cometh knowledge and understanding.

Proverbs 2:6

Chapter 1

Introduction

Classroom and take-home assignments represent a way of evaluating the knowledge gained on a course by students. These assignments are graded either by the instructor or by the teaching assistant (TA). Grading involves comparing the submission of the student with a standard sample solution. The effort and time consumption in grading depends on the type of assignment.

Assignments vary in complexity from simpler assignments with multiple choice answers, to more complex assignments such as open-ended essays and reports, graphic design and model building, programs written to execute on a particular electronic device, etc.

Evaluating some assignments involves human inspection, but there are also assignments whose evaluation involves a comparison of the output with an expected sample output. This chapter describes the process of grading computer programs in Section 1.1. Developments and insights into the evaluation of SQL queries from other works are presented in Section 1.2. Section 1.3 briefly provides a high level overview of the project work.

1.1 Automatic Evaluation of Computer Programs

Computer programs have an advantage over any other type of assignments in grading. Essays, reports, and graphics need extra tools to decipher and test, while computer programs

have the advantage that, like a Math problem, they often have a single correct result. Just like the steps involved in solving a Math problem vary based on the method or formula used, and the time the method/formula is used, computer programs are also susceptible to such complexity. In general, for a programming question there are multiple ways of achieving the desired result.

Automatic evaluation can help the instructors and TAs in eliminating redundant tasks and save precious time. Generally, grading requires the collection and extraction of all the student submissions to an environment. Then, the grader will go through each assignment and will run the program comparing the result with the expected outcome. When the program encounters errors, as a result of running in an environment other than the one program is developed on, the grader has to examine and setup/change specific parameters. An incomplete program submission will involve extra effort on the grader's side in order to find the errors in the program. Once the grading is done, the grades and feedback will be made available to the students.

An ideal automatic evaluation tool will take the student submissions as input, execute each submission, identify errors, if present, and finally assign a grade. This will eliminate the need for the grader to log in to the specific program execution environment, as well as the need for the transfer of data from the grader's machine to the execution environment. Furthermore, in addition to benefits from automatic evaluation of the program, the process of submission can also be improved by an automated tool. Such a tool will check for syntax errors in the program and will warn the students of an incorrect syntax, thereby alerting students to re-check their submission, and allowing only correct syntax submissions. As a result, this can free the graders from evaluating submissions for syntax errors. Furthermore, the process will enable the students to test their submissions in the grading environment, which will eliminate errors due to the program migration. As the same environment is used for student testing and grader evaluation, this will reduce the effort of both students and graders. The grader will have all the submissions in the desired environment, which can be accessed through any remote system. The submission portal will also offer timely feedback to the students, as students can test multiple times before submitting. The final grades

and feedback can be published quickly when the assignments are still fresh in the minds of students.

The automatic grading described by Cheang et al. (2003) involves three things: correctness, efficiency, and maintainability. Correctness is verified using the output of the program. Efficiency aspects in grading refer to feedback on performance, by identifying the resource usage of the program. Thirdly, maintainability is measured by the ease of readability and modifiability of the program. The priority in grading, as part of learning a programming language, should be to achieve correctness. Once this has been established, one needs to focus on maintainability and understandability.

In this project, we focus on automatic evaluation of queries written in the SQL language, specifically MySQL and PostgreSQL variants of the standard SQL.

1.2 Previous Work in SQL Query Evaluation

Queries can be written in multiple forms including casing, aliasing, inner queries, conditions and projection clauses, etc., and are evaluated for correctness with a set of test cases. *ActiveSQL* (Russell and Cumming, 2005) is a tool which contains a standard dataset, visible to the students and useful for testing the correctness of their queries, and a hidden dataset used for submission evaluation and grading. The hidden test cases are developed with a different set of data, to find and penalize the hard-coded results in the query. Grading using *ActiveSQL* is done by checking the accuracy of a number of column cells retrieved for the solution with the number of cells from the expected result.

ActiveSQL has the limitation that it checks with only two test cases/datasets when grading. Each test case should involve its specific data, which the grader has to setup while creating new questions. Partial credit is provided based on query output, but query clauses, which play an important role in the solution, are not considered in evaluation. To check for the clauses in the query, the instructor should provide a correct sample query solution. Since there are multiple ways to write queries, the system should identify all the possible solutions to a query, so that it will look for the suitable solution that will best match with the student's

submission. This method is followed in building the *XData* system (Chandra et al., 2016), which provides an automated platform for learning and grading SQL queries. The queries are canonicalized to identify the predicates, projections, relations and other clauses, and then submitted queries are awarded or penalized based on the conformity with the solution query. This will result in partial credit irrespective of the query output. Although the pre-processing steps will reduce a query to a standard form, the instructor should provide a number of possible ways to write the query.

SQLify (Dekeyser et al., 2007) uses a method of feedback on a query with assessment tool and peer review. Immediate feedback is provided by the system for syntactical and query correctness. Two other students randomly selected by the system provide elaborate feedback, which enables students to develop higher order thinking, while evaluating other students' work.

1.3 Proposed Approach

We have developed a tool that can help students test their queries and, at the same time, help graders evaluate submissions. Building the tool for testing and evaluating SQL queries involves developing a front-end user access, and a two-step validation process. The first step is to check the syntax of the query, and the second step is to test the semantic conformity of the query with respect to the sample solution query.

The user interface should allow students to validate the credentials and connect to their database account. It will provide the list of questions from an assignment for the students to work on. Then, it will allow students to test queries, see the output to their queries, and also to get feedback on the assignment questions. Students can then submit their solution for grading using the same interface.

In the first step of validation, the query submitted by the student/user is executed in the database on either *MySQL* or *PostgreSQL* server. The error message thrown by the database for an invalid query is shown to the user. When the query is free of errors, the output is displayed to the student. To test for semantic conformity, the query is verified

for the presence of required SQL clauses, identified based on the sample solution query. To achieve this, the query parser will bring the SQL query to a standard form. The output of a parser is stored in JSON format, and is used to check for conformity with the solution query.

The final step is to improve the aesthetics in the user interface and code of the tool. The user experience is enhanced by the responsive and dynamic nature of the web-page. Furthermore, the code of the tool is optimized for increased readability, re-usability and modularity.

Chapter 2

Tool Implementation

2.1 Architecture

The portal contains a front-end user interface, which the user can access through the web browser. Students and graders will interact with the systems through the user interface. For a single question testing, the input query is provided through an HTML text-box, while for multiple questions, the input queries are provided as a file. The back-end server acts as a middle-ware to execute operations requested from the browser. Based on the user inputs, the browser interacts with the file system and with the database to serve the requests. The result of an operation will be rendered and displayed by the server on a web-page. Figure 2.1 shows the order of interactions between different components in the project.

The front-end will be used as a channel for users to provide a query to test in a specific database. It will also allow the users to submit a query for evaluation, and allow graders to provide feedback on the correctness of query. The database is used to identify errors in query syntax, fetch query output and meta-data information of the query and tables used. The file system is used to store student submissions as a persistent data and to retrieve and display the assignment questions.

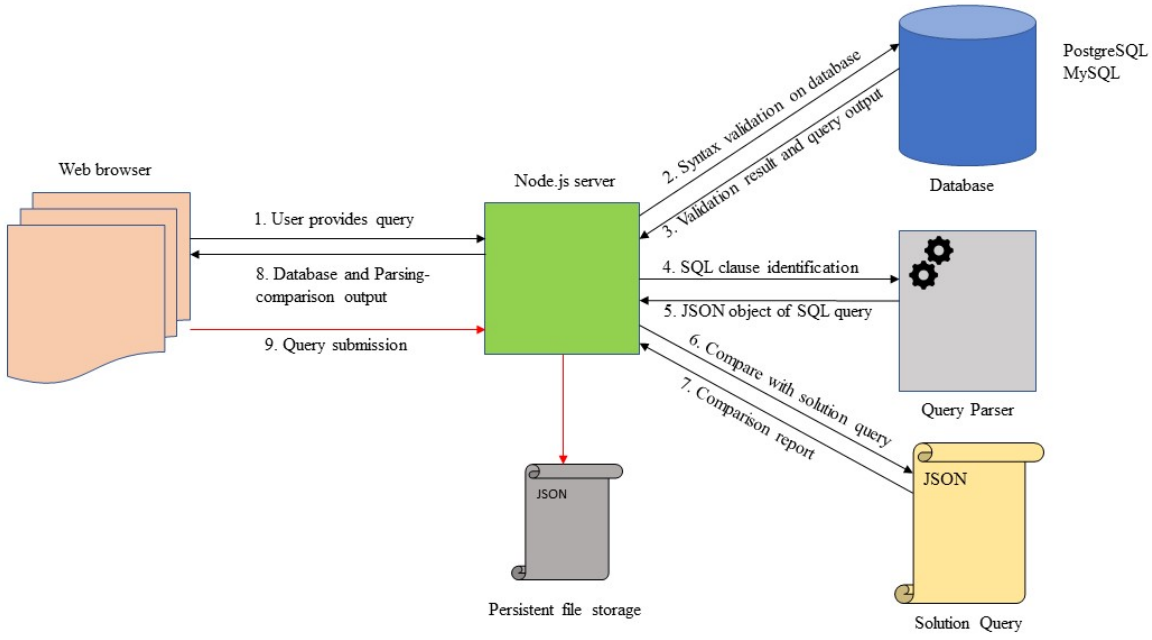

Figure 2.1: *Architecture and Components of the Grading Tool*

2.2 Project Components

As mentioned above, development of a portal which encompasses user interface, database, and file system access, and is accessible through any remote system, has to be developed on a web server. The server should be robust, secured, scalable to handle large requests. *Node.js* is used in constructing such a server. *Node.js* serves request through the asynchronous execution. Additional packages are available through Node Package Manager (*npm*), and can be used to extend the features of *Node.js*.

Connection and database operations are enabled by using *mysql* and *pg npm* packages for *MySQL* and *PostgreSQL* databases, respectively. Reading and writing operation to a file system is achieved through the *fs* package. *Express.js*, a *Node.js* web framework is used to build the web application. To extract the specific components of the SQL query, an SQL parser package, *node-sqlparser* is used. The user interface is built using *HTML*, *CSS* with *Bootstrap* framework. The user interface is made responsive and interactive with *Javascript*,

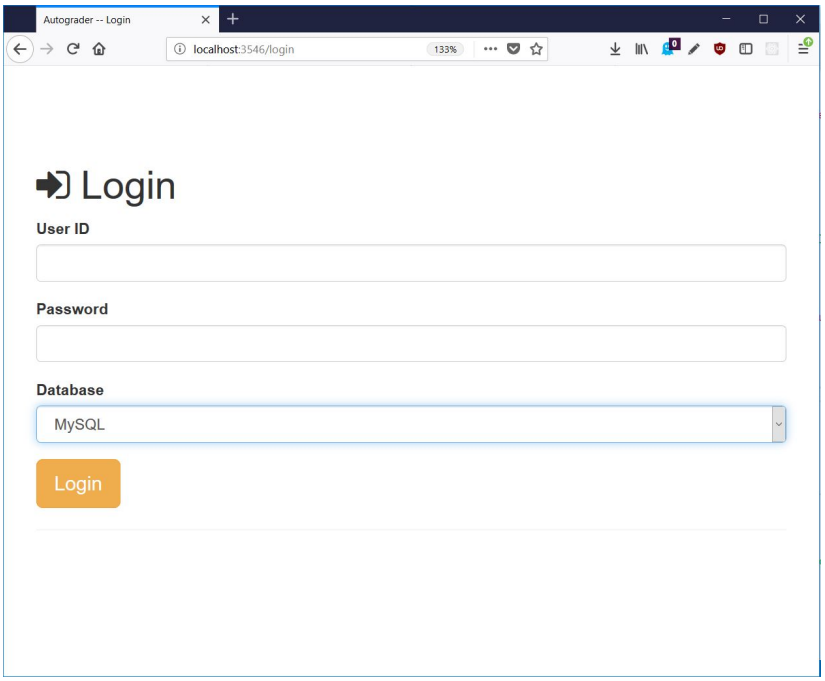

Figure 2.2: *User Login Page*

jQuery, *Embedded Javascript* (*EJS*). Figure 2.2 shows the user login screen of the tool, while Figure response shows a response page for a MySQL query.

The organization of the project file structure is influenced by the *Express.js* framework. Each web page has a dedicated router method to handle the *GET* and *POST* method. The router method invokes a series of *JavaScript* function to process the *POST* methods. Database and file transactions are handled through a dedicated function. Every time data is retrieved from the database or from a file file, this dedicated function is invoked. Rendering of the output as *HTML* pages is done through the *EJS* templates. Each page includes multiple templates, used as a series of partial templates to render a specific web page.

2.3 Parsing SQL Queries

The project's main component will focus on the correctness of the query, which is a task achieved by the SQL parser. The SQL parser will identify all the tokens from the query string and categorize them to different SQL clauses. Then, they are converted to the JSON format. The resulting JSON file contains a representation of the query string, and is used

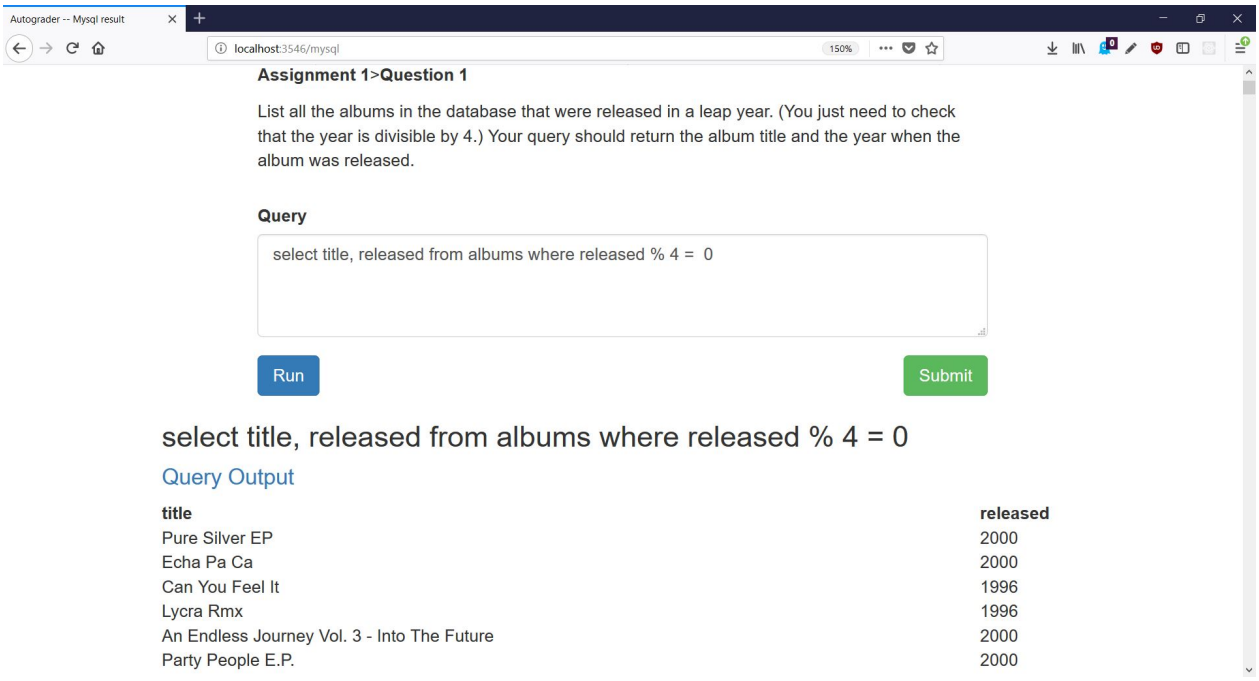

Figure 2.3: *MySQL Query Response Page*

for evaluation by extracting and comparing the specific clauses of the submitted query, to the equivalent clauses from the solution query.

To help with the parsing, the parser should identify the SQL select statement. The available packages in *npm* for parsing SQL queries are *node-sqlparser* (Fish, 2015) and *sql-parser* (Kent, 2015). The *sql-parser* encompasses a tokenizer and a parser, the output of the parser is a formatted string with the identified SQL expression from the original query. The *sql-parser* can parse `where`, `group by`, `order by` and `limit` clauses. The *node-sqlparser* package can identify the type of expressions, literals and arguments, and writes the output to a JSON file.

For our purposes, the *node-sqlparser* is better, as compared to *sql-parser*, as it can parse more clauses and outputs JSON file. However, it has a limitation in terms of the clauses it can identify. Specifically, the parser fails with an unidentified keyword error for inner queries, functions with more than one argument, `having`, `not null` and `case` statements. To overcome this limitation, the capability of the parser is extended by explicitly parsing each of the above-mentioned clauses, as described in detail in Section 2.3.1, before running

the parser on *node-sqlparser*. Thus, the extended clauses are parsed by our tool, and the equivalent JSON form is produced. A word/placeholder is used to replace an expression in the query which cannot be parsed by *node-sqlparser*. This word acts as a key and the JSON form of the SQL expression acts as a value. The key-value information is stored as a dictionary using a map data type. Once the query is parsed by the *node-sqlparser*, in the generated JSON, the key is replaced with the value from the dictionary, and thus the parser functionality is extended to parse more SQL clauses. Further details on the replacement of the key with the value from the dictionary are provided in Section 2.3.2.

Algorithm 1 Pre-parsing algorithm

```
1:  function preParseQuery(query)
2:      result = {query:query,store:new Map(),json:{}}
3:      result ← replaceAllString(result)
4:      result ← replaceNotNull(result)
5:      result ← removeExtraSpaces(result)
6:      result ← replaceFunction(result)
7:      result ← buildCaseJSON(result)
8:      result ← reduceSubQuery(result)
9:      json = parseSQL(json.query)
10:     json.having = buildHavingJSON(json.query)
11:     result.json ← json+json.having          ▷ Having is appended to JSON
12:     return result
```

To identify the presence of a specific expression or SQL clause, a regular expression is used. Both expression types and values are extracted through regular expressions and string operations like substring and match functions. The regular expressions used to identify SQL clauses are listed in Table 2.1. These clauses are stored in a dictionary data type. Dictionary keys are assigned a unique sequence number for each of these SQL clauses. This step acts as an extension to the *node-sqlparser* and is classified as a pre-parsing step, i.e., a step performed before calling the *node-sqlparser*. All operations performed in the pre-parsing step are shown in Algorithm 1.

Table 2.1: *Regular expressions to identify SQL clauses*

Regular expression	SQL clause
'[^']*'	String identifier
\s+(true|false)\s+	Boolean identifier
[0-9\.]+	Number identifier
[0-9A-Za-z_.$]+	Column names
\(\s*select.*\)	Inner query
([\w$.]+)\(\(\s*[\w$.]+\s*,\s*)*\s*([\w$.]+){1}\s*\)	Function call
case\s.*when\s.*then\s.*end	Case statement
\s+(having)\s+	Having clause

2.3.1 Pre-parser Operations

The pre-parsing operation will identify those SQL clauses that should be reduced/replaced in the JSON representation of the query, before passing the query to the *node-sqlparser*. A regular expression is used to identify the presence of a clause or identifier, and the clause is reduced in the equivalent JSON form of the query. The dictionary key is the SQL clause string from the query, and the value is the JSON form which is stored in this phase. The dictionary is used in a post-parser operation which is described in Section 2.3.2. Table 2.1 defines the regular expressions used to identify the SQL clauses, and the JSON forms generated for each clause are shown in Table 2.2. Listing 1 shows a sample query that needs to be verified. The output of the pre-parsing step executed on the sample query is shown in Listing 2 12 and the dictionary stored for extending the parsing functionality is shown in Listing 3.

Listing 1 Sample query for parsing

```
SELECT title,CASE ('released')  WHEN 1987 THEN 'before' WHEN 1988 THEN 'same'
END AS output FROM albums WHERE released IS NOT NULL GROUP BY title,released
HAVING released=1987 OR released=1988;
```

The first step is to find strings in the query. String literals represented by single quotes can have any printable characters. If quotes need to be added to the string, extra quotes or special characters are used to escape the quotes, and to allow the interpretation as a string literal. The *node-sqlparser* doesn't handle strings that contain quotes. To avoid such strings preventing the identification of other clauses, they are replaced with a key and the actual

Listing 2 Intermediate result of query and JSON in the pre-parser step

```
{ query: 'SELECT title,rep_string_5 AS output FROM albums where released is
rep_string_3 group by title,released having released=1987 or released=1988',
  json:
   { type: 'select',
     distinct: null,
     columns: [ { expr: { type: 'column_ref', table: '', column: 'title' },
                                              as: null },
          { expr: { type: 'column_ref', table: '', column: 'rep_string_5' },
                                              as: 'output' } ],
     from: [ { db: '', table: 'albums', as: null } ],
     where:{ type: 'binary_expr',
           operator: 'IS',
              left: { type: 'column_ref', table: '', column: 'released' },
              right: { type: 'column_ref', table: '', column: 'rep_string_3' } },
     groupby: [ { type: 'column_ref', table: '', column: 'title' },
                  { type: 'column_ref', table: '', column: 'released' } ],
     orderby: null,
     limit: null,
     params: [],
     having:
      { type: 'binary_expr',
                  operator: '=',
                  left: { type: 'column_ref', table: '', column: 'released' },
                  right: { type: 'number', value: 1988 } } } }
```

replaced string is stored in the dictionary for later recovery.

The NULL keyword is used to check uninitialized columns. Similarly NOT NULL keyword is used to identify the columns which are initialized with an actual value. The *node-sqlparser* handles NULL comparisons but not the NOT NULL comparisons. To account for this, NOT NULL is replaced with a constant word representing NOT NULL where-ever it is present in the query and its JSON is stored in a dictionary.

Functions with a single argument are classified as aggregate functions by *node-sqlparser*. Aggregate functions handle distinct keyword and all columns represented by *. Functions with more than one argument are generated in the pre-parser step. The argument key in the corresponding JSON form has an array type to store each argument type.

Case expressions in *MySQL* and *PostgreSQL* are written with a value which is used for

Listing 3 Dictionary for storing clauses for later processing in the pre-parser step

```
Map {
     'rep_string_0' => { type: 'string', value: '\'released\'' },
     'rep_string_1' => { type: 'string', value: '\'before\'' },
     'rep_string_2' => { type: 'string', value: '\'same\'' },
     'rep_string_3' => { type: 'not null', value: 'not null' },
     'rep_string_4' => { type: 'expr_func', name: '',
     args: [ { type: 'column_ref', table: '', column: 'rep_string_0' } ] },
     'rep_string_5' => { type: 'case_expr',
     value: { value: { type: 'column_ref', table: '', column: 'rep_string_4' },
          cond_result:[ { condition: { type: 'number', value: '1987' },
    result: { type: 'column_ref', table: '', column: 'rep_string_1' } },
  { condition: { type: 'number', value: '1988' },
    result: { type: 'column_ref', table: '', column: 'rep_string_2' } } ],
  default: null } }
  }
```

comparison or as an expression condition statement. Both these forms are handled using a regular expression. The JSON form of the case consists of three key-value pairs. The first and third key-value pairs are optional, and have a value used for comparison and a default case if none of the condition expression is applicable. The second key-value pair has an array of condition expressions. Table 2.2 shows the definition of the JSON expressions for case statements.

Handling the `having` expression is different from the other pre-parsing steps. Instead of replacing an expression with a word, `having` is attached as a new key-value pair to the select statement. The `having` expression is similar to the `where` expression but it is applied on a `group by` expression for filtering aggregated results. Generating the JSON form for `having` leverages the way the `where` expression is parsed in *node-sqlparser*.

If the query has inner queries (a.k.a., subqueries), the inner queries are isolated and each is parsed as an independent simple query. The parsed inner queries are then attached to the final JSON as sub-expressions in the post-parser step.

Table 2.2: *JSON form of SQL clause*

SQL Clause	JSON
Identity	{type:<type>,value:<value>}
Column name	{type:'column_ref',table:<table_name>,column:<column_name>}
Table name	{db:<db_name>, table:<table_name>, as:<alias>}
Select query	{ type: 'select',distinct: null, columns: [<expression>], from: [<expressions>], where:<binary_expressions>, groupby:[<expressions>], orderby:[<expressions>], limit:[<start_limit>[,<end_limit>]], params: [<params_list>] }
Function call	{type:'expr_func',name:<function_name>, args:[<args_list>] }
Inner Query	{type:'inner_query',value:<json_query_expression>}
Having	{having:<expression>}
Case Expression	{type:'case_expr', value:{value:<value>,cond_result:[<sub_expressions>], default:<value>}}

2.3.2 Post-parser Operations

The pre-parser step simplifies the query to be parsed by the *node-sqlparser* package. The output has the corresponding JSON form and partial JSON forms of the SQL clauses that can't be parsed by *node-sqlparser* in a dictionary. The partial JSON forms consist of a string constant which is a key (i.e., an SQL expression) and a value which is the JSON form of the corresponding SQL expression. The post-parser function iterates through all the key-value pairs in the parsed JSON, by identifying the type of expression it encounters. Then, it checks whether a value is a string constant corresponding to a dictionary key which was replaced as part of the pre-parser step. If it is found to be a key, then it is replaced with the value from the dictionary. The function starts with the outermost part of the query and go into depth until each key is reduced to its JSON form recursively. After every key is replaced, the final JSON generated is used as the representation of the given query. The JSON for the sample query shown in Listing 1 is provided in Listing 4, and the post-parser algorithm is described in Algorithm 2.

Listing 4 Output of the post-parser step

```
{ type: 'select',
     distinct: null,
     columns: [ { expr: { type: 'column_ref', table: '', column: 'title' },
    as: null },
  { expr: { type: 'case_expr', value: [Object] }, as: 'output' } ],
     from: [ { db: '', table: 'albums', as: null } ],
     where:
      { type: 'binary_expr',
               operator: 'IS',
           left: { type: 'column_ref', table: '', column: 'released' },
           right: { type: 'not null', value: 'not null' } },
     groupby: [ { type: 'column_ref', table: '', column: 'title' },
                  { type: 'column_ref', table: '', column: 'released' } ],
     orderby: null,
     limit: null,
     params: [],
     having:
      { type: 'binary_expr',
           operator: '=',
           left: { type: 'column_ref', table: '', column: 'released' },
           right: { type: 'number', value: 1988 } } }
```

2.4 Query Evaluation Phase

The validation of the query is done by comparing the JSON of the student submitted query with the JSON of the solution query. The evaluation is carried out in two phases, one by comparing the output of the query, and another by the comparing the clauses in the query.

To compare the output of the query, columns and tables from both the parsed query and the solution query are used. Both these queries are joined using a full outer join, and the output will be used to see if there are any differences in the two solutions.

Individual columns, tables, and clauses such as `group by`, `having` extracted from student's submitted solution are validated by checking their presence among the expected SQL clauses extracted from the sample solution. If all the expected clauses are present, then the query is deemed as correct. Otherwise, feedback is provided with respect to which of the expected elements are missing.

Algorithm 2 Post-parsing algorithm

```
1: function postParseQuery(clause)
2:     if clause.type = identity then                          ▷ Basic datatypes
3:         return clause.value
4:     else if clause.type =′ column_ref′ then
5:         clause.type = postParseQuery(clause.value)
6:         return clause
7:     else if clause.type =′ case_expr′ then                  ▷ Case statement
8:         clause.value.value = postParseQuery(clause.value.value)
9:         for var clause in clause.value.condition_result do
10:            clause.condition_result = postParseQuery(clause.condition_result)
11:        clause.value.default = postParseQuery(clause.value.default)
12:        return clause.value
13:    else if clause.type =′ aggr_func′ then                  ▷ Function calls
14:        clause.args = postParseQuery(clause.args)
15:        return clause
16:    else if clause.type =′ binary_expr′ then                ▷ Conditions and filters
17:        clause.right = postParseQuery(clause.right)
18:        clause.left = postParseQuery(clause.left)
19:        return clause
20:    else if clause.type =′ expr_list′ then
21:        for var subClause in clause.value do
22:            subClause = postParseQuery(subClause)
23:        return clause
```

Both of these results are provided in the web browser for the students to review. Students can improve queries by correcting and re-testing, or can submit the query for evaluation which will be stored persistently for grading.

Chapter 3

Tool Evaluation

This chapter concentrates on testing the implementation as well as on improving the effectiveness of the grading of an SQL query. In Section 3.1, testing methods for verifying the working of modules, desired functionality, and end-to-end correctness are described. Then, the tool is tested on a set of queries, and quantitative and qualitative observations based on the results are described in Section 3.2.

3.1 Testing Life Cycle

Unit, integration and end-to-end testing are three different testing phases employed to check for bugs and verify the correct implementation of the project. The project is decomposed into files and each file has one or more functions dedicated to some specific functionality. In the unit testing phase, functions are tested by providing the function inputs and observing the return values. The mutated value in a variable is tracked to check whether the data is propagated as expected. The unit test cases are available for each function under its description and before the declaration in the source file.

In integration testing, data and control propagation from more than one function is verified. Functions residing in other files are tested. End-to-end testing involves testing of complete flow from the user input through the web browser, then a call to the server, reading

of data from file or database, processing of the data and finally the time until the response is received and rendered on the web page. End-to-end testing verifies the correct interaction between different technologies. Testing was conducted with queries similar to those that can be used in a relational database assignment to verify the behavior in a production environment.

3.2 Evaluating An SQL Assignment

The accuracy of grading an assignment is evaluated with a set of queries. The quantitative analysis provides the accuracy of the tool in recognizing correct and wrong queries. The qualitative analysis provides an elaborate description of the distinct observations from the testing.

3.2.1 Quantitative analysis

Table 3.1: *Assignment evaluation: quantitative report*

Class	**Number of queries**
Total queries	120
Parser error	30
Syntax error	2
True positive	62
False positive	0
False negative	5
True negative	21

The application was used for grading an assignment with six questions. The assignment questions were stored in a JSON file along with their answers as SQL queries. For each question, 20 different queries are used to test the behavior of the application. The responses are categorized into several classes as follows: a query that failed to parse (parser error), a query with syntactical errors (syntax error), wrong query identified as incorrect (true negative), correct query reported as correct (true positive), queries that are correct but the tool that failed to recognize the correct output (false negative) and queries that are wrong

but the tool identified them as correct (false positive). The table 3.1 lists the count of queries that fall into each of these categories.

In the testing phase, minor issues were identified and fixed in the parser, and the testing was resumed from the beginning. Issues which required major changes in the parser were the star "*" symbol in a query (which represents all columns), a query having back quote, double quotes for strings, a function call in projection without alias name and limit clause. These changes were performed on the query and tested. Out of 120 total queries tested, 40 queries were found with these issues. Queries were changed as a workaround for successful parsing. Major changes in the parser will be fixes that are taken up as future work.

3.2.2 Qualitative analysis

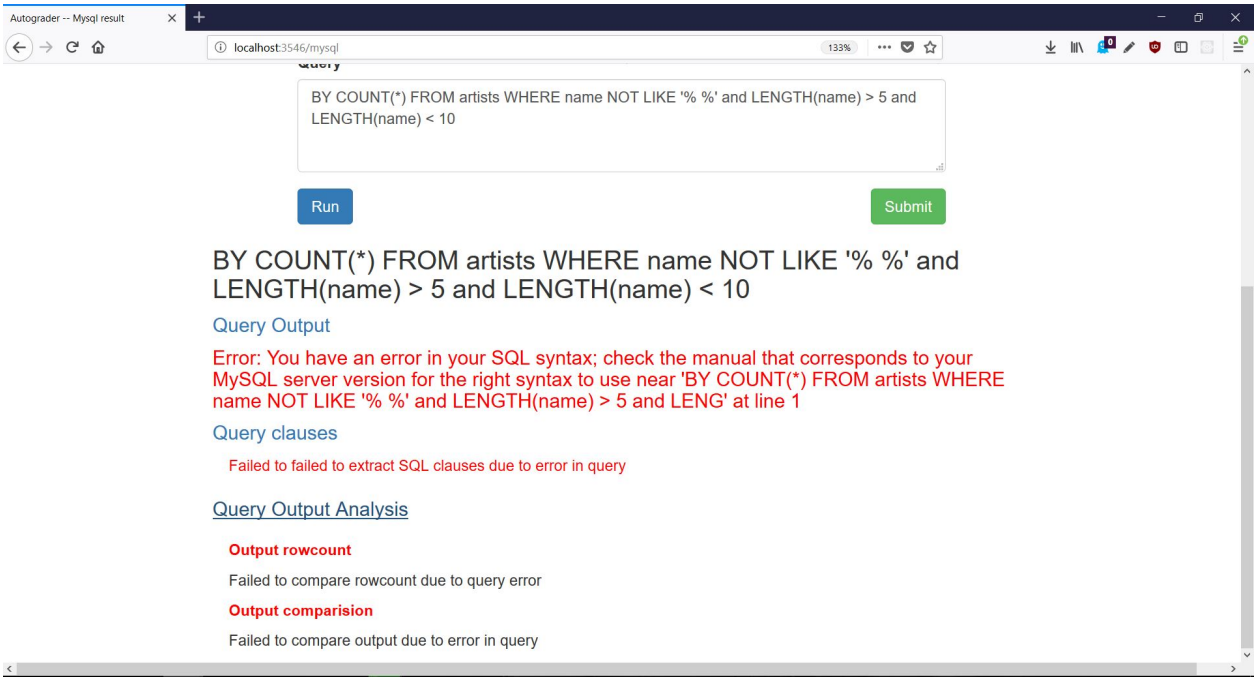

Figure 3.1: *Tool response on a query with syntax error*

The tool imitates the behavior of a database when a query with a syntax error is executed in it. The tool stalls the query parsing and analysis operations by displaying the error message from the database to the user. Figure 3.1 shows the result of the query with a syntax error where, `by` is used instead of the `select` keyword.

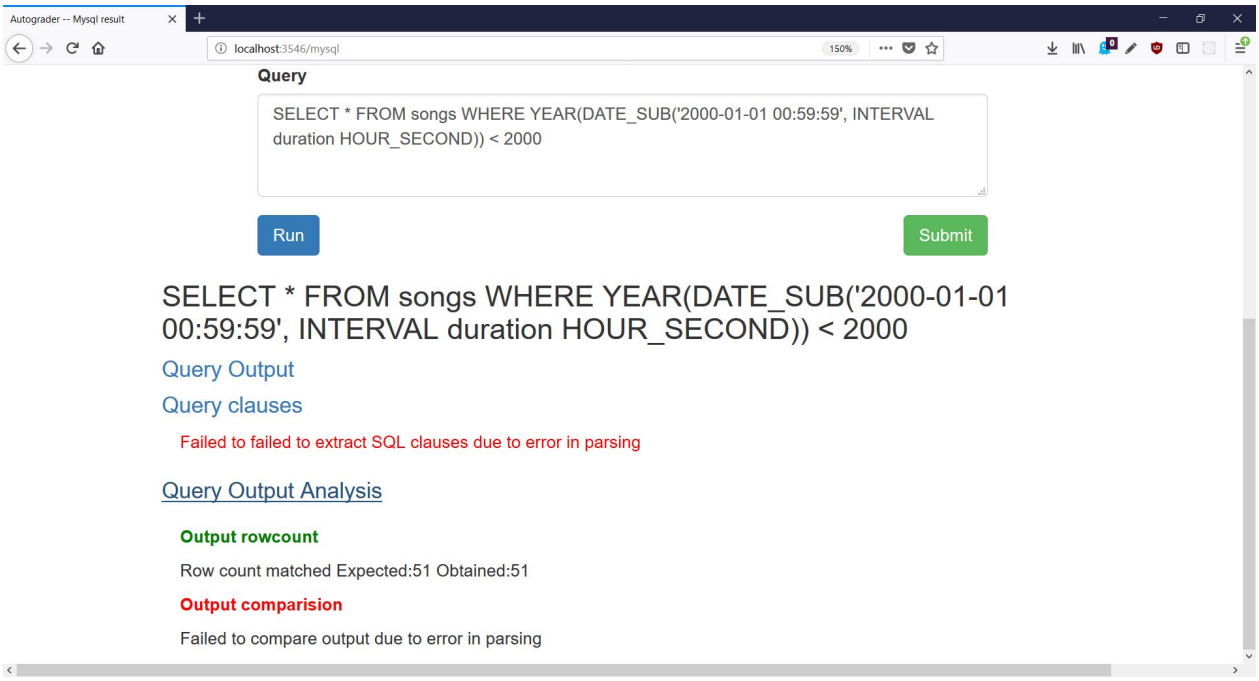

Figure 3.2: *Tool response on a valid query that failed to parse*

The parser fails with an exception when it encounters unexpected tokens. Even the valid SQL queries fail due to this limitation of the parser implementation. The query from Figure 3.2 is one such example, where the parser failed to process it. The parser was unable to interpret the `DATE_SUB` function. Comparison of the row count operation is independent of parsed JSON, hence the result of the query output and the row count alone is displayed.

A query parsed successfully will be analyzed for its output and each of its clauses. The tool displays the identified output and clauses, then analysis on it for row count and output comparison are displayed. Feedback will be provided on missing and undesirable tokens in the query. Figure 3.3 shows a query with all evaluation step successfully accomplished. The query has all necessary tokens and no undesirable ones, hence the query clause analysis displays no suggestion.

A valid query that yields an incorrect output will be revealed when comparing the row count of the sample query with the row count of the output. Figure 3.4 has a query where the filter was supposed to be on the `released` column, but it is on the `title` column. The query analysis provides feedback on the column expected in `where` clause.

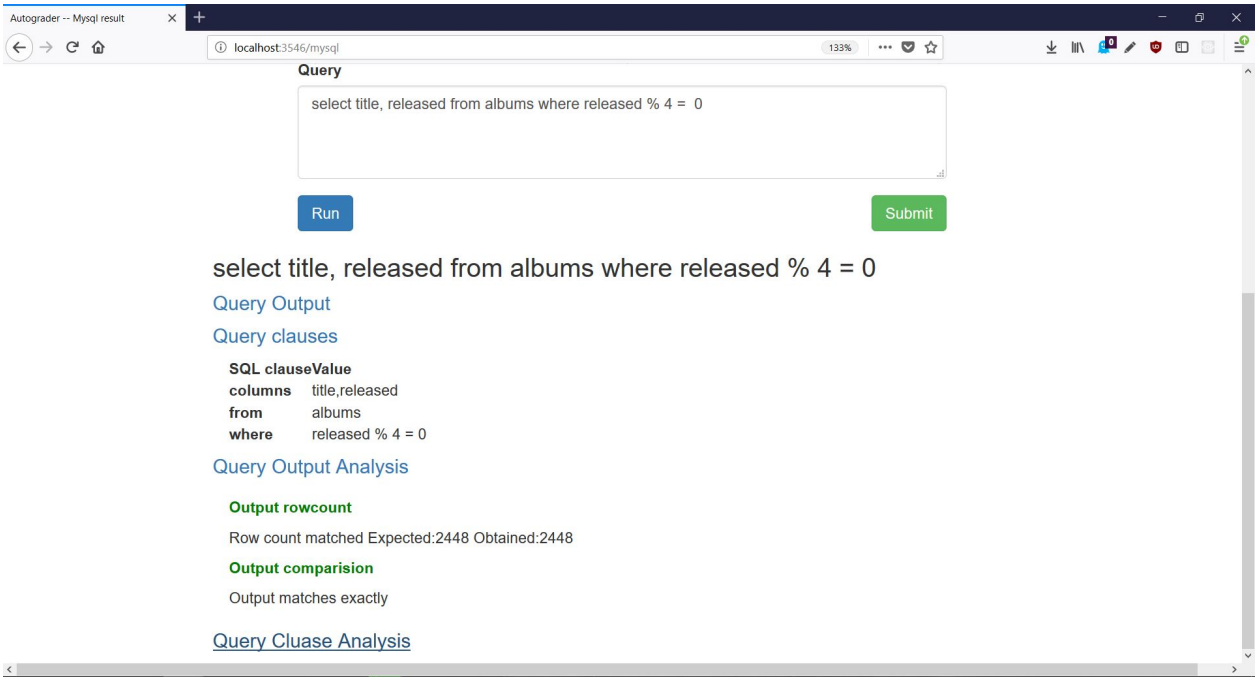

Figure 3.3: *Identifying a correct query and its output*

Figure 3.5 shows a false positive case of the evaluation tool. The select statement uses `album_id` instead of `artist_id` inside of an aggregate function. The tool failed to recognize that the output is the same irrespective of any column name used inside the count function.

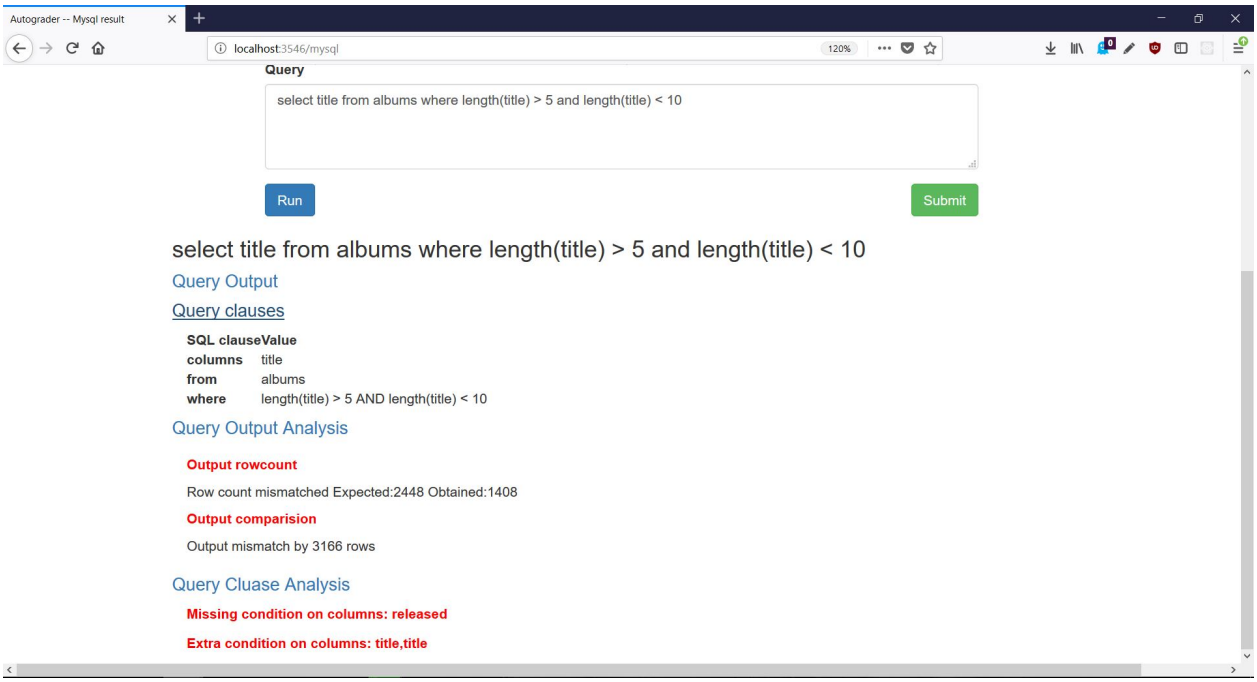

Figure 3.4: *Query with an incorrect result*

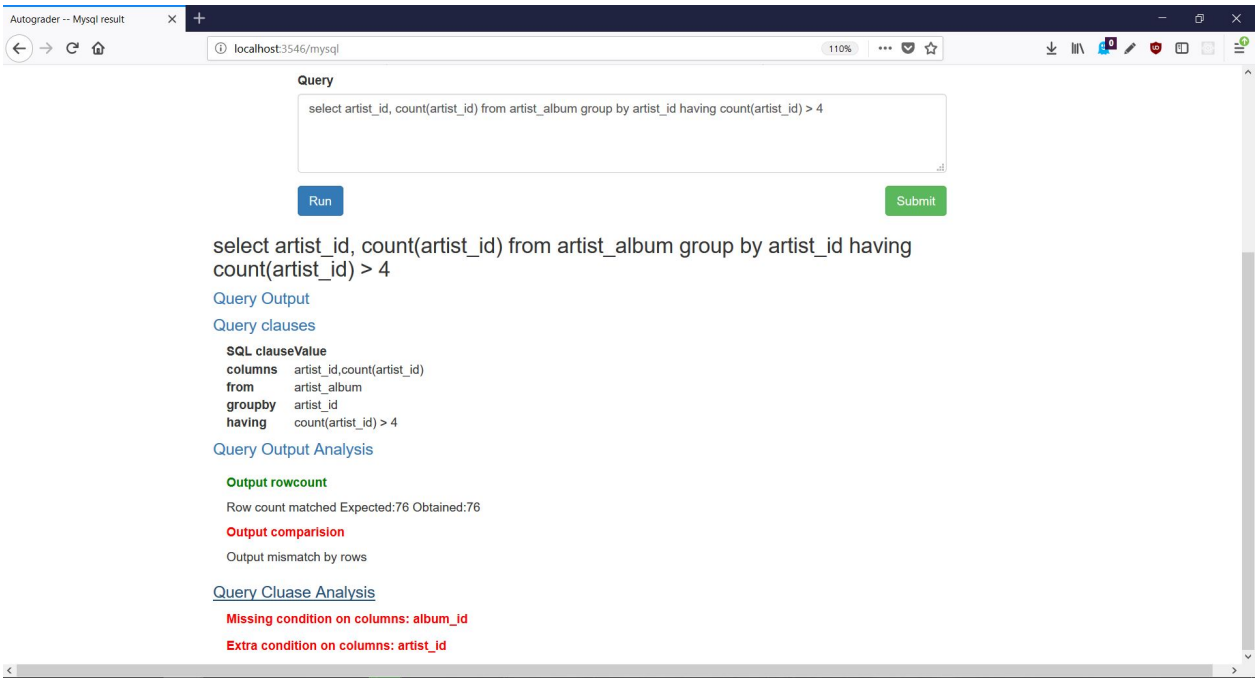

Figure 3.5: *Valid query interpreted as wrong*

Chapter 4

Tool Readability And Reusability

This chapter covers the usage of the tool, by providing details on the environment that tools requires and the steps involved in setting up the project and running it, in Section 4.1. Section 4.2 provides a style guide which enlightens on project structure, modularity of code and naming conventions. The SQL parser has other applications, some of which are provided in Section 4.3.

4.1 Project Setup and Execution

The project is built using the *npm* tool. The environment should have *Node.js*, *PostgreSQL* and *MySQL* databases configured, for the project to run. The project code is available for download at git repository (Kumar, 2018). The *Javascript* dependency packages for the project are imported by `npm install` command in the root directory of the project folder. Then, the server needs to be started with the `npm start` command. The default port is 3546. To use a different port, the port number is provided in the command as an additional parameter as `PORT=8080 npm start`. This sequence of commands are listed in Listing 5.

Listing 5 Steps on setting up the project

```
$ git clone https://github.com/vijaykumardev/SQLAutoGrader .git
$ cd .git
$ npm install
$ npm start
```

4.2 Style Guide

The project follows a standard convention to enable readability, re-usability, and extensibility. The standard naming convention is followed for the developer familiarity and uniformity within the project. Constants are used with all uppercase and underscore delimited, constants used to store library imports are in lowercase without any spacing, functions, files and variable names are in camel casing.

Listing 6 Coding conventions followed

```
/**
 * @function checkValueType
 * @description check the type of the value stored in a variable
 * by identifying it's match to regular expression condition
 * @param {String} identifier
 * @return {"null"|"bool"|"number"|"column_ref"|"string"}
 * returns the type of value stored in the variable
 */
function checkValueType(identifier){
        if(NULL_REG_EXP.exec(identifier))
            return 'null'
        else if(NOT_NULL_REG_EXP.exec(identifier))
            return 'not null'
        else if(BOOL_REG_EXP.exec(identifier))
            return 'bool'
        else if(NUMBER_REG_EXP.exec(identifier))
            return 'number'
        else if(STRING_REG_EXP.exec(identifier))
            return 'string'
        //If the value is of extended sql JSON type
        else if(COLUMN_REG_EXP.exec(identifier))
            return 'column_ref'
    return 'string'
}
```

Strings are wrapped around single quotes unless strings contain quotes in themselves, in which case, the SQL queries are wrapped using double quotes. Line delimiter such as semi-colon for *Javascript* statements are omitted. Block statements delimited by curly braces are excluded if the block contains just one *Javascript* statement. *ECMAScript* 6 arrow function is used for anonymous function declaration. JSON is used as a return type for returning more than one variables. Comments for each function have a description, parameters used, return value, function's type and function call with the sample argument value. An example of the convention used is shown in Listing 6.

4.3 Additional Application

The extended SQL parser has potential for a wide range of applications. The parser will allow users to quickly verify the syntax of SQL queries without any active database connection. It can also enlighten the user about all the tables required to run a query. This is helpful for setting up a new database, rather than using the brute force approach to finding the missing database components. The parser will give the list of database components.

Finally, the parser can be used as a channel for learning the SQL language. For specific values in the SQL clause, the parser using its JSON representation will generate equivalent SQL queries, which the user can use to test the behavior and explore the SQL language.

Chapter 5

Future Work and Conclusion

The portal enables the student to get access to assignment questions, test their queries, check query correctness and submit queries for evaluation. The query parser identifies the constructs and clauses in the query such as tables, projections, filters, conditions, `group by`, `order by`, `having`, `functions` and `case` expressions. The correctness of a query is validated with the help of a query parser by comparing the presence of expected clauses from the solution query in the query submitted by a student.

The functionality of the query parser is limited to identifying and parsing SQL constructs and clauses in one SQL query. Combinations of two or more queries using `union`, `intersect`, `except` are not supported. The parser works only on valid SQL statements. Issues in the parser that need major code fixes are described in Section 3.2.1. Implementing these fixes will enable query evaluation without a workaround. Identifying the syntax correctness will enable better feedback for users. Improving functionality by analyzing the query statement on the usage of indexes, removal of duplicates, restructure of the query, best practice structures and explain plans will help students to not only write a correct query but also an efficient query. This can be done as an extension in future work.

Furthermore, the user interface can be further enhanced to make it more accessible by providing meta-data information about the query. The interface can be redesigned for a cleaner, dynamic and uniform user experience. For front-end rendering, *EJS* can be replaced

with *Vue.js* to meet the redesign goals in future work.

The full-functionality portal will contribute to a greater learning experience of the students, by allowing them to concentrate their time and effort on enhancing the knowledge of writing SQL queries and minimizing the time spent on database configuration or environment setup. The portal will also give swifter feedback on assignment submissions. For the graders, it will automate mundane tasks and streamline the process of assignment grading.

Bibliography

Chandra, B., Joseph, M., Radhakrishnan, B., Acharya, S., and Sudarshan, S. (2016). Partial marking for automated grading of sql queries. *Proceedings of the VLDB Endowment*, 9:1541–1544.

Cheang, B., Kurnia, A., Lim, A., and Oon, W. (2003). On automated grading of programming assignments in an academic institution. *Computers & Education*, 41:121–131.

Dekeyser, S., de Raadt, M., and Lee, T. (2007). Computer assisted assessment of sql query skills. *ADC Conference*, 63:53–62.

Fish (2015). node-sqlparser. https://www.npmjs.com/package/node-sqlparser.

Kent, A. (2015). sql-parser. https://www.npmjs.com/package/sql-parser.

Kumar, V. (2018). SQLAutoGrader git location. https://github.com/vijaykumardev/SQLAutoGrader/.

Russell, G. and Cumming, A. (2005). Automatic checking of sql: Computerised grading. *International Journal of Learning*, 12:127–134.