A Static-Based Approach to Detect SQL Semantic Bugs

Master's Thesis

Claudiu Ion

A Static-Based Approach to Detect SQL Semantic Bugs

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Claudiu Ion
born in Bucharest, Romania

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
`www.ewi.tudelft.nl`

A Static-Based Approach to Detect SQL Semantic Bugs

Author:	Claudiu Ion
Student id:	5105080
Email:	`c.ion@student.tudelft.nl`

Abstract

While SQL engines are now capable of detecting a large number of syntactic mistakes, most often semantic errors are not detected, which can lead to serious performance issues or even security vulnerabilities being introduced in the system. This thesis proposes a set of 25 validated heuristics together with a new rule-based static analysis tool for detecting the most common types of semantic bugs in SQL queries, based on evidence from previous research. We conduct an empirical study on the prevalence of semantic bugs in SQL on two datasets with queries collected from different open source industry projects as well as on a large dataset of queries collected from StackOverflow posts. Manual analysis of more than 500 queries shows that our tool is able to detect semantic bugs in SQL queries with an accuracy of 97%. Furthermore, out of all 191,994 collected queries, we identified a total of 36,818 queries which contain at least one semantic bug, meaning that 19.17% of queries contained some semantic problem in their formulation. To the best of our knowledge, this is the largest dataset of SQL queries extracted from StackOverflow and could later be used for subsequent studies as well.

Preface

As I write these final words of my master thesis, my mind and heart are filled with the memories and emotions I had when I first embarked on this great adventure to a new country, when I came to study in The Netherlands. For three years, I got to call Eindhoven my home. Over there, while doing my bachelor studies, I had the great opportunity of meeting very talented and dedicated professors, as well as other fellow student colleagues, whom I later got to call my friends. My journey then continued when I moved to Delft, where I started my master studies. Here, I again got the chance to meet some amazing people whom I would like to thank in this preface.

To my supervisor, Maurício Aniche, I would like to express my sincere gratitude for all the help and time he has dedicated to me. His support during the literature review and throughout the thesis, over the past nine months, was unparalleled. He has provided me with the best guidance I could have asked for. I am very grateful for the enlightening explanations and support material that I have received from him, for the impromptu meetings and our regular progress calls. I would also like to thank the members of my thesis committee, Arie van Deursen and Christoph Lofi, for their time and effort. Having both as professors during my studies at TU/Delft, I would like to thank them for their hard work and dedication, especially during these difficult times.

Additional thanks go to my dear friend, mister Tudor, for his entire support during my master studies. I could always count on his constructive feedback, insightful remarks and keen eye for detail.

I would also like to thank my colleagues at Optiver, where over the past two years I have been working part-time. Ever since I started my master studies, they have always shown great understanding and curiosity towards my university work, and for this I would like to thank them all.

Finally, I would like to thank my family for their constant support. I am very fortunate to have an elder brother whom I can always count on. I can not thank him enough for listening to my monologues about the thesis and not only, during our routine evening walks, and I know that I can always rely on him to hear me out. I would also like to thank my parents for their unconditional love and support.

Even though you have been far away, you were always there for me and I am forever grateful for this.

Thank you all for being part of my journey in becoming a computer scientist!

Claudiu Ion
Delft, the Netherlands
June 22, 2021

Contents

List of Figures

Chapter 1

Introduction

The Structured Query Language, also known as SQL, is a domain specific programming language used for managing and interacting with relational database management systems. It was also one of the first languages to utilize the relational model presented by Codd [12] in his paper and later on became both an ANSI and ISO standard, making it the most widely used database language, with more than 70% of developers using SQL as shown by a recent study [38].

What makes it even more impressive, is the fact that SQL is not only used in IT but also in various other industries, such as banking, accounting, aviation or commerce. This makes this programming language one of the most widely spread and at the same time it means that there are a lot of people, with different skill levels and backgrounds, writing and using SQL queries. Depending on the understanding level of SQL, some users might directly use queries from forums or other questions and answers websites, such as StackOverflow, which has inherent risks if these queries contain bugs. For this reason, it is important to therefore have tools which can assist developers in detecting not only syntactic errors in SQL queries, but also semantic ones, just as modern integrated development environments suggest code reformatting for various other programming languages.

Currently, the tools which exist for SQL are mostly focused on detecting only syntactic errors in queries that developers write. While this is still an important aspect, queries should also be checked for semantic issues, since these types of problems can have a huge impact on the system's performance, as we will later explain in this study. Furthermore, most of the existing tools require the complete database schema in order to analyse queries and detect potential semantic errors. Another important point is that most of the current tools are not able to detect issues in dynamically generated SQL queries either, which is very limited since applications might create queries at runtime depending on various input as described by Gould et al. [21].

In the background study that we carried out for this thesis we also identified the lack of open source tools focused on detecting semantic bugs in SQL queries. The lack of such tooling, in turn, makes it harder for developers to identify these issues. As pointed out by Ahadi et al. [1] in their study on semantic bugs in student written queries, these types of issues are the ones requiring more knowledge and a deeper understanding of the underlying SQL principles in order to solve. Furthermore, these bugs are not addressed as long as the query contains syntax errors, however, most of the time even after the query formulation is correct, developers tend to overlook the final step of checking for potential semantic bugs. Therefore, having automated tools for checking this would offer tremendous help in solving this problem.

The goal of this thesis is two-fold. First, we propose a set of 25 validated heuristics for detecting the most common semantic bugs encountered in SQL queries, based on evidence from previous research. We then implement a tool, using our proposed heuristics, for detecting semantic bugs in SQL queries and measure the accuracy of the tool by performing an extensive manual analysis on a large collection of queries. Second, we explore the prevalence of these semantic bugs in two different SQL datasets, one provided by Castelein et al. [11] containing queries from three different open source projects tracked on GitHub and the other, which we specifically built for this study, containing queries extracted from StackOverflow posts. To the best of our knowledge, this is the largest and most complete dataset of SQL queries collected from StackOverflow, containing not only queries but also various other additional metadata information, which could be used for other interesting research in the future as well.

Our rule based static analysis tool is able to detect semantic bugs in SQL queries with a 97% accuracy. Furthermore, we observed that out of all the 191,994 collected queries, 36,818 contain at least one semantic bug, which means that 19.17% of queries contained some semantic problem in their formulation. Also quite interesting, we show that more complex queries, in terms of number of joins, predicates and functions used, tend to suffer from more semantic bugs, an interesting finding which could be used in the future as a metric for early prediction as to whether a query might contain semantic bugs or not.

Our thesis leads to the following four contributions:

- A set of validated heuristics to detect semantic bugs in SQL queries together with a prototype implementation of a tool which implements the proposed heuristics and detects semantic bugs in SQL queries.
- An empirical study on the prevalence of semantic bugs in two SQL datasets.
- A dataset of 172,000 queries extracted from StackOverflow together with additional metadata information.
- A replication package containing the queries and other scripts used in our evaluation process that can help with reproducing and improving our results.

Chapter 2

Background

SQL is a domain specific language used in handling structured data held in a relational database management system. In this chapter, we discuss technologies related to the thesis and give background information to the problem at hand. This chapter is divided in two main sections. In Section 2.1 bugs in SQL are explained and related work on the topic of identifying semantic bugs in SQL is shown. In Section 2.2 the importance of StackOverflow as a source for collecting relevant data is discussed and previous research on this topic is presented.

2.1 Bugs in SQL queries

We organized the collected previous work related to bugs in SQL queries in four different categories: semantic bugs in SQL, teaching systems for SQL, empirical studies on SQL queries and testing SQL queries. In the following subsections we present each category together with the most relevant work that we identified. We give an overview of previous studies per research area in Table 2.1.

Research area	**References**	**Qty**
Semantic bugs in SQL	[9] [10] [44] [8] [23]	5
Teaching systems for SQL	[29] [30] [31] [7] [43] [42]	6
Empirical studies on SQL queries	[1] [2] [34] [41]	4
Testing SQL queries	[11] [36] [37]	3
Total		**18**

Table 2.1: Literature collection related to bugs in SQL queries

2.1.1 Semantic bugs in SQL

The work of Brass and Goldberg [9] and Brass et al. [10] was the first to focus on the class of SQL queries which are correct from a syntactic point of view but contain semantic issues, regardless of the query task. The authors classify SQL errors in two groups, one related to syntactic issues and the other to semantic ones. In the first group, the database management system will detect the error since the query can not be executed at all. These errors are usually easier to detect and fix since the developer gets an immediate error or warning when trying to run the query. In the second group, the queries are correct however they do not produce the intended results. These types of errors are harder to detect and fix since the problem is not immediately obvious.

In their work, the authors further classify semantic errors in two groups, one for which the task must be known beforehand and another where the single SQL query is enough to detect semantic issues. The focus is then placed on the second group and this is also the case for this thesis. The main contribution of their work represents a taxonomy of semantic errors which frequently appear in SQL queries collected from homework and exam materials for one of the database courses at the University of Halle.

Apart from this list, Brass and Goldberg [8] carried out another study for analysing two other specific types of semantic issues that appear in SQL queries, more specifically inconsistent conditions and queries which might generate runtime errors. One potential threat to validity for their work is represented by the limited size of the dataset on which the study was conducted, more specifically the errors were extracted from six SQL questions that appeared on two exams with roughly 150 participants. It is worth pointing out that Goldberg [20] carried out a study using the same taxonomy of semantic bugs on a larger dataset, however the queries were still extracted from previous exams. In this thesis, apart from implementing a tool for detecting these semantic errors, a study on a large dataset of queries extracted from real-world applications from StackOverflow is also conducted which brings further insight into how these issues appear.

An extensive taxonomy of SQL code smells is also presented by Karwin [23]. In his book, the author presents the most frequently encountered SQL antipatterns and pitfalls. The provided list of issues also takes into account the most common SQL problems which the author encountered while answering various question posts on online forums, mailing lists and newsgroups over a time period of more than 15 years. The book places the antipatterns in one of the following categories: logical database design, physical database design, query related and finally application development issues. In the query related category, we find the most common types of semantic issues encountered in SQL queries (also defined as code smells), more specifically a list of six query antipatterns: fear of the unknown, related to the special nature of `NULL` in SQL, ambiguous groups, which relates to the use of the `GROUP BY` clause, the random selection antipattern, related to the issue when some random row is selected from the database and finally the implicit columns issue which refers to use of the select all star operator (*) in `SELECT` statements instead of providing an explicit select list. A large number of subsequent studies are based on the taxonomy provided in this book in an effort to both analyse as well as better understand how and why these issues appear in SQL queries.

2.1.2 Empirical studies in SQL education

Other similar studies presented by Mitrovic [29, 30, 31] and Bider and Rogers [7] focus on building knowledge based teaching systems for SQL. The authors in these studies present two very similar systems which are intended to be used as guided discovery learning environments for SQL. The motivation behind building such systems is that although SQL is a relatively simple and highly structured language, students still have many difficulties when first learning it, therefore such intelligent teaching systems aim to overcome these difficulties and serve as an additional practice environment for SQL learners. Furthermore, both of the presented systems focus on detecting semantic errors in SQL queries since syntactic problems can already be detected by the database engines.

Moreover, the two approaches presented in these papers also require having knowledge about the task which has to be solved, in the form of a correct query for a given SQL question. In contrast, the approach taken in this thesis assumes no such knowledge which makes the tools and algorithms work for queries written at large, not only in a teaching environment. The system built by Mitrovic [29] only works on `SELECT` SQL statements while the one presented by Mitrovic [31] works for `SELECT` and `CREATE VIEW` statements. The approach taken in this thesis is such that the rule-based semantic error detection tool should work on all SQL statements. Finally, in both of the previously mentioned studies, the presented tools for detecting semantic issues in SQL were tested on fairly small datasets which only contained SQL statements originating from assignments or exams, nevertheless the research did show the great potential these tools have in a teaching environment, helping most students better understand the SQL syntax.

In Taipalus and Seppänen [43] and Taipalus and Perälä [42], further SQL teaching practices in higher education are also presented as well as an overview of educational SQL research topics and most frequent issues which appear in SQL queries written by students. As with other previous research, the authors note that future studies should be carried out on more diverse datasets, with queries from real world projects, in an effort to understand whether their findings generalize on software at large as well, an aspect which we try to address in this thesis.

2.1.3 Empirical studies on SQL queries

Another interesting study that focuses on conducting the first large-scale quantitative analysis of syntactic errors that appear in SQL queries written by students is presented by Ahadi et al. [1]. Apart from going over the most common mistakes that students make, the study also describes how an automatic classifier can be used in predicting the performance of students when writing SQL statements. Although the paper is not directly concerned with analysing semantic errors in SQL, the authors point out that in most cases students abandon answering an SQL related question due to a syntactic error. Their findings show that syntax errors in different clauses of the `SELECT` statement, undefined referred column errors and grouping errors are the most common types of mistakes that novices SQL users make when writing their statements.

It is therefore interesting to point out that while semantic errors are the ones requiring more knowledge and a deeper understanding of the underlying SQL principles in order to solve, students will most certainly not come to grips with these while the query formulation contains syntax errors, as the authors of the study also conclude.

Furthermore, in a subsequent study Ahadi et al. [2] also conduct the first large-scale analysis of semantic mistakes in SQL `SELECT` queries written by students. The dataset used in this paper contained queries pertaining to seven different key SQL concepts: `GROUP BY` without `HAVING`, self `JOIN`, `GROUP BY`, natural `JOIN`, simple subquery, simple query with only one table and finally correlated subqueries. Again, the study also made use of knowledge related to the question for which the SQL statement was written, therefore a query was considered semantically incorrect when the result set returned by it was different from the one returned by the accepted solution query. The authors find out that the majority of semantic bugs are of type omission, showing that students have trouble in selecting the correct type of query to use as well as lack a systematic approach to formulating the query. Queries that require a `JOIN`, a subquery or a `GROUP BY` are the most susceptible to these types of omission errors. Furthermore, results also show that the majority of semantic mistakes occur in the `WHERE` clause of the `SELECT` statement which, as the authors point out, might be the result of memory overload when dealing with complex queries.

Another large-scale study on SQL code smells, which investigates the prevalence, impact, evolution and co-occurrence of SQL code smells with traditional code smells was conducted by Muse et al. [34]. In their work, they analysed 150 open source software projects which made use of popular database access APIs such as JDBC, JPA and Hibernate, in an attempt to investigate how often SQL code smells appear in these systems.

While traditional code smell detection has been an important research topic for several years now, recently there has been growing interest in analysing and detecting such code smells in SQL code. The authors of this study make use of the SQLInspect tool and focus on analysing four SQL code smells. These code smells were selected from the extensive catalog published by Karwin [23], which is the first book to present a comprehensive list of SQL antipatterns. More specifically, the authors of the study looked at implicit columns code smells, where the select all operator (`*`) is used instead of a smaller list which has implications on network bandwidth wastage, improper handling of `NULL` values smells, where comparison operators are used instead of the `IS NULL` or `IS NOT NULL` expressions, ambiguous grouping code smells, where `GROUP BY` clause terms are misused and finally random selection smells when developers query a single random row wrestling in a full scan of the table.

The findings of the research indicate that SQL code smells indeed exists persistently in data-intensive systems, however these are independent from other traditional code smells. The most prevalent code smell was the use of the select all operator, the implicit columns usage smell. Also quite an interesting finding is that SQL code smells have a weaker association with bugs than traditional code smells as well as the fact that SQL smells are more likely to be introduced at the start of the project, and are not addressed as quickly as other more traditional smells, which might also be due to the fact that tools for detecting and informing developers about these issues are not yet in place or not widely adopted enough.

In a recent study by Taipalus [41] the author presents a detailed analysis on the types of SQL errors which are most likely to be encountered in SQL queries formulated by novice users. Furthermore, the author also attempts to explain the causes behind the most common errors. The study uses a taxonomy of different syntactic and semantic SQL errors previously defined by Taipalus et al. [44] in their work on errors and complications in the formulation of SQL queries as well as a dataset of queries collected from four iterations of an SQL course. The reported results show that the three most common errors were missing expressions, extraneous or omitted grouping columns and missing join statements.

Moreover, the authors observed that the cause of the error is strongly related to the query concept rather than the error type. The study finally recommends a way to mitigate the three most commonly encountered errors in SQL queries by teaching students techniques for recognizing natural language patterns and their SQL equivalents, before proceeding to writing more complex queries.

2.1.4 Testing SQL queries

An interesting study on generating test data for SQL queries is presented by Castelein et al. [11]. As with any other piece of code, testing is of utmost importance in order to guarantee that software is working correctly, and this should also be the case for applications involving SQL queries. In their work, the authors model the problem of generating test data for SQL queries as a search-based problem and present three novel approaches, based on search algorithms, to solving this problem. More specifically, the paper proposes using genetic algorithms, random search and biased random search and analyses each approach on three open source projects and one industrial software system.

The algorithms take as input some SQL query to be tested as well as the database schema and use the three different techniques for populating the tables with test data which should meet certain testing criteria. The results show that the genetic algorithm approach is able to cover 98.6% of all queries in the dataset and also has the advantage of being able to properly test queries with `JOIN` statements, multiple subqueries as well as string manipulation functions. One of the main contributions of the research was the implementation of the EvoSQL tool for generating test data for SQL queries as well as an empirical study on the effectiveness and performance of the tool on a dataset of queries extracted from four different software projects.

Furthermore, the study shows that full coverage for one query can be reached in almost all cases within 2 to 15 seconds, which makes the tool usable in a practical setting as well. The dataset collected for this paper will be used for testing the rule-based static analysis tool that we will present in this paper, in order to further analyse the prevalence of semantic bugs in queries extracted from different software projects.

In two other studies by Nagy and Cleve [36, 37] we find the first attempt at implementing a tool for identifying code smells in SQL queries embedded in Java code. The tool uses a combination of static analysis techniques, together with database schemas as well as the data present in the database in order to detect potential semantic issues in SQL queries extracted from source code.

Furthermore, depending on the context of the smell, the tool is also able to determine its severity. What is interesting for this research, is that the tool is provided both as a command line interface as well as an Eclipse plug-in, which should help with providing better support for developers as well as speed up adoption of the tool, something which most previous research in this area is lacking.

It is worth mentioning that in Nagy and Cleve [37] the authors provide an extensive evaluation of their tool on a number of open source Java projects of various complexity and size, with results showing the potential that such tools have for inspecting SQL queries. The SQL code smells implemented by the authors of these studies again come from the book by Karwin [23] which provides a list of six common SQL antipatterns.

Interestingly enough, the authors decided to not implement a detection rule for the issue concerning the unnecessary index scan, when a wildcard character is used at the beginning of a string for the `LIKE` expression, which results in a missing access predicate, however our tool does detect this issue as well. The SQL code smell detector presented by the authors takes as input some SQL statements extracted from a source code file and optionally the schema of the database along with any database contents that might be present. After this, both the abstract syntax tree (AST) as well as the abstract semantic graph (ASG) are constructed, on which the implemented rules for detecting the various smells are run. As future work, the authors plan on extending the list of supported code smells for the detection tool as well as adding more features to the Eclipse plug-in.

2.2 SQL data collection from StackOverflow

With more than 10 million visits per day[1] and 14 million active users, StackOverflow has become the main questions and answers website (Q&A), related to programming, that exists on the internet. It was launched in 2008 and has quickly grown to now contain 21 million questions, 31 million answers, with an answer rate of 70%

[1]`https://stackexchange.com/sites?view=list`

and an average of 6500 questions new questions being asked on the platform every day. This huge popularity means that StackOverflow is now a large knowledge base for several programming languages and hence it represents a valuable resource for researchers to tap into. There have been numerous papers on extracting various types of information from StackOverflow, however only recently there has been some growing interest in extracting SQL related information from this website.

In Nagy and Cleve [35] the authors present the first attempt at extracting SQL related information from StackOverflow, more specifically queries. The methods presented in the paper are the closest to our approach, however the authors make use of data dumps of StackOverflow provided by Stack Exchange in XML format, as opposed to our approach which makes use of real time data from the platform by taking advantage of the Stack Exchange API. Furthermore, the algorithm for extracting the SQL queries from the code block sections that we use for building our dataset is also different than the one presented in the paper. Finally, the study looks at whether the extracted queries are error prone by manually investigating the queries and reporting on some general descriptive statistics such as the number of joins per query or functions used. As the authors of the paper also conclude, the queries should further be investigated with automated tools for properly detecting issues, which is something that our study covers. Nevertheless, this does show the huge potential of the StackOverflow platform in extracting a large number of SQL queries for various research topics. Moreover, this should help in bridging the current gap which exists in this research field, in the sense that most studies on identifying semantic issues in SQL queries are conducted on datasets which come from SQL courses, where queries are written by students on either homework assignments or exams. By using these new mining techniques for extracting queries from StackOverflow this should now allow for building better datasets and obtaining new insight as to what types of issues might appear in queries found in real world applications.

Another study by Allamanis and Sutton [3], uses information extracted from StackOverflow questions in an attempt to understand which programming concepts are the most confusing and whether there are any similarities across different languages. For this study, a number of programming languages were considered, among which Java, Python and SQL. By looking at the top words used, the authors were able to extract the concepts of a question post and build two categories one related to programming concepts and the other related to the type of information that users were looking for. Each question was then placed in either of the two categories. Furthermore, the questions were also clustered by using topic models, with one of the main findings of the paper being that the distribution over question types is not different among programming languages. This means that the types of questions being asked do not vary across programming languages. Interestingly, a method was also presented for how to identify what question types were mostly associated with a certain programming language, such as SQL for example, as well as a distribution of what are the most popular times of the week when users ask questions. As the authors of the study also conclude, this type of insight could be useful for future IDEs or smart documentation systems for providing guidance to developers depending on the context.

In Zhang et al. [47] a study on C/C++ code weaknesses found in StackOverflow posts is presented. The aim of the paper is to analyse common weakness enumeration errors (CWE) [27, 28] in code snippets found on StackOverflow. This is especially important since such code might for example contain security vulnerabilities and sharing it carelessly might introduce these problems in other systems. The authors find out that 36% of the CWE issue types are detected, particularly the improper restriction of operation within the bound of a memory buffer. More alarming, the study shows that the proportion of code which contains issues on StackOverflow has doubled from 2008 to 2018. For building the dataset, the authors first extract code snippets from StackOverflow answers tagged with the C/C++ tag. Furthermore, snippet codes with less than 5 lines of code are removed in order to reduce bias for frequent code. Finally, the paper uses the Gueslang[2] open source tool for detecting which of the extracted code snippets actually contain C/C++ code. This tool is shown to detect with a 90% accuracy the programming language from a list of 20 pre-defined languages including C/C++. The authors also analysed the revision history of answer posts in order to detect whether issues are also corrected when detected by other users, which is not always the case or in some situations this happens with a big delay. This study therefore brings into discussion the issue of sharing bug-induced code on StackOverflow, which inexperienced developers might directly use in their applications without realizing the implications of potentially introducing errors or semantic issues into their systems. Prior studies by Wu et al. [45] and Fischer et al. [19] show that code snippets on StackOverflow are indeed widely shared and used by developers and look into the implications that arise from this. Moreover, Baltes et al. [4] build an open dataset containing code extracted from StackOverflow in order to provide a means for understanding how these snippets are maintained and evolve over time.

Dietrich et al. [14] present an interesting study on detecting the programming language used in StackOverflow code snippets. The authors look at two different methods for classifying the language used. The first one uses metadata provided by users such as tags and other filters. The second approach uses the GitHub linguist tool[3], which according to the developers is an industry-strength library for automatically classifying code and identifying the underlying programming language used. The authors of the study remark that both approaches proved to be scalable enough to be employed on diverse code repositories of various sizes. The results of the paper however show that the two approaches produce results which are often not consistent, which indicates that these tools should be used with caution. This further indicates that a hybrid approach, which combines the strengths of both methods, user provided metadata as well as automatic detection tools, should be investigated in future research.

[2] `https://github.com/yoeo/guesslang`

[3] `https://github.com/github/linguist`

In Ponzanelli et al. [40] the authors construct a full island grammar which is capable of modelling Java tagged StackOverflow posts. The intention of the research was to build a heterogeneous abstract syntax tree (H-AST) for each post type in StackOverflow, that is questions, answers and comments, which is then used to construct a structured dataset for the posts information, to be used in future research. This solves one of the main problems with StackOverflow data mining, more specifically the heterogeneous nature of data which can be found in posts. Using the provided dataset, the contents of a post can be navigated by differentiating between code, which in this case is Java, and natural language fragments. Moreover, the authors also provided other meta information for each post in the dataset such as term frequency vectors and mentioned terms. In a previous study Ponzanelli et al. [39] present an interesting Eclipse plugin which leverages information from StackOverflow posts and provides assistance to developers using the active context in the IDE. More specifically, the plugin formulates queries to StackOverflow using the information available in the active context of the IDE and then presents a ranked list of results to the developer who can then directly insert the retrieved code from the posts.

Previous research has also used StackOverflow data to gain insights into trends and most common topics of interest in the development community. Barua et al. [5] and Linares-Vásquez et al. [26] use LDA modeling techniques to discover the most frequent topics present in post discussions. The studies also analyse these topics as well as their evolution over time and show the increasing popularity of web development and mobile applications related discussions. Similarly Beyer and Pinzger [6] use manual techniques to categorize Android related posts on StackOverflow and find dependencies between question types and certain problems, providing a better understanding of the most frequent issues appearing in mobile app development. This again shows the great potential of using the huge knowledge base that StackOverflow has become in extracting interesting information for various studies. Although we do not provide this analysis in our paper, it would be very interesting to use our collected dataset and perform LDA analysis on it in order to identify the main topics of interest for SQL related questions present in StackOverflow posts. Furthermore, it could also be very insightful to perform manual analysis on some of the posts for which our tool reports semantic issues found in the extracted queries, in order to understand whether developers notice the presence of these types of SQL issues as well as if the errors are addressed over time. Finally, Khatoon et al. [24] provide a complete overview and evaluation of current source code mining tools and techniques, mostly focused on sources such as code tracking websites or other APIs, however relevant nonetheless in presenting different methods for source code extraction. Other studies on mining source code from various APIs are also presented by Lamba et al. [25], Xie and Pei [46] and Kagdi et al. [22].

Chapter 3

Heuristics for SQL Semantic Bug Detection

In this chapter we present in Section 3.1 a detailed description of semantic bugs found in SQL queries as well as detection strategies for these in Section 3.2. Furthermore, details about the implementation of the rule-based static analysis tool for detecting these bugs are given in Section 3.3. The process used for validating the tool is presented in Section 3.4 followed by the research methodology in Section 4.1 and finally the results which are shown in Section 4.3.

3.1 Semantic bugs in SQL queries

A semantic bug in an SQL query, as defined by Brass and Goldberg [9], refers to a legal query which does not (always) produce the intended result. To be more precise, it means that the query uses correct SQL syntax, however the results it produces are incorrect for the given task, which means the query has some semantic bug. These are the types of errors that the tool presented in this paper tries to detect. This task is however non-trivial especially when there is no information about the task for which the query was written or how the queried data is structured.

Semantic bugs are especially common to appear in queries written by students who are yet to master the SQL language, however these can also be present in real world applications, which makes them more dangerous. Since these issues do not usually generate any warnings from the database engines, it is therefore harder for inexperienced developers to catch them, especially due to the fact that in some cases the queries might actually seem to produce the correct results. For this very reason, it can be argued that semantic bugs in SQL are more dangerous than syntactic errors, which can very easily be detected as well as corrected. This is why tools for detecting these types of issues should be directly integrated in either development environments or even better, plugins that can check the correctness of queries during runtime.

Furthermore, semantic bugs can often impact the overall performance of the query leading to increased computation time or inefficient resource usage. This is again quite important in applications where speed is critical and executing queries which contain semantic bugs might lead to bottlenecks as discussed by Muse et al. [34] in their paper. Although previous studies did not show significant correlation between SQL code smells and other traditional code smells, it was shown by Muse et al. [34] that once an SQL semantic bug is introduced in some system, it tends to have a longer lifespan than other more traditional semantic code smells, which makes the task of detecting these types of errors quickly even more important.

3.2 Detection strategies for SQL semantic bugs

In this section we propose several heuristics used for detecting semantic bugs in SQL queries. Each of these are later implemented as rules in the static analysis tool we developed for this project. The errors, which are detected by our proposed heuristics, were initially presented and classified in the work of Brass and Goldberg [9] which represents a complete list of semantic bugs encountered in SQL. A small subset of semantic bugs was also collected from the SQL Enlight tool. This is a closed source static analysis tool developed by Yubitsoft Eood which aims to detect a number of issues in SQL queries as well. Other tools for SQL analysis and code smell detection are TOAD and SQL Prompt, both of which are also closed source and require a set of input queries as well as database schemas in order to detect any issues. Furthermore, none of the previously mentioned tools are able to analyse queries embedded in source code.

The rule-based static analysis tool developed for this project uses a hard implementation, meaning that the worst is always assumed, since there is no additional information related to tasks or database schemas. This means that for certain queries, the tool might generate false positive warnings, as will later be discussed in Section 3.4, however this was preferred since it offers a good tradeoff for the setting when little information about the queries is known.

In the following subsections we provide a description of each bug, examples of SQL queries that contain the semantic bug and present the implementation of each detection strategy together with any assumptions made by the heuristics which are used in our tool. A summary of all the semantic bugs that our tool detects is also presented in Table 3.1.

Semantic bug	**Description**	**Ref.**
Inconsistent tuple variable	Tuples matched on key attributes access different values for some other attribute	[9]
Constant output column	The value of an output column is constant and can be derived from the query	[9]
Duplicate output column	The same attribute is present inside the `SELECT` clause multiple times	[9]
Unnecessary `JOIN` clause	Tuple can be replaced with another already existing one (removing unnecessary join)	[9]

Identical tuple variables	Tuples matched on key attributes are identical	[9]
Comparison with `NULL`	Using normal comparison operators with `NULL` instead of `IS NULL` expression	[9] [34]
Unnecessary general comparison	Using greater than or equals or less than or equals operators instead of the equals to operator	[9]
`LIKE` used without wildcards	Like can be replaced with equals to operator	[9] [16]
Unnecessary `SELECT` list in `EXISTS`	There is no need to add a complicated `SELECT` list inside `EXISTS` subqueries	[9]
Unnecessary index scan	Using a wildcard character at the beginning of a string results in missing the access predicate	[17]
Unnecessary `DISTINCT` in aggregations	For certain aggregation function `DISTINCT` keyword has no effect on the result	[9] [18]
Unnecessary `COUNT` argument	In certain cases the `COUNT` argument is not necessary and can be replaced with something simple	[9]
Unnecessary `GROUP BY` attribute	Attribute under `GROUP BY` which does not appear under `SELECT` or `HAVING` outside aggregations is unnecessary	[9] [34]
Unnecessary `GROUP BY` clause	If no aggregation functions are used, `GROUP BY` can be replaced with `SELECT DISTINCT`	[9] [34]
Unnecessary `UNION` condition	Two `UNION` clauses with the same `SELECT` and `FROM` conditions and mutually exclusive `WHERE` conditions can be merged	[9]
Unnecessary `ORDER BY` terms	A constant term is not necessary to be included in the `ORDER BY` clause	[9]
Unnecessary `GROUP BY` terms	A constant term is not necessary to be included in the `GROUP BY` clause	[9]
Inefficient `HAVING` clause	Condition inside the `HAVING` clause that can be moved inside the `WHERE` clause making the query more efficient	[9]
Missing `JOIN` conditions	Missing references for the joined tables	[9]
Uncorrelated `EXISTS` subqueries	Subqueries contain no reference to tables or tuples defined in the higher level portions of the query	[9]
Mismatch in subquery `SELECT`	For `IN` subqueries, the `SELECT` list from the inner query should match the attributes used in the outer query	[9]
Strange subquery conditions	A condition inside an SQL subquery which accesses only tuple variables from the outer query is strange	[9]
Strange `HAVING` clause	A query using a `HAVING` clause without a `GROUP BY` clause is strange	[9]
Strange wildcards without `LIKE`	Using wildcard characters without `LIKE` is strange	[9]
Division by zero	Potential divisions by zero due to order of operation execution	[9]

Table 3.1: Summary of semantic bugs detected by our tool

3.2.1 Inconsistent tuple variable

Motivation: This bug is present whenever the key attributes of two different tuple variables over the same relation are matched while at the same time for some other attribute the two tuple variables are matched to different values. Since our tool does not require information about the database schema, we can not determine the key attributes, therefore the condition for this semantic bug is relaxed by dropping this constraint. More specifically, whenever two different tuple variables over the same relation have a set of attributes on which they are matched as well as at least one attribute for which there are different requirements, then this means the query is inconsistent, since the conditions can not be satisfied at the same time, hence the result set will be empty. The following example should be detected as having a semantic bug[1].

```
SELECT * FROM audit a1
WHERE NOT EXISTS (
SELECT * FROM audit a2
WHERE a2.status='approve' AND a1.status='decline' AND a2.id=a1.id);
```

In this example, the two tuple variables `a1` and `a2` over the same relation `audit` are being matched inside the `EXISTS` subquery on the `id` attribute, however, the query is inconsistent since `a1.status = 'decline'` and `a2.status = 'approve'` can not have two different values at the same time, hence the result set for this subquery will always be empty. Without knowing the intent of the developer, we can not propose a refactored version of the query, however an equivalent query would be:

```
SELECT * FROM audit a1;
```

Detection strategy: Detecting this semantic bug is done by building, for each tuple variable used, a set of attributes which are equated with other tuple variables over the same relation. Furthermore, a second set is built which stores the attributes that are equated with other constant values. After the query is parsed, for all tuple variables over the same relation, we check the set of matching attributes as well as the set of attributes which are equated to different values. If there is a full match between the attributes in the first set, then we check to see whether there are any common attributes from the second set which might be equated to different values. If this is the case, then the query has inconsistent conditions and a warning for this semantic bug is generated by our tool.

3.2.2 Constant output column

Motivation: This semantic bug appears whenever the value of an output column is constant and can be derived from the query without any additional information

[1]`https://stackoverflow.com/questions/34064071/how-to-select-data-based-on-special-condition`

regarding the state of the database. More specifically, whenever an attribute under the WHERE clause is equated to some constant value, if it is also present in the SELECT list, then it will always have the same value. This attribute can therefore be removed from the SELECT clause. The following example should be detected as having a semantic bug[2].

```sql
SELECT * FROM person WHERE name = 'robert';
```

In this example, there is a constant column which is present in the output of the query, more specifically the name attribute is equated to some constant value, thus it can be removed from the SELECT clause. We can not directly provide a potential fix for this query without having more information about the structure of the person schema, however the query can easily be corrected by specifying only the individual columns that are needed.

Detection strategy: This bug is detected by looking for attributes under the WHERE clause which are equated to some constant value. If this attribute is also present under the SELECT clause or the select all operator (*) is used, then it means that it will always have a constant value, hence it can be removed from the output, and a warning indicating this will be generated by our tool.

3.2.3 Duplicate output column

Motivation: This bug appears when the SELECT list of a query or subquery contains the same attribute multiple times. In this case, it means that duplicate output columns will be present in the result set, therefore these columns can be removed from the SELECT clause, making the query easier to understand. The following example should be detected as having a semantic bug[3].

```sql
SELECT e.empid, e.fname, c.description AS hair, c.description AS race
FROM employee2 e INNER JOIN code c;
```

In this example, there is a duplicate output column, since the c.description attribute is specified two times under the SELECT clause of the query. A potential fix for the example query is given below.

```sql
SELECT e.empid, e.fname, c.description AS hair_or_race
FROM employee2 e INNER JOIN code c;
```

Detection strategy: For detecting this semantic bug, the SELECT clause as well as the FROM clause of each query is parsed. We then keep track of tuple variables declared in the FROM clause and referenced under SELECT, in order to detect different tuple

[2]https://stackoverflow.com/questions/16449377/combine-two-queries-to-check-for-duplicates-in-mysql

[3]https://stackoverflow.com/questions/10845201/tsql-join-query

variables referring to the same attribute. All attributes present in the `SELECT` clause are then verified in order to determine whether there are duplicates in this list. Furthermore, if the select all operator (*) is present, then there is also no need for selecting other attributes.

3.2.4 Unnecessary `JOIN` clause

Motivation: This bug appears in queries containing a tuple variable `A`, for which only the key attributes are accessed, and this key is then equated with the foreign key of another tuple variable `B`. In this case, the tuple variable A is not needed, as this results in an unnecessary `JOIN` being used. This semantic bug can therefore have an impact on the performance of the query and should be corrected. For our tool, the requirement on key attributes is dropped, since only the query is presented as input, and there is no way of detecting which attributes represent the keys for a tuple variable. Therefore, we check if for some tuple variable `A`, the attributes which are being accessed for it are then further equated with the same attributes of another tuple variable `B`, in which case `A` might not be needed, resulting in the unnecessary `JOIN`. The following example should be detected as having a semantic bug[4].

```
SELECT t.* FROM terms t
JOIN term_relationships tr ON tr.term_id = t.term_id
JOIN posts p ON p.post_id = tr.post_id WHERE p.post_id = 1;
```

In this example, there is an unnecessary `JOIN` for `posts` on `post_id` since the condition inside the `WHERE` clause, `p.post_id = 1`, can be replaced by `tr.post_id = 1` and moved up inside the second `JOIN` clause. This not only helps with making the query easier to understand, but also improves the performance of the query since there is one less `JOIN` that needs to be executed. A potential fix for the example query is given below.

```
SELECT t.* FROM terms t
JOIN term_relationships tr ON tr.term_id = t.term_id
WHERE tr.post_id = 1;
```

Detection strategy: Detecting this semantic bug is done by keeping track, for each tuple variable, of the comparisons in which they are involved throughout the query. After the query is parsed, for each tuple variable `A`, we check all the comparison expressions in which it was used. If all accessed attributes are also equated with a different tuple variable `B`, under the assumption that these attributes are a foreign key of this tuple variable `B`, then a warning for an unnecessary `JOIN` is raised.

[4] `https://stackoverflow.com/questions/25953303/multiple-select-result-in-mysql-depend-on-other-select`

3.2.5 Identical tuple variables

Motivation: If a query contains two different tuple variables over the same relation for which the key attributes are equated, then these two variables must always point to the same tuple. Since our tool does not take as input any information regarding the key attributes of the schema for which the query was written, the detection rule checks for tuple variables over the same relation for which at least one attribute is equated, with the assumption that this attribute represents the key. In some cases, this might not hold and the generated warning should then be considered as a false positive. The following example should be detected as having a semantic bug.

```
SELECT * FROM employees X, employees Y WHERE X.employee_id=Y.employee_id;
```

In this example, the two tuple variables `X` and `Y` over the same relation `employees` are matched on `employee_id`, thus under the assumption that this is a unique key, the two tuple variables are identical, meaning they will always refer to the same rows. In this case, we can not directly provide a potential fix for this query without having more information about the structure of the `employees` schema and the underlying task for which the query was written.

Detection strategy: The implementation for detecting this semantic bug first stores all tuple variables used in the query as well as the tables to which the variables refer. Whenever the equals to operator is detected, we check to see whether the tuple variables used in the comparison are pointing to the same underlying table and the same attribute is used on both sides of the comparison. If this is the case, under the assumption that the compared attribute represents a unique key on the table, then the two tuple variables must always point to the same tuple, thus they are identical and a warning is generated indicating the presence of this semantic bug.

3.2.6 Comparison with `NULL`

Motivation: In some database management systems it is syntactically valid to use `A = NULL`, however this expression always has a constant truth value of either null or unknown. To avoid such situations, the `IS NULL` or `IS NOT NULL` should always be used instead. The following example should be detected as having a semantic bug[5].

```
SELECT r.id, r.authtoken.instagram, r.username FROM root r
WHERE r.abc <> null;
```

In this example, `r.abc` is being compared to `null`. For some database engines, this might not be considered a syntactic error, however the condition will return either null or unknown. To avoid these types of situations, when `null` is involved, we

[5]https://stackoverflow.com/questions/52203347/check-if-field-exists-in-cosmosdb-json-with-sql-nodejs

should always use the `IS NULL` or `IS NOT NULL` expressions instead of the comparison operators. A potential fix for the example query is given below.

```
SELECT r.id, r.authtoken.instagram, r.username FROM root r
WHERE r.abc IS NOT null;
```

Detection strategy: For detecting this semantic bug, whenever the equals to or not equals to operators are used inside a query, we check to see whether one of the terms is the `NULL` keyword. If this is the case, then the comparison operator should be replaced with the expression `IS NULL` or `IS NOT NULL` respectively.

3.2.7 Unnecessary general comparison

Motivation: This semantic bug concerns the greater than or equals as well as the less than or equals operators. When the greater than or equals operator is used inside the `WHERE` clause and the rightmost term of the comparison operator contains a subquery where the `SELECT` list only uses the `MAX` function with the same attribute as the leftmost comparison term, then the greater than or equals can be replaced with the equals operator. Similarly, the same holds for the less than or equals operator and the `MIN` function. The following example should be detected as having a semantic bug[6].

```
SELECT name FROM world
WHERE gdp >= (SELECT max(gdp) FROM world WHERE continent = 'europe');
```

In this example, the greater than or equals operator from within the `WHERE` clause can be replaced by the equals operator, because of the `SELECT max(gdp)` expression from the subquery, which will make the query easier to understand. A potential fix for the example query is given below.

```
SELECT name FROM world
WHERE gdp = (SELECT max(gdp) FROM world WHERE continent = 'europe');
```

Detection strategy: This semantic bug is detected by checking the greater than or equals or less than or equals operators inside the `WHERE` clauses of queries. Whenever such operators are detected, the clause following the operator is checked in order to determine whether a `SELECT MAX` or `SELECT MIN` is used, where the `MAX` and `MIN` functions use the same attribute as the leftmost term of the comparison operator. In these cases, the comparison can be replaced by the equality operator making the query easier to understand.

[6]https://stackoverflow.com/questions/43248530/max-or-all-in-sql

3.2.8 LIKE used without wildcards

Motivation: If a query uses a `LIKE` expression which does not contain any wildcard characters, then the `LIKE` can and should be replaced by the equals to operator. When using the equality operator, the entire strings are compared as opposed to comparing one character at a time when the `LIKE` expression is used without any wildcards. The following example should be detected as having a semantic bug[7].

```
SELECT description FROM tproduct WHERE description LIKE 'diamond';
```

In this example, the `LIKE` expression is used, however it does not contain any wildcard characters. In this case, the `LIKE` expression can be replaced by the equality operator. A potential fix for the example query is given below.

```
SELECT description FROM tproduct WHERE description = 'diamond';
```

Detection strategy: For detecting this bug, the `LIKE` expressions from within the parsed query are checked in order to see whether any wildcard characters are being used. If this is not the case, a warning is raised indicating the presence of this semantic bug.

3.2.9 Unnecessary SELECT list in EXISTS clause

Motivation: When using an `EXISTS` subquery, the `SELECT` list from the subquery is not important since `EXISTS` only checks for row existence, independent of the selected columns. When the query is executed, as soon as the `EXISTS` subquery returns one row, its execution is stopped and `TRUE` is returned to the parent query. Therefore, in `EXISTS` subqueries, the `SELECT` list is not important and something simple such as the select all operator (∗) should be used. However, this does not hold for `IN` subqueries, where the `SELECT` list from within the subquery is indeed important. The following example should be detected as having a semantic bug[8].

```
SELECT id, name FROM tbl_bkp t1
WHERE NOT EXISTS (
SELECT id, name FROM tbl_namecode WHERE id=t1.id AND name = t1.name);
```

In this example, inside the `EXISTS` subquery, the `SELECT` list contains multiple attributes. This is redundant as the only `TRUE` or `FALSE` will be returned to the parent query, therefore the `SELECT` list can be replaced with something simpler such as the select all operator. A potential fix for the example query is given below.

[7] `https://stackoverflow.com/questions/6502134/query-for-exact-match-of-a-string-in-sql`

[8] `https://stackoverflow.com/questions/27633978/compare-combination-of-two-column-with-combination-of-other-two-in-sql`

```
SELECT id, name FROM tbl_bkp t1
WHERE NOT EXISTS (
SELECT * FROM tbl_namecode WHERE id=t1.id AND name = t1.name);
```

Detection strategy: Detecting this semantic bug is done by parsing the query and searching for `EXISTS` subqueries. When these are detected, the `SELECT` list is checked and if it does not use a simple operator such as the select all operator or a single attribute, then a warning for this bug is raised.

3.2.10 Unnecessary index scan

Motivation: This semantic bug concerns the use of wildcard characters used at the beginning of a word while searching with the `LIKE` operator. In SQL, the usage of the `LIKE` operator is often the cause of unexpected performance issues because of inefficient indexing. The position of the wildcard characters inside the word that is being matched can cause significant drops in performance. When using a `LIKE` filter, only the characters which appear before the first wildcard can be used as indices to speed up the pattern matching. The remaining characters are just filter predicates which do not help with reducing the scanned index range. Therefore, `LIKE` expressions can contain an access predicate, the part before the first wildcard, and the filter predicate, the remaining characters. The scanned index range becomes smaller if the access predicate is more selective, resulting in better performance for the queries since the index lookup is faster. Thus, if a `LIKE` expression starts with a wildcard character, then no access predicate is present which defeats the purpose of an index, resulting in a scan of the entire database table. The following example should be detected as having a semantic bug[9].

```
SELECT * FROM street WHERE street_name LIKE '%park%ave%10%';
```

In this example, the search term for the `LIKE` expression starts with a wildcard character (matching zero or more characters), therefore there is no access predicate, thus the query will perform a full table index scan. In order to optimize the query it would be better to either provide an access predicate or, in case this is not possible, use a search function which will result in a full text index scan. In this case, the query can be fixed by using the `MATCH` and `AGAINST` syntax. The `MATCH` operator takes as input a comma separated list which names the columns to be searched, while the `AGAINST` clause takes as input the string to search for together with an optional modifier that includes the type of search. This can either be a natural language search, a boolean search or a query expansion search. A potential fix for the example query is given below.

```
SELECT * FROM street WHERE MATCH(street_name)
AGAINST('park ave 10' WITH QUERY EXPANSION);
```

[9]`https://stackoverflow.com/questions/11754192/like-operator-to-retrieve-the-results`

Detection strategy: The implementation for detecting this semantic bug parses the query in search for `LIKE` expressions. Whenever such an expression is found, the search term is checked in order to determine whether or not it starts with a wild-card character. If this is the case, a warning is generated indicating that an access predicate is missing, thus a full index scan will be performed.

3.2.11 Unnecessary `DISTINCT` in aggregations

Motivation: For the `MIN` and `MAX` aggregation functions the `DISTINCT` keyword is never necessary since this has no effect on the underlying query results. Furthermore, in most cases the presence of duplicates is likely significant in computing the results of the `SUM` and `AVG` aggregation functions, thus duplicates should not be excluded unless there are strong reasons for this. Therefore, the presence of the `DISTINCT` keyword inside these aggregation functions is strange and should raise a warning. For the other aggregation functions, it can not be determined whether `DISTINCT` is necessary or not inside the clause without having additional information about the query task, thus the other aggregation functions are excluded from this check. The following example should be detected as having a semantic bug[10].

```sql
SELECT SUM(DISTINCT money) AS sum_money, m.created_at FROM receipt r
JOIN material m GROUP BY m.created_at;
```

In this example, the `DISTINCT` keyword is used inside the `SUM` function. Unless there are strong reasons for this, the presence of `DISTINCT` here is considered strange since duplicates are most likely significant in this case, however without having additional information about the task for which the query was written, our tool will raise a warning for a potential semantic bug. Without having more information about the underlying task for which the query was written, we can not provide a potential fix for this query.

Detection strategy: The detection strategy checks whether any of the `MIN`, `MAX`, `SUM` or `AVG` aggregation functions are present in the query. For each of them, it then detects whether the `DISTINCT` keyword was used inside the function clause and if so a warning is raised.

3.2.12 Unnecessary `COUNT` argument

Motivation: The `COUNT` aggregation function can be used with or without an argument, in which case the star symbol is used (*) most commonly. Whenever there is no `DISTINCT` being used in the `COUNT` function and the argument can not be null, these two versions of the `COUNT` function are equivalent and the one without any argument is preferred since it makes the query easier to understand and read. In our implementation for this detection strategy, we can not check the condition related to the

[10]`https://stackoverflow.com/questions/53067977/combine-two-tables-unique-column`

arguments of the `COUNT` function not being null, since this would require additional information about the database schemas, therefore we make the assumption that the arguments used inside the `COUNT` functions are not null. The following example should be detected as having a semantic bug[11].

```
SELECT m.time, COUNT(r.seconds) FROM dbo.minutes m GROUP BY m.time;
```

In this example, there is no `DISTINCT` being used in the `COUNT` function and only one argument is used. Therefore, the clause inside the `COUNT` can be simplified and the start operator can be used instead. A potential fix for the example query is given below.

```
SELECT m.time, COUNT(*) FROM dbo.minutes m GROUP BY m.time;
```

Detection strategy: For detecting this semantic bug, we parse the query and check if any `COUNT` functions are being used. For each of these, we then check whether `DISTINCT` is used inside the `COUNT` clause. If this is not the case, and the clause only contains an argument, then we raise a warning indicating that the `COUNT` clause can be simplified by using the start operator instead.

3.2.13 Unnecessary `GROUP BY` attribute

Motivation: Whenever an attribute which appears under the `GROUP BY` clause is functionally determined by other attributes and if it does not appear under the `SELECT` or `HAVING` clauses outside of aggregation functions, then it can be removed from the `GROUP BY` altogether. For our tool, this condition is relaxed since we are looking only at the query without knowing the database schema, thus it can not be determined whether an attribute is functionally determined by other attributes or not. Therefore, we only check for an attribute which appears inside the `GROUP BY` clause without being used in either the `SELECT` or `HAVING` clauses outside of any aggregation functions, in which case a warning for this semantic bug will be triggered. The following example should be detected as having a semantic bug[12].

```
SELECT count(*) AS countbyid FROM items
WHERE fkid = 2003799 GROUP BY fkid
HAVING countbyid>1 ORDER BY countbyid;
```

In this example, the `fkid` attribute is present under the `GROUP BY` clause without being used in either the `SELECT` or the `HAVING` clauses outside aggregation functions, therefore it can be removed from the `GROUP BY`. This will further simplify the query since the `fkid` is the only attribute present under `GROUP BY`. Furthermore, this query contains another semantic bug since `fkid` can only have one value as it is equated to some constant in the `WHERE` clause, thus the grouping will have no effect on the result. A potential fix for the example query is given below.

[11] https://stackoverflow.com/questions/761700/how-can-i-check-for-average-concurrent-events-in-a-sql-table-based-on-the-date

[12] https://stackoverflow.com/questions/4818207/sql-aggragates-and-aliass

```
SELECT count(*) AS countbyid FROM items
WHERE fkid = 2003799
HAVING countbyid>1 ORDER BY countbyid;
```

Detection strategy: The detection strategy for this bug parses the query and keeps track of the attributes appearing under the SELECT and HAVING clauses outside of aggregation functions. Furthermore, the attributes appearing under the GROUP BY clause are also stored. When the parsing of the query is finished, we check for attributes found under the GROUP BY clause that are not present in either of the two lists with attributes detected inside SELECT or HAVING clauses. For each of these attributes a warning is raised indicating this semantic bug being detected in the query.

3.2.14 Unnecessary GROUP BY clause

Motivation: If a query has exactly the same attributes under the SELECT clause listed inside the GROUP BY clause as well and if no aggregation functions are used in either of the two clauses, then the GROUP BY clause can be replaced by SELECT DISTINCT. In most cases, the SQL optimizer will produce the same or similar execution plans for the two queries, so in these cases there is not always a gain in terms of performance, however by rewriting the query this becomes shorter and clearer. In most of these cases, using the GROUP BY clause essentially removes duplicates from the result set, therefore the DISTINCT operator is better suited to be used, as opposed to situations where aggregation functions are used in which case a GROUP BY is indeed required. The following example should be detected as having a semantic bug[13].

```
SELECT p.name FROM pc p LEFT JOIN sc s ON p.color=s.color
WHERE s.color IS NOT NULL GROUP BY p.name;
```

In this example, all the SELECT attributes are listed under the GROUP BY clause and also no aggregation functions are used, therefore the GROUP BY clause can be dropped and replaced by SELECT DISTINCT, making the query shorter and clearer. A potential fix for the example query is given below.

```
SELECT DISTINCT p.name FROM pc p LEFT JOIN sc s ON p.color=s.color
WHERE s.color IS NOT NULL;
```

Detection strategy: The implementation for this strategy first checks whether any aggregation functions are used in either the SELECT or the GROUP BY clauses. If this is the case, then there is no need to continue with further checking the query for this semantic bug. Otherwise, we continue by detecting all the terms used in the SELECT clause as well as those used under the GROUP BY clause. If there is a perfect match between these two sets of terms, then it means that the GROUP BY clause can be replaced by a SELECT DISTINCT and a warning for this semantic bug is triggered.

[13] https://stackoverflow.com/questions/33840072/how-can-you-have-a-column-equal-all-items-in-a-list

3.2.15 Unnecessary UNION condition

Motivation: A query for which the two UNION subqueries use the same SELECT expressions, the same FROM lists as well as mutually exclusive WHERE conditions, can be simplified by joining the two WHERE clauses and connecting them using an OR, replacing the UNION altogether. The same holds for queries using a UNION ALL clause, in which case the query can again be simplified and the two different WHERE clauses joined using an OR. For our tool, the condition for mutually exclusive WHERE conditions can not be fully checked without having additional database schema information, therefore this was relaxed to only having different WHERE clause expressions. The following example should be detected as having a semantic bug[14].

```
SELECT id, name, age FROM student WHERE age < 15 UNION
SELECT id, name, age FROM student WHERE name LIKE '%a%' ORDER BY name;
```

In this example, the query contains one UNION clause and we can observe that the SELECT expressions of the two subqueries are the same as well as the FROM lists, more specifically the same table student is used in both subqueries. Furthermore, the two WHERE clauses are mutually exclusive, therefore, the query can be simplified by replacing the UNION with an OR and joining the two WHERE clauses. This query also suffers from another semantic bug, more specifically the unnecessary index scan, since the LIKE expression is starting with a wildcard character. A potential fix for the example query is given below.

```
SELECT id, name, age FROM student WHERE age < 15 OR
MATCH(name) AGAINST ('a' WITH QUERY EXPANSION)
ORDER BY name;
```

Detection strategy: The implementation for this strategy first detects whether the query involves any UNION or UNION ALL clauses. If this is the case, then we continue with storing the SELECT expressions, as well as the FROM lists and the expressions under the WHERE clauses. After the whole query is parsed, we pairwise check the collected lists involving the three different clauses. Whenever two SELECT list expressions as well as FROM lists from two different subqueries are the same, the WHERE clauses are checked and if different then a warning is triggered, informing the user that the UNION can be replaced by an OR condition.

3.2.16 Unnecessary ORDER BY terms

Motivation: This bug concerns an ORDER BY term which can only have one possible value, in which case it is not necessary for this term to be included under the ORDER BY clause. More specifically, whenever a term inside the WHERE clause is equated to some constant value, it does not make sense for this term to also be present inside the ORDER BY clause as this will not have any effect on the result of the query. Moreover,

[14] https://stackoverflow.com/questions/4715820/how-to-order-by-with-union-in-sql

the same holds in the case where an ORDER BY term is functionally determined by previous terms which are used in the clause. However, since these cases can not be detected without having additional information about the database schema, our tool will not check for these situations. Furthermore, whenever a CASE expression is present inside the ORDER BY clause, then the involved terms are needed regardless of whether they have only one possible value or are otherwise functionally determined by previous terms used inside the clause. The following example should be detected as having a semantic bug[15].

```
SELECT q.postid, a.postid, c.commentid FROM posts q
WHERE q.postid = 1234 ORDER BY q.postid, a.postid, c.commentid;
```

In this example, the `q.postid` term from the ORDER BY clause is not necessary since it has a constant value, as can be observed from the `q.postid = 1234` condition inside the WHERE clause. A potential fix for the example query is given below.

```
SELECT q.postid, a.postid, c.commentid FROM posts q
WHERE q.postid = 1234 ORDER BY a.postid, c.commentid;
```

Detection strategy: For detecting this semantic bug, the query is parsed and the constant columns, those which are equated to a single constant value inside the WHERE clause, are being stored. For these terms, we then check if they are not present inside the ORDER BY clause and also not included inside of any CASE expression inside the ORDER BY clause. In these cases, the terms are not necessary in the ORDER BY clause and a warning should be raised by our tool.

3.2.17 Unnecessary GROUP BY terms

Motivation: This bug concerns a GROUP BY term which can only have one possible value, in which case it is not necessary for this term to be included under the GROUP BY clause. More specifically, whenever a term inside the WHERE clause is equated to some constant value, it does not make sense for this term to also be present inside the GROUP BY clause as this will not have any effect on the result of the query. The following example should be detected as having a semantic bug[16].

```
SELECT division, count(id) AS ct FROM test
WHERE role >=101 AND division=1 GROUP BY division;
```

In this example, the `division` term from the GROUP BY clause is not necessary since it has a constant value, as can be observed from the `division=1` condition inside the WHERE clause. A potential fix for the example query is given below.

[15] https://stackoverflow.com/questions/1918398/how-do-you-query-for-comments-stackoverflow-style

[16] https://stackoverflow.com/questions/1917516/sql-count-with-group-by-not-returning-0-zero-records

```
SELECT division, count(id) AS ct FROM test
WHERE role >=101 AND division=1;
```

Detection strategy: The strategy for detecting this semantic bug involves parsing the query and storing the constant columns, those which are equated to a single constant value inside the WHERE clause, as well as not being used in any other expressions. For these terms, we then check if they are present inside the GROUP BY clause as well, in which case a warning should be raised by our tool.

3.2.18 Inefficient HAVING clause

Motivation: If a condition inside the HAVING clause of an SQL query uses only GROUP BY attributes and no aggregation functions are used in either the HAVING or the GROUP BY clauses, then the condition can be moved under the WHERE clause. Since the WHERE clause is executed before the HAVING clause, it is much cheaper to check the condition already under the WHERE clause as this will result in faster running times allowing the SQL optimizer to better run the query. It should not be considered a semantic bug if the condition inside the having clause uses any of the SQL aggregation functions, since these are not allowed to be present in the WHERE clause of the query. The following example should be detected as having a semantic bug[17].

```
SELECT first_name FROM students GROUP BY class HAVING class IN('a');
```

In this example, the IN condition inside the HAVING clause can be moved under the WHERE clause as it does not make use of any aggregation functions. This will result in a faster query as the condition will be executed when the WHERE clause is evaluated thus the GROUP BY will run on a smaller result set. A potential fix for the example query is given below.

```
SELECT first_name FROM students WHERE class = 'a';
```

Detection strategy: For detecting this semantic bug, the GROUP BY and HAVING clauses are first checked to determine whether any aggregation functions are being used. If this is not the case, the HAVING clause is then further analysed in search of conditions that only use the attributes under the GROUP BY clause. These conditions can therefore be moved under the WHERE clause making the query more efficient.

3.2.19 Missing JOIN conditions

Motivation: This bug appears in queries involving joined tables for which the joined table sources do not have any column referenced neither in the JOIN condition nor in the WHERE clause. When no JOIN predicate is present, the query will include the

[17] https://stackoverflow.com/questions/40992352/sql-group-by-and-having-not-working

cartesian product of all rows, also known as the cross product, most certainly resulting in performance overhead for the queries and potentially wrong results as well. Therefore, it is important that joined tables are referenced in the `JOIN ON` or the `WHERE` clauses in order to avoid these types of issues. There is however one exception, more specifically when a `CROSS JOIN` is used. In these cases, there is no need for the involved table sources to have any columns referenced since for these queries the intention is clearly to obtain the cross product of all rows, thus a warning should not be generated. The following example should be detected as having a semantic bug[18].

```
SELECT station, slot, subslot, compid, compname FROM devicetrace AS dt
INNER JOIN complist as cl;
```

In this example, there are two joined tables, `devicetrace` and `complist` which do not have any columns reference in either the `JOIN` or the `WHERE` conditions. This means that the query result will include the cross product of all rows. Whether this was the intention of the query's author or not, missing join conditions are a potential cause for performance overhead and should be signaled as semantic bugs. Without having more information about the underlying task for which the query was written, we can not provide a potential fix for this query.

Detection strategy: Detecting this semantic bug is done by parsing the query and keeping track of all the used table sources. If less than two tables are used or a `CROSS JOIN` is involved, there is no need to further continue with checking for missing join conditions, so no warnings will be raised. If at least two table sources are used, then the `ON` and `WHERE` clauses of the involved (sub)query are checked in order to detect whether these table sources have any referenced columns. For any table which is not referenced, a warning will be generated for missing a join condition.

3.2.20 Uncorrelated `EXISTS` subqueries

Motivation: Uncorrelated subqueries are those that do not contain any reference to tables or tuple variables defined in the higher level portions of the query. If an `EXISTS` or `NOT EXISTS` subquery does not contain any reference to tuple variables from outer queries, this means that the subquery is uncorrelated, making it either globally true or globally false. This is unusual behaviour since both `EXISTS` and `NOT EXISTS` clauses should be used with correlated subqueries. When used in correlated subqueries, the expression is evaluated once for every row in the parent query and either true or false is returned as a result. Thus, when `EXISTS` subqueries are correctly used, in correlated subqueries, the SQL optimizer can take advantage of evaluating the subquery without materializing intermediate results as well as caching results in order to reuse previously computed values of the predicate for the same values of

[18]`https://stackoverflow.com/questions/16632901/how-to-do-where-clause-before-inner-join`

the outer query tuple reference. Therefore, whenever an `EXISTS` subquery makes no reference to any tuple variable from the outer query, this is a semantic error and a warning should be raised. The following example should be detected as having a semantic bug[19].

```
SELECT * FROM final_combined_result wfcr
WHERE NOT EXISTS (SELECT contact_id, account_id FROM temp_wfcr);
```

In this example, the `NOT EXISTS` subquery is uncorrelated since there are no references to outer tuple variables inside the subquery. This means that the subquery will be either globally true or globally false, returning the same result set each time it is executed for the rows in the parent query. Furthermore, this example also contains another semantic bug, more specifically the `SELECT` list inside `EXISTS` subqueries is not important, thus this can be simplified to just use the select all operator (*). Again, without having more information about the underlying task for which the query was written, we can not provide a potential fix for this query.

Detection strategy: For detecting this type of semantic bug, the query is parsed in search for `EXISTS` subqueries. A list of previously encountered tables and tuple variables is stored and whenever a condition inside an `EXISTS` subquery is encountered we check to see whether any reference to outer tables or tuple variables is made. If at least one such reference is detected, then the `EXISTS` subquery is considered to be correlated and no warnings are generated for it. Whenever an `EXISTS` subquery is not correlated, then the tool will trigger a warning for this semantic bug.

3.2.21 Mismatch in subquery `SELECT`

Motivation: When using the `IN` operator together with a subquery expression, the `SELECT` list from the inner query should match the attributes used in the outer query. This semantic error is especially dangerous since database engines will not detect any syntax error if there is a mismatch between the subquery `SELECT` list and the attributes used in the parent query. It is also true that there might be cases for which such mismatches might not actually represent an issue with the query, however we believe that SQL developers should still be informed about these mismatches, in order to be able to further analyse whether the query contains any problem or not. Therefore, when the `SELECT` clause from a subquery contains different attributes that the ones used in the outside query a warning should be raised. The following example should be detected as having a semantic bug[20].

```
SELECT t1.articleno, t1.artdescription, t2.dateyear FROM t1
JOIN t2 WHERE NOT t1.articleno IN (SELECT t2.dateyear FROM t2);
```

[19]`https://stackoverflow.com/questions/25978882/select-combination-of-columns-from-table-a-not-in-table-b`

[20]`https://stackoverflow.com/questions/4177671/inverted-sql-query`

In this example, there is a mismatch between `t1.articleno` in the parent query and `t2.dateyear` in the `SELECT` clause of the inner query. While this might not always represent an underlying problem with the query, our tool will raise a warning in these situations. Again, without having information about the task for which the query was written or information about the database schema, it is not possible to detect for sure these types of issues, nor is it possible to provide a fix for these queries, however we can assume that whenever there is a mismatch between these attributes special attention is required, therefore developers should be informed.

Detection strategy: The implementation for this rule only looks at the `IN` expressions from queries and compares the list of attributes used in the outside parent query with the one used by the `SELECT` clause in the inner subquery. Whenever there is a mismatch between the column names used in these two clauses, the tool keeps track of this and after the whole query is parsed, a warning is displayed for all the detected mismatches. In order to avoid a large number of potentially false positive warnings, the implementation checks for substring matches as well in which case a warning will not be generated.

3.2.22 Strange subquery conditions

Motivation: A condition inside an SQL subquery which accesses only tuple variables from the outer query is strange as it can be moved up, as part of a condition in the parent query. In some cases, this could even mean that unnecessary joins are present and therefore the overall performance of the query is impacted, since inner queries are executed before the outer ones and their returned results are then used inside their parent queries. Furthermore, if a subquery condition only uses tuple variables from outer queries, moving these conditions up helps with making the overall query more understandable and at the same time makes it easier to detect other potential issues. The following example should be detected as having a semantic bug[21].

```sql
SELECT * FROM t1 WHERE t1.id IN (SELECT t2.id FROM t2 WHERE t1.a = 'aa');
```

In this example, the condition `t1.a = 'aa'` can be extracted from the inner query and moved up, inside the `WHERE` clause of the outer query. This will not only help with improving the readability of the query, but will also improve the performance of the inner query as well. A potential fix for the example query is given below.

```sql
SELECT * FROM t1 WHERE t1.a = 'aa' AND t1.id IN (SELECT t2.id FROM t2);
```

Detection strategy: For implementing this rule in our tool, a map of tuple variables encountered for inner and outer queries is built. Whenever a condition such as equalities, comparisons or `BETWEEN` expressions are encountered, the tuple variables

[21] `https://stackoverflow.com/questions/39871799/sql-with-ambiguous-column`

used in these conditions are checked against the map of tuples. If the condition only contains tuple variables that are not defined inside the current subquery, then the condition is marked as strange and the tool raises a warning. Since tuple variables are defined in the `FROM` clause, this allows for building both the map of tuple variables as well as checking subquery conditions while parsing the query at the same time, hence only one iteration over the complete query is required for this rule.

3.2.23 Strange `HAVING` clause

Motivation: A query using the `HAVING` clause without a `GROUP BY` clause is considered strange, since it can have only one result or none at all. The `HAVING` clause is similar to the `WHERE` clause, however it only applies to groups as a whole rather than individual rows. In standard SQL, the `FROM` clause is executed first, then the `WHERE` clause followed by the `GROUP BY` and finally the `HAVING` and `SELECT` clauses. This means that if the `GROUP BY` clause is not present, all the rows which are not excluded by the `WHERE` clause will be returned as a single group to the `HAVING` clause. In this case, since no grouping is performed between the `WHERE` and `HAVING` clauses, the latter will act like a `WHERE` clause except it operates on the result set as a group and aggregation functions are allowed as well. Furthermore, when joins are present, the `HAVING` clause without a `GROUP BY` should generate a syntax error in most database engines. Therefore, since `HAVING` is filtering groups, if no `GROUP BY` clause is present, then all rows represent one group, in which case if the predicate inside the `HAVING` clause evaluates to true, the query will return one row, otherwise nothing will be returned. Thus, in these situations, a query using a `HAVING` clause without a `GROUP BY` clause being present is strange and should be considered as having a semantic bug. The following example should be detected as having a semantic bug[22].

```
SELECT height FROM sashelp.class HAVING height = max(height);
```

In some database engines, this SQL query might even be considered as having a syntax error as well since there is a reference to `height` in the `HAVING` clause without it being contained in an aggregate function or the `GROUP BY` clause. In this case, our tool will detect a semantic error since the `HAVING` clause is being used without `GROUP BY` being present. A potential fix for the example query is given below.

```
SELECT height FROM sashelp.class GROUP BY height
HAVING height = max(height);
```

Detection strategy: The implementation for this rule will parse the query in search for `GROUP BY` and `HAVING` clauses. Two boolean flags are used for tracking the presence of either two clauses in the query. When done with parsing either the whole query or a subquery, the two flags are checked. If the `HAVING` clause is present without a `GROUP BY` clause also being used, then the tool marks this as a semantic bug.

[22]https://stackoverflow.com/questions/28032638/finding-the-id-associated-to-the-maximum-in-sas

3.2.24 Strange wildcards without LIKE

Motivation: In SQL a number of special symbols are reserved to be used as wildcards. More precisely, these symbols carry special meaning and are used to substitute one or multiple characters in a string. Wildcards are used in combination with the `LIKE` operator, which is allowed in the `WHERE` clause of the query, in order to search for the specified pattern in the column entries. Since these special characters are written inside a string, using one of these symbols in a comparison expression without the `LIKE` operator will not generate a syntactic error as the database engine has no way of knowing whether the author of the query intended to use the wildcard operator or its underlying character inside the string. Thus, whenever a wildcard is used without the `LIKE` operator, a warning should be triggered. The following example should be detected as having a semantic bug[23].

```
SELECT * FROM employees WHERE name = 'chris%';
```

In this example, the database engine will treat the wildcard character `%` as a normal literal. This is most certainly not what the developer actually had in mind when writing this query since the returned result will probably be the empty set as there are no real names ending in `%`. A potential fix for the example query is given below.

```
SELECT * FROM employees WHERE name LIKE 'chris%';
```

Detection strategy: The implementation for this rule parses the query in search for comparison expressions involving columns and string values containing one or more wildcard characters. Whenever these types of expressions are found, a warning for wildcards used without the `LIKE` operator is triggered. The underscore (`_`) wildcard character is not considered for this check since it is quite common for these types of characters to appear in column values, therefore in these situations it can no be determined if a missing `LIKE` operator is indeed a semantic bugs.

3.2.25 Division by zero

Motivation: One common problem involving data type operators is dividing by zero. In SQL, since there is no guarantee on the evaluation sequence of statements within the `WHERE` clause, it is therefore difficult for developers to avoid such situations. The following example is considered unsafe and should be detected as a semantic bug[24].

```
SELECT itm_num FROM itemconfig
WHERE (pal_qty / case_qty) > 500 AND case_qty > 0;
```

[23] `https://stackoverflow.com/questions/543580/equals-vs-like`
[24] `https://stackoverflow.com/questions/11692384/division-in-a-sql-statement`

In this example, although the condition `case_qty > 0` is present in the `WHERE` clause, since the order of operations is not enforced, this query is still unsafe. A warning should therefore be triggered when there is a division present in the query, except situations in which the division is under the `SELECT` clause and the divisor column is checked for not being equal to zero under the `WHERE` clause, since in these cases the `WHERE` clause will be evaluated prior to the `SELECT` by any database engine.

Detection strategy: The implementation for this rule parses the query and stores all the encountered division terms. Apart from this, a boolean flag is also created which indicates whether the division term was under the `SELECT` clause or not. Furthermore, all columns which are checked for not being equal to zero under the `WHERE` clause are also stored. Finally, the divisions which are under the `SELECT` clause but for which the divisor column is checked in the `WHERE` clause are removed from the initial set and the remaining division terms are returned as a result for this rule which generates a warning for potential divisions by zero.

3.3 Implementation

For detecting the previously described semantic bugs in SQL queries, we implemented a static analysis tool, where each detection strategy represents one rule for catching a different semantic bug. The tool was developed in Java and JSQLParser was used for parsing the SQL queries. JSQLParser is an open source library which translates SQL into a traversable tree of Java classes, for which the visitor design pattern can then be used to inspect the whole query. The library was chosen for this project because of its vast support for different syntaxes such as Oracle, SqlServer, MySQL and PostgreSQL as well as its open source nature. The overall flow starts with some input query being presented to the tool. Then, JSQLParser is used to parse the query. Each detection rule, implemented as a JSQLParserVisitor, then runs on the input query and whenever a rule is broken, the query is marked as being problematic and a warning signaling the detected semantic bug is raised by the tool and displayed to the user. An overview of the overall flow of the tool is presented in Figure 3.1.

The detection strategies for each of the semantic bugs described in the previous sections are implemented as individual rules which make use of the visitor design pattern to parse SQL queries in search for different semantic bugs. Each rule focuses on detecting a single bug type. Furthermore, the tool is easily open for extension allowing for the implementation of numerous other rules for detecting other types of issues. Since for most rules there are a number of similarities such as parsing the queries in search for table sources or tuple variable names, there is a common interface shared by all the rules. One key aspect that we wanted to have for our tool was the ease of adding other rules in the future. In our current framework, this can easily be done by creating a new Java class and implementing the main interface which, among others, already takes care of parsing SQL queries. Developers then

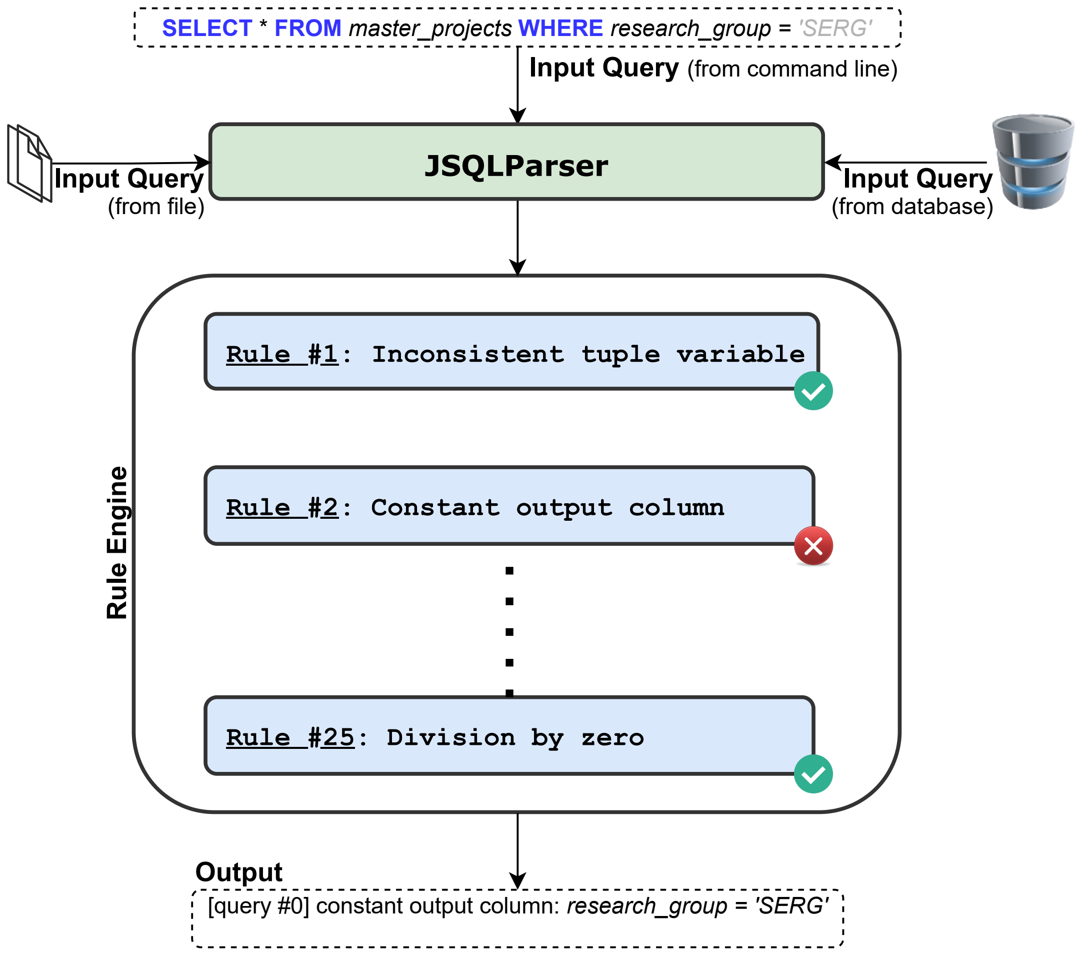

SELECT * FROM master_projects WHERE research_group = 'SERG'
Input Query (from command line)
Input Query (from file)
JSQLParser
Input Query (from database)
Rule Engine
Rule_#1: Inconsistent tuple variable
Rule_#2: Constant output column
Rule_#25: Division by zero
Output
[query #0] constant output column: research_group = 'SERG'

Figure 3.1: Overall flow of the semantic bug detection tool

only have to override the methods for which custom functionality for the detection strategy is needed. This, for example, results in being able to write complicated heuristics for the rules without having to rewrite the parsing logic which saves both time as well as potential mistakes. Another important aspect is that rules can better be tested by only focusing on the new functionality which is specifically tailored for the detection strategy since the parsing logic is already tested by our framework.

The tool has three different modes of operation. In the first one, a query is entered via the command line and all the rules will be checked for finding semantic bugs in the input. This can be used whenever a fast check needs to be run for a single query in order to validate if this is semantically correct or not. The second mode of operation allows for an input file to be specified as input. The tool will then read all the queries from the file, which should be separated by a semicolon character (;) and will display all the errors it detected for each of the SQL queries present in the file. This setting is perhaps more useful for conducting analysis on a larger input dataset. Finally, in the third mode of operation, the tool makes use of the Spring framework[25] in order to connect to a MySQL database for retrieving input queries and running the rules on these. This mode was used during this research for analysing the large dataset of SQL queries collected from StackOverflow posts. The tool then also saves the results in this database which can later be used for further analysis.

Each rule is heavily covered by unit tests which try to account for general as well as more special edge cases. In total, there are more than 150 unit tests for the 25 rules implemented for our detection tool. Other tests include an automated script which uses a collection of more than 500 manually verified queries that runs against the tool and compares the outcome to the expected result. Some other 50 queries were manually checked in order to verify they do not contain any semantic errors after which they were presented as input to the tool in order to verify no warnings are raised, and later were added to the automated tests as well. The overall code quality for the tool was tracked on Better Code Hub, achieving an overall score of 9 out of 10 for quality, with the only area where maximum points were not scored being the separation of concerns in modules due to the shared interface used by all our rules. The source code for the tool together with all other scripts used for testing and analysing the results for this project can be found in GitHub at the following link: `https://github.com/SERG-Delft/sql-bug-finder`.

Furthermore, we also make our tool available in the form of an interactive website, where developers can directly check their queries for semantic bugs, as shown in Appendix C.

This website can be found at the following link: `https://sqlbugfinder.com`.

3.4 Validation

Apart from the various unit tests implemented for checking each rule, the tool was validated using a manual process of investigating more than 500 SQL queries and verifying whether the output of the tool was correct or not. For each of the 25 semantic bug detection rules implemented in our tool, a random set of 20 queries collected from StackOverflow posts, for which the tool reported semantic bugs, was first selected in order to be manually analysed. The tool was configured such that each rule was run in isolation, meaning that every rule was analysed in detail, separately from the others, in order to confirm the correctness of both the implementation as well as the underlying heuristics used. Whenever a semantic bug was erroneously reported by the tool for a specific query, the code was analysed and fixed.

Furthermore, the unit tests were also improved to ensure the issues would not appear at a later stage when further modifications might be made to the rule. If multiple issues were found for a specific rule, after the initial set of 20 queries was verified, the process was repeated by selecting a new random set of 20 queries from the dataset. When no more errors were detected in the random sample, or none of

[25]`https://spring.io/`

the reported errors could be addressed, due to the fact that the query could not be parsed properly by the JSQLParser library or otherwise, the rule was considered to be sufficiently checked. A dataset of roughly 500 queries was created, where for each rule the final set of 20 randomly selected queries together with the semantic bugs reported by the tool are registered. Furthermore, a script was implemented which uses this dataset together with the expected output to perform automated tests on the tool, in order to improve the testing of the rules. The final implementation of our tool, correctly detected 487 queries with semantic bugs as well as 50 queries without any bugs, out of the test dataset of 550 queries, bringing the accuracy of the tool to 97%. The results of this manual analysis for our tool are provided in Appendix A and in Table 3.2.

	Predicted NO	**Predicted YES**
Actual NO	$TN = 50$	$FP = 13$
Actual YES	$FN = 0$	$TP = 487$

Table 3.2: Tool performance confusion matrix

Chapter 4

An Empirical Study on SQL Semantic Bugs in the Wild

4.1 Research methodology

In this section we will present the research questions that this paper tries to answer. Furthermore, we also provide some more details regarding the datasets used in this research.

RQ1: *What is the prevalence of semantic bugs in SQL queries?*

With this research question, the aim is to investigate which are the most prevalent semantic bugs that appear in SQL queries. In order to analyse this, one of the most important aspects is being able to collect queries used in real world systems. For this, two datasets are used, one containing queries extracted from StackOverflow posts, and another dataset provided by Castelein et al. [11] in their study on search-based test data generation for SQL queries. Both of these contain queries used in various open source projects and should provide a clear answer as to how often semantic bugs appear in SQL queries. To answer the question, we will run the rule-based static analysis tool designed for this project on all the collected queries and report on the total number of issues detected for each semantic bug. Furthermore, we also determine the median prevalence for each semantic bug as the ratio between the total number of queries having the semantic bug and the total number of collected queries in the dataset.

RQ2: *What are the co-occurrences of semantic bugs in SQL queries?*

The aim of this research question is to understand whether certain semantic bugs might tend to appear in pairs. It is valuable to know whether this is the case since if certain errors are detected in a query, then further attention can be paid to bugs that might co-occur. For answering this question, we use the results obtained for RQ1 from running the tool on all the collected queries, and then build a co-occurrence matrix for the semantic bugs. Furthermore, we also use the techniques described by Eck and Waltman [15] in order to compute the Jaccard index for normalizing co-occurrence data and present this in the form of a heat map. The Jaccard index is defined as the ratio between the number of times two bugs co-occur and the number of times for which at least one bug is observed to be present.

RQ3: *What is the correlation between the complexity of a query and the number of semantic bugs it has?*

Another interesting insight is knowing whether the complexity of a query has any correlation with the number of semantic bugs it has. By having this knowledge, whenever a query is detected as being too complex, this can already signal that potential semantic bugs might be present in the formulation of the query. In order to answer this question, we define and compute the complexity of a query as the total number of predicates, joins, subqueries, functions and columns it contains.

For analysing the correlation between the complexity of the query and the number of semantic bugs it has we will provide a box plot, which is the preferred visualization technique when dealing with both numerical and categorical data such as the one in our case, as described by Munzner [33] in their book on data visualization techniques. Furthermore we also provide the distribution for the query complexity scores. The data used in these visualizations will include our entire collection of SQL queries, from both the StackOverflow and the EvoSQL datasets.

Finally, we perform the one-way analysis of variance (ANOVA) test as described by Cuevas et al. [13] in order to determine whether or not there is a statistically significant difference between the means of our groups, which are given by the number of semantic bugs per query.

4.2 Data collection

In this section we describe the approach used for building a large dataset of SQL queries collected from StackOverflow posts. StackOverflow is the main website where developers can ask questions and discuss various computer programming related issues which makes it one of the largest knowledge bases in this field. Apart from the 170,000 queries collected from StackOverflow, we also used for our study the dataset provided by Castelein et al. [11] in their paper, which contains an additional 19,159 SQL queries extracted from various open source projects on GitHub.

The technique we used for extracting SQL queries from StackOverflow posts focused on retrieving the code block sections from each post and parsing these for finding valid SQL statements. For this, using an API for accessing the available StackOverflow data was preferred over other techniques such as web crawling since using an API is less intrusive for the StackOverflow platform. Google's BigQuery public datasets now also include StackOverflow data, however these are not always up to date and further analysis showed these datasets are also not complete. In the end, for this paper, it was decided to use the Stack Exchange API[1] which provides an interface for retrieving posts data from StackOverflow. Our query extraction tool only looks at SQL tagged questions, since these are the most likely to contain the queries we are interested in. Furthermore, before starting to process the posts, we filtered them on the number of votes they had, and sorted them in descending order, such that the more relevant and higher quality posts were processed first. In total, there are roughly 11,000 pages with SQL tagged questions on StackOverflow. For our study, we processed and extracted the queries from the top ranked 2000 pages. Our tool is capable of processing all pages, however we considered that enough queries were extracted to build a good dataset on which we could test our rule-based static analysis tool. Further studies could use the tool we developed in order to obtain an even larger dataset of queries should this be needed.

The Stack Exchange API offers functionality for creating a custom filter which is then included in the request providing fine control over the returned data. For our project, all metadata associated with a post was collected since the goal was building a complete dataset which might later be used for other research as well. Furthermore, it is important to point out that a distinction should be made between a question post and an answer post. A question post can have multiple answers associated with it and should also have at most one accepted answer. All posts, questions and answers, are created by one user, and have a score associated with them which indicates the number of upvotes a question or an answer has received. The intuition behind the score value for questions is that the higher the number, the more developers found that particular question useful, whereas for answers the higher the score the better the answer.

The collected data was organized in a MySQL database with four important schemas: owners, questions, answers and queries. In the owners table, the data about the user who created the post is stored together with any other metadata associated with the user. In the questions table, the data for the questions posts is stored and similarly in the answers table the data for all other posts is stored. Finally, in the queries table all extracted queries are saved together with an identifier which links the query to the post from which it was extracted. Since StackOverflow has no way of enforcing SQL syntax to be written in code block sections, this means that users are free to input any type of text inside the code blocks. Because of this, a special algorithm has to be used for extracting only the SQL statements from these sections.

Our approach for query extraction can be seen as an island parsing technique, similar to the solution proposed in Moonen [32], where typically some recognizable structures are being parsed with detailed rules describing the constructs of interest, in our case the SQL keywords (the islands), and the remaining parts (the water), which in our case represent the overall query statement, are captured with other more generic rules. The algorithm used for extracting the queries from the code block section starts by first looking at every line and detecting whether it starts

[1]`https://api.stackexchange.com/`

with a valid SQL keyword. If not, a number of additional checks are carried out in order to determine whether the line is part of an enumeration which was split on multiple rows. If this is still not the case, then the line is eliminated from the code block. After all lines from a code block section are parsed, all of them are collected into a single query. In case users input multiple SQL queries the algorithm also detects this by looking for the special semicolon character which indicates the end of an SQL statement. Another approach was also considered for the query extraction algorithm, which constructed the abstract syntax tree (AST) for each of the code blocks, however since these blocks might also contain non-SQL syntax as well, it was observed that more queries were extracted with the previous approach. Furthermore, another potential method could have been using a tool such as GitHub linguist[2] for automatically detecting the type of code present in the snippets and continue with further processing if SQL would have been detected. However, we did not explore this approach, but it might be something interesting to investigate in future studies on mining data from StackOverflow. Finally, all extracted queries are saved in the MySQL database and can later be processed by other tools such as SQL parsers like JSQLParser for determining whether the syntax of the query is indeed valid. We provide some descriptive statistics for the collected dataset in Table 4.1. The tool which extracts the queries is also made available on the GitHub page for our project at the following link: `https://github.com/SERG-Delft/sql-bug-finder/tree/main/queries_stack_exchange`

Statistic	**Measurement**
Total queries	394,477
Valid syntax	172,232
Invalid syntax	22,2245
Average response time for SQL questions	14m 54s
Queries with semantic bugs	28,386
Queries with 1 semantic bug	24,229
Queries with 2 semantic bugs	3572
Queries with 3 semantic bugs	551
Queries with 4 semantic bugs	32
Queries with >5 semantic bugs	2

Table 4.1: General statistics for the StackOverflow dataset

4.3 Results

In the following sections we discuss the results for each of the research questions and present our findings.

[2] `https://github.com/github/linguist`

4.3.1 RQ1: What is the prevalence of semantic bugs in SQL queries?

We ran the rule-based static analysis tool containing 25 rules for detecting various SQL semantic bugs, as explained in Section 3.2, on all collected queries. We observe that out of all 191,994 queries, a total of 36,818 queries which contain at least one semantic bug were identified, meaning that 19.17% of queries contained some semantic problem in their formulation. Again, this shows the need for providing semantic bug detection tools for developers in order to improve the overall quality of SQL queries used in real world applications.

In Figure 4.1 we show the prevalence of the detected semantic bugs in our StackOverflow dataset. Out of all these bugs, the most frequent one is the missing join predicates semantic bug (`E019`), with a median prevalence of 5.98%. The second most frequent semantic bug is the constant output column (`E002`), which has a median prevalence of 3.62% closely followed by the unnecessary count argument bug (`E012`) with a median prevalence of 3.35%. We did not detect any semantic bugs related to the use of identical tuple variables (`E005`) in our collected dataset. For most of the other rules, the median prevalence was below 0.5%, which means that out of 200 queries, one will be affected by a semantic bug. We further analysed the rule for detecting missing join predicates by selecting 20 random queries for which this error was reported and manually checking the results of our tool. In all cases, we concluded that there were no false positive warnings and all errors were detected correctly. The same manual analysis was also conducted for all the other rules and we included the results in Appendix A.

Other previous studies such as those conducted by Brass and Goldberg [9] and Goldberg [20] also concluded that the missing join predicates semantic bug is indeed frequent in SQL queries. One noticeable difference between previous studies and our work comes from the datasets used. Most of the past work is based on queries collected from SQL courses, which means that queries are written by students on exams and other homework material. In our dataset, we used queries from various open source projects, hence the slightly lower median prevalence. Interestingly enough, Goldberg [20] reports a 3.6% prevalence for the constant output column semantic bug, again on a dataset of queries written by students on various exams, which is very close to the median prevalence we found in our study for the StackOverflow dataset, of 3.62%.

In Figure 4.2 we show the prevalence of the detected semantic bugs in the EvoSQL[3] dataset. This dataset only contains SQL `SELECT` queries found in three open source projects tracked on GitHub. Upon further manual inspection, it was observed that most of the queries suffered from the implicit columns SQL semantic code smell as described in the works of both Muse et al. [34] and Karwin [23]. In our implementation, this SQL semantic error corresponds to the constant output column bug (`E002`), and as it can be observed from the results of our tool, this bug is indeed the most frequent one in the EvoSQL dataset with a median prevalence of

[3] `https://www.zenodo.org/record/1166023`

43.23%. For all other semantic bugs, our tool detected a median prevalence of less than 0.4% for this dataset.

RQ1 summary: Out of all the collected queries, 19.17% contained at least one semantic bug. The most common bugs are the missing join predicates (`E019`) followed by the constant output column (`E002`) and unnecessary count argument (`E012`) bugs.

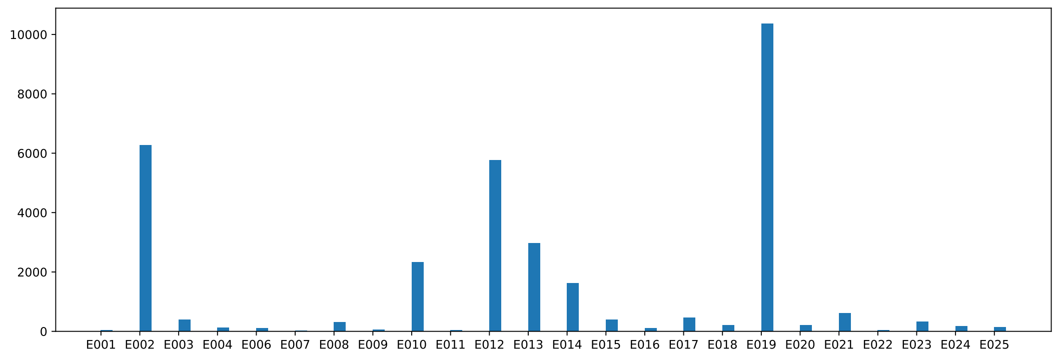


Figure 4.1: Prevalence of SQL semantic bugs for the StackOverflow dataset

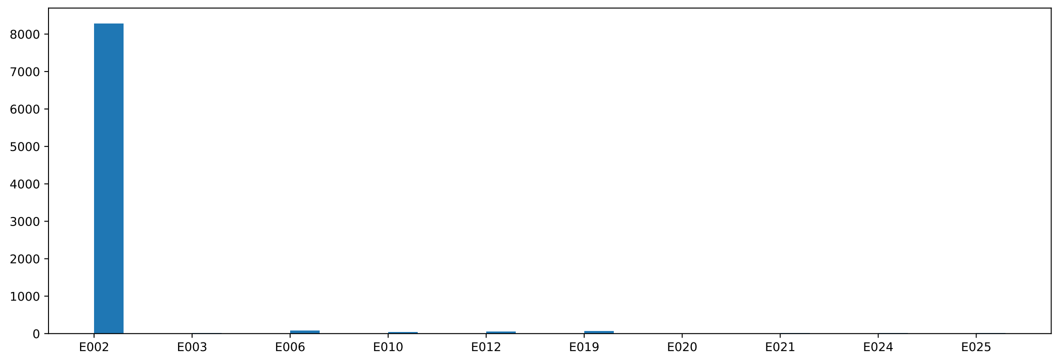


Figure 4.2: Prevalence of SQL semantic bugs for the EvoSQL dataset

4.3.2 RQ2: What are the co-occurrences of semantic bugs in SQL queries?

In Figure 4.3 we show the co-occurrence matrix for our entire dataset, where each entry in the matrix represents the Jaccard similarity coefficient as described by Eck and Waltman [15] (empty cells have value zero). From this, we observe that most of the similarity values are rather low, however there is a 20% similarity between bugs

E012 and E013, as well as a 15% similarity between bugs E012 and E019, followed by a 15% similarity for bugs E002 and E019. The Jaccard index is a measure of similarity between two sets of data, ranging from 0% to 100%, which means the higher the percentage, the more similar the two sets are to one another. In our case, for two bugs say A and B, if these have a 30% similarity measure, this means that if bug A is detected in one query, then there is a 30% probability that bug B is also present in this query.

This type of insight can be useful in designing future detection tools as well as prediction systems that could be integrated within IDEs in order to alert developers regarding certain bugs that their queries might contain given bugs which have already been detected. Further manual inspection for the queries where both the unnecessary count argument bug (E012) as well as the unnecessary group by attribute error (E013) both occur reveals that indeed these two types of semantic bugs are more likely to co-occur since usually the GROUP BY clause is used with the intention of aggregating some data in the SELECT clause.

Another interesting finding is that the missing join predicate bug (E019) is the most likely one to co-occur with other types of bugs. More concretely, there is a 15% similarity between queries containing bug E019 and either bug E002 or E012 and a 10% similarity between queries containing bug E019 and either bug E013 or E015. This also matches with the findings from **RQ1** which showed that bug E019 is the most prevalent one in our entire dataset, and as a result it co-occurs with 4 other different error types.

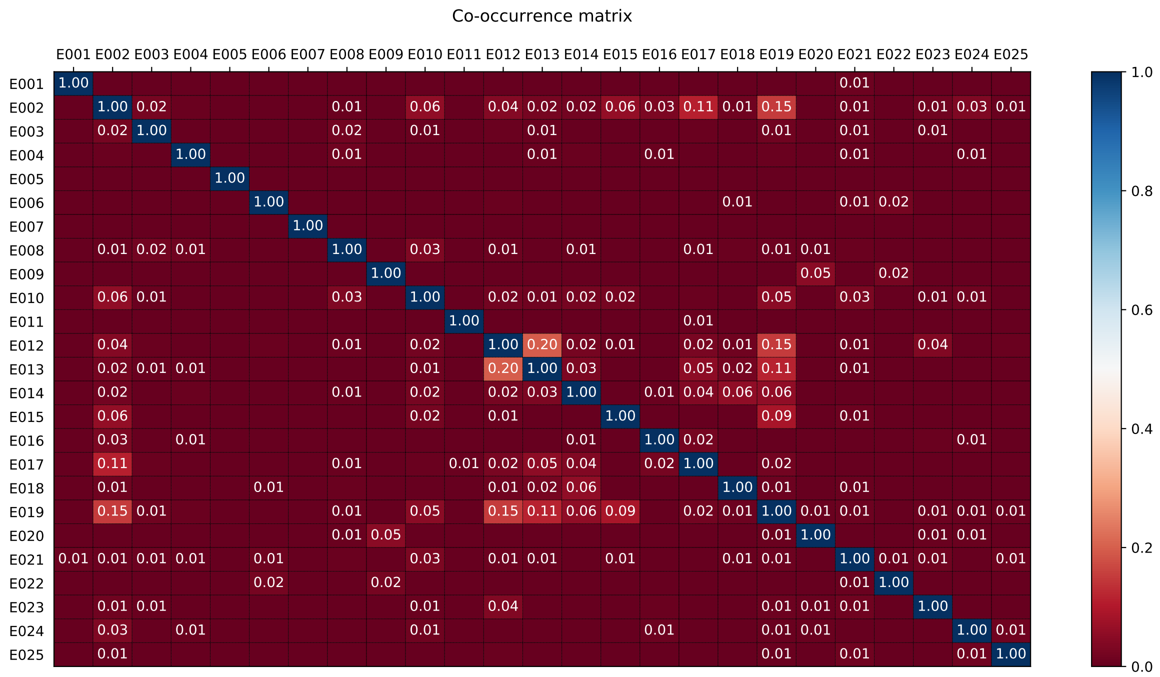
RQ2 summary: The co-occurrence of semantic bugs in SQL queries for our entire dataset is rather low, indicating that queries rarely contain more than one semantic bug. The highest similarity between two bugs is 20%, for the unnecessary count argument (E012) and unnecessary group by attribute (E013).

4.3.3 RQ3: What is the correlation between the complexity of a query and the number of semantic bugs it has?

For analysing the correlation between the complexity of the query and the number of semantic bugs it has we provide a box plot in Figure 4.4. In Figure 4.5 we also show the distribution of the query complexity scores. The data presented in these figures includes our entire collection of SQL queries, from both the StackOverflow and the EvoSQL datasets.

To further determine whether there is an correlation between the complexity of a query and the number of semantic bugs it has, we perform a one-way analysis of variance (ANOVA) test. For our one-way ANOVA test ($\alpha = 0.05$) we define the following null and alternative hypotheses:

- H_0 **(null hypothesis):** all the groups have the same mean, $\mu_1 = \mu_2 = \cdots = \mu_5$
- H_1 **(alternative hypothesis):** at least one of the means is different



	E001	E002	E003	E004	E005	E006	E007	E008	E009	E010	E011	E012	E013	E014	E015	E016	E017	E018	E019	E020	E021	E022	E023	E024	E025
E001	1.00																				0.01				
E002		1.00	0.02					0.01		0.06		0.04	0.02	0.02	0.06	0.03	0.11	0.01	0.15		0.01		0.01	0.03	0.01
E003		0.02	1.00					0.02		0.01			0.01						0.01		0.01		0.01		
E004				1.00				0.01					0.01			0.01					0.01			0.01	
E005					1.00																				
E006						1.00												0.01			0.01	0.02			
E007							1.00																		
E008		0.01	0.02	0.01				1.00		0.03		0.01		0.01			0.01		0.01	0.01					
E009									1.00											0.05		0.02			
E010		0.06	0.01					0.03		1.00		0.02	0.01	0.02	0.02				0.05		0.03		0.01	0.01	
E011											1.00						0.01								
E012		0.04						0.01		0.02		1.00	0.20	0.02	0.01		0.02	0.01	0.15		0.01		0.04		
E013		0.02	0.01	0.01						0.01		0.20	1.00	0.03			0.05	0.02	0.11		0.01				
E014		0.02						0.01		0.02		0.02	0.03	1.00		0.01	0.04	0.06	0.06						
E015		0.06								0.02		0.01			1.00				0.09		0.01				
E016		0.03		0.01										0.01		1.00	0.02							0.01	
E017		0.11						0.01			0.01	0.02	0.05	0.04		0.02	1.00		0.02						
E018		0.01				0.01						0.01	0.02	0.06				1.00	0.01		0.01				
E019		0.15	0.01					0.01		0.05		0.15	0.11	0.06	0.09		0.02	0.01	1.00	0.01	0.01		0.01	0.01	0.01
E020								0.01	0.05										0.01	1.00			0.01	0.01	
E021	0.01	0.01	0.01	0.01		0.01				0.03		0.01	0.01		0.01			0.01	0.01		1.00	0.01	0.01		0.01
E022						0.02			0.02												0.01	1.00			
E023		0.01	0.01							0.01		0.04							0.01	0.01	0.01		1.00		
E024		0.03		0.01						0.01						0.01			0.01	0.01				1.00	0.01
E025		0.01																	0.01		0.01			0.01	1.00

Figure 4.3: Co-occurrence matrix for SQL semantic bugs

We get that the corresponding p-value is 1.14e-168, thus since this values is less our chosen $\alpha = 0.05$, we reject the null hypothesis, and can therefore say that there is significant difference between the considered groups.

Computing the mean complexity score per category, we get that queries with a single semantic bug have a mean complexity of 7, queries with two bugs have a mean complexity of 10, queries with 3 bugs have mean complexity of 12, queries with 4 bugs have a mean complexity of 13 and finally queries with 5 bugs have a mean complexity of 26.

RQ3 summary: Performing an ANOVA test shows there is significant difference between the queries' complexities. There is also strong evidence which suggests that complex queries are more prone to suffer from SQL semantic bugs.

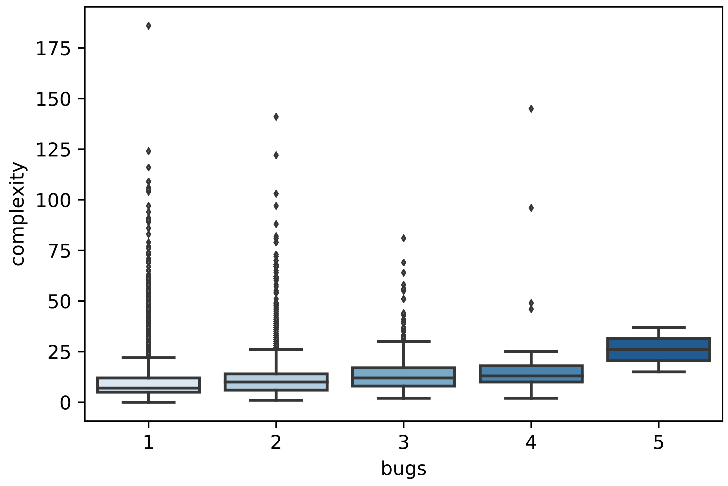


Figure 4.4: Number of semantic bugs per query complexity

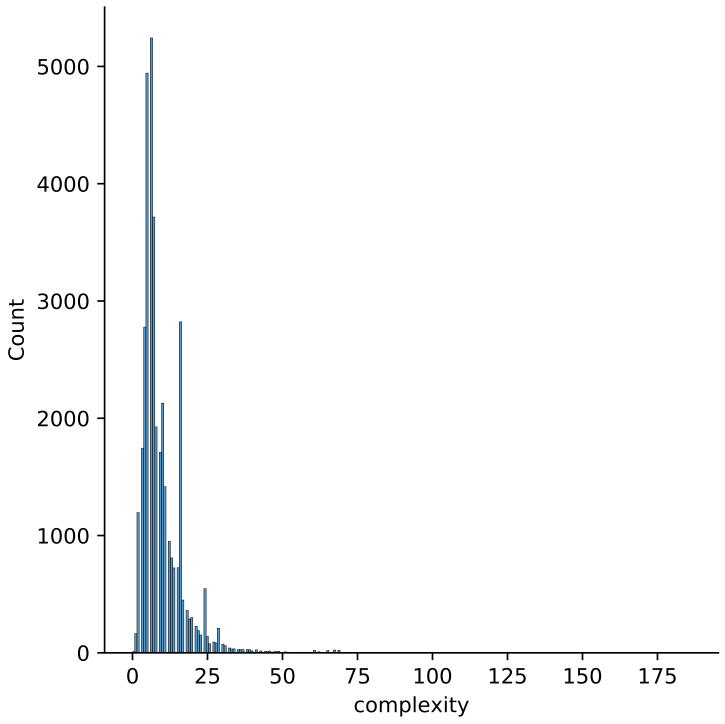


Figure 4.5: Distribution of query complexity

4.4 Threats to validity

In this section, we discuss the threats to the validity of this study and the actions we took to mitigate them.

Internal validity. The implementation of the heuristics for our rule-based static analysis tool can be subject to internal validity threats. To help with overcoming this, each of the 25 rules is covered by an extensive suite of unit tests, for both general as well as special edge case scenarios. Furthermore, we carried out a manual analysis of more than 500 queries, selecting for each of the 25 rules a random subset of 20 queries which were manually checked to ensure the results returned by the heuristics are correct. Finally, we also tried to align, as much as possible, the implementation of our heuristics, with the various implementation details found in previous research.

External validity. Although the collected dataset contains a diverse set of SQL queries, we observed that there are considerably more `SELECT` queries than other types, such as `INSERT`, `UPDATE` or `DELETE` queries, hence these latter types might be underrepresented in this study. This is especially noticeable in the EvoSQL dataset which only contains `SELECT` queries. However, by extracting queries from StackOverflow posts, we hope to have overcome some of these issues, build a diverse enough collection of queries and at the same time have our results generalize on SQL queries in any software system.

Chapter 5

Conclusion

SQL is the most widely used database language, with more than 70% of developers actively using it, as noted in a previous study [38], however there is lacking support for detecting semantic issues, also known as code smells, in SQL queries. This is especially worrying seeing the high popularity of this language and its wide adoption in numerous industries, therefore the impact of potential problems with such queries can not be overstated.

Our goal with this thesis is two-fold. First, we propose a set of 25 validated heuristics for detecting the most common types of semantic bugs which appear in SQL queries, based on evidence from previous research. Second, we conduct an empirical study on the prevalence of semantic bugs in SQL using two datasets with queries extracted from three open-source projects as well as more than 172,000 queries collected from StackOverflow posts, building the largest collection of SQL queries to our knowledge. The correctness of the heuristics was manually verified from a random sample of more than 500 queries.

We also provide a prototype implementation for a rule based static analysis tool, containing our 25 heuristics, for detecting semantic bugs. Our tool is able to detect semantic bugs in SQL queries with a 97% accuracy. Furthermore, we observed that out of all the 191,994 collected queries, 36,818 contain at least one semantic bug, which means that 19.17% of queries contained some semantic problem in their formulation.

Analysing the prevalence of semantic bugs in SQL queries, we find that the most common bugs are missing join predicates (`E019`) followed by the constant output column (`E002`) and unnecessary count argument (`E012`) bugs. We also found out that co-occurrence of semantic issues in SQL queries for our entire dataset is rather low, indicating that queries rarely contain more than one semantic bug. The highest similarity between two issues is 20%, for the unnecessary count argument (`E012`) and unnecessary group by attribute (`E013`).

Also quite interesting, we show that more complex queries, in terms of number of joins, predicates and functions used, tend to suffer from more semantic bugs, an interesting finding which could be used in the future as a metric for early prediction as to whether a query might contain semantic bugs or not.

5.1 Recommendations

In this section, we give some recommendations for developers, tool makers and researchers based on the findings made in our research.

Recommendations for developers. With so many applications relying on SQL for querying purposes, now more than ever it is crucial for developers to understand the implications of their queries suffering from various semantic issues. As we show in our empirical study, there are already quite a number of queries being shared online which already suffer from these types of problems, as it can be seen from Figure 4.1. Developers should therefore try to familiarize themselves not only with these semantic issues and how they appear when working with SQL, but they should also pay more attention to detecting these issues as well as addressing them as soon as they are discovered, just as bad code smells are addressed as well.

Recommendations for tool makers. The main focus here should be placed on designing tools for mass adoption. Choosing this as the main driving factor when building new tools should help with making sure that developers actually see value in the tool. More specifically, our findings show that it is currently hard for developers to check their SQL queries for semantic issues due to one underlying problem, which is the lack of adequate tools and support. Current implementations of tools that integrate with IDEs are lacking a lot when it comes to support for detecting semantic issues for queries, furthermore, there are also no tools, to our knowledge, which are integrated with popular development frameworks, such as SpringBoot[1], for checking these types of issues during application runtime. Especially interesting for future tools could be to integrate issue detection based on similarity measures, as we show in Figure 4.3 as well as early warnings based on metrics such as query complexity as we present in Figure 4.4.

Recommendations for researchers. Based on the findings made when carrying out this study, we can say that current research places more focus on trying to develop techniques and methods for identifying syntactic issues in SQL queries. However, as we show in this paper and as well as other studies have pointed out, more emphasis should be placed on detecting semantic issues in queries as well. With implications ranging from performance issues to security vulnerabilities, having these types of semantic errors in any system is definitely detrimental. Therefore, it is quite crucial that in the near future, more research involving semantic issue detection for SQL will be carried out.

[1]`https://spring.io/`

5.2 Future work

In this section we discuss the possible improvements that could be made to our detection tool as well as the query extraction algorithm and provide some ideas for future research which could make use of our large-scale SQL query dataset.

- **Integrate the rule-based static analysis tool for detecting semantic issues in SQL queries into an IntelliJ/Eclipse plugin.** By providing support for our tool as an IDE plugin, developers can take advantage of it more easily. Other benefits could be real time detection of issues, when queries are written in the IDE as well as during runtime, when the application is executed, as an integration with the SpringBoot framework.
- **Improve the query extraction algorithm.** For retrieving queries from StackOverflow it would be interesting to investigate whether better results (more queries extracted) could be achieved by using a tool such as GitHub's linguist[2] or Guesslang[3] for determining the programming language from within a code snippet.
- **Use our dataset to investigate whether developers notice SQL semantic issues in StackOverflow posts.** It would be interesting to analyse in future studies whether developers notice various types of semantic issues when either asking or replying to questions on StackOverflow. Another potential avenue for research could be to investigate the evolution of posts which might suffer from semantic issues in time and see if the problems are addressed.
- **Improve the rule-based static analysis tool by expanding the number of semantic issues which can be detected.** Although the tool is able to already detect the most common types of semantic issues which might occur in SQL queries, it would be very interesting to keep expanding this list by including other issues as well, such as the ones present in the closed source tool SQLEnlight. There is a comprehensive list[4] with some of the semantic rules which this tool supports, so in the future one could port these into our detection tool as well.

[2]`https://github.com/github/linguist`
[3]`https://github.com/yoeo/guesslang`
[4]`https://sqlenlight.com/support/help/analysis-rules/`

Bibliography

[1] Alireza Ahadi, Vahid Behbood, Arto Vihavainen, Julia Prior, and Raymond Lister. Students' syntactic mistakes in writing seven different types of sql queries and its application to predicting students' success. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, pages 401–406, 2016.

[2] Alireza Ahadi, Julia Prior, Vahid Behbood, and Raymond Lister. Students' semantic mistakes in writing seven different types of sql queries. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, pages 272–277, 2016.

[3] Miltiadis Allamanis and Charles Sutton. Why, when, and what: analyzing stack overflow questions by topic, type, and code. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 53–56. IEEE, 2013.

[4] Sebastian Baltes, Christoph Treude, and Stephan Diehl. Sotorrent: Studying the origin, evolution, and usage of stack overflow code snippets. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 191–194. IEEE, 2019.

[5] Anton Barua, Stephen W Thomas, and Ahmed E Hassan. What are developers talking about? an analysis of topics and trends in stack overflow. *Empirical Software Engineering*, 19(3):619–654, 2014.

[6] Stefanie Beyer and Martin Pinzger. A manual categorization of android app development issues on stack overflow. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 531–535. IEEE, 2014.

[7] Ilia Bider and David Rogers. Yasqlt–yet another sql tutor. In *International Conference on Conceptual Modeling*, pages 197–206. Springer, 2016.

[8] Stefan Brass and Christian Goldberg. Detecting logical errors in sql queries. In *Grundlagen von Datenbanken*, pages 28–32. Citeseer, 2004.

[9] Stefan Brass and Christian Goldberg. Semantic errors in sql queries: A quite complete list. *Journal of Systems and Software*, 79(5):630–644, 2006.

[10] Stefan Brass, Christian Goldberg, and Alexander Hinneburg. Detecting semantic errors in sql queries. Technical report, Technical Report, University of Halle, 2003.

[11] Jeroen Castelein, Maurício Aniche, Mozhan Soltani, Annibale Panichella, and Arie van Deursen. Search-based test data generation for sql queries. In *Proceedings of the 40th international conference on software engineering*, pages 1220–1230, 2018.

[12] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970. ISSN 0001-0782. doi: 10.1145/362384.362685. URL `https://doi-org.tudelft.idm.oclc.org/10.1145/362384.362685`.

[13] Antonio Cuevas, Manuel Febrero, and Ricardo Fraiman. An anova test for functional data. *Computational statistics & data analysis*, 47(1):111–122, 2004.

[14] Jens Dietrich, Markus Luczak-Roesch, and Elroy Dalefield. Man vs machine–a study into language identification of stack overflow code snippets. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 205–209. IEEE, 2019.

[15] Nees Jan van Eck and Ludo Waltman. How to normalize cooccurrence data? an analysis of some well-known similarity measures. *Journal of the American society for information science and technology*, 60(8):1635–1651, 2009.

[16] SQL Enlight. E0178: Like operator is used without wildcards. *SQL Enlight, Retrieved April*, 2021. URL `https://sqlenlight.com/support/help/SA0178/`.

[17] SQL Enlight. E0007: Pattern starting with "%" in like predicate. *SQL Enlight, Retrieved April*, 2021. URL `https://sqlenlight.com/support/help/SA0007/`.

[18] SQL Enlight. E0102: Do not use distinct keyword in aggregate functions. *SQL Enlight, Retrieved April*, 2021. URL `https://sqlenlight.com/support/help/SA0102/`.

[19] Felix Fischer, Konstantin Böttinger, Huang Xiao, Christian Stransky, Yasemin Acar, Michael Backes, and Sascha Fahl. Stack overflow considered harmful? the impact of copy&paste on android application security. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 121–136. IEEE, 2017.

[20] Christian Goldberg. Do you know sql? about semantic errors in database queries. In *7th Workshop on Teaching, Learning and Assessment in Databases, Birmingham, UK, HEA*. Citeseer, 2009.

[21] Carl Gould, Zhendong Su, and Premkumar Devanbu. Jdbc checker: A static analysis tool for sql/jdbc applications. In *Proceedings. 26th International Conference on Software Engineering*, pages 697–698. IEEE, 2004.

[22] Huzefa Kagdi, Michael L Collard, and Jonathan I Maletic. An approach to mining call-usage patternswith syntactic context. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 457–460, 2007.

[23] Bill Karwin. *SQL antipatterns: avoiding the pitfalls of database programming.* Pragmatic Bookshelf, 2010.

[24] Shaheen Khatoon, Guohui Li, and Azhar Mahmood. Comparison and evaluation of source code mining tools and techniques: A qualitative approach. *Intelligent Data Analysis*, 17(3):459–484, 2013.

[25] Yash Lamba, Manisha Khattar, and Ashish Sureka. Pravaaha: Mining android applications for discovering api call usage patterns and trends. In *Proceedings of the 8th India Software Engineering Conference*, pages 10–19, 2015.

[26] Mario Linares-Vásquez, Bogdan Dit, and Denys Poshyvanyk. An exploratory analysis of mobile development issues using stack overflow. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 93–96. IEEE, 2013.

[27] MITRE. Weaknesses in software written in C. *MITRE, Retrieved April*, 2021. URL `https://cwe.mitre.org/data/definitions/658.html`.

[28] MITRE. CWE Top 25 most dangerous software errors. *MITRE, Retrieved April*, 2021. URL `https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html`.

[29] Antonija Mitrovic. A knowledge-based teaching system for sql. In *Proceedings of ED-MEDIA*, volume 98, pages 1027–1032, 1998.

[30] Antonija Mitrovic. Learning sql with a computerized tutor. In *Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education*, pages 307–311, 1998.

[31] Antonija Mitrovic. An intelligent sql tutor on the web. *International Journal of Artificial Intelligence in Education*, 13(2-4):173–197, 2003.

[32] Leon Moonen. Generating robust parsers using island grammars. In *Proceedings Eighth Working Conference on Reverse Engineering*, pages 13–22. IEEE, 2001.

[33] Tamara Munzner. *Visualization analysis and design.* CRC press, 2014.

[34] Biruk Asmare Muse, Mohammad Masudur Rahman, Csaba Nagy, Anthony Cleve, Foutse Khomh, and Giuliano Antoniol. On the prevalence, impact, and evolution of sql code smells in data-intensive systems. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 327–338, 2020.

[35] Csaba Nagy and Anthony Cleve. Mining stack overflow for discovering error patterns in sql queries. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 516–520. IEEE, 2015.

[36] Csaba Nagy and Anthony Cleve. A static code smell detector for sql queries embedded in java code. In *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 147–152. IEEE, 2017.

[37] Csaba Nagy and Anthony Cleve. Sqlinspect: A static analyzer to inspect database usage in java applications. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 93–96, 2018.

[38] Stack Overflow. Stack overflow annual developer survey 2020. *Retrieved February*, page 2021, 2021.

[39] Luca Ponzanelli, Alberto Bacchelli, and Michele Lanza. Seahawk: Stack overflow in the ide. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 1295–1298. IEEE, 2013.

[40] Luca Ponzanelli, Andrea Mocci, and Michele Lanza. Stormed: Stack overflow ready made data. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 474–477. IEEE, 2015.

[41] Toni Taipalus. Explaining causes behind sql query formulation errors. In *2020 IEEE Frontiers in Education Conference (FIE)*, pages 1–9. IEEE, 2020.

[42] Toni Taipalus and Piia Perälä. What to expect and what to focus on in sql query teaching. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, pages 198–203, 2019.

[43] Toni Taipalus and Ville Seppänen. Sql education: A systematic mapping study and future research agenda. *ACM Transactions on Computing Education (TOCE)*, 20(3):1–33, 2020.

[44] Toni Taipalus, Mikko Siponen, and Tero Vartiainen. Errors and complications in sql query formulation. *ACM Transactions on Computing Education (TOCE)*, 18(3):1–29, 2018.

[45] Yuhao Wu, Shaowei Wang, Cor-Paul Bezemer, and Katsuro Inoue. How do developers utilize source code from stack overflow? *Empirical Software Engineering*, 24(2):637–673, 2019.

[46] Tao Xie and Jian Pei. Mapo: Mining api usages from open source repositories. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 54–57, 2006.

[47] Haoxiang Zhang, Shaowei Wang, Heng Li, Tse-Hsun Peter Chen, and Ahmed E Hassan. A study of C/C++ code weaknesses on stack overflow. *IEEE Transactions on Software Engineering*, 2021.

Appendix A

Tool Validation

For validating the correctness of the heuristics we implemented for our tool, a manual analysis of more than 500 queries was done. Each query was manually checked and the results returned by our tool were verified to detect, for each rule, how many of the reported warnings were valid. As presented in Section 3.4, a random set of 20 queries was selected for each rule. All queries can be found in our replication package on GitHub as well. We present the analysis below.

No.	RuleID	Query	Correct
1	E001	`select a1.owner,a1.table_name from all_tab_columns a1 , all_tab_columns a2 where a1.owner=a2.owner and a1. table_name=a2.table_name and a1.column_name='location ' and a2.column_name='asset_id' order by a1.owner,a1. table_name;`	*YES*
2	E001	`select t1.userid from userrole t1 join userrole t2 on t1 .userid = t2.userid and t2.roleid = 2 join userrole t3 on t2.userid = t3.userid and t3.roleid = 3 and t1.roleid = 1;`	*YES*
3	E001	`select deptno from dept where exists(select * from emp x where x.deptno = 20 and exists(select * from emp y where y.job = x.job and y.deptno = dept.deptno)) and deptno <> 20;`	*YES*
4	E001	`select count(distinct patients.id) from public. patients, public.subscriptions, public.users, public. calendar_days where patients.user_id = users.id and patients.id = calendar_days.patient_id and subscriptions .user_id = patients.user_id and (date_trunc('day', patients.last_sync) > current_date - inter...`	*YES*
5	E001	`select * from tmp t0 where exists (select * from tmp t1 join tmp t2 on t2.clientid = t1.clientid and t2. serverid = t1.serverid and t2.logtime > t1.logtime where t1. status = 'aborted' and t2. status = 'failed' and t1 .clientid = t0.clientid and t1.serverid = t0.serverid and t1.logtime = t0.logtime o...`	*YES*

6	E001	`select distinct t.name from hospital_a t, hospital_a v where t.op_date=v.date_of_birth and t.surname=v.surname and t.op_id = 'p619920' and v.op_id = 'i552015';`	*NO*
7	E001	`select im1.imageid as id, im1.attributevalue as color, im2.attributevalue as quality from imageattribute im1, imageattribute im2 where im1.imageid = im2.imageid and im1.attributetype = "color" and im2.attributetype = " quality" and im2.attributevalue = "good";`	*YES*
8	E001	`select t1.a3 from test t1, test t2 where t1.a1 = t2.a1 and t2.a2 = t1.a2 and t1.a1 = t2.a2;`	*YES*
9	E001	`select * from audit a1 where not exists( select * from audit a2 where a2.status='approve' and a1.status=' decline' and a2.id=a1.id );`	*YES*
10	E001	`select a.post_id, b.post_id, a.ul_value as "likes", b .ul_value as "dislikes" from wp_like_dislike_counters as a, wp_like_dislike_counters as b where a.post_id = b.post_id and a.ul_key = 'u_like' and b.ul_key = ' u_dislike';`	*NO*
11	E001	`update users set online = 1 where exists(select null from users t where t.email = in_email and t.password = in_password and t.id = id) and id = 'result_id';`	*YES*
12	E001	`select p1.domain_id, p2.domain_id, count(p1.domain_id) as d1, count(p2.domain_id) as d2 from pdb as p1, interacting_pdbs as i1, pdb as p2, interacting_pdbs as i2 where p1.id = i1.pdb_first_id and p2.id = i2. pdb_second_id and i1.id = i2.id group by p1.domain_id, p2.domain_id having d1 > 100 and d2 > ...`	*YES*
13	E001	`select a.object from mytable a, mytable b where a.object = b.object and a.key = 'a' and a.value = 'a' and b.key = 'b' and b.value = 'b';`	*YES*
14	E001	`select * from cats outside where not exists(select * from cats cat where exists( select dog.foo,dog.bar from dogs dog where cat.foo = dog.foo and cat.bar = dog.bar) and outside.foo = cat.foo and outside.bar=cat.bar );`	*YES*
15	E001	`select username.name, useremail.email, userphone.phone from users as username inner join users as useremail on username.user = useremail.user and username.field = 'name' and useremail.field = 'email' inner join users as userphone on username.user = userphone.user and userphone.field = 'phone';`	*NO*
16	E001	`select count(*) from ( select u1.userid from vote u1, vote u2 where u1.itemid = u2.itemid and u1.userid = user1 and u2.userid = user2);`	*YES*
17	E001	`update mark set mark= case when mark.val<= 5 then val*1.1 else val end where mark.id_classes = classes. id_classes and classes.id_subject = subject.id_subject and subject.id_subject = 5;`	*YES*
18	E001	`select a1.owner,a1.table_name from all_tab_columns a1 , all_tab_columns a2 where a1.owner=a2.owner and a1. table_name=a2.table_name and a1.column_name='location ' and a2.column_name='asset_id' order by a1.owner,a1. table_name;`	*YES*

19	E001	`select p1.gameid from participants as p1, participants as p2 where p1.name = 'team1' and p2.name='team2' and p1 .gameid = p2.gameid;`	*NO*
20	E001	`select n1.id, n1.name,n2.name, a3.age from name n1, name n2, age a3 where n1.id=n2.id and n1.id=a3.id and n1. type=0 and n2.type=1;`	*YES*

Table A.1: Manual analysis of tool performance for rule E001

No.	RuleID	Query	Correct
21	E002	`select * from mytable1 ca left outer join mytable2 dcn on dcn.dstrct_code = ca.dstrct_code where ca. dstrct_code = '0001' and ca.req_232_type = 'p' and ca .requisition_no = '264982 000' and ca.alloc_count = '01' order by ca.alloc_count asc;`	*YES*
22	E002	`select * from ranked where ranking = 1;`	*YES*
23	E002	`select * from orc_users as t1 inner join orc_files as t2 on t1.id = t2.userid where t1.email='sdfsdf';`	*YES*
24	E002	`select * from person where name = 'robert';`	*YES*
25	E002	`select * from myuser as u join friend_pivotal as p on u. id = p.user_rec_id or u.id = p.user_inv_id where status = 'accepted';`	*YES*
26	E002	`select * from (select * from table order by value desc, date_column) where rownum = 1;`	*YES*
27	E002	`select * from a_table where a_column = '&avariable ' union select * from a_table where b_column = '&& avariable';`	*YES*
28	E002	`select incurredcharges.procedure_no, incurredcharges. patient_no, charges.procedure from incurredcharges inner join charges where incurredcharges.patient_no=12;`	*YES*
29	E002	`select u.uid ,u.uname,p.uid , p.profname,e.uid,e.eduname from user u inner join profession p on u.uid=p.pid inner join education e on u.uid = e.uid where u.uid=p. uid and u.uid=e.uid and i.uid=1;`	*YES*
30	E002	`select arbogast.node.nid as anid, mcguffin.node.nid as mnid, arbogast.node.title as atitle, mcguffin.node. title as mtitle from arbogast.node, mcguffin.node where arbogast.node.nid = 1 and mcguffin.node.nid = arbogast. node.nid;`	*YES*
31	E002	`select imagename, sum(ratingvalue) as "lol" from ratings group by imagename having imagename = 'myimagename';`	*YES*
32	E002	`select max(id) as id, type, max(other_id) as other_id, max(def_id) as def_id, ref_def_id from t where type = ' ref' group by type, ref_def_id;`	*YES*
33	E002	`select distinct name from sys.objects where type in ('u ','v') and name= 'myname';`	*YES*
34	E002	`select customerid, orderdate, count(1) cnt from sales .salesorderheader where customerid = 11300 group by customerid, orderdate order by cnt desc;`	*YES*

35	E002	`select booking.bookno, booking.courno, course.coursename from booking, course, coursename where booking.bookno = 6200 and booking.courno = course.courno and coursename. coursenameno = course.coursenameno;`	*YES*
36	E002	`select table1.personcode, table1.name, table2.location, max(table2.servicedate) from table1 inner join table2 on table1.id = table2.table1id where table1.personcode = 'xyz' group by table1.personcode,table1.name, table2. location;`	*YES*
37	E002	`select t1.userid, t2.date, min(t1.time) as in_time, max( t1.time) as out_time from test t1 join (select distinct date from test where userid = 609) t2 where t1.date = t2 .date and userid = 609 group by t1.userid, t2.date;`	*YES*
38	E002	`select * from mm_tfs where product_description like '% football%' and schoolid = '8' and category_id ='21';`	*YES*
39	E002	`select * from customer where c_role = 'dev' order by c_id limit 2;`	*YES*
40	E002	`select attrsmall, attrlarge, max(rating_id) as ratingmax from (select case when c.attr1_id < c.attr2_id then c .attr1_id else c.attr2_id end as attrsmall, case when c .attr1_id < c.attr2_id then c.attr2_id else c.attr1_id end as attrlarge, c.rating_id from compatibility c) as c1 group by atrrsmall, a...`	*YES*

Table A.2: Manual analysis of tool performance for rule E002

No.	RuleID	Query	Correct
41	E003	`insert into personpet(fk_person, fk_pet) select id, id from person;`	*YES*
42	E003	`select identity( int ) as tempid, *, sectionid as fix2ids from files_sections;`	*YES*
43	E003	`select *, cond2 as cond2, cond3 as cond3 from table having cond1 and (cond2 or cond3);`	*YES*
44	E003	`select country, count(*) as cnt1, count(*) as cnt2 from orders group by country having cnt1=2 and cnt2>2;`	*YES*
45	E003	`select c77,c77,c125,c126,c127,c74 from mytable;`	*YES*
46	E003	`select itemnumber as '@id', itemnumber as 'itemnumber', price as 'price/@value', datefrom as 'price/datefrom', dateto as 'price/dateto' from #tempxml;`	*YES*
47	E003	`select videos.id, videos.id as video_id, videos. video_title as video_title, group_concat(distinct t. tag_name separator '|') as tag_names from videos join video_tags as vt join tags as t where videos.id <= ( select max_doc_id from sph_counter where counter_id = 1 ) group by videos.id;`	*YES*
48	E003	`select distinct t.date_effective, t. acct_account_transaction_id, p.method, t.amount, c. business_name, t.amount from contact c join contact_role cr on cr.contact_fk = c.contact_id join acct_account a on a.contact_fk = c.contact_id join acct_account_tran...`	*YES*

49	E003	`insert into test_table(column_1, column_2) select val, val from x;`	*YES*
50	E003	`select e.empid,e.fname,e.lname,c.description as hair,c.description as race from employee2 e inner join code c;`	*YES*
51	E003	`select uuid, uuid from data;`	*YES*
52	E003	`select value, value from public.customers where nodevalue.key3 = 'key3' and nodevalue.key4 = 'key4';`	*YES*
53	E003	`select owner, count(distinct object_name || 'this is a really long string in my expression, don''t you think? actually, it''s really, reall...`	*YES*
54	E003	`select projects.project_id, projects.title, projects.start_time, projects.description, projects.user_id, projects.winner_user_id, users.username as owner, users.username as winner from projects,users where projects.user_id=users.user_id and projects.winner_user_id=users.user_id;`	*YES*
55	E003	`select productgroupid as product23_1_, articleid as articleid1_, articleid as articleid18_0_, inventory_name as inventory3_18_0_, inventory_unitofmeasure as inventory4_18_0_, businesskey as business5_18_0_, name as name18_0_, servespeople as servespe7_18_0_, instock as instock18_0_, description as d...`	*YES*
56	E003	`select *, t1.image_id as image from event.dbo.dia_tracker t1 where exists (select 1 from event.dbo.dia_tracker t2 where t2.patient_id = 'dsma' group by case when t2.active = '1' then t2.image_id end having max(t2.image_id ) = t1.image_id);`	*YES*
57	E003	`select *, 'full' from abc;`	*YES*
58	E003	`select 1 except select 1;`	*NO*
59	E003	`select *, regexp_replace(port, '[0-9/.]', '', 'g') port_name, string_to_array(regexp_replace(port, '[a-za-z-]', '', 'g'), '/')::float[] port_number from device order by name, regexp_replace(port, '[0-9/.]', '', 'g'), string_to_array(regexp_replace(port, '[a-za-z-]', '', 'g'), '/')::float[];`	*YES*
60	E003	`select *, match(pages) against('doodle') as score from books where match(pages) against('doodle') order by score desc;`	*YES*

Table A.3: Manual analysis of tool performance for rule E003

No.	**RuleID**	**Query**	**Correct**
61	E004	`select emp.salary from employee emp inner join sap s on emp.id = s.id where s.id = 111;`	*YES*
62	E004	`select * from child c,parent p where c.id=p.parentid and c.parentid !=0;`	*YES*
63	E004	`select t1.col_1 from table t1 join table t2 on t1.col_2=t2.col_1 where t1.col_1=t2.col_2 and t1.col_1='one';`	*YES*

64	E004	select m.actor from movies m where m.movie = 'pulp fiction' and not exists (select 1 from movies m1 join movies m2 on m1.movie = m2.movie and m1.actor <> m2.actor and m2.movie <> 'pulp fiction' and m2.actor in (select actor from movies where movie = 'pulp fiction') where m.actor = m1.actor);	*YES*
65	E004	select c.name, if(find_in_set('type_a', group_concat(substring_index(f1.filename,'_',2))), 'yes', 'no') as type_a, if(find_in_set('type_b', group_concat(substring_index(f1.filename,'_',2))), 'yes', 'no') as type_b, if(find_in_set('type_c', group_concat(substring_index(f1.filename,'_',2))), 'yes', 'n...	*YES*
66	E004	select u.firstname, u.lastname, u.rep, u.email, u.password, u.gender, u.level, u.birthday, u.achievements, u.height, u.unit, u.cityid, u.countryid, r.regdate, ci.name as city, co.name as country from users u, registry r, cities ci, countries co where u.id = 1 and r.uid = u.id and u.cityid = ci.id an...	*YES*
67	E004	select images.*, users.username from images left join users on images.user_id = users.id left join user_follow on images.user_id = user_follow.follow_id where images.user_id = 3 or user_follow.user_id = 3 order by images.date desc;	*YES*
68	E004	select * from tblpe t1 where t1.date = (select max(date) from tblpe t2 where t1.id = t2.id) and t1.id = 39;	*YES*
69	E004	select pd.id,pd.price_date,pd.name_id,pd.class_id,pd.currency_id,pd.price, pd.price - (select price from price_data as x where x.price_date < pd.price_date and x.name_id = pd.name_id and x.class_id = pd.class_id and x.currency_id = pd.currency_id having max(x.price_date)) as `change` from ...	*YES*
70	E004	select activitytext, actiontext from activity join activityaction on activity.activityid = activityaction.activityid join action on activityaction.actionid = action.actionid where activity.activityid = 1;	*YES*
71	E004	select studentid, firstname, lastname, gender from student join major on student.majorid = major.majorid where major.majorid = 2;	*YES*
72	E004	select col.column_name, col.constraint_name from information_schema.constraint_column_usage col where col.constraint_name = tab.constraint_name and col.table_name = tab.table_name and constraint_type = 'primary key' and col.constraint_name like '%adhoc%';	*YES*
73	E004	update purchase as p inner join artwork as a on p.purchaseid = a.purchaseid set p.total = sum(a.price) where a.purchaseid = 'd4758';	*YES*
74	E004	select client.*, cat1id.client_catid_1 as cat1, cat2id.client_catid_2 as cat2 from tb_clients as client left join tb_clients_categories cat1id on client.client_id = cat1id.client_id left join tb_clients_categories cat2id on client.client_id = cat2id.client_id where client.client_id = 65447;	*YES*

75	E004	`select c.c_id, s.s_fname, count(r.c_id) from results r, candidates c, student s,positioning p, organization o where r.c_id = c.c_id and c.sid = s.sid and c.pos_id = p .pos_id and o.org_id = c.org_id and o.org_id = 1 group by c.c_id;`	*YES*
76	E004	`select pt.projecttaskid, isnull(sum(case when pte. assigneduserid = @userid then 1 end),0) as assignedtome, isnull(sum(case when pte.assigneduserid <> @userid then 1 end),0) as assignedtoothers from projecttask pt inner join project p on p.projectid = pt.projectid left join projecttaskentity pte on p...`	*YES*
77	E004	`select ip_settings.ip from server inner join network_interfaces on network_interfaces.id = server .fk_network_interfaces inner join ip_settings on ip_settings.id = network_interfaces.fk_ip_settings where server.id=6;`	*YES*
78	E004	`update table_to tt, table_from tf set tt.name = "chandi" where tt.id = tf.id and tf.id = 1;`	*YES*
79	E004	`select t.* from terms t join term_relationships tr on tr.term_id = t.term_id join posts p on p.post_id = tr. post_id where p.post_id = 1;`	*YES*
80	E004	`select t1.service from table1 t1, table2 t2 where t1. cnty = t2.cnty and t1.zip = t2.zip and t2.zip = '1234';`	*YES*

Table A.4: Manual analysis of tool performance for rule E004

No.	RuleID	Query	Correct
81	E006	`select * from (values (1), (2)) as tbl(col) where not ( col = null or col = 1);`	*YES*
82	E006	`select r.id, r.authtoken.instagram,r.username from root r where r.abc <> null;`	*YES*
83	E006	`select loans.*, if(ul.name = null, ub.name, ul.name) as name, if(ul.id = null, ub.id,ul.id) as uid from loans left join users ul on users.id = loans.lender_id left join users ub on users.id = loans.borrower_id where ul. id = 2 or ub.id = 2;`	*YES*
84	E006	`select count(*) from table1 where request_time < timestamp'2012-05-19 14:00:00' and (end_time > timestamp '2012-05-19 14:00:00' or end_time=null);`	*YES*
85	E006	`select * from my_table where some_field = null;`	*YES*
86	E006	`select * from customer c where so.orderid = null;`	*YES*
87	E006	`select name from bbc where gdp > all (select gdp from bbc where region = 'africa' and gdp<>null);`	*YES*
88	E006	`select * from hotel where address != null;`	*YES*
89	E006	`select * from t1 where a=b or (a=null and b=null);`	*YES*
90	E006	`select concat(area, yearlevel, code) as subjectcode, count(student) from studenttakessubject where result < 50 and result <> null group by code having count(student ) > 1;`	*YES*

91	E006	select checklists.id from checklists left join (select checklist_id from checklist_items where completed = 'f' union distinct select checklist_id from checklist_items group by checklist_id having count (*) = 0) partial_checklists where partial_checklists. checklist_id = null;	*YES*
92	E006	select lefttable.id, righttable.id as nullid from lefttable left join righttable on righttable.id = lefttable.id where nullid = null;	*YES*
93	E006	select e.id from employments e where not exists (select 1 from emails em where em.employment_id=e.id and em. deactivated_on = null);	*YES*
94	E006	select secid, min(date) as startdate, max(date) as enddate, price from bigtable group by secid, enddate having min(date) != max(date) and date != null;	*YES*
95	E006	select * from student left join course where student. std_id = null or student.std_name = null or student. std_start_date = null or student.std_end_date = null or student.std_gender = null or student.course_id = null;	*YES*
96	E006	select top (10) ctry, sales from table1 union all select 'other', sum(sales) from table1 where table2.ctry = null group by table1.ctry;	*YES*
97	E006	update salesorders o set transref = (select t.transref from transfers t where o.orderno = t.orderno and (o. ordext = t.ordext or (t.ordext=null and o.ordext=null)) and t.transref <> null) where ordext = null;	*YES*
98	E006	select h.clientnumber, iif(h.checkoutdate=null,"yes ","") as currentvisitor from visitstable as h inner join (select clientnumber, max(lastvisitdate) as lastvisitstart from visitstable group by clientnumber) as t;	*YES*
99	E006	select * from t left join m on t.sub_id = m.id where t. sub_id != null;	*YES*
100	E006	select m.provider_id, m.provider_name, p. purchase_order_code, null as purchase_order_sample_code, p.provider_id as provider_order from mst_provider as m left join trx_purchase_order as p where p.provider_id != null union select m.provider_id, m.provider_name, p. purchase_order_code, null as purchase_...	*YES*

Table A.5: Manual analysis of tool performance for rule E006

No.	**RuleID**	**Query**	**Correct**
101	E007	select sum(amt) where session=x and order >= (select max(order) where atype='set' and session=x);	*YES*
102	E007	select t.*, row_number() over(partition by "name" order by "date") as rank from tablea t where "date" >= (select max("date") from tablea where "type" = 'b');	*YES*
103	E007	select * from `table_name` where id >= (select floor(max(id) * rand()) from `table_name`) order by id limit 30;	*YES*

104	E007	`select id from table_name where id >= (select max(id) from table_name where id <= 12);`	*YES*
105	E007	`select thedate from datetable where thedate >= (select max(thedate) from datetable where thedate < getdate());`	*YES*
106	E007	`select distinct maker, price from product inner join printer where color = 'y' and price <= (select min(price ) from printer where color = 'y');`	*YES*
107	E007	`select * from numbers where nr >= (select max(nr) from numbers where nr < 6.20) and nr <= (select min(nr) from numbers where nr > 6.20);`	*YES*
108	E007	`select name, message from flux_chat_messages where id >= ( max( id ) -5 ) order by id asc;`	*YES*
109	E007	`select submissionid, input, value from yourtable where submissionid >= (select max(submissionid) from yourtable ) - 1 order by submissionid;`	*YES*
110	E007	`select * from yourtable where id >= 4 and id <= (select min(id) from yourtable where b = 'f' and id >= 4);`	*YES*
111	E007	`select * from my_table where datetime_column >= date_sub ((select max(datetime_column) from my_table), 7);`	*YES*
112	E007	`select distinct d.phonenum,d.sourcetable,n.fullname ,c.fk_applicationid as ref,t.subject,t.createddate from dial d join database.dbo.dm_phonenumbers p on p. phonenum1 = d.phonenum collate latin1_general_ci_as join database.dbo.dm_phonenumbers on p.phonenum2 = d. phonenum collate latin1_general_ci_as jo...`	*YES*
113	E007	`select * from yourtable where id >= 4 and id <= coalesce ( (select min(id) from yourtable where b = 'f' and id >= 4), (select max(id) from yourtable ) );`	*YES*
114	E007	`select * from users where id >= (select floor((max(id) - min(id) - 1) * rand()) + min(id) from users) limit 1;`	*YES*
115	E007	`select foo from bar where id >= (abs(random()) % (select max(id) from bar)) limit 1;`	*YES*
116	E007	`select ename, job, sal from emp where sal >=(select max (sal) from emp where sal < (select max(sal) from emp where sal < (select max(sal) from emp))) order by sal;`	*YES*
117	E007	`select foo from bar where _rowid_ >= (abs(random()) % ( select max(_rowid_) from bar)) limit 1;`	*YES*
118	E007	`select * from customers where cno = ( select cno from ( select count(*) as ordcount, cno from orders group by cno having ordcount >= ( select max(ordcount) from ( select count(*) as ordcount, cno from orders group by cno ) ) ) );`	*YES*
119	E007	`select * from table where id >= (select max(id) from table) - 10 order by id desc;`	*YES*
120	E007	`select name from world where gdp >= (select max(gdp) from world where continent = 'europe');`	*YES*

Table A.6: Manual analysis of tool performance for rule E007

No.	**RuleID**	**Query**	**Correct**

121	E008	`update run inner join snp_hgvs set run.comment=concat(' rs',snp_hgvs.snp_id) where run.compute not like 'tron';`	*YES*
122	E008	`insert into cesc_pf_stmt_ext_wrk( pf_emp_code , pf_dept_code , pf_sec_code , pf_prol_no , pf_fm_seq , pf_seq_no , pf_sep_tag , pf_source) select pfl_emp_code , pfl_dept_code , pfl_sec , pfl_prol_no , pf_fm_seq , pf_seq_no , pfl_sep_tag , pf_...`	*YES*
123	E008	`select id into id_historical from historical where volume_id = id_volume and (action like 'expedicao' or action like 'conference');`	*YES*
124	E008	`select * from first_test_name where firstname like 'bob ';`	*YES*
125	E008	`select * from table where fiels1 not like 'x' and field2 not like 'y';`	*YES*
126	E008	`select name from memberdb where name like 'lim %' or name like '% lim' or name like '% lim %' or name like ' lim';`	*YES*
127	E008	`select forum.user.userid, forum.user.usergroupid, forum .user.password, forum.user.salt, forum.user.pmunread, forum.subscriptionlog.expirydate from forum.user inner join forum.subscriptionlog where forum.user.username like 'someuser';`	*YES*
128	E008	`select column_name from information_schema.columns where table_name='lime_survey_98673' and column_name like '98673';`	*YES*
129	E008	`select * from emp where ename like 'king';`	*YES*
130	E008	``select *, count(`event_target`) `totalsum` from `testdb ` where `account_id` = ? and (`event_target` like ' searchquery' or `event_title` like 'searchquery' or ` event_name` like 'searchquery') group by `username`, ` event_title`, `event_target` order by `totalsum` desc;``	*YES*
131	E008	`select * from your_name where category_id not like '90';`	*YES*
132	E008	`insert into table1(user_uuid, login_id) select users_uuid, '1234' from table2 where first_name like 'ortal';`	*YES*
133	E008	`select * from products where p.name not like 'brand1%' and p.name not like 'specific product';`	*YES*
134	E008	`select busname, email, render_pic, area,logo, url, email, map, description, tag, catch_phrase, region from results where style like 'varstyle' and region like ' varregion' and bedrooms like 'varbedrooms' and bathrooms like 'varbathrooms' and price between varminprice and varmaxprice order by id de...`	*YES*
135	E008	``select c.id from `tags` `t` left join comictags as ct left join comics as c where ((t.tag like 'tag1') or (t. tag like 'tag2')) group by c.id;``	*YES*
136	E008	`select description from tproduct where description like 'diamond';`	*YES*
137	E008	`select distinct motor from general g where g.year between 1998 and 2004 and g.motor like '4 cil 1.8 lts';`	*YES*
138	E008	`select * from administrators where username like ' thierry';`	*YES*

139	E008	`select * from all_updatable_columns where column_name like 'reqd col name';`	*YES*
140	E008	`select 'inactive' "status",nvl(region,'total') region, count(region) regcount from temppivottest where status like 'inactive' group by rollup (region);`	*YES*

Table A.7: Manual analysis of tool performance for rule E008

No.	RuleID	Query	Correct
141	E009	`select paramone, paramtwo from tablename where search_param = 'x' union all select null, null from dual where not exists ( select paramone, paramtwo from tablename where search_param = 'x' );`	*YES*
142	E009	`insert into a (vala1, vala2, vala3, vala4) select valb1 , valb2, valb3, valb4 from b where not exists(select vala1, vala2, vala3, vala4 from a where a.vala1 = b. valb1 and a.vala2 = b.valb2 and a.vala3 = b.valb3 and a. vala4 = b.valb4);`	*YES*
143	E009	`select match_ref, count(match_ref) from raw_mwbe where exists( select match_ref, wbe from raw_mwbe where wbe like "p" or wbe like "n") group by match_ref, wbe having count(match_ref)>1;`	*YES*
144	E009	`select id, type, number from roads where number = '12' union select id, type, number from roads where number like '12%' and not exists ( select id, type, number from roads where number = '12' );`	*YES*
145	E009	`select l.email as email, l.date as date, l.ip as user_ip , l.logid as id from table as t where exists( select email,max(date)date from table group by email) and (ip is not null) group by t.email;`	*YES*
146	E009	`select * from address a where not exists ( select country,state union select 'us' as country, 'la' as state union select 'ind' as country, 'del' as state where e.country != a.country and e.state != a.state );`	*YES*
147	E009	`select * from cats outside where not exists(select * from cats cat where exists( select dog.foo,dog.bar from dogs dog where cat.foo = dog.foo and cat.bar = dog.bar) and outside.foo = cat.foo and outside.bar=cat.bar );`	*YES*
148	E009	`select a.id,a.name,split.item,a.flag from source_excel a where not exists ( select a.id,split.item from source_excel a join source_dw b );`	*YES*
149	E009	`delete from mycard t1 where t1.idmoney = 5 and t1.idcard = 80 and exists (select idcard, year, money from mycard t2 where t2.idcard = t1.idcard and t2.year = t1. year and t2.money = t1.money group by t2.idcard, t2.year , t2.money having count(t2.idcard) > 1) and t1.id not in (select min(id) from my...`	*YES*

150	E009	`select users.userid, year(membership.memyear) as memyear , users.mailto, users.streetaddress, users.address2, users.city, statelookup.state, users.zip from users inner join membership inner join statelookup where year (membership.memyear) = '2013' and not exists (select x. userid, year(membership.memye...`	*YES*
151	E009	`select sgtins_tmp_table.epc, sgtins.store from sgtins_tmp_table, sgtins where exists (select organization_id, sgtin from sgtins where client_id = 4 and sgtins.sgtin = sgtins_tmp_table.sgtin) and sgtins. client_id = 4 and sgtins_tmp_table.sgtin = sgtins.sgtin;`	*YES*
152	E009	`select r12.id, r23.id, r13.id from relations r12 join relations r23 join relations r13 where not exists ( select relation1_id, relation2_id, relation3_id from triangles where relation1_id = r12.id and relation2_id = r23.id and relation3_id = r13.id );`	*YES*
153	E009	`select item, size from t1 where t1.date = '2013-02-11' and not exists ( select item, size from t1 where t1.date = '2013-02-13' );`	*YES*
154	E009	`update pm set pm.amt = newvalue where exists ( select t. acct, t.amt from pm t where pm.acct = t.acct group by t. acct, t.amt having count(t.acct)>1 );`	*YES*
155	E009	`insert into tablename (col1,col2) select @par1, @par2 where not exists (select col1,col2 from tablename where col1=@par1 and col2=@par2);`	*YES*
156	E009	`select id,name from tbl_bkp t1 where not exists (select id,name from tbl_namecode where id=t1.id and name=t1. name );`	*YES*
157	E009	`select i.agent , i.agency , i.customer , i.company from table_1 as i where not exists ( select p.agent , p. agency , p.customer , p.company from table_2 as p where i.agent = p.agent and i.agency = p.agency and i.customer = p.customer and i.company = p.company );`	*YES*
158	E009	`select t1.id, avg(t1.score) avg1 from foo t1 group by t1 .id having not exists ( select t2.id, avg(t2.score) avg2 from foo t2 group by t2.id having avg(t2.score) > avg( t1.score));`	*YES*
159	E009	`select valb1, valb2, valb3, valb4 from b where not exists(select vala1, vala2, vala3, vala4 from a where a .vala1 = b.valb1 and a.vala2 = b.valb2 and a.vala3 = b. valb3 and a.vala4 = b.valb4);`	*YES*
160	E009	`select a.number, a.c1, a.c2, a.c3 from table a where exists (select count(1), b.number from table b where b. number = a.number group by b.number having count(1) = 1) ;`	*YES*

Table A.8: Manual analysis of tool performance for rule E009

No.	**RuleID**	**Query**	**Correct**
161	E010	`select * from user where name like '%mike%' or color = ' blue';`	*YES*

162	E010	`select sc.id, sc.number, sc.name from tempdb..syscolumns sc inner join tempdb..sysobjects so on sc.id = so.id where so.name like '%mytable%';`	*YES*
163	E010	`update person set addr = left(addr, len(addr) - 2) + ' street' where addr like '% st';`	*YES*
164	E010	`select * from emp where name like '%a%e%';`	*YES*
165	E010	`select columnname from tablename where columnname not like '%stack%';`	*YES*
166	E010	`select * from street where street_name like '%park%ave %10%';`	*YES*
167	E010	``select * from `words` where `word` like '%person%';``	*YES*
168	E010	`select * from fiberbox where field like '1740 %' or field like '%1938 ' or field like '%1940 % test';`	*YES*
169	E010	`select * from parameters where name like '%n%';`	*YES*
170	E010	`select * from tablename where column like '%company%';`	*YES*
171	E010	`select name from table where age like '%5%';`	*YES*
172	E010	`select a.s_oid, a.s_id, a.area_acre, a.power_peak, a. nearby_city, a.solar_total from global_site a cross join na_utility_line b where (a.power_peak between 1.0 and 100.0) and a.area_acre >= 500 and a.solar_avg >= 5.0 and a.pc_num <= 1000 and (a.fips_level1 = '06' and a. fips_country = 'us' and a.f...`	*YES*
173	E010	`select * from table_name where name like '%word1%' and name like '%word2%';`	*YES*
174	E010	`select m.* from dbo.mytable m where acolumn like '%val' order by (case when acolumn like '[a-z]%' then 0 else 1 end), acolumn;`	*YES*
175	E010	`select distinct city from station where city like '%a' or city like '%e' or city like '%i' or city like '%o' or city like '%u';`	*YES*
176	E010	`select * from yourtable where field like '%a%' or field like '%b%' or field like '%c%';`	*YES*
177	E010	`select * from table1 where table1.id like '%1234';`	*YES*
178	E010	`update table set col = substring(col,1,charindex('like', col,0)-1) where column like '%like%';`	*YES*
179	E010	`select movie.id, movie.title, alsokownas.aka from movie left join alsoknowas on movie.id = alsoknowas.movieid where title like '%searchterm%';`	*YES*
180	E010	`select a.noteid, a.firstname, a.lastname, a.status, a. category, a.name, a.followupdate from (select n.noteid , t.firstname, t.lastname, s.name as status, tc.name as category, d.name, n.followupdate as followupdate, row_number() over (partition by t.firstname, t.lastname order by n.noteid desc) rnk f...`	*YES*

Table A.9: Manual analysis of tool performance for rule E010

No.	**RuleID**	**Query**	**Correct**

181	E011	`select `utm source`, `users` , count(*) from ( select distinct `utm source`, `company id`, sum(distinct ` active users`) as users from customers group by `utm source`, `company id`) customers_2 group by `utm source `, `users`;`	*YES*
182	E011	`select sum(distinct money) as sum_money, m.created_at from receipt r join material m group by m.created_at;`	*YES*
183	E011	`select itemid from stock group by itemid having sum( distinct warehouseid) = (select sum(warehouseid) from warehouse);`	*YES*
184	E011	`select department, sum(distinct price) from products join ratings on product_id=products.id where rating=5 group by department;`	*YES*
185	E011	`select sum( distinct case when a.gender like 'm%' then 1 else null end) as males;`	*YES*
186	E011	`select sum(distinct value) as total from table;`	*YES*
187	E011	`select col1, sum(distinct col2) as s from tbl1 where col1='abbc' group by col1 order by s asc;`	*YES*
188	E011	`select col1, sum(distinct col2) as s from tbl1 where col1='abbc' group by col1 order by s asc;`	*YES*
189	E011	`select product_type, segment_type, sum(distinct promotion_value)promotion_value from sample_data group by product_type, cube(segment_type) order by product_type;`	*YES*
190	E011	`select p.color, array_agg(distinct pg.group_id) as groups, sum(distinct installs) as installs from performance p join performance_groups pg group by color;`	*YES*
191	E011	`select avg(distinct event_count), min(event_count), max (event_count) from numstest a join (select patient_id , count(*) as event_count from numstest group by patient_id) b;`	*YES*
192	E011	`select sum(distinct money_spent.value), sum(distinct money_earned.value) from user join money_spent on money_spent.userid = user.userid join money_earned on money_earned.userid = user.userid;`	*YES*
193	E011	`select sum(distinct cast(ar_all_bills.a_unpaid_balance as decimal(5,2))) as "total unpaid balance" from ar_all_bills where a_ar_customer_cid = 100059 group by a_ar_customer_cid;`	*YES*
194	E011	`select blocks.user_id, sum(distinct payout_history. amount) as amount from blocks left join payout_history where confirms > 520 group by blocks.user_id;`	*YES*
195	E011	`select `utm source`, `users` from ( select distinct `utm source`, `company id`, sum(distinct `active users`) as users from customers group by `utm source`, `company id `) customers_2;`	*YES*
196	E011	`select work.workid, work.description, machine. machinedescription, name.name, work2.regmin, work. minutes, (select sum(distinct minutes) from work w where w.machineid = machine.machineid) total_minutes from work work join machine machine left join work2 work2 left join name name;`	*YES*

197	E011	`select `utm source`, `users` from ( select distinct `utm source`, `company id`, sum(distinct `active users`) as users from customers group by `utm source`, `company id `) as customers_2 group by `utm source`;`	*YES*
198	E011	`select a.user_id , count(distinct b.client_id) count_client_id, sum( distinct c.client_price) sum_client_price, max(b.act_date) max_act_date from tbl_user a inner join tbl_sactivity b inner join tbl_client c group by a.user_id;`	*YES*
199	E011	`select `utm source`, sum(`users`) from ( select distinct `utm source`, `company id`, sum(distinct `active users `) as users from customers group by `utm source`, ` company id`) customers_2 group by `utm source`;`	*YES*
200	E011	`select id_a, id_b, nu_b, date_trunc('month', dt_date) :: date as dt_month, sum( distinct (1 << (date_part('day', dt_date)::int) -1) ) as nu_bit_days from test group by id_a, id_b, nu_b, date_trunc('month', dt_date) :: date ;`	*YES*

Table A.10: Manual analysis of tool performance for rule E011

No.	RuleID	Query	Correct
201	E012	`select analytics.source as referrer, count(analytics.id) as frequency, sum(if(transactions.status = 'completed', 1, 0)) as sales from analytics left join transactions on analytics.id = transactions.analytics where analytics .user_id = 52094 group by analytics.source order by frequency desc limit 10;`	*YES*
202	E012	`select count(order_header_id) from order_lines where sum (accounting_total) <= 500;`	*YES*
203	E012	`select a.source as referrer, count(a.id) as frequency , sum(t.sales) as sales from (select id, source from analytics where user_id = 52094) a left join (select analytics, case when status = 'completed' then 1 else 0 end as sales from transactions) t on a.id = t.analytics group by a.source order by fr...`	*YES*
204	E012	`select source, count(id) as frequency from analytics where user_id = 52094 group by source order by frequency desc limit 10;`	*YES*
205	E012	`select count(order_id) from orders;`	*YES*
206	E012	`select e.id, e.name, count(d.social_security) as number_of_departments from employee e inner join department d where d.social_security=e.social_security group by social_security;`	*YES*
207	E012	`select count(id) from profile where registration_date between now() - interval 7 day and now();`	*YES*
208	E012	`select * from table_name where primarykey in ( select primarykey from table_name group by primarykey having count(primarykey) > 1 ) order by primarykey;`	*YES*
209	E012	`select customerid, count(customerid) from maintenance where actiontype = 2 group by customerid having count( customerid) >= 1;`	*YES*

210	E012	select t1.id, t1.name, t1.country, count(bid) as bookings, sum(case when t2.vehicle = 'plane' then 1 else 0 end) as plane, sum(case when t2.vehicle = 'train' then 1 else 0 end) as train, sum(case when t2.vehicle = 'bus' then 1 else 0 end) as bus from table1 t1 inner join table2 t2 on t1.id = t2.orig...	*YES*
211	E012	select id from table group by id having count(id) > 2;	*YES*
212	E012	select id, names, count(names), total from tbl_products , (select count(distinct names) as total from tbl_products) as total where type = '1' group by names;	*YES*
213	E012	select name, count(email) from users group by email having count(email) > 1;	*YES*
214	E012	select a.post_name,a.post_title,a.id,b.meta_value as _sku from wp_posts a inner join (select meta_value,max(post_id) as post_id from wp_postmeta where meta_key='_sku' group by meta_value having count(meta_value) > 1) b on a.id=b.post_id where post_type = 'product' and post_status = 'publish';	*YES*
215	E012	select id, count(id) from table1 group by id having count(id)>1;	*YES*
216	E012	select customer, count(liked) as total_likes from table where liked = 'true' group by customer;	*YES*
217	E012	select m.start_time, count(r.seconds) from dbo.minutes m group by m.start_time;	*YES*
218	E012	select a.id from result a cross join result b where b.count = (select max(count) from result);	*YES*
219	E012	select orders.entrydate as "date", count(orders.orderno) as "orders", sum(case when orders.reason is null then 1 else 0 end) as "replacements" from orders where orders.reason is null and orders.entrydate = '09-may-2014' and orders.customerno = 'a001' group by orders.entrydate;	*YES*
220	E012	select count(f.id_foo) from my_remote_server_public.my_remote_server_public_foo f where f.date < _my_date;	*YES*

Table A.11: Manual analysis of tool performance for rule E012

No.	RuleID	Query	Correct
221	E013	select count(*) as countbyid from items where fkid = 2003799 group by fkid having countbyid>1 order by countbyid;	*YES*
222	E013	select v.concode,c.conname, min(v.rate), v.vendor from venprices v, country c where c.conid = v.concode group by c.conid;	*YES*
223	E013	select u from user u where size(u.comments) = (select max(count(c.id)) from user u2 inner join u2.comments c group by u2.id);	*YES*
224	E013	select t.person_id from table t group by t.personid having count(t.personid) > 3;	*YES*

225	E013	`select import_values.id, import_values.part_id, import_values.qty, import_values.note, parts. partterminologyname, group_concat(basevehicle.yearid, ' ', make.makename, ' ', model.modelname, ' ', submodel .submodelname separator ', '), group_concat(distinct(enginedesignation.enginedesignationname) sepa...`	*YES*
226	E013	`select count(*) from tbl t group by t.ip_address order by count(*) desc limit 10;`	*YES*
227	E013	`select name, max(b), max(c), min(b), min(c) from tablename group by name, b, c;`	*YES*
228	E013	`select customer_name, count(purchases.customer_id) as number_of_purchaes from customer left join purchases on customer.customer_id = purchases.customer_id group by customer.customer_id;`	*YES*
229	E013	`select c.name, count( * ) as mycount from coupon c left join coupon_users u on c.id = u.coupon_id group by c.id order by mycount desc;`	*YES*
230	E013	`select game.mdate, game.team1, sum(case when goal.teamid =game.team1 then 1 else 0 end) score1, game.team2, sum (case when goal.teamid=game.team2 then 1 else 0 end) score2 from game inner join goal group by game.mdate, goal.matchid, game.team1, game.team2;`	*YES*
231	E013	`select itemprices.id, min(itemprices.lowprice) as minprice, itemprices.locationid from itemprices left join t where t.id is null group by locationid;`	*YES*
232	E013	`select employees.empid, sum(workhours.hoursworked) as totalhours from employees inner join workhours where wh_month = 4 group by lastname, firstname, employees. empid;`	*YES*
233	E013	`select sum(points) as total_points from sometable where total_points > 25 group by username;`	*YES*
234	E013	`select count(1) from file_item p join type t on t.id = p.family_type_id group by p.family_type_id order by t. name;`	*YES*
235	E013	`select group_name, group_id, sum(case when pass_fail = 'pass' then 1 else 0 end) as pass, sum(case when pass_fail = 'fail' then 1 else 0 end) as fail from log a join group b group by b.group_name, a.group_id;`	*YES*
236	E013	`select knowledge.*, sorting.* from knowledge, sorting where knowledge.id = sorting.kid group by kid having count(id) < 2;`	*YES*
237	E013	`select t.historyid from synk_isheet_1_int t where t. historyid = 6 and t.group1id = 27 and t.group2id = 4 group by t.historyid,t.requestentityid;`	*YES*
238	E013	`select classes.classname, students.userid from classassociation join students join classes where classassociation.classid = 1 group by classname;`	*YES*
239	E013	`select avg(m.a) from maintable m inner join #temptable t on m between t.startdate and t.enddate group by t. startdate;`	*YES*

240	E013	`select p.p_pid, p.p_name, p.p_url from products p inner join activity a on p.p_pid = a.a_pid where a.a_uid= ". $uid_int." group by p_pid, p_name, p_url order by max(a. a_time) desc limit 6;`	*YES*

Table A.12: Manual analysis of tool performance for rule E013

No.	**RuleID**	**Query**	**Correct**
241	E014	`select customerkey from factinvoices i where i. dossierkey =2 and i.reportingdate between '2016-01-01' and '2017-12-31' group by customerkey;`	*YES*
242	E014	`select ins1 as insurance from insauth2 where ins1 is not null group by ins1 union select ins2 as insurance from insauth2 where ins2 is not null group by ins2 union select ins3 as insurance from insauth2 where ins3 is not null union select ins4 as insurance from insauth2 where ins4 is not null union ...`	*YES*
243	E014	`select cast(somecol as decimal(10,3)) from sometable group by cast(somecol as decimal(10,3));`	*YES*
244	E014	`select distinct productid from x_product_ship group by productid having shipid <> 3;`	*YES*
245	E014	`select country from yourtable group by country;`	*YES*
246	E014	`select reservations.idcustomer from reservations ( nolock) where excludedreservations.idcustomer is null and reservations.idcustomer is not null group by reservations.idcustomer;`	*YES*
247	E014	`select zip from table1 where created between '2014-08-04 00:00:00' and '2014-08-08 23:59:59' group by zip order by count desc;`	*YES*
248	E014	`select trackid from tracks where timestamp between tmin and tmax group by trackid;`	*YES*
249	E014	`select t.purchase_date, t.qty, approx_percentile(t1.qty, 0.9) tp90 from mytable t inner join mytable t1 group by t.purchase_date, t.qty;`	*YES*
250	E014	`select parent_id, listagg(child_id, ',') within group ( order by child_id) as "children" from parentchildtable where parent_id = 0 group by parent_id;`	*YES*
251	E014	`select product_name from purchase_history group by product_name;`	*YES*
252	E014	`select animal from pets group by animal having animal not in ( select animal from pets where name not in (' homer','bart','marge','lisa','maggie') group by animal ) ;`	*YES*
253	E014	`select pers_key, pers_name from visit_info a join valid_dates b where a.visit_date between b.start_date and b.end_date group by pers_key, pers_name;`	*YES*
254	E014	`select id, group_concat(name order by name asc separator ', ') from my_table group by id;`	*YES*
255	E014	`select country from countries group by country having sum(building) is null;`	*YES*

256	E014	`select t1.groupguid, t2.memberguid from temp t1, temp t2 where t2.isgroup = 0 group by t1.groupguid, t2.memberguid;`	*YES*
257	E014	`select p.name from pc p left join sc s on p.color=s.color where s.color is not null group by p.name;`	*YES*
258	E014	`select maker_id, status_id from cars where status_id != 0 group by maker_id, status_id union all select maker_id, max(status_id) max_status_id from cars group by maker_id having max_status_id = 0;`	*YES*
259	E014	`select tb.id from tablea ta inner join tableb tb group by tb.id;`	*YES*
260	E014	`select student_id, name, group_concat(subject separator ' ') from table1 join table2 on table1.student_id = table2.student_no group by student_id, name;`	*YES*

Table A.13: Manual analysis of tool performance for rule E014

No.	RuleID	Query	Correct
261	E015	`select count(*) from t1 where value='0' union select count(*) from t1 where value='1';`	*YES*
262	E015	`select * from mytable where a=x union all select * from mytable where b=y and a!=x;`	*YES*
263	E015	`select cc.contactpersonid, cc.clientcontactid, ad.city, ad.addressid from savedlist sl inner join clientcontacts cc on cc.contactpersonid = sl.objectid inner join clients c on c.clientid = cc.clientid inner join address ad on c.clientid = ad.objectid where sl.savedlistid = 2117 and (ad.city is not n...`	*YES*
264	E015	`select id,name,age from student where age < 15 union select id,name,age from student where name like '%a%' order by name;`	*YES*
265	E015	`select * from citizen c where c.name = 'smith' union select * from citizen c where p.area = 'moon';`	*YES*
266	E015	`select * from a_table where a_column = :a_var union select * from a_table where b_column = :a_var;`	*YES*
267	E015	`select id, name from color where parentid=4 union select id, name from color where parentid=(select id from color where parentid=4);`	*YES*
268	E015	`select * from table where id1 = :var1 and id2 = :var2 union all select * from table where id1 = :var2 and id2 = :var1;`	*YES*
269	E015	`select resourceid from mytable where startdate between '2009-01-01' and '2009-01-20' and datediff(day, case when enddate < '2009-01-20' then enddate else '2009-01-20' end, startdate) >= 5 union select resourceid from mytable where enddate between '2009-01-01' and '2009-01-20' and datediff(day, endda...`	*YES*

270	E015	`select fiscal_period, fiscal_year, amount from maxtable where fiscal_period = 5 union all select fiscal_period, fiscal_year, amount from maxtable where fiscal_period = (select max(fiscal_period) from maxtable) and not exists (select * from maxtable where fiscal_period = 5);`	*YES*
271	E015	`select * from xx where f_colour = "green" union all select * from xx where id not in (select distinct id from xx where f_colour = "green");`	*YES*
272	E015	`select customers.firstname, customers.surname, customers .dob, customers.customeraddress from customers where customers.customeraddress like '%'+ 'main' + '%' union select customers.firstname, customers.surname, customers .dob, customers.customeraddress from customers where customers.customeraddress...`	*YES*
273	E015	`select * from stock where stockid in (33) union select * from stock where stockid in (12) union select * from stock where stockid in (53) union select * from stock where stockid in (4) union select * from stock where stockid in (99) union select * from stock where stockid in (88);`	*YES*
274	E015	`select id from foo where a = 1 and b = 2 union all select id from foo where a = 3 and b = 4 union all select id from foo where a = 5 and b = 6;`	*YES*
275	E015	`select a.id, b.model, c.color from cars a join models b join colors c join brands d where b.id=1 union all select a.id, b.model, c.color from cars a join models b join colors c join brands d where b.id=3;`	*YES*
276	E015	`select id, name, group_id from mytable where id in ( select min(id) from mytable group by group_id) union all select id, name, group_id from mytable where id in ( select min(id) from mytable where id not in ( select min (id) from mytable group by group_id ) group by group_id) order by group_id;`	*YES*
277	E015	`select meetingid, billid from mytable where billid is not null group by billid union all select meetingid, billid from mytable where billid is null;`	*YES*
278	E015	`select id, event, seen, time_stamp from notifications n where id_user = :id and seen is null union all (select id, event, seen, time_stamp from notifications n where id_user = :id and seen is not null limit 15);`	*YES*
279	E015	`select name from foo where name like 'm%' order by name desc union all select name from foo where name not like 'm%' order by name asc;`	*YES*
280	E015	`select friend_user_id from friends where user_id = 1 union select friend_user_id from friends where user_id in (select friend_user_id from friends where user_id = 1);`	*YES*

Table A.14: Manual analysis of tool performance for rule E015

No.	**RuleID**	**Query**	**Correct**

281	E016	`select parent.object_id,parent.event_time,parent.state, min(child.event_time) as ch_event_time, case when parent .state<>'done' and min(child.event_time) is null then (select localtimestamp)-parent.event_time else min(child .event_time)-parent.event_time end as step_time from objectstate parent where p...`	*YES*
282	E016	`select title, pubid as 'publisher id', pubdate as ' publish date' from books where pubid = 4 or pubdate > '01-jan-01' order by pubid asc;`	*YES*
283	E016	`select * from tv_watchers where mins_watching_tv <= 60 or id=10 order by mins_watching_tv desc, id asc limit 6;`	*YES*
284	E016	`select t.name as tablename, p.rows as rowcounts, convert (decimal,sum(a.total_pages)) * 8 / 1024 / 1024 as totalspacegb, sum(a.used_pages) * 8 / 1024 / 1024 as usedspacegb , (sum(a.total_pages) - sum(a.used_pages)) * 8 / 1024 / 1024 as unusedspacegb from sys.tables t inner join sys.indexes i on t.obj...`	*YES*
285	E016	`select a.column1, a.column2, b.column1, c.column1, cc. column1, cc.column2, cc.column3, d.column1 from table_a a inner join table_b b inner join table_c c left join table_d d inner join table_c cc where a.column1 = 1000 and b.column3 = 1 and c.column3 = 0 order by a.column1 asc;`	*YES*
286	E016	`select q.postid, a.postid, c.commentid from posts q where q.postid = 1234 order by q.postid, a.postid, c. commentid;`	*YES*
287	E016	`select salary from table_name where name='inputfromphp' and (surname='inputfromphp' or country='inputfromphp') order by case surname when 'inputfromphp' then 0 else 1 end, case country when 'inputfromphp' then 0 else 1 end limit 1;`	*YES*
288	E016	`select name, score_string, place from scores s where game_id =1 and game_size =15 and game_level =1 and id = (select id from scores si where si.game_id =1 and si. game_size =15 and si.game_level =1 and si.name = s.name and si.place = s.place and si.date > "2010-10-01" order by si.game_id, si.game_si...`	*YES*
289	E016	`select id,title,release_date from tbl_movies where release_date > '2014-02-20' or release_date='' order by release_date asc;`	*YES*
290	E016	`select m.discovery_time, m.ip, m.matched_id, m.uuid from mydata m where m.ip = '12.34.56.78' order by m.ip, m. discovery_time;`	*YES*
291	E016	`select max(speed) as speed, time from info where id = 1 and time > 1234 order by id;`	*YES*
292	E016	`select p1.store_number as st# from personnelfile as p1 left join payrollfile as p2 where p2.pay_date > '2010-05-14' and p2.uniform_allowance_amt_cppd in (8.25,8.50,300) and p1.jobs_series in ('2380','1458') and p1.ssn = '123456789' order by p1.ssn,p2.pay_date;`	*YES*
293	E016	`select column1, column2, column3, coalesce(column4, 'foo ') as column4 from tablename where column1 = 'bar' order by column1, column2;`	*YES*

294	E016	select v.id, v.active, v.reg_no, p.install_date, p. remove_date from vehicle v left join period p on (v.id = p.car_id) where v.id = 1 order by v.id, p.install_date asc;	*YES*
295	E016	select code, name from table1 where scope1='here' or scope2='room' group by code, name order by min(scope1), min(scope2), min(seq);	*YES*
296	E016	select br.bm_tracking_number, (select tolist(appt.fact_date) from bm_fact appt where appt. bm_review_sk = br.bm_review_sk and appt.fact_type_code =183050) "appointments" from bm_review br where row_delete_date_time is null order by min(select appt .fact_date from bm_fact appt where appt.bm_review_sk = ...	*YES*
297	E016	select \`rp_products\`.\`product_code\`, \`rp_log\`.\` customer_id\`, \`rp_products\`.\`product_name\` from rp_products, rp_log where (\`rp_log\`.\`customer_id\` = '111') order by \`rp_products\`.\`product_code\` asc, \` rp_log\`.\`customer_id\` asc;	*YES*
298	E016	select * from label where tenantid = 'jim' and tagfieldname = 'work' and tagid = 'work1' order by value, tenantid, tagfieldname, tagid, key limit 2;	*YES*
299	E016	select pd.id,pd.price_date,pd.name_id,pd.class_id,pd. currency_id,pd.price, pd.price - (select price from price_data as x where x.price_date < pd.price_date and x .name_id = pd.name_id and x.class_id = pd.class_id and x .currency_id = pd.currency_id having max(x.price_date)) as \`change\` from price_data...	*YES*
300	E016	select mq, im, pf, kmct from vehicules where mq='renault ' order by mq;	*YES*

Table A.15: Manual analysis of tool performance for rule E016

No.	**RuleID**	**Query**	**Correct**
301	E017	select count(*) as countbyid from items where fkid = 2003799 group by fkid having countbyid>1 order by countbyid;	*YES*
302	E017	select t.project, t.employee_id, sum(timestampdiff(hour , t.start_time, t.end_time)) as number_of_hours, max(e. billable_rate) as unit_price, sum(timestampdiff(hour, t .start_time, t.end_time)) from my_db.timesheet t join my_db.employee e on e.id = t.employee_id where t.project = 'ait' group by t.proje...	*YES*
303	E017	select \`periode_class_members\`.\`id\`, \`classes\`.\`id \` as \`class\`, \`periode_class_members\`.\`periode\`, \` user\`.\`firstname\` as \`firstname\`, \`user\`.\`lastname \` as \`lastname\`, \`periode_class_members\`.\`status\`, min(\`periode_class_subjects\`.\`id\`) as pcs from \` periode_class_subject_members\` left join \`periode_c...	*YES*
304	E017	select class, min(grade) as highestgrade, freq, ranking from ranked where ranking = 1 group by class, freq, ranking;	*YES*

305	E017	select car.personid as person, count(car.carid) as cars , null as pets from car where car.personid = 1 group by car.personid union all select pet.personid as person, null as cars, count(pet.petid) as pets from pet where pet.personid = 1 group by pet.personid;	*YES*
306	E017	select product.productname, component.componentname from product_component join component join product where product.productname = 'bread' group by product. productname;	*YES*
307	E017	select name, year, sum(hours) from test_hours join test_users where year = 2020 group by users_id, year;	*YES*
308	E017	select speed , lat as lat1 , lon as lon1 from table1 where speed <> 0 union all select speed , lat as lat1 , lon as lon1 from table1 where speed = 0 group by speed, lat1,lon1 order by lat1;	*YES*
309	E017	select id, code, sum(sale) as sale from tablename where code = 11 group by id, code;	*YES*
310	E017	select d.full_date, count(*) from actions a join date_dimension d where d.full_date = '2010/01/01' group by d.full_date;	*YES*
311	E017	select distinct from_member_id, to_member_id from ` single_chat` where from_member_id = 175 or to_member_id = 175 group by from_member_id, to_member_id;	*YES*
312	E017	select customer_id, tax_code from orders where customer_id = 'some customer id' group by customer_id , tax_code;	*YES*
313	E017	select service_name, metric_name, array_agg(value_textarray) from service_data where service_name = 'activitydataservice' group by service_name, metric_name ;	*YES*
314	E017	select s.transaction_id, group_concat(i.item_name) from stock s inner join ref_item i where s.transaction_id = 123 group by s.transaction_id;	*YES*
315	E017	select label1, label2, sum(number) as mysum from mytable where label1 = 'foo' group by label1, label2 having mysum > 0;	*YES*
316	E017	select product.productname, component.componentname from product_component join component join product where product.productname = 'bread' group by product. productname;	*YES*
317	E017	select p.proj_uid, p.proj_name,p.agency,p.district ,p.division,p.projstatus,civilbill80.billcount as civilbill80, civilbill20.billcount as civilbill20 , civilbillpay.billcount as finalcivilbill,civilworkslip. billcount as civilworkslip, electribill80.billcount as electricbill80, electribill20.billcount...	*YES*
318	E017	select sum(quantity) as total from tablename where productid =1 and shopid in (1,2) group by productid;	*YES*

319	E017	`select a.userid, a.code, a.country, listagg(b.email, ',') within group (order by b.email) as "emails" from tab1.a, tab2.b where a.userid = b.userid and a.code = b.code and a.userid = 'rishi' group by a.userid, a.code, a.country;`	*YES*
320	E017	`select division, count(id) as ct from test where role >=101 and division=1 group by division;`	*YES*

Table A.16: Manual analysis of tool performance for rule E017

No.	RuleID	Query	Correct
321	E018	`select sc.studentid, c.classname, u.usergrouporganizationname, c.academicyearid , c.usergroupid, c.schoolid, sc.classid , u.usergrouporganizationstatusid from studentclasscrossreference sc inner join class c on sc.classid = c.classid inner join school s on s.schoolid = c.schoolid inner join dbo.usergr...`	*YES*
322	E018	`select table1_id, count(*) as count from table1_table2 group by table1_id having count > 2;`	*YES*
323	E018	`select active, is_featured, count(*) from tbl_sales group by active, is_featured having (active=0 or active =1) and (is_featured=0 or is_featured=1);`	*YES*
324	E018	`select sum(child_id) from children group by child_id having child_id = 5;`	*YES*
325	E018	``select `businessid`, count(*) c from `biz_listing` where updated_date between '2014/06/01' and last_day ('2014/07/01') group by `businessid`,`type` having c = 2;``	*YES*
326	E018	`select a, b, sum(d) as c from tbla inner join (tblb inner join tblc on tblb.a = tblc.a) group by a, b having f like '*808*';`	*YES*
327	E018	`select d.doctorid, d.doctorname, count(p.patientid) as patients from doctor d inner join patient p group by d.doctorid, d.doctorname having patients > 1;`	*YES*
328	E018	`select company, project from table group by company, project having type = 'dummy';`	*YES*
329	E018	`select val, array_agg(fkey) fkeys from mytable group by val having array_length(array_agg(fkey),1) > 1;`	*YES*
330	E018	`select first.subscriber_id, second.tag_id, count(*) as c from content_hits first join content_tag second on first.content_id=second.content_id group by second.tag_id,first.subscriber_id having c = 0;`	*YES*
331	E018	`select lr.ansattnr, lr.rom, rb.behandling from rom_behandling rb inner join lege_rom lr on lr.rom = rb.rom group by lr.rom having rb.behandling = 'konsultasjon';`	*YES*
332	E018	`select id parent_id, title, ( select count(id) from tbltree group by id having parent = parent_id) child_count from tbltree;`	*YES*

333	E018	select id, kmstand, count(*) as cnt from (select id , kmstand, vacationame, vacationvalue from `db_1`.`table_new` where (vacationame='vacation1' or vacationame ='vacation2' or vacationame='vacation3' or vacationame ='vacation4') group by id, kmstand, vacationame, vacationvalue having count(*) = 1) t gr...	*YES*
334	E018	select label1, label2, sum(number) as mysum from mytable group by label1, label2 having mysum > 0 or label1 = ' foo';	*YES*
335	E018	select c.country from orders o left join customers c on o.customer_id = c.customer_id join order_detail od where o.order_id = od.order_id group by country, month having rank <= 3;	*YES*
336	E018	select first_name, last_name, class from students group by class having class = 'jss1';	*YES*
337	E018	select first_name from students group by class having class in('a');	*YES*
338	E018	select s.buyer_id, p.product_name from sales s join product p group by s.buyer_id having p.product_name = " s8";	*YES*
339	E018	select name from schools group by city having city = city;	*YES*
340	E018	select string_agg(title, ', ') as titles, genre, count (*) from films group by genre, title having genre='SciFi ';	*YES*

Table A.17: Manual analysis of tool performance for rule E018

No.	**RuleID**	**Query**	**Correct**
341	E019	select a.brand from brands a join cars b group by a. brand;	*YES*
342	E019	select a.empid, a.name, b.name as dept_name from emp a left join department b;	*YES*
343	E019	select o.orderid from orders as o join orderitems as oi join products as p;	*YES*
344	E019	select e.id, e.name, count(d.social_security) as number_of_departments from employee e inner join department d group by e.id, e.name;	*YES*
345	E019	select a.x, a.y from table_a a left join table_b b where b.x is null;	*YES*
346	E019	select e.id, e.name, count(d.social_security) as number_of_departments from employee e inner join department d group by e.id, e.name;	*YES*
347	E019	select table1.id, table1.start_date as table1_start_date , table1.end_date as table1_end_date, table2.start_date as table2_start_date, table2.end_date as table2_end_date from table1 inner join table2 order by table1.id, table1.start_date, table2.start_date;	*YES*
348	E019	select food, description from tbl_foods join tbl_dishes;	*YES*

349	E019	`select ts.pagesize from syscat.tablespaces ts join syscat.tables tb where tb.tabschema = 'sysibm' and tb.tabname = 'dual';`	*YES*
350	E019	`select distnct station, slot, subslot, compid, compname from devicetrace as dt inner join complist as cl;`	*YES*
351	E019	`select * from category c join ( select distinct pt.category_id from part pt ) pqparts;`	*YES*
352	E019	`select * from a inner join b;`	*YES*
353	E019	`select p.personname from people p left join addresses a where a.addressid is null;`	*YES*
354	E019	`select d.end_date, extract(hour from end_time) as end_hour, count(t.users) as total_users from (select distinct cast(end_time as date) as end_date from table) d cross join table t group by e.end_date, h.end_hour;`	*YES*
355	E019	`select omode.vehicle, omode.ordercode, omode.actiondate -imode.actiondate from (select vehicle, ordercode, actiondate from table where mode='o' ) omode, (select vehicle, ordercode, actiondate from table where mode='i' ) imode where omode.vehicle = imode.vehicle and omode.ordercode = imode.ordercode;`	*YES*
356	E019	`select a.id, a.account, a.mydate from awesometable as a join ( select account, max(mydate) as maxdate from awesometable group by account having maxdate < date_sub( now(), interval 12 month)) as b;`	*YES*
357	E019	`select t1.id as t1_id,t3.data as t3_data from table3 t3 inner join table2 t2 inner join table1 t1;`	*YES*
358	E019	`select distinct saletype.id, saletype.code, saletype.name from customer left join saletype_customer where (saletypeid = saletype.id or saletypeid is null) and customer.id= 4;`	*YES*
359	E019	`select foo from footable foo join bartable bar where bar .anotherid=:another;`	*YES*
360	E019	`select * from meetings m join (select attendee_id, max(meeting_date) from meetings group by attendee_id) attendee_max_date;`	*YES*

Table A.18: Manual analysis of tool performance for rule E019

No.	RuleID	Query	Correct
361	E020	`select fname, lname from employee where not exists ( ( select pnumber from project where dnum = 5) );`	*YES*
362	E020	`select case when exists (select patientid from table2 t2 where t2.patientid =t1.patientid) then 'yes' else 'no' end as patientexists from table1 t1;`	*YES*
363	E020	`select * from tblrecords where not exists ( select personid from tblpeople group by personid having count( personid) > 1 );`	*YES*
364	E020	`select * from event where exists ( select 1 from dual where mod(start_date - to_date(1, 'j') + level - 1, 7) = 6 or (mod(start_date - to_date(1, 'j') + level - 1, 7) = 3 and to_char(start_date + level - 1, 'dd') = '13') );`	*YES*

365	E020	`select (select max(if(index=80, value, null)) from unnest(customdimensions)) as is_app, (select hits.eventinfo.eventaction) as ea from ` + "`table-big-query.105229861.ga_sessions_201711*`" + `, unnest(hits) hits where totals.visits = 1 and _table_suffix between '21' and '21' and exists(select 1 from unnest(hi...`	*YES*
366	E020	`select * from a a where not exists (select 1 from ab m where m.a_id = a.id and exists (select 1 from b b where m.b_id = b.id and b.type = 'c'));`	*YES*
367	E020	`insert into tableb select sum(a), sum(b), sum(c) from tablea where not exists (select * from table b where a=a and b=b) group by a, b;`	*YES*
368	E020	select * from \`riders\` where exists(select * from \`ridersclasses\` where ridersclasses.rid = riders.id and \`cid\` = '6') order by \`first_name\` asc;	*NO*
369	E020	`select * from test1 t1 where exists(select 1 from test2 t2 where (t1.id = t2.idc or t1.id = idp) and exists( select 1 from test3 where t2.idc = id or t2.idp = id));`	*YES*
370	E020	`select * from students where exists (select studentid from student_to_hostel where hostelid=2);`	*YES*
371	E020	`select e.email, case when exists(select * from userstbl tu where e.email = tu.username) then 'exist' else 'not exist' end as ex from (values('email1'),('email2'),('email3'),('email4')) e(email);`	*NO*
372	E020	`select 1 from dual where exists (select 1 from user_objects where object_name = upper('client_sys.clear_info') and object_type = 'procedure') or exists (select 1 from user_procedures where object_name = substr(upper('client_sys.clear_info'), 1, instr(upper('client_sys.clear_info'), '.') - 1) ...`	*YES*
373	E020	`select * from t1 where exists ( select null from t2 where y = x );`	*YES*
374	E020	`select staff_no from doctor where not exists (select * from patient where staff_no = consultant_no);`	*YES*
375	E020	`select * from final_combined_result wfcr where not exists (select contact_id, account_id from temp_wfcr);`	*YES*
376	E020	`select id from usertable where exists ( select 1 from ( values (user_addedon, user_deletedon, user_modified)) ud (ud) where ud >= ri and ud < re );`	*YES*
377	E020	`select * from current_stock where not exists ( select * from stock_record);`	*YES*
378	E020	`select count(*) as row_count from catat where catat.id = 1007642 and catat.is_parent = 1 and exists(select 1 from cg where cg.c_id = catat.id and exists(select 1 from ccsd where ccsd.g_id = cg.id));`	*YES*
379	E020	`select id from dbo.splitstringtotable('2,3,6,7') where not exists (select 1 from tab where col = id);`	*YES*
380	E020	`select c.custname, p.pjtitle from customer as c join project as p on p.custno = p.custno join tack as t on t.pjno = p.pjno where t.empid = 'glc' and not exists ( select null from task where empid = 'glc' and empid = 'cac');`	*YES*

Table A.19: Manual analysis of tool performance for rule E020

No.	RuleID	Query	Correct
381	E021	`select * from formfields where formfields.id in(select fields from form);`	*YES*
382	E021	`select ancestorid from myview where ancestorid in ( select id from #t);`	*YES*
383	E021	`select eid, employee_name from employee where eid not in ( select user1 from assign ) and eid not in ( select user2 from assign );`	*YES*
384	E021	`select eid from entidades e where distrito in ( select id from distritos where distrito_t like '%lisboa%' );`	*YES*
385	E021	`select ids from my_temp_table where ids not in ( select id from table_one );`	*YES*
386	E021	`select m.msg_id, m.uid_fk, m.message, m.alert, m.created , m.uploads, m.owner, u.uid, u.first_name, u.last_name from users u join messages m on m.uid_fk = u.uid where u.uid = :uid or u.uid in ( select f.friend_two from friends f where f.friend_one = :uid ) order by m.created desc limit 10;`	*YES*
387	E021	`select categoryname from categories where id_category not in ( select supercategoryid from supercategories );`	*YES*
388	E021	`select * from data d inner join user u on 1=1 where (u.id, d.id) not in (select user_id, data_id from collection);`	*YES*
389	E021	`select * from temp where part_in in (select count( part_id) as duplicates from temp where 1 group by part_id) and duplicates > 1;`	*YES*
390	E021	`select table1.* from table1 where table1.id not in ( select table2.key_to_table1 from table2 where table2.id = some_parm );`	*YES*
391	E021	`select * from table1 where info1 in (select info2 from table1) and info2 in (select info1 from table1) and id not in (select id from table1 where (info1 in (select info2 from table1) and info2 not in (select info1 from table1)) or (info2 in (select info1 from table1) and info1 not in (select info2 f...`	*YES*
392	E021	`select id from user where id not in ( select owner_id from hd_ticket union all select submitter_id from hd_ticket union all select owner_id from hd_archive_ticket union all select submitter_id from hd_archive_ticket );`	*YES*
393	E021	`select * from table1 a where a.col1 in (select b.col2 from table2 b) ;`	*YES*
394	E021	`select f.field1, f.field2, f.field3 from foo f where f. field4 in ( select b.bar from bar b where b.type = 4 and b.other = 7 );`	*YES*

395	E021	`select * from worklog w where 1=1 and w.class = ' activity' and w.recordkey in (select wonum from woactivity where parent = 'm2176') union all select * from worklog w where 1=1 and w.class = 'workorder' and w.recordkey in (select wonum from workorder where parent = 'm2176');`	*YES*
396	E021	`select no, action_dt, request_type, status_cd, min ( action_dt ) over ( partition by no, grp ) as request_start_dt from ( select w.*, count( case when status_cd in ( 'approved', 'denied' ) then 1 end ) as grp from w ) order by action_dt;`	*YES*
397	E021	`delete from r2_table where r02_r01_id_fk in (select column_value from table(r01_ids));`	*YES*
398	E021	`select id,name,xs1 as download, xs2 as upload from sc_params where rfen = 'service_requested' and code = 'speed' or code in ( select code from sc_params where rfen = 'service_requested' and id = ( select parent from sc_params where rfen = 'service_requested' and code = ' speed' ) );`	*YES*
399	E021	`select table1.articleno,table1.artdescription,table2. year from table1 join table2 where not table1.articleno in (select table2.year from table2);`	*YES*
400	E021	`select * from conversation where (least(sender_id, receiverid), greatest(sender_id, receiverid), date) in ( select least(sender_id, receiverid) x, greatest (sende_id, receiverid) y, max(date) max_date from conversation ) and '$uid' in (sender_id, receiverid);`	*YES*

Table A.20: Manual analysis of tool performance for rule E021

No.	RuleID	Query	Correct
401	E022	`select a.name from users a, friends b where a.id=b. user_b and b.user_a = (select b.user_a from friends where a.name='s1');`	*YES*
402	E022	`select * from tablea a where not exists( select * from tableb b where a.pid = b.pid and a.startdate >= '20-jun -10' );`	*YES*
403	E022	`select cola,colb,colc,cold from mytable where exists (select 1 from (select i.itemid from items as i where iitemname like '%xxx%') as itm where itm.itemid=mytable. cola or itm.itemid=mytable.colb);`	*YES*
404	E022	`select count(distinct i.third_party_id) as uniques from db.ids i where i.third_party_type = 'cookie_1' and i. first_party_id not in ( select i.first_party_id where i. third_party_id = 'cookie_2' );`	*YES*
405	E022	`select * from t1 where t1.id in (select t2.id from t2 where t1.a = 'aa');`	*YES*

406	E022	`select atc.owner, atc.table_name, atc.column_name from all_tab_columns atc where not exists (select acc.owner , acc.table_name, acc.column_name from all_cons_columns acc join all_constraints ac on acc.owner = ac.owner and ac.constraint_name = acc.constraint_name and ac. constraint_type in ('p', 'r') ...`	*YES*
407	E022	`select * from table1 a where a.d > ( select b.d from table2 b where a.id = b.id and a.something = 1 );`	*YES*
408	E022	`select * from (select e.id,e.name,sum(s.amount) as ' total_amount' from employee e inner join sale s on e.id= s.emp_id group by s.emp_id,e.id,e.name) as t1 where(0) =(select count(distinct(total_amount)) from(select e.id ,e.name,sum(s.amount) as 'total_amount' from employee e inner join sale s on e.id...`	*YES*
409	E022	`select * from table1 a where a.d > coalesce((select b.d from table2 b where a.id = b.id and a.something = 1), '0');`	*YES*
410	E022	`select u.name from user u inner join (select uid from user_profile where p.address = 'some constant') p on u. uid = p.uid;`	*YES*
411	E022	`select t2.word, t1.frequency, t2.secondword, t2. secondfrequency from (select * from (select word, secondword, secondfrequency, row_number() over(partition by word order by secondfrequency desc) as num from table_2) t where t.num <= 3) t2 join table_1 as t1 on t2.word = t1.word order by t2.secondfre...`	*YES*
412	E022	`select u.e_id, case when e_type_id = 1 then u.e_name else ' - ' + u.e_name end e_name, e_type_id, su.n_id from table1 u inner join table3 su on u.e_id = su.e_id where exists (select n_id from table2 where n_id = case when u.e_type_id = 1 then u.e_id else n_id end) order by e_type_id, u.e_name,n_id;`	*YES*
413	E022	`select s.*, u.a, u.b, u.c, u.d, u.e, u.f, c. location_country, st.location_state, ct.location_city from (select user_id from tags where t.skillid = 52772) as t left join search s on t.user_id = s.user_id left join users u on t.user_id = u.user_id left join countries c on s.countryid = c.countryid lef...`	*YES*
414	E022	`select timeslots.timeslot, users.role, users.surname, users.clinic from timeslots, users where timeslots.id not in (select timeslot from appointments where appdate = getdate() and (users.clinic = 'werrington') and (users .role = 'doctor' or users.role = 'nurse')) and (users. role = 'doctor' or users....`	*YES*
415	E022	`select * from customer_tbl where exists( select 2 as customer_tbl where customer_tbl.country = 'mexico' );`	*YES*
416	E022	`update e set e.employeenumber = (select top 1 employeenumber from #employees where e.id = e.id order by newid()) from #employees e;`	*YES*

417	E022	`select distinct winner.person from (select case when t2_1.last_post > t2_2.last_post then person1 else person2 end as person from t1 inner join t2 t2_1 on t1.person1 = t2_1.person inner join t2 t2_2 on t1.person2 = t2_2.person) winner left join (select case when t2_1.last_post < t2_2.last_post then ...`	*YES*
418	E022	``select `name` , count(*) as `count` from `t1`, `t2` where `t2`.`id` = `t1`.`id` group by `t2`.`id` union select name, 0 as count from t1 where not exists (select 1 from t2 where `t2`.`id` = `t1`.`id`);``	*NO*
419	E022	`select a.id,a.type,a.date,b.status1,a.status2,a.status3 from table1 a inner join table2 b inner join table2 c group by a.type having count(a.type)>0 and b.status1='aaa' union select a.id,a.type,a.date,b.status1,a.status2,a.status3 from table1 a inner join table2 b inner join table2 c group by a.type...`	*YES*
420	E022	`select * from bi_employee e where exists (select null from bi_user_access ua where ua.division_id = e.division_id and ua.product_id = e.product_id and ua.sub_product_id = e.sub_product_id and ua.region_id = e.region_id and (e.confidential = 'n' or ua.confidential = 'y') and ua.user_id = :user_id);`	*YES*

Table A.21: Manual analysis of tool performance for rule E022

No.	RuleID	Query	Correct
421	E023	`select dbp.mob_num as mobile_number, dbp.name as name, dbp.area_code as area_code from db_phonebook dbp inner join (select mob_num from db_phonebook dbp where dbp.area_code = 4817 having count(distinct dbp.mob_num) = count(dbp.mob_num)) c_dbp where dbp.area_code = 4817;`	*YES*
422	E023	`select uv.id, if(uv.voc_id = 0,uv.word,sv.word) as word from user_vocabulary uv left join system_vocabulary sv having word like '%user_input%';`	*YES*
423	E023	`select name, height * weight as inchpounds from sashelp.class having inchpounds > 5000;`	*YES*
424	E023	`select *, cond2 as cond2, cond3 as cond3 from table having cond1 and (cond2 or cond3);`	*YES*
425	E023	`select * from card c where exists (select 1 from history h where h.cardid = c.cardid having count(case when h.statusid = 310 then 1 end) = 0);`	*YES*
426	E023	`select department.dname from department join deptloc where deptloc.city in ('boston', 'dallas') having count(distinct deptloc.city) = 1;`	*YES*
427	E023	`select * from item a having orderid not in (select orderid from table_excluded_item);`	*YES*
428	E023	`create table tallest as select name, height from sashelp.class having height = max(height);`	*YES*
429	E023	`select * from a t where exists ( select 1 from a where id = t.id having count(distinct partid) > 1);`	*YES*

430	E023	select customerid, salesorderid, year(orderdate) as ' year' from sales.salesorderheader where year(orderdate) in (2011,2014) having count(year(orderdate))=2;	*YES*
431	E023	select client_id from my_table having balance <> 0;	*YES*
432	E023	select tag from tagging having count(distinct resource) > 2;	*YES*
433	E023	select name from (select name, min(track) as track from horses group by 1 having count (distinct track) = 1) horses_one_race where track = 'sa';	*YES*
434	E023	select userid, sum(money_spent), sum(money_spent_on_candy) / sum(money_spent) as percentcandyspend from moneytable where date >= '2010-01-01' having percentcandyspend > 0.1;	*YES*
435	E023	select distinct name, version from table1 where name in ("asdf", "ghjk") having max(version) = version;	*YES*
436	E023	select column_name, count(column_name) as column_name_tally from table_name where column_name < 3 having count(column_name) >= 3;	*YES*
437	E023	select title.id, title.title from titles as title having points > 0 union all select title.id, title.title from titles as title having points > 1;	*YES*
438	E023	select person_id, max(salary) from yourtable a where exists (select 1 from yourtable b where a.person_id = b. person_id having (a.salary < max(b.salary) and count(*) > 1) or count(distinct salary) = 1);	*YES*
439	E023	select quizzes.*, count(submissions.id) as submissions_count from "quizzes" inner join "submissions " having count(distinct submissions.correct) >= 2;	*YES*
440	E023	select id, name, count(name) from table group by 2,1 having count(name) = 2;	*YES*

Table A.22: Manual analysis of tool performance for rule E023

No.	**RuleID**	**Query**	**Correct**
441	E024	select * from table where entry='cow' or entry = 'appl%' or entry = 'roo%';	*YES*
442	E024	select medications.clinname from medications right join patientdata on patientdata.medid=medications.medid where patientdata.id='*the actual patient id*';	*YES*
443	E024	select airline, flt_no, fairport, tairport, depart, arrive, fare from flights inner join airports from_port on (from_port.code = flights.fairport) inner join airports to_port on (to_port.code = flights.tairport) where from_port.code = '?' or to_port.code = '?' or airports.city='?';	*YES*
444	E024	select count(tweet_id) from tweets where from_user = '%s ';	*YES*
445	E024	select flights.*, fromairports.city as fromcity, toairports.city as tocity from flights left join (airports as fromairports, airports as toairports) where flights.fairport = '?' or fromairports.city = '?';	*YES*

446	E024	`select * from table where foo='#foo#';`	*YES*
447	E024	`select * from employees where name = 'chris%';`	*YES*
448	E024	`select * from providers where id='$[var1]';`	*YES*
449	E024	`select * from sysobjects where name = '#temp_table';`	*YES*
450	E024	``select * from `members` where `memberid` = '[id]' limit 1 union select * from `members`;``	*YES*
451	E024	`select * from information_schema.columns where table_name = '[table name]';`	*YES*
452	E024	`select column_name, table_name from information_schema .columns where schema_name = 'db_name' and table_name ='%98673%' and column_name like '%98673%';`	*YES*
453	E024	`delete from tshirt where sku='%s';`	*YES*
454	E024	`delete from tshirt del where del.sku = '%s';`	*YES*
455	E024	`select count(*) from bo_labels l left join bo_contract_hardwood_deal c on (c.bo_document_fkey = l.bo_doc_base_fkey) where 1=1 and exists ( select 1 from bo_party party where 1=1 and party.id = l.bo_party_fkey and party.inn = '?' );`	*YES*
456	E024	`select * from tablename where fieldname = '#value#';`	*YES*
457	E024	`select distinct owner, object_name from dba_objects where object_type = 'table' and owner = '[some other schema]';`	*YES*
458	E024	`select * from table where entry='cow | appl* | roo*';`	*YES*
459	E024	`select pg_terminate_backend(pg_stat_activity.pid) from pg_stat_activity where pg_stat_activity.datname = '[ database to copy]' and pid <> pg_backend_pid();`	*YES*
460	E024	`update wp_postmeta set meta_value = '0.25' from wp_postmeta as a inner join wp_woocommerce_order_itemmeta as b inner join wp_woocommerce_order_item as c where c.value = '%250g%';`	*YES*

Table A.23: Manual analysis of tool performance for rule E024

No.	**RuleID**	**Query**	**Correct**
461	E025	`select clicks / impressions as probability, round(100 * probability, 1) as percentage from raw_data;`	*YES*
462	E025	`select dev_cost / sell_cost from software ;`	*YES*
463	E025	`select (won/total) as 'rankpercentage' from dbo. filmranking order by rankpercentage desc;`	*YES*
464	E025	`select itm_num from itemconfig where (pal_qty/case_qty) > 500 and case_qty > 0;`	*YES*
465	E025	`update mytable set ave_cost = cost / num;`	*YES*
466	E025	`select * from table where (col1 / col2 ) between 1 and 8 and (col1 / col2 ) = floor(col1 / col2 );`	*YES*
467	E025	`select capacity_used / capacity_total from tablename where capacity_total is not null and capacity_total <> 0;`	*YES*

468	E025	`select (a.quantity + b.quantity + c.quantity) as totalquantity, sum(a.quantity * a.rate) + sum(c.quantity * c.rate) as totalamount, totalquantity/totalamount as result from a, b, c where (a.userid = 1 and a.companyid = 1) and (a.userid = b.userid and a.userid = c.userid and a.companyid = b.companyid...`	*YES*
469	E025	`select (select count(distinct s.lastfirst) from students s join cc on s.id = cc.studentid join courses c on cc.course_number = c.course_number where cc.schoolid ='109' and c.course_name like 'ap %' and substr(cc.termid,0,1) <> '-' and cc.dateenrolled between to_date ('08/01/2010','mm/dd/yyyy') and to...`	*YES*
470	E025	`select h.total_sale, s.f1 / h.total_sale as f1_percent from sales s, (select id, f1 + f2 as total_sale from sales) h where s.id = h.id;`	*YES*
471	E025	`update employee e set e.payroll = e.payroll + 1000 where e.payroll > (select department.dep_payroll / department .dep_amount from department where department.dep_id = e.dep_id);`	*YES*
472	E025	`select *, r1.value / r.value - 1 as return from rownums r inner join rownums r1;`	*YES*
473	E025	`select id, count / maxcount as score from result;`	*YES*
474	E025	`select round(noofboys / noofgirls) as ration from student;`	*YES*
475	E025	`select * from xyz where x/y = (select max(x/y) from xyz) limit 1;`	*YES*
476	E025	`select (current_salary/start_salary) appraisal, * from employee;`	*YES*
477	E025	`select * from properties order by (price / floorsize);`	*YES*
478	E025	`update factsales set unitcost = ( select revenue / quantity from factsales );`	*YES*
479	E025	`select material_id, cost/v_cost_total from materials where material_id >=0 and material_id <= 10;`	*YES*
480	E025	`select *, (100-(table.price/table.oldprice))*100 as discount from table;`	*YES*

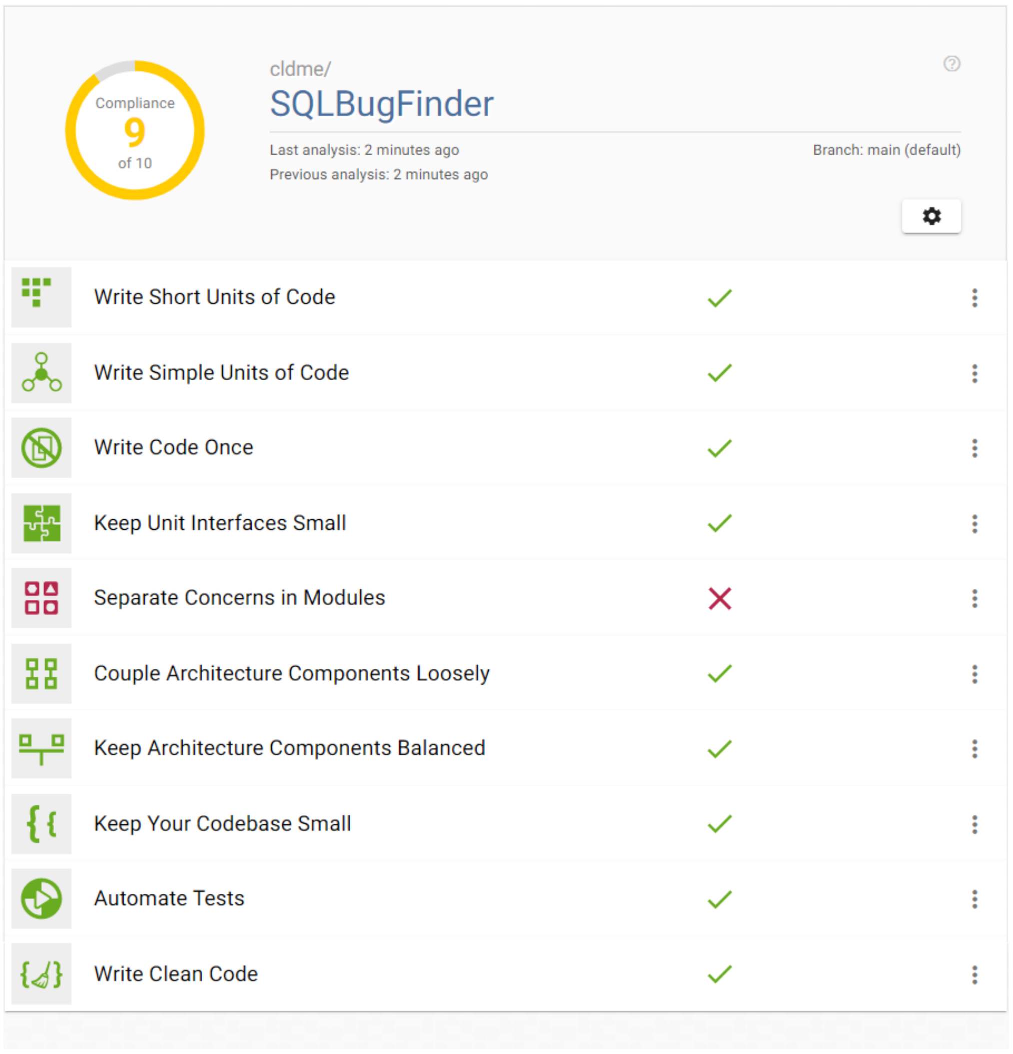
Table A.24: Manual analysis of tool performance for rule E025

Appendix B

Code Quality

The BetterCodeHub[1] platform was used for tracking the code quality for our code. In Figure B.1 we show the metrics generated after running the analysis on our code base. The separate concerns in modules guideline is violated since we chose to have a common abstract class, implemented by all our other rules, which holds most of the common functionality for parsing the queries. This was preferred since it makes it easier to add and test new rules in the future.

[1]`https://bettercodehub.com/`



Compliance 9 of 10

cldme/

SQLBugFinder

Last analysis: 2 minutes ago

Previous analysis: 2 minutes ago

Branch: main (default)

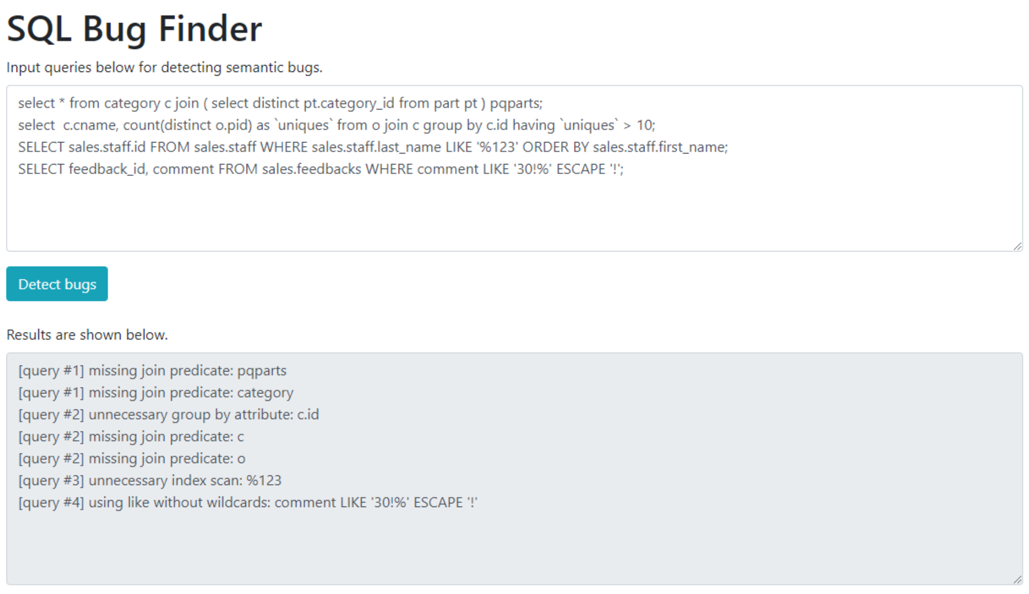
Write Short Units of Code	✓
Write Simple Units of Code	✓
Write Code Once	✓
Keep Unit Interfaces Small	✓
Separate Concerns in Modules	✗
Couple Architecture Components Loosely	✓
Keep Architecture Components Balanced	✓
Keep Your Codebase Small	✓
Automate Tests	✓
Write Clean Code	✓

Figure B.1: Code quality analysis

Appendix C

Tool Website

We make our tool available in the form of an interactive website, where developers can directly input and check their quires for semantic bugs as shown in Figure C.1. This website can be found at the following link: `https://sqlbugfinder.com`.



SQL Bug Finder

Input queries below for detecting semantic bugs.

```
select * from category c join ( select distinct pt.category_id from part pt ) pqparts;
select  c.cname, count(distinct o.pid) as `uniques` from o join c group by c.id having `uniques` > 10;
SELECT sales.staff.id FROM sales.staff WHERE sales.staff.last_name LIKE '%123' ORDER BY sales.staff.first_name;
SELECT feedback_id, comment FROM sales.feedbacks WHERE comment LIKE '30!%' ESCAPE '!';
```

Results are shown below.

```
[query #1] missing join predicate: pqparts
[query #1] missing join predicate: category
[query #2] unnecessary group by attribute: c.id
[query #2] missing join predicate: c
[query #2] missing join predicate: o
[query #3] unnecessary index scan: %123
[query #4] using like without wildcards: comment LIKE '30!%' ESCAPE '!'
```

Figure C.1: Interactive tool website