Errors and Complications in SQL Query Formulation

Errors and Complications in SQL Query Formulation

TONI TAIPALUS and MIKKO SIPONEN, University of Jyvaskyla, Finland
TERO VARTIAINEN, University of Vaasa, Finland

SQL is taught in almost all university level database courses, yet SQL has received relatively little attention in educational research. In this study, we present a database management system independent categorization of SQL query errors that students make in an introductory database course. We base the categorization on previous literature, present a class of logical errors which has not been studied in detail and review and complement these findings by analyzing over 33,000 SQL queries submitted by students. Our analysis verifies error findings presented in previous literature and reveals new types of errors, namely logical errors recurring in similar manners among different students. We present a listing of fundamental SQL query concepts we have identified and based our exercises on, a categorization of different errors and complications and an operational model for designing SQL exercises.

CCS Concepts: • **Social and professional topics** → **Computing education**; **Computer science education**; *Model curricula*;

Additional Key Words and Phrases: Human factors, Languages, Standardization, Errors, Exercise Design, Query Languages, SQL

ACM Reference Format:
Toni Taipalus, Mikko Siponen, and Tero Vartiainen. 2010. Errors and Complications in SQL Query Formulation. *ACM Trans. Comput. Educ.* 9, 4, Article 39 (March 2010), 28 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

Relational databases still dominate the field of enterprise applications [32], and Structured Query Language (SQL) continues to be the de facto database query language. In addition to SQL's current popularity as a topic in database courses [7], the role of SQL in both courses and practice in the future is strengthened by the recent emergence of NewSQL systems, which use SQL as their query language [8, 41].

Different errors that users make have been studied extensively in programming [24] and to some extent in other computer languages such as HTML and CSS [29]. Such studies contribute to increased understanding of the difficulties that users experience in the process of learning new languages [11]. Even though SQL was standardized by ANSI/ISO as early as 1986/1987 [14] and is used widely today, SQL has received less attention in educational research than programming [1].

SQL still lacks a unified error categorization, and error categorizations presented in previous studies [1, 34, 39] have focused on specific errors relevant only to each individual study or have allowed a specific database management system (DBMS) to perform the categorization. DBMS-specific SQL implementations differ from one another [31] and there exists no error categorization

Authors' addresses: Toni Taipalus, toni.taipalus@jyu.fi; Mikko Siponen, mikko.t.siponen@jyu.fi, University of Jyvaskyla, Faculty of Information Technology, P.O. Box 35 (Agora), FI-40014, Jyvaskyla, Finland; Tero Vartiainen, University of Vaasa, Department of Computer Science, Wolffintie 34, P.O. Box 700, 65101, Vaasa, Finland, tero.vartianen@uva.fi.

1946-6226/2010/3-ART39 $15.00
https://doi.org/0000001.0000001

that is both DBMS-independent and also addresses a wide variety of different errors. Additionally, no study is focused on categorizing student errors in SQL in detail in regard to data demand. As defined by Buitendijk [6], data demand is a task expressed in natural language, e.g., "give me all suppliers in London", to which a student is required to write an equivalent SQL query. In other words, all previous studies focused on errors that are apparent, even if the data demand was not known [e.g., 4] or the focus of those studies was not in error categorization [e.g., 6].

An SQL standard based, unified error categorization will benefit database query teaching and research by providing a framework by which study results are comparable, regardless of the DBMS used or error types studied. Furthermore, a DBMS-independent error categorization is not limited to pre-2010s relational implementations but extends to NewSQL systems as well. In terms of teaching, this means that by teaching SQL in a way that conforms to the SQL standard, we are teaching students the use of a query language that is usable in NewSQL systems [cf. 33] as well. With a categorization explaining what errors occur, we can move toward understanding why those errors occur and how we can adjust our teaching methods accordingly so that the occurrence of these errors can be minimized or avoided.

Our work makes two main contributions. First, we review the previous literature on SQL errors to form a basis for the unified error categorization in a DBMS-independent framework. We analyze student data to review and complement these findings and present a categorization of different errors. Second, we address an error class (namely logical errors, see Section 2.2) that has been identified in a previous study [4] but has not been studied in detail, and we categorize SQL errors within that class based on the analysis of student data.

In the next section, we review relevant studies on SQL errors. In Section 3, we describe the course from which the student data was collected, the exercises and the procedure for error categorization. In Section 4, we present the results of our study and compare them to previous research. In Section 5, we discuss the practical implications of our results and in Section 6 present conclusions and future work. Appendix A contains the database diagram presented to the students, and Appendix B contains the SQL exercises with example answers.

2 BACKGROUND

We explored previous research in both computer languages in general and SQL in particular. We conducted a keyword search in scientific databases, such as ACM, IEEE and Elsevier, with keywords and keyword pairs *sql, query, error, category* and *taxonomy*. The publications we found also provided us with references to related studies in other publications.

We have identified two error classes in previous studies: syntax and semantic errors. Before we review error categories identified in previous studies, we discuss which error classes are the focus of those studies and how these error classes are defined.

2.1 Terminology

The previous studies on SQL errors that we examined sometimes used conflicting definitions for key terms, e.g., a *query* could be interpreted either as a statement in SQL [6] or a statement in natural language [39]. Next, we define key terms for this work.

A *query* is an answer that is usually written by a student in SQL for a particular data demand. A query is submitted to the DBMS, and the DBMS outputs a result table or an error message. Although a query is commonly understood as any Data Manipulation Language (DML) statement, this work focuses on data retrieval queries. For convention and clarity, SQL keywords are written in all capital letters.

A *database object* is any object that exists in a database or can be defined with a Data Definition Language (DDL) statement. Database objects include, but are not limited to, tables, columns, schemas, aggregate and scalar functions, triggers and catalogs.

A *clause* is a part of a query initiated by one of the following keywords: SELECT, FROM, WHERE, GROUP BY, HAVING or ORDER BY, and is followed by one of the previous keywords or a semicolon terminating the whole statement. A *source table* is a table from which a query projects or calculates values into the result table. A *subject table* is a table that is utilized to restrict rows in the result table. From a single query's point of view, a table may be a source and a subject table at the same time. A query may contain zero to many source and subject tables. In a special case such as a self-join, the same table may be a subject table multiple times from a single query's point of view.

A WHERE clause is the main restriction clause of an SQL query, and we make a distinction between restrictions: table join conditions are called *joins* and other restrictions are called *expressions*.

2.2 Error Classes

Error categorization has been studied extensively in programming [e.g., 18, 23, 25], and these languages, regardless of whether they were studied in the 1970s or 2010s, are all imperative by nature, whereas SQL is a fundamentally different language in its declarative nature and purpose.

Errors in SQL query formulation have been studied in the past, and different categorizations for errors have been presented [e.g., 39, 40]. Although these studies are still largely relevant, the SQL standard has undergone several changes and received additional features, including the JOIN predicate, outer joins and new data types. Furthermore, more easily understood teaching methods have been proposed, e.g., teaching relational algebraic division with aggregate functions rather than multiple NOT EXISTS subqueries [26].

We aim to establish an error categorization that is independent of the DBMS and whether or not the DBMS is a traditional relational DBMS or a NewSQL system. Different DBMSs implement the SQL standard differently, and although these differences may be minor, we cannot base our error categorization on one DBMS. If we were to base, e.g., our syntax error categorization on SQL Server, that syntax error categorization would not be ours, and the errors would be categorized by the DBMS. Consequently, categorizing the same data with a different DBMS would result in a different categorization. Because of these differences and the aim of our study, we use the SQL standard as a reference when categorizing the errors.

Previous research [1, 2, 4, 6, 34, 39] categorizes SQL errors under two classes: syntax and semantic errors. The division is intuitive because DBMSs detect syntax but not semantic errors. All the aforementioned research agrees that a query containing a syntax error is not valid SQL, and the DBMS returns an error message. We add that, whether or not a query contains a syntax error can depend on the DBMS, e.g., query rewriters of some DBMSs like MySQL may automatically add type casts to expressions such as *order_number LIKE 100*, while others like PostgreSQL may return a syntax error. Because we consider syntax errors in accordance with the SQL standard, all syntax errors are not necessarily caught (or tolerated) by the DBMS.

In regard to semantic errors, previous research presents definitions for different levels of precision. Buitendijk [6, p. 79] states that a query that contains a semantic error is produced with "an erroneous train of thought". Buitendijk's [6] study contains the word pair *logical errors*, but the SQL errors that are not syntax errors are called semantic errors. However, Buitendijk [6] points out that it is possible for a user to commit a logical error, which results in the query being incorrect for the particular data demand. Smelcer [34] and Ahadi et al. [1] propose that the semantically incorrect query is syntactically correct but returns information not intended by the user. Ahadi et al. [3] state that a query with a semantic error produces either an empty result table or a result table that is not the same as desired.

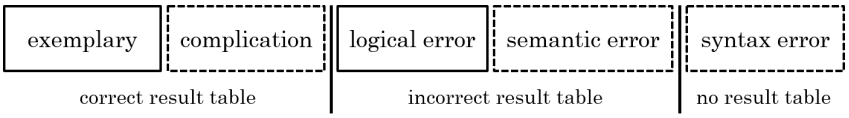


Fig. 1. Error classes and their relationship with the result table and data demand

Brass and Goldberg [4] detail the semantic error definition by stating that the SQL query is legal (i.e., syntactically correct) but does not always produce the intended results for a given data demand (they use the word *task*). Brass and Goldberg [4] make a further distinction between queries that are incorrect regardless of the data demand and queries that are incorrect for a particular data demand. These authors leave the latter class of semantic errors outside of their study and state that it is possible to implement functionality to DBMSs to detect the former class of semantic errors. We consider these former and latter classes of semantic errors fundamentally different by definition and from now on call the latter class *logical errors* and the former class *semantic errors*. To clarify the difference between semantic and logical errors, consider queries Q1 and Q2 in a database of a single table, EMP(empno, fname, sname, job):

Q1 [4]:
```
SELECT *
FROM emp
WHERE job = 'clerk' AND job = 'manager';
```

Q2:
```
SELECT *
FROM emp
WHERE job = 'clerk' OR job = 'manager';
```

Given that the database is in the first normal form, both of the expressions in Q1 can never be true, and, therefore, the WHERE clause could be reduced to *WHERE False*, which will always return an empty result table. It follows that, even without knowing what the data demand is, Q1 contains a semantic error. It could be argued that Q1 is correct if the data demand is to "list all the information about employees whose job is both clerk and manager", but this data demand is not valid because, with this database structure, it is not possible to store more than one job per employee. Q2, however, does not contain a semantic error. Whether or not Q2 contains a logical error depends on the data demand associated with the query. If the data demand for Q2 is to "list all the information about clerks and managers", Q2 is correct. If the data demand is something else, Q2 contains a logical error. Our work essentially builds upon Brass and Goldberg's study [4]; they recognize the existence of logical errors, but they leave logical errors outside of their study. Our study verifies their semantic errors, discovers new semantic errors and studies logical errors in detail, namely, what logical errors occur in query writing, and characteristics and frequencies of those errors.

Brass and Goldberg [4] further propose that queries that are unnecessarily complicated are also considered to contain a semantic error. While Brass and Goldberg focus on compiler warnings and optimization in their study, and we focus on students learning SQL, we make a further distinction between semantic errors and complications. Both semantic errors and complications share the characteristic of being evident by reading the query without knowledge of the data demand, but while semantic errors affect the data in the result table, complications do not (see Section 4.4).

Figure 1 summarizes the definitions we use in this study. The rectangles represent queries with a certain characteristic, the leftmost query being without errors or complications. The text below the rectangles represents the nature of the result table that the query returns. The rectangles with dotted lines represent error classes that can be recognized without knowing the data demand.

3 STUDY

In order to create an unified SQL error categorization, our study had the research goal of categorizing errors and complications in students' SQL retrieval statements in an introductory database course. As we categorize students' errors during their learning processes, our research approach is interpretive in nature. Interpretive studies may be divided into the types of "outside researcher" or "involved researcher" [38], and our study represents the involved type, as the first author of our study carries out the teaching activity and aims to develop database education based on the results of this study. This "close involvement" [38] means that, as the course teacher, the first author is close to the students who produce the SQL sentences and is therefore able to understand the context of the data production, the problems students face and the issues emerging. Although this kind of involvement is time and resource consuming, the first author's involvement makes it possible to create an in-depth understanding of how to develop education with respect to database teaching. The error categorization is based on analysis of over 33,000 SQL queries submitted by 237 students via an e-learning environment developed in the university of the first two authors. The queries were collected during a one-semester course taught by the first author.

Writing the correct SQL query for a course assignment can take several tries, and making an error or several errors does not necessarily mean that a student is unable to complete an assignment [7]. However, our e-learning environment logs all queries submitted to the DBMS, thus offering more fine-grained data than is usually available for analysis. The granularity of logging allows us to move closer to the student in our analysis as proposed by Fincher et al. [13], as opposed to only analyzing the final answers. If we only analyzed the final answers, it is possible that we would miss some errors. For this course, the e-learning environment provides a minimal interface for the students to communicate with the DBMS via a web browser. The user interface is an interactive SQL prompt similar to those provided by most relational DBMSs. The difference in the e-learning environment's SQL prompt is that it is embedded to a web page and that the students are not required to install anything. Even though different SQL learning environments have been studied in the past [e.g., 5], for this work, the e-learning environment simply provided a method of collecting data and was not the subject of the study.

3.1 Course

The course in question mostly follows the ACM/AIS IS 2010 curriculum [37] guidelines for the core course in data and information management. Topics of the course include conceptual modeling, relational model and algebra, SQL, normalization up to fourth normal form and data warehousing. The course is primarily taken by second-year CS and IT majors but also by students from other departments minoring in ICT with no previous experience in SQL.

Query languages are taught in five phases in the course. First, students are introduced to the concept of relationally complete relational algebra, which lays the foundation for different operations that queries utilize to retrieve data from a relational database. In the next four phases, SQL is introduced gradually by the four sublanguages DML, DDL, Data Control Language (DCL) and Transaction Control Language (TCL).

In order for the reader to understand the scope of our study, it is worth discussing how and what aspects of DML, namely data retrieval, are taught in the course. All the DML aspects in the course are taught according to the SQL standard and without a product-specific dialect. Our e-learning environment, however, utilizes the SQLite DBMS to which the student queries are sent. SQLite returns a result table or an error message depending on the query submitted. In addition to the basic concepts like SELECT, FROM and ORDER BY clauses, and basic operators like classic comparison, LIKE, IS and IN, table joins are taught in four different methods: subqueries with IN, subqueries

with EXISTS, explicit join without a subquery and the keyword JOIN and its features like OUTER and NATURAL. The differences in syntax and the logic behind each of the methods are explained. After that, a student is free to choose the methods to his or her liking and use those in practice.

Grouping concepts are also taught in the course to the extent of GROUP BY and HAVING clauses as well as aggregate functions. More advanced SQL concepts like CASE, WINDOW, recursion, non-primitive grouping [22, p. 345] or runtime SQL are not discussed (see Section 3.2 for more details on the query concepts). We expect error categorization to provide the disciplinary knowledge and the fine-grained student data to provide knowledge on how students learn, as proposed by Tenenberg and McCartney [36].

The students were given the opportunity to complete assignments to earn points toward a better grade, and among these assignments were 15 SQL retrieval statements. The students could choose whether to omit or include their queries from the study, and everyone chose to participate. Points were given regardless of participation. The students were given seven days and unlimited tries to complete each of the three sets of assignments (see sets A, B and C in Table 1) wherever and using whatever material available (course material, internet and any means of communication) to more accurately mimic today's work environments. The exercises within a set could be completed in whatever order, and the correct result table was presented for each of the exercises during the whole process so that the students could compare it with the result table produced by their query. Although existing literature on this approach is limited, Ahadi et al. [2] note that, compared to work environments, providing the correct result table constitutes in making the environment unnatural, as nobody knows the correct result table when writing a query. Prior [30], however, utilizes an approach similar to ours and suggests that providing the correct result table facilitates query formulation skills. The students were also presented a database schema diagram representing tables, their columns, data types and primary and foreign keys (see Appendix A). SQLite allowed the students to obtain more detailed information on the database objects, if necessary.

3.2 Exercises

The exercises were completed using a database with 11 tables (see Appendix A). The data were handcrafted, and each table contained 5-125 rows, with an average of 60 rows. Table 1 lists all the fundamental concepts associated with each of the exercises. The data demand and one example answer for each exercise are listed in Appendix B. While concepts like *single-table, multi-table, expressions, nesting, ordering* and *grouping* are self-explanatory and discussed in many course books [e.g., 12, 27], the more ambiguous concepts are explained below.

The concept named *facing foreign keys* describes a situation in which table A has a foreign key constraint linking to table B, and table B linking to table A (see foreign keys between tables *store* and *employee* in Appendix A). The concept named *does not exist* is expressed as a negated existential quantifier ($\neg\exists$) in tuple relational calculus. Syntactically, the concept is simple and is achieved by the logical operator *not* and in SQL with NOT EXISTS or NOT IN, followed by a subquery or with OUTER JOIN.

The concept named *equal subqueries* stands for two or more subqueries that are not nested, but on the same level (compare example answers B6 and B8 in Appendix B). *Aggregate function evaluated against a column value* (Q3) and *a constant* (Q4) are special cases in which the result returned by a aggregate function must be evaluated against a column value or a constant using a comparison operator. The column or constant side of the evaluation is in the upper level query, and the aggregate function is in the subquery. In the scope of our course, evaluation against a constant usually requires a correlated subquery, whereas evaluation against a column value requires an uncorrelated subquery. Note that such comparisons related to grouping restrictions accomplished by the HAVING clause are not included in this concept.

Table 1. Each exercise's fundamental concepts and the number of source and subject tables required to write a correct query. Tables total lists the number of tables in the correct query's FROM -clause or clauses. Concepts marked with an asterisk are also used in Ahadi et al. [2]

#	Fundamental concepts	Source tables	Subject tables	Tables total
A1	single-table*; expressions	1	1	1
A2	single-table*; expressions; ordering	1	1	1
A3	single-table*; wildcard; expressions with nesting	1	1	1
B4	multi-table*; expressions; facing foreign keys	1	1	2
B5	multi-table*; expressions with nesting; ordering	1	3	3
B6	multi-table*; expressions; does not exist	1	2	3
B7	multi-table*; expressions; does not exist	1	2	2
B8	multi-table*; expressions; does not exist; equal subqueries	1	2	3
B9	single-table*; expressions; aggregate functions*	1	1	1
B10	multi-table*; expressions; multiple source tables	2	3	4
B11	multi-table*; expressions; self-join*; aggregate function evaluated against a column value; correlated subquery*	1	2	2
B12	multi-table*; expressions; aggregate function evaluated against a constant; uncorrelated subquery*; parameter distinct	1	1	2
B13	multi-table*; expressions; self-join*;	1	5	5
C14	multi-table*; multiple source tables; aggregate functions*; grouping*	2	1	2
C15	multi-table*; multiple source tables; aggregate functions*; grouping*; grouping restrictions*; ordering	2	1	2

Q3:
```
[...]WHERE price =
  (SELECT MAX(price)
  FROM[...]
```

Q4:
```
[...]WHERE 8 <
  (SELECT COUNT(*)
  FROM[...]
```

Not all of the exercises test a novel fundamental concept but rather the student's ability to combine previously learned concepts in different contexts. For example, in multi-table queries, simple expressions require the necessary understanding of the query language as well as the database structure regarding an expression's placement in the correct clause. Based on previous teaching experience, we believe that a number of these concepts invite the possibility of different logical errors. For example, a data demand that requires expressions with nesting invites the possibility of missing or incorrect nesting, whereas a data demand with grouping restrictions invites the possibility of insufficient grouping, a missing HAVING clause or missing expressions.

3.3 Methodology

From the diversity of interpretive research methods, we selected directed and conventional content analysis [19] to study students' SQL queries. Directed content analysis is used when prior research exists about a phenomenon but the literature is perceived to be incomplete. In directed content analysis, the existing theory is used to direct the analysis of the data, for example. The strength of directed content analysis is that existing literature can be extended. Conventional content analysis

is used to study the phenomenon when research literature or existing theory on the phenomenon is limited. The emergence of categories is data-driven, meaning that preconceived categories are not used [19]. In this study, we first produced a synthesis of errors based on existing literature, and then the first author used directed content analysis to determine error categories in students' SQL queries. Most of the errors were categorized, but there were errors not found in pre-existing studies, especially in the class of logical errors. Therefore, conventional content analysis was then used to study those uncategorized logical errors. Next, we report our analysis step-by-step.

First, we gathered the errors, complications and error categories listed or discussed by Welty [39], Smelcer [34], Brass and Goldberg [4] and Ahadi et al. [1]. We did not discuss runtime SQL in our course and left runtime errors discussed by Brass and Goldberg [4] outside our categorization. Next, when it was possible in terms of the level of detail used in the previous studies, we considered to which of the four classes the error or complication belonged.

Second, we grouped similar errors and complications in previous studies together. This was done because some of the studies discussed similar concepts using slightly different names or descriptions, e.g., "misspellings" by Smelcer [34] and "using AVE instead of AVG" by Welty [39] were grouped together as "misspellings" in our listing.

Third, the first author used the lists of syntax errors, semantic errors and complications produced in the previous step to perform directed content analysis on the student data, as proposed by Shannon and Hsieh [19]. He grouped the SQL queries submitted by students by exercise number and then by student, and then he sorted the queries by the timestamp the query was submitted to the DBMS. We wanted the DBMS to influence the results as little as possible, and the first author analyzed the data without any computerized automation, e.g., a DBMS or scripts. It was possible for a single query to demonstrate several errors and complications. The first author coded queries with appropriate codes corresponding to errors and complications found in previous literature. Any query demonstrating a new error or complication was marked. The queries that were marked were analyzed again to determine the classes of errors the query demonstrated. New syntax errors, semantic errors and complications were given a new code, and if the query did not demonstrate a logical error, the mark was removed.

Fourth, the first author analyzed the queries that were left marked in the previous step, i.e., queries with logical errors, using conventional content analysis as proposed by Shannon and Hsieh [19]. Every time a new logical error was encountered, it was given a code and a brief description. After all the data had been analyzed, the first author considered whether or not the logical errors were sufficiently similar to be grouped together, e.g., he grouped "join with > operator, when equijoin is required" and "join with < operator, when equijoin is required" together as "join with incorrect comparison operator". We decided not to disregard errors just because they were infrequent because, intuitively, different exercises invite different errors, and just because an error is infrequent in our exercises does not necessarily mean that it would be infrequent in some other exercises.

Finally, the first author grouped errors and complications together into categories within each of the four classes. These eighteen categories each represent a distinct family of errors or complications that are similar in nature. These eighteen categories, and whether they are new or previously identified, are discussed in detail in Section 4.

4 RESULTS

All the SQL examples provided in this section contain an error demonstrating the related error category. The queries submitted by students were often complicated but the errors simple; because of this, the example queries in Section 4 are not actual student answers but modified by us for brevity and clarity to represent the error in question as clearly as possible. Unless stated otherwise, the database schema for the example queries is as presented in Appendix A.

We gathered syntax, semantic and logical errors found in previous literature and student data into Table 2, Table 4 and Table 5, respectively. Complications are presented in Table 6. The references in italics represent errors we also encountered in the student data. The errors without a reference are new errors we did not find in previous studies but encountered in the student data. The numbers inside brackets in the Discussed in -column correspond to errors numbered by Brass and Goldberg [4], and the IDs in the ID column are defined by us in the coding phase, as reported in Section 3.3. We have also briefly elaborated on some of the previously identified errors in the tables.

4.1 Syntax Errors

Syntax errors were numerous in the student data. We identified 23 errors in the previous literature, some of which had different levels of overlapping. Overlapping is a result of the level of detail errors are discussed in different previous studies, e.g., Ahadi et al. [1] list different cases where the query refers to nonexistent database objects (error IDs 4-8), whereas Smelcer [34] abstracts these cases to misspellings and synonyms (error IDs 9-10).

4.1.1 SYN-1 Ambiguous Database Object. Database objects, such as tables, schemas and catalogs, each form namespaces on different levels in the database, and the DBMS prevents naming conflicts within a namespace. In multi-table, multi-schema or multi-catalog queries, however, namespaces are merged and, e.g., without additional qualifiers, column names can become ambiguous. In SQL, these qualifiers are called correlation names (i.e., aliases), and omitting them leads to a syntax error if the DBMS cannot resolve the database object to which the query refers.

Cleve et al. [9] researched a schema evolution of a database in which the number of tables grew roughly from 100 to 450. A database of hundreds of tables creates a need for namespaces of different levels, and the queries to such databases need correlation names of corresponding levels. We see no need to divide this subcategory to more fine-grained categories, such as ambiguous columns, tables, schemas or functions, since future standards may introduce new database objects.

4.1.2 SYN-2 Undefined Database Object. Queries with references to nonexistent database objects cause a syntax error for the same reason as references to unambiguous database objects because the DBMS cannot resolve the database object based on the information schema.

This category combines syntax errors for undefined columns, tables and functions and extends the categorization to include all other database objects as well. Smelcer's [34] names "synonyms" and "misspellings" give some insight as to why these types of errors occur, such as the student making a typographical error or remembering the object name incorrectly, e.g., *customers* instead of *customer*. In these two cases, the error is relatively easy to fix, but the error can also be caused by more severe problems, such as the failure to understand the database structure.

4.1.3 SYN-3 Data Type Mismatch. As pointed out in Section 2.2, some DBMSs handle some type conversions automatically while others do not. We argue that the type conversion, i.e., using the correct operator, should always rest on the query writer, especially when teaching SQL in order to simulate a DBMS-independent environment. When appropriate operators are used and when necessary type casts are explicit and in accordance with the SQL standard, the queries are more portable from one DBMS to the other. Next, we elaborate on the common cases of this error based on the student data.

First, we observed using an unfit operator for a column data type. Examples of these errors include cases of using LIKE instead of a classic comparison operator, IS instead of LIKE and IN with something other than a list with atomic values. Second, we observed omitting quotes around character strings (error ID 11) or adding quotes around other data types such as numerical values or Booleans.

Table 2. Syntax errors and error categories

ID	Error	Discussed in
SYN-1 Ambiguous database object		
1	omitting correlation names	*[34, 39]*
2	ambiguous column	*[1]*
3	ambiguous function	[1]
SYN-2 Undefined database object		
4	undefined column	*[1]*
5	undefined function	*[1]*
6	undefined parameter	*[1]*
7	undefined object	*[1]*
8	invalid schema name	*[1]*
9	misspellings	*[34, 39]*
10	synonyms	*[34]*
11	omitting quotes around character data	*[34]*
SYN-3 Data type mismatch		
12	failure to specify column name twice	*[34]*
13	data type mismatch	*[1]*
SYN-4 Illegal aggregate function placement		
14	using aggregate function outside SELECT or HAVING	*[39]*
15	grouping error: aggregate functions cannot be nested	*[1]*
SYN-5 Illegal or insufficient grouping		
16	grouping error: extraneous or omitted grouping column	*[4] (21)*
17	strange HAVING: HAVING without GROUP BY	*[4] (32)*
SYN-6 Common syntax error		
18	confusing function with function parameter	*[39]*
19	using WHERE twice	*[34, 39]*
20	omitting the FROM clause	*[34]*
21	comparison with NULL	*[4] (9)*
22	omitting the semicolon	*[1]*
23	date time field overflow	[1]
24	duplicate clause	
25	using an undefined correlation name	
26	too many columns in subquery	
27	confusing table names with column names	
28	restriction in SELECT clause (e.g., SELECT fee >10)	
29	projection in WHERE clause (e.g., WHERE firstname, surname)	
30	confusing the order of keywords (e.g., FROM customer SELECT fee)	
31	confusing the logic of keywords (e.g. grouping instead of ordering)	
32	confusing the syntax of keywords (e.g., LIKE ('A', 'B'))	
33	omitting commas	
34	curly, square or unmatched brackets	
35	IS where not applicable	
36	nonstandard keywords or standard keywords in wrong context	
37	nonstandard operators: (e.g., &&, || or ==)	
38	additional semicolon	

Third, we observed failure to understand that both sides of logical operators AND and OR must be evaluated as Boolean values, i.e., arguments of WHERE and HAVING clauses must be Boolean type (error ID 13). This was a particularly common syntax error with the LIKE operator, and, in some rarer cases, a classic comparison operator was used to compare a value to a set of values. The former case was demonstrated by Smelcer [34]. Fourth, we observed using a column as a function parameter even though the column data type does not match the parameter's required data type (usually results in error ID 13).

4.1.4 SYN-4 Illegal Aggregate Function Placement. The only two clauses in which aggregate functions COUNT, SUM, AVG, MIN and MAX can be used in the scope of our course are the SELECT and HAVING clauses. Placing an aggregate function in any other clause or using an aggregate function as a parameter to another aggregate function causes a syntax error. This syntax error was common with exercises involving aggregate function evaluation against a column or a constant.

4.1.5 SYN-5 Illegal or Insufficient Grouping. According to the SQL standard, two approaches exist to implement grouping, and we call the non-additional implementation *strict* [22, p. 321-328]. The strict approach determines that a query with at least one aggregate function and at least one grouping column in the query's main SELECT clause must contain a GROUP BY clause that groups the result table according to all grouping columns and only the grouping columns. Execution of such a query should result in a syntax error if grouping is missing altogether, grouping is not applied to all grouping columns or grouping is applied to non-grouping columns.

The additional, less strict approach (optional feature T301) states that all the grouping columns in the SELECT clause must be functionally dependent on the columns listed in the GROUP BY clause [22, p. 341-349]. Although theories and implementations of automated functional dependency discovery to various levels of reliability have been proposed [e.g., 17, 20, 21], the query processors of relational DBMSs are seldom aware of the functional dependencies present in the database. Because of this, we consider the T301 approach problematic when learning SQL. To briefly demonstrate the matter, let us assume Table 3 and queries Q5 and Q6 for the remainder of this subsection only.

Table 3. Customer table

cno	fee
1	0
2	10
3	10

Q5:
```
SELECT cno, MAX(fee)
FROM customer;
```

Q6:
```
SELECT cno, MIN(fee), MAX(fee)
FROM customer;
```

We found this error particularly interesting because of different popular DBMS default settings. Because the grouping is missing, both Q5 and Q6 return a syntax error in PostgreSQL, Oracle Database, SQL Server and DB/2, all of which are among the most popular relational DBMSs. MySQL and SQLite, however, return a result table that contains one row based on the execution plan. It is worth noting that the result table returned by Q6 contains a spurious row, either (2, 0, 10) or (3, 0, 10). Furthermore, the row in the result table changes based on the order of the aggregate functions. The outcome of implementing the T301 approach and the fact that the query processor is not aware of functional dependencies is that grouping can be applied to whatever columns or omitted

completely. We extend this category to the HAVING clause as well because it is closely related to grouping, and the same approaches (strict and T301) apply to it. Our e-learning environment utilized SQLite, which implements the T301 approach for grouping by default, and because we overlooked this possibility when designing the exercise data, approximately 59% of the 158 students who solved exercise B11 used a similar erroneous query and, satisfied with the result table, moved on to the next exercise. Queries Q7 and Q8 explain the name of this error category.

Q7:
```
SELECT cno, AVG(fee)
FROM customer
GROUP BY city;
```

Q8:
```
SELECT cno, city, AVG(fee)
FROM customer
GROUP BY city;
```

Since customer number (cno) is not functionally dependent on city, Q7 displays illegal grouping and Q8 insufficient grouping. To sum up, insufficient or illegal grouping causes no syntax error in some popular DBMSs, but we cannot find any data demand by which having a result table with a spurious row or a row that is seemingly but not truly random could be useful. Based on the arguments above, we consider this a syntax error, and not a semantic or logical error unless functional dependencies can be reliably identified and utilized by the DBMS to prevent behavior not conforming to the SQL standard.

4.1.6 SYN-6 Common Syntax Error. Because syntax errors were numerous and diverse, we categorized all syntax errors not fitting in clear patterns as common syntax errors, similar to Ahadi et al. [1]. Examples of common syntax errors were misspellings of SQL keywords, missing semicolon, brackets or commas, projection in a wrong clause, incorrect clause ordering and missing clauses.

Without the FROM clause, no column values can be projected or calculated since no tables or views are declared. Although Smelcer [34] categorizes FROM clause omission as a semantic error, we consider it a syntax error since the SQL standard requires the FROM clause [22, p. 55-56]. Although some of the popular implementations conform to the SQL standard in this regard, others such as PostgreSQL allow the FROM clause to be omitted, e.g., to perform calculations or calling scalar functions that do not necessarily operate on database data. In such cases, omitting the FROM clause is not a semantic or logical error.

4.2 Semantic Errors

Brass and Goldberg [4] state that their semantic error listing has "a certain degree of completeness" and our findings support theirs; thus, we found few semantic errors not listed in their study. We did not encounter errors regarding UNION in the student data because none of our exercises required the use of UNION. Similarly, errors regarding OUTER JOIN were rare in the student data due to the fact that most of the students chose to solve the exercises dealing with the concept of *does not exist* with NOT IN or NOT EXISTS rather than using OUTER JOIN.

4.2.1 SEM-1 Inconsistent Expression. An inconsistent expression is an expression that causes the result table to be empty or to contain all rows. Smelcer [34] lists four sample cases of AND/OR difficulties, some of which are semantic errors and some clearly syntax errors.

A common example of an inconsistent expression in the student data was when the data demand implied an AND operator even though OR was required (e.g., exercise A1). We also grouped problems with wildcards to this subcategory. Examples of wildcard errors included using an asterisk instead of a percent sign, confusing the functionality of percent and underscore signs, using wildcards with classical comparison operators or with IN and errors with null comparison.

Table 4. Semantic errors and error categories

ID	Error	Discussed in
SEM-1 Inconsistent expression		
39	AND instead of OR (empty result table)	*[34]*
40	implied, tautological or inconsistent expression	*[4] (1, 8)*
41	DISTINCT in SUM or AVG	*[4] (33)*
42	DISTINCT that might remove important duplicates	*[4] (38)*
43	wildcards without LIKE	*[4] (34)*
44	incorrect wildcard: using _ instead of % or using, e.g., *	
45	mixing a >0 with IS NOT NULL or empty string with NULL	
SEM-2 Inconsistent join		
46	NULL in IN/ANY/ALL subquery	*[4] (10)*
47	join on incorrect column (matches impossible)	*[4] (29, 31)*
SEM-3 Missing join		
48	omitting a join	*[4, 34] (27, 28)*
SEM-4 Duplicate rows		
49	many duplicates	*[4] (37)*
SEM-5 Redundant column output		
50	constant column output	*[4] (3)*
51	duplicate column output	*[4] (4)*

4.2.2 SEM-2 Inconsistent Join. An inconsistent join is an error that causes the result table to be empty or, if there are no other restrictions, contain all the rows, and it is certainly not intended by the query writer. Examples of inconsistent joins in the student data were multi-table join conditions using columns with data types that did not match or that matched but had data ranges that never overlapped, resulting in the join condition returning no rows. Self-joins using a wrong correlation name were also observed.

Q9:
```
SELECT cust_id
FROM customer
WHERE cust_id IN
  (SELECT renno
  FROM rental);
```

Q10:
```
SELECT s1.stono, s1.street
FROM store s1, store s2
WHERE s1.city = s1.city
AND s2.stono = 100
AND s1.stono <>100;
```

Q9 demonstrates a join condition using columns that have different data types and no overlapping values. The join condition in Q10 by itself does not restrict any rows because one of the correlation names in the join is incorrectly s1 instead of s2.

4.2.3 SEM-3 Missing Join. Based on the student data, we noticed that table joins using IN or JOIN contained fewer missing joins than joins with EXISTS or explicit join conditions without subqueries. Even though a table join is usually required in multi-table queries, some special cases exist in which omitting the join is desired. This argument is also supported by the SQL standard that specifies an optional feature (F401-04) specifically for joinless (i.e., no column values are compared) multi-table queries; CROSS JOIN [22, p. 1197]. Practical implications for joinless multi-table queries, however, are scarce. We emphasize the meaning of the word *missing* here. A query contains this

semantic error if a join is clearly implied (but omitted) for the result table to contain meaningful data.

4.2.4 SEM-4 Duplicate Rows. Duplicate rows in the result table serve no purpose. We consider a query that by design invites the possibility of duplicate rows to contain a semantic error. Duplicate rows returned by the DBMS cause more data to be transferred between the disk and buffer, between software tiers and possibly in the network. Duplicate rows also make the result table more difficult to read for the end user. The error is remedied by using DISTINCT when needed. It is worth noting that DISTINCT should not be used unnecessarily because it can decrease performance [4].

4.2.5 SEM-5 Redundant Column Output. Redundant column output demonstrates a projection error that causes the result table to contain one or more columns that provide no useful data. Brass and Goldberg [4] demonstrated two semantic errors related to redundant column outputs: constant column output and duplicate column output. Even though it could be argued that SEM-4 and SEM-5 represent a similar concept, i.e., redundant data in the result table, we argue that they demonstrate a different kind of error. SEM-5 represents errors that are possible for the student to identify relatively easily even before running the query. SEM-4, however, can result from the student using different methods for joining tables, e.g., a join with IN is less likely to produce duplicate rows than an explicit join without a subquery.

4.3 Logical Errors

We discovered 30 logical errors, 28 of which were not discussed in previous studies. Contrary to syntax and semantic errors that have been demonstrated in previous studies, we have provided more examples of logical errors listed in Table 5.

4.3.1 LOG-1 Operator Error. For this category, we grouped together errors concerning comparison and logical operators. Operator errors cover errors with missing, extraneous (i.e., not required) or misplaced operators, such as NOT, confusions with classical comparison operators and BETWEEN, and using OR instead of AND. Note that using OR instead of AND can be a logical or semantic error depending on the columns compared and other expressions in the clause. Two particularly interesting and common examples of operator errors (IDs 55 and 56) were related to existence negation in exercises B7 and B8. For brevity, let us consider the data demand "list the surnames of actors who have never acted in a movie released in 2015":

```
Q11:
SELECT a.sname
FROM actor a
WHERE EXISTS
  (SELECT *
  FROM acts s);
  WHERE a.actno = s.actno
  AND EXISTS
    (SELECT *
    FROM movie m
    WHERE s.movno = m.movno
    AND m.year <>2015)
  );
```

```
Q12:
SELECT a.sname
FROM actor a
WHERE EXISTS
  (SELECT *
  FROM acts s);
  WHERE a.actno = s.actno
  AND NOT EXISTS
    (SELECT *
    FROM movie m
    WHERE s.movno = m.movno
    AND m.year = 2015)
  );
```

Both Q11 and Q12 contain a logical error and answer to different data demands: "list the names of actors who have acted in at least one movie not released in 2015" and "list the names of actors

Table 5. Logical errors and error categories

ID	Error	Discussed in
LOG-1 Operator error		
52	OR instead of AND	
53	extraneous NOT operator	
54	missing NOT operator	
55	substituting existence negation with <>	
56	putting NOT in front of incorrect IN/EXISTS	
57	incorrect comparison operator or incorrect value compared	
LOG-2 Join error		
58	join on incorrect table	
59	join when join needs to be omitted	
60	join on incorrect column (matches possible)	
61	join with incorrect comparison operator	
62	missing join	
LOG-3 Nesting error		
63	improper nesting of expressions	*[34]*
64	improper nesting of subqueries	
LOG-4 Expression error		
65	extraneous quotes	*[39]*
66	missing expression	
67	expression on incorrect column	
68	extraneous expression	
69	expression in incorrect clause	
LOG-5 Projection error		
70	extraneous column in SELECT	
71	missing column from SELECT	
72	missing DISTINCT from SELECT	
73	missing AS from SELECT	
74	missing column from ORDER BY clause	
75	incorrect column in ORDER BY clause	
76	exraneous ORDER BY clause	
77	incorrect ordering of rows	
LOG-6 Function error		
78	DISTINCT as function parameter where not applicable	
79	missing DISTINCT from function parameter	
80	incorrect function	
81	incorrect column as function parameter	

who have acted in at least one movie but not in a movie that was released in 2015", respectively. Although both of the examples above are syntactically and semantically correct, they display a lack of understanding of either the functionality of the language, the data demand or both. Moreover, both of these queries will likely return a seemingly correct result table in terms of the number of rows returned. Although we discuss these errors in the lectures prior to the exercises, the error demonstrated in Q11 was common in the student data.

4.3.2 LOG-2 Join Error. The student data showed a common join-related error in one of the exercises. Because the tables *employee* and *store* had facing foreign key constraints between them, the students were required to choose the appropriate columns for the table join. Consider Q13 with respect to the data demand "list the city and phone number of the store in which Jaakko Mattila works":

```
Q13:
SELECT s.city, s.phone
FROM store s, employee e
WHERE s.empno = e.empno
AND e.fname = 'Jaakko'
AND e.sname = 'Mattila';
```

```
Q14:
SELECT COUNT(*) AS total
FROM movie m, acts a
WHERE m.movno = a.movno
AND m.year BETWEEN 1970 AND 2000;
```

Because the join condition is erroneous (error ID 60), the result table shows the cities and phone numbers of the stores of which Jaakko Mattila is responsible for (but does not necessarily work in). Again, the result table can be seemingly correct depending on the number of stores an employee can be responsible for. Although this type of error might be mitigated by renaming the columns with more descriptive names, e.g., *manager_empno*, this exercise served as a demonstration of the importance of choosing the correct columns for a table join and that a NATURAL JOIN, although preferable in terms of enforcing uniform column names and code maintainability, is not always possible.

Q14 demonstrates an extraneous join condition for the data demand "list the number of movies released between the years 1970 and 2000." The example contains a join condition that checks whether or not the movie has any actors, thus possibly limiting the results. In this case, the join must be omitted for the query to be logically correct. A join on incorrect table (error ID 58) means that a join was required by the data demand, and the query contained a syntactically and semantically legal join but on an incorrect table. Missing join (error ID 62) as a logical error differs from missing join listed in semantic errors. Consider the example answer for exercise B8 in Appendix B; if we omit the last subquery, the query is still semantically correct but logically incorrect.

4.3.3 LOG-3 Nesting Error. Difficulties with Boolean logic are a recognized and studied phenomenon in query writing [16], and we categorized these errors as nesting errors, i.e., erroneous use of brackets. This subcategory is also listed by Smelcer [34] under AND/OR difficulties as improper nesting. Smelcer demonstrated improper nesting with an example of improper nesting of expressions. We extend this category with improper nesting of subqueries.

```
Q15:
SELECT fname, sname
FROM actor
WHERE sname LIKE 'F%'
OR sname LIKE 'S%'
AND dob IS NULL
OR dod IS NOT NULL;
```

```
Q16:
SELECT c.fname, c.sname, c.dob
FROM customer c
WHERE NOT EXISTS
  (SELECT *
  FROM rental rt
  WHERE c.cust_id = rt.cust_id
  AND EXISTS
    (SELECT *
    FROM review rv
    WHERE c.cust_id = rv.cust_id)
  );
```

Q15 (see exercise A3 in Appendix B) is similar to the one demonstrated by Smelcer [34]. A query that has both AND and OR operators in a single WHERE clause usually requires nesting the expressions. Errors with subquery nesting were expected in exercises with equal subqueries (B8). For Q16, consider the data demand "list the full names and dates of birth of customers who have never rented a movie but who have given at least one review." Because the subqueries are nested and not equal (i.e., on the same level) in Q16, the query functions like an exclusive OR operator even though that might not be outright evident by reading the query. In other words, the query returns a result table that contains the data of customers who have rented or reviewed at least one movie but have not done both.

4.3.4 LOG-4 Expression Error. We identified three straightforward logical errors related to expressions: missing expression, extraneous expression and expression on incorrect column. We observed missing expressions to be common in all queries and expressions on incorrect columns particularly common in queries involving self-joins. Consider the data demand "list the phone numbers of stores which are in the same city as store #100" for both Q17 and Q18.

Q17:
```
SELECT DISTINCT s1.phone
FROM store s1, store s2
WHERE s1.city = s2.city
AND s2.stono = 100;
```

Q18:
```
SELECT DISTINCT s1.phone
FROM store s1, store s2
WHERE s1.city = s2.city
AND s1.stono = 100
AND s2.stono <>100;
```

Q17 demonstrates a missing expression: the result table also contains the phone number of store #100, which is not desired. Q18 demonstrates an expression on the wrong column: the result table contains only one row representing store #100 if the store is in the same city as some other store. We also identified two more intricate expression errors. The first, extraneous quotes, is demonstrated by Welty [39]. The second error was placing the expression in an incorrect clause, e.g., placing the expression in WHERE clause instead of HAVING clause or vice versa, or placing the expression in an incorrect WHERE clause.

Q19:
```
SELECT sname, SUM(fee)
FROM customer
WHERE fee >= 100
GROUP BY sname;
```

Q20:
```
SELECT sname, SUM(fee)
FROM customer
GROUP BY sname
HAVING SUM(fee) >= 100;
```

Consider Q19 with the data demand "list the sums of fees of customers by surname. Omit surnames with sums below 100 euros" and Q20 with the data demand "list the sums of fees of customers by surname. Calculate only individual fees of 100 euros and above." Both Q19 and Q20 contain a logical error and answer to each other's data demands.

4.3.5 LOG-5 Projection Error. Projection errors are related to column listings in ORDER BY and the main SELECT clauses. We identified four projection errors in the main SELECT clause: missing column, extraneous but non-redundant column, missing DISTINCT and missing AS when it was required to rename a column in the result table.

Regarding ORDER BY clause, we identified four projection errors: fully or partially missing ORDER BY clause, extraneous column in ORDER clause, incorrect column in ORDER BY clause and incorrect ordering of rows. Although an extraneous ORDER BY clause may not affect the result

table, ordering will usually negatively affect performance. Therefore, ordering should not be used if it is not required.

```
Q21:
SELECT stono, city, zip
FROM store
ORDER BY city ASC, zip DESC;
```

```
Q22:
SELECT stono, city
FROM store
ORDER BY stono ASC, city ASC;
```

Q21 demonstrates incorrect ordering when the data demand is to "list the store numbers, cities and zip codes and sort the results in ascending order by city and then in ascending order by zip code." Q22 demonstrates incorrect ordering when the data demand requires that the results are to be sorted using city first and then, within a city, by store number. Q22 achieves the opposite.

4.3.6 LOG-6 Function Error. We identified four aggregate function errors in the student data: incorrect function, incorrect column as function parameter, using DISTINCT as function parameter where not applicable and missing DISTINCT from function parameter.

```
Q23:
SELECT SUM(fee)
FROM customer
WHERE fee >10;
```

```
Q24:
SELECT COUNT(dob)
FROM customer
WHERE fee >10;
```

The data demand for both Q23 and Q24 is to "list the number of customers with a fee over 10 euros", but Q23 instead lists the sum of all customers' fees and demonstrates the use of an incorrect function. Q24 demonstrates the use of an incorrect column as a function parameter. Using date of birth as a function parameter for counting customers with a fee may yield erroneous results since the value of the date of birth can be null.

Our exercise database contained data on movies, actors and actors' participation in different movies. As in real life, the exercise database contained movies in which the same actor acts several roles. Q25 demonstrates a missing DISTINCT when the data demand is to list the names of actors who have acted in at least eight different movies.

```
Q25:
SELECT ar.fname, ar.sname
FROM actor ar
WHERE 7 <
  (SELECT COUNT(ac.movno)
  FROM acts ac
  WHERE ar.actno = ac.actno);
```

```
Q26:
SELECT COUNT(DISTINCT zip)
FROM store;
```

Because the DISTINCT keyword is omitted from the function parameter, the function may count the same movie multiple times. Only one of the exercises required the use of a parameter DISTINCT, and approximately 45% of the 152 students who attempted exercise B12 could not formulate the correct query. Q26 demonstrates using DISTINCT as function parameter where it is not applicable. The data demand for Q26 is to "list the number of stores with a known zip code".

4.4 Complications

Similar to semantic errors, complications are evident without knowledge of the data demand, but unlike semantic errors, complications do not affect the result table. Queries with complications

return the correct result table but could be formulated in a simpler fashion. Such complications can affect the readability of the query and cause performance issues.

Table 6. Complications

ID	Complication	Discussed in
82	unnecessary complication	*[4] (unnumbered)*
83	unnecessary DISTINCT in SELECT clause	*[4] (2)*
84	unnecessary join	*[4] (6)*
85	unused correlation name	*[4] (5)*
86	correlation names are always identical	*[4] (7)*
87	unnecessarily general comparison operator	*[4] (11)*
88	LIKE without wildcards	*[4] (12)*
89	unnecessarily complicated SELECT in EXISTS subquery	*[4] (13)*
90	IN/EXISTS can be replaced by comparison	*[4] (14)*
91	unnecessary aggregate function	*[4] (15)*
92	unnecessary DISTINCT in aggregate function	*[4] (16)*
93	unnecessary argument of COUNT	*[4] (17)*
94	unnecessary GROUP BY in EXISTS subquery	*[4] (18)*
95	GROUP BY with singleton groups	*[4] (19)*
96	GROUP BY with only a single group	*[4] (20)*
97	GROUP BY can be replaced with DISTINCT	*[4] (22)*
98	UNION can be replaced by OR	[4] (23)
99	unnecessary column in ORDER BY clause	*[4] (24)*
100	ORDER BY in subquery	
101	inefficient HAVING	*[4] (25)*
102	inefficient UNION	[4] (26)
103	condition in the subquery can be moved up	*[4] (30)*
104	condition on left table in LEFT OUTER JOIN	[4] (35)
105	OUTER JOIN can be replaced by INNER JOIN	[4] (36)

Most of the complications were already recognized by Brass and Goldberg [4], who proposed four different reasons why a query writer (i.e., in our study, a student) would formulate a query that is unnecessarily complicated. One of the proposed reasons is that the student does not even consider an exemplary query. The other three reasons propose that the student considers an exemplary query but discards it as either erroneous, suboptimal in terms of performance or more difficult to read or maintain compared to his or her unnecessarily complicated query.

Common complications in the student data were unnecessary joins, using DISTINCT when it is not needed, and declaring correlation names that are never used. Another complication also common in the student data was an expression with no impact on the evaluation of the WHERE or the HAVING clause, e.g., *WHERE fee >= 0*, even though the expression is enforced by a CHECK constraint. One complication we did not observe in previous studies was using ORDER BY in a subquery. It is worth noting that, given the nature of complications, we should focus primarily on the errors in teaching, which affect the result table, and secondarily on complications, which affect performance.

Table 7 summarizes the frequencies of all eighteen categories for each of the exercises. As speculated in Section 3.2, certain query concepts appear to invite certain errors. Conversely, some

errors are absent in some exercises. For example, function errors (LOG-6) are not observed before exercise B9, possibly because the data demands in exercises A1-B8 do not imply the use of aggregate functions, or possibly because the students are not yet even aware of the concept of aggregate functions.

Table 7. Error and complication frequencies by category. A number represents the percentage of queries that contained an error or complication belonging to a category, among the queries for a specific exercise.

	A1	A2	A3	B4	B5	B6	B7	B8	B9	B10	B11	B12	B13	C14	C15
SYN-1	0.0	0.0	0.0	0.9	0.5	0.0	0.6	0.0	0.0	0.9	0.0	0.6	0.0	0.3	4.4
SYN-2	25.9	20.4	1.9	19.5	15.5	4.0	10.1	10.7	8.0	11.1	8.7	4.7	2.0	4.9	6.5
SYN-3	22.3	7.8	4.2	2.6	1.0	30.3	4.9	3.3	2.3	9.8	6.7	3.4	0.5	1.0	1.5
SYN-4	0.0	0.0	0.0	0.0	0.0	0.0	1.2	0.9	0.0	0.0	9.3	1.7	0.0	0.0	2.5
SYN-5	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.4	47.3	15.1	0.0	62.2	67.3
SYN-6	28.9	8.7	30.7	26.0	31.6	17.4	17.5	15.3	23.9	37.8	11.3	20.4	22.9	12.8	26.5
SEM-1	6.6	0.0	19.2	0.4	1.5	3.0	6.5	10.7	19.3	0.4	10.0	4.2	1.5	9.0	12.4
SEM-2	0.0	0.0	0.0	3.9	4.9	3.1	0.6	0.5	0.0	1.8	0.0	6.1	0.0	2.8	0.0
SEM-3	0.0	0.0	0.0	1.7	0.0	3.5	8.1	1.9	0.0	19.6	2.0	8.9	13.9	14.9	11.3
SEM-4	0.6	0.0	0.0	0.8	0.8	0.2	0.4	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0
SEM-5	2.4	3.9	3.5	0.0	0.0	0.5	0.0	0.0	0.0	5.8	0.0	0.0	0.0	0.3	0.0
LOG-1	0.0	0.0	3.2	0.4	16.1	0.0	62.2	43.7	2.3	12.4	9.3	17.6	1.1	0.0	0.4
LOG-2	0.0	0.0	0.6	58.0	27.2	7.0	7.1	45.6	0.0	24.4	47.3	23.2	23.9	0.7	6.2
LOG-3	0.0	0.0	47.3	0.0	0.0	2.2	0.0	38.6	0.0	0.0	0.0	0.0	0.0	0.0	0.0
LOG-4	10.2	28.2	32.9	49.8	35.9	35.3	30.2	59.5	18.2	9.3	58.7	21.8	38.3	31.3	40.4
LOG-5	2.4	60.2	5.1	11.3	67.2	62.7	15.6	2.3	36.4	40.4	10.7	6.7	12.9	67.4	60.1
LOG-6	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	18.2	0.0	5.3	56.1	0.0	21.5	13.8
COMP	41.6	76.7	0.6	50.6	10.7	64.2	75.3	28.4	4.5	48.4	89.3	49.4	66.2	24.0	9.5

5 DISCUSSION

In this study, we categorized student errors in SQL, a long-lived and popular database query language, and proposed a DBMS-independent categorization of syntax, semantic and logical errors and complications. In this section, we present considerations regarding our results, propose an operational model for designing SQL exercises for a database course and discuss the limitations of our study.

5.1 Regarding the Results

Based on the analyzed data, our findings showed similar syntax errors reported by Smelcer [34] and Ahadi et al. [1]. We encountered similar syntax errors as Ahadi et al. [1, 2], even though they used a different DBMS than us. In terms of syntax errors, we expected and encountered many errors regarding grouping. As criticized by Date [10] over thirty years ago, grouping rules in SQL can cause somewhat anticipated confusion, and this has not changed considering the analyzed data. While some syntax errors were clear misspellings and did not relate to a student's skill, other syntax errors, such as illegal aggregate function placement or illegal grouping, displayed inadequate knowledge about the query language.

Grandel et al. [15] report in their study a comparison of student errors in Java and Python and conclude that the syntax of the language can reduce both syntax and logical errors. Furthermore,

Stefik and Siebert [35] reported that the syntax of a programming language influences both the perceived and actual difficulty of the task. Given this information, the simple syntax of SQL might encourage students to try solving the given exercises. On the other hand, more complex tasks may be perceived simpler than they actually are, resulting in fallacies such as those demonstrated in Section 4.1.5.

Since no result table is returned, we considered syntax errors to be the easiest for a student to spot and fix because of the error message received. We considered other kinds of errors and complications to be more problematic, e.g., complications can cause additional workload on the system, network or both, and may never be fixed as the system, apart from possible performance issues, works as expected. Semantic errors identified in the student data support Brass and Goldberg's [4] listing, where applicable.

It is worth noting that a recent study by Ahadi et al. [3] investigated errors made by students when learning SQL. Ahadi et al. [3] make no distinction between semantic and logical errors but list similar logical errors we encountered in the student data, namely error IDs 62, 66-68, 70-72 and 74-76 in Table 5. In fact, even though their study does not focus on what errors students make, but more on why certain errors occur, errors listed in their study show clear similarities to our findings. These similarities support our findings and show that the errors students make share common patterns, no matter the teacher, teaching methods or the database domain. The error types listed by Ahadi et al. [3] did not affect our categorization because their study was published after our data analysis was completed.

The previously little-studied class of logical errors displayed patterns that we have encountered in our teaching in the past, such as errors with nesting, misplaced NOT operators, missing expressions and confusions with *does not exist*. Our error findings benefit future SQL research by providing an a unified and DBMS-independent categorization of various errors: future studies may use this categorization to identify and analyze errors without the need to establish their own. Our study also adds to the understanding of what difficulties students face when learning SQL, specifically the difficulties with understanding how the language logically operates.

5.2 Designing SQL Exercises

Our categorization of errors provides teachers with a framework of what kind of errors to expect when teaching SQL in an introductory database course. Specifically, considering the results of the study, logical errors are among the key points that cause difficulties with SQL logic among students. In this section, we propose an operational model for designing SQL exercises based on the fundamental concepts used in our exercises, our error categorization framework and positive experiences in the course.

After designing the exercise database structure, we design each of the query exercises around selected fundamental concepts, e.g., similar to those presented in Table 1, and recognize different errors those exercises may invite. When fundamental concepts for each exercise are recognized and acknowledged, we know what errors we as teachers might expect. As we know what errors to expect, we can mitigate the problems students face with purposeful data. Next, we populate the exercise database tables with appropriate amounts of data. We design the exercise data with the expected errors for all exercises in mind. When we are aware of erroneous result tables and the expected errors they are associated with, we can provide students with appropriate feedback. Concerning Figure 2, a query q_1 will have a correct result table RTc and expected logical errors e_1, e_2 and e_3 with their associated, erroneous result tables RTe_n. For example, in exercise B7, the teacher could identify three likely errors. First, there is the potential for confusion with *does not exist*. If a student erroneously formulates *does not exist* similarly to Q11, the result table will contain movies released in 2000-2009, of which there exists at least one copy that is not a BluRay

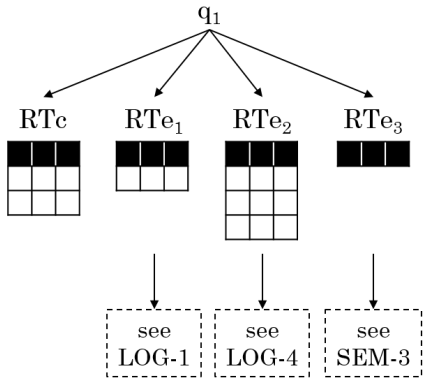


Fig. 2. Query q_1, expected result tables and errors associated with each result table.

(LOG-1). Second, if a student places the year expression into the subquery's WHERE clause instead of the main query's, the result table will contain movies released in 2000-2009, of which there does not exist a copy in BluRay format, and movies which are not released in 2000-2009 (LOG-4). Third, if the student formulates the query as in the example answer for B7 but omits the join, the result table will be empty (SEM-3). Each of these erroneous result tables can be identified, and feedback can be provided accordingly to the student. This feedback can be integrated into e-learning environments by comparing the actual result table with result tables produced by queries with expected errors. Such e-learning environments already exist and could benefit from more accurate feedback concerning errors. For example, SQL-Tutor [28] provided feedback on possible semantic errors in the student's query, and strove to provide more understandable syntax error messages than the DBMS. AsseSQL [30] was used to test the student's query formulation skill, but similar to our e-learning environment, AsseSQL provided no feedback about the correctness of the query, besides presenting the correct result table. Brusilovsky et al. [5] discuss several additional SQL learning environments and tools.

We provide the students with the correct result table for each exercise. If no result table is provided or the correct result table is the same as returned by the queries that contain an expected error, the students might continue to the next exercise even though the query contains an error. Providing the students with the correct result table, however, can be problematic if the expected errors are not recognized, as mentioned in Section 4.1.5.

5.3 Limitations

The first group of limitations of our study is related to the exercises. As stated earlier, the student data provided the means to complement previous error categorization and to create new knowledge on previously unstudied errors, and because the errors encountered are closely related to the exercise design, it may be possible that there exists a key concept that would fit the scope of an introductory-level database course. Even though we taught division operation in the course, it was not included in the exercises. Additionally, even though the fundamental concepts listed in Table 1 were recognized over many years from many different sources and teaching experiences involving diverse target domains, it is possible that some other domains introduce novel, domain-specific fundamental concepts. When analyzing the student data, we made several remarks regarding different errors. In order to identify semantic and logical errors, we needed to inspect more than just the result table and recognize whether or not the query was unnecessarily complicated. Furthermore, a single

query may contain a number of errors, both syntax and semantic [4], in addition to logical errors and complications. Finally, in order to categorize all errors in a query according to our proposed framework, we should not just focus on data demand and the query, but we also need knowledge on the database structure, data types and constraints and the business logic of the target domain.

On the other hand, creating a similar research environment to this study is relatively simple: the database structure and queries are reported in the appendices if the same database domain and queries are warranted. If not, the fundamental concepts and query complexities are listed in Table 1. The environment in which the students complete the exercises is minimally controlled. Students are given the correct result table for each exercise. The error messages from the DBMS will likely have a role in how the student will try to correct a syntactically incorrect query, as discussed by Hristova et al. [18] in the context of programming language compiler errors. Therefore, we suggest using SQLite with default settings in order to more accurately replicate our research environment.

The second group that affects our findings is related to the SQL standard. First, the standard is not an implementation and leaves room for interpretation. Second, the standard is divided into different levels, such as Entry, Intermediate and Full, which serve as instructions on how many features should be implemented. Different products implement the SQL standard for different levels, which may hinder the use of our framework with some DBMSs. Third, the standard contains optional features, some of which overlap with the core features. As demonstrated in Section 4.1.5, some DBMSs have adopted an optional feature over a core feature as their default behavior. All of these points considered, no obvious, single position exists to what a standard-conforming implementation is. As stated by Randolph [31], there is no conformance testing of SQL implementations anymore, and different implementations allow conflicting features. Although the SQL standard should be the reference point for different implementations, the companies behind DBMSs are known to add nonstandard proprietary features to their products' SQL dialects while omitting standard features.

The third group of limitations is related to the research methods and data. For directed content analysis, Shannon and Hsieh [19] discuss that the researchers approach the data with strong bias. Therefore, had we analyzed the data for syntax and semantic errors without examining previous studies beforehand, it is possible that our error categorization might have been different. For conventional content analysis, Shannon and Hsieh [19, p. 1281] argue that it might be challenging for the researchers to develop a "complete understanding of the context, thus failing to identify key categories". Additionally, only the first author coded the errors. However, we have tried to minimize the error margin for logical error categorization by prolonged exposure to the context, as discussed in Section 3.3. Even though our data was collected over one teaching period, the number of queries is relatively high for manual qualitative analysis.

6 CONCLUSIONS AND FUTURE WORK

This study presents a DBMS-independent categorization of SQL errors made by students in an introductory database course. The discovered errors add to the knowledge on what problems students face when learning SQL, specifically problems with the logic of the language.

We suggest applying our categorization framework in practice to proceed from what errors occur to why errors occur, namely, what types of errors have a tendency to stay unresolved, what the reasons are behind these errors and how much the error frequencies affect course performance. When and if we answer these questions in future research, we can begin to identify the students who have problems with SQL queries based on the exercise data, provide additional support for those students and adjust our teaching methods accordingly. Additionally, current research does not discuss the effects of providing versus not providing students with the correct result table.

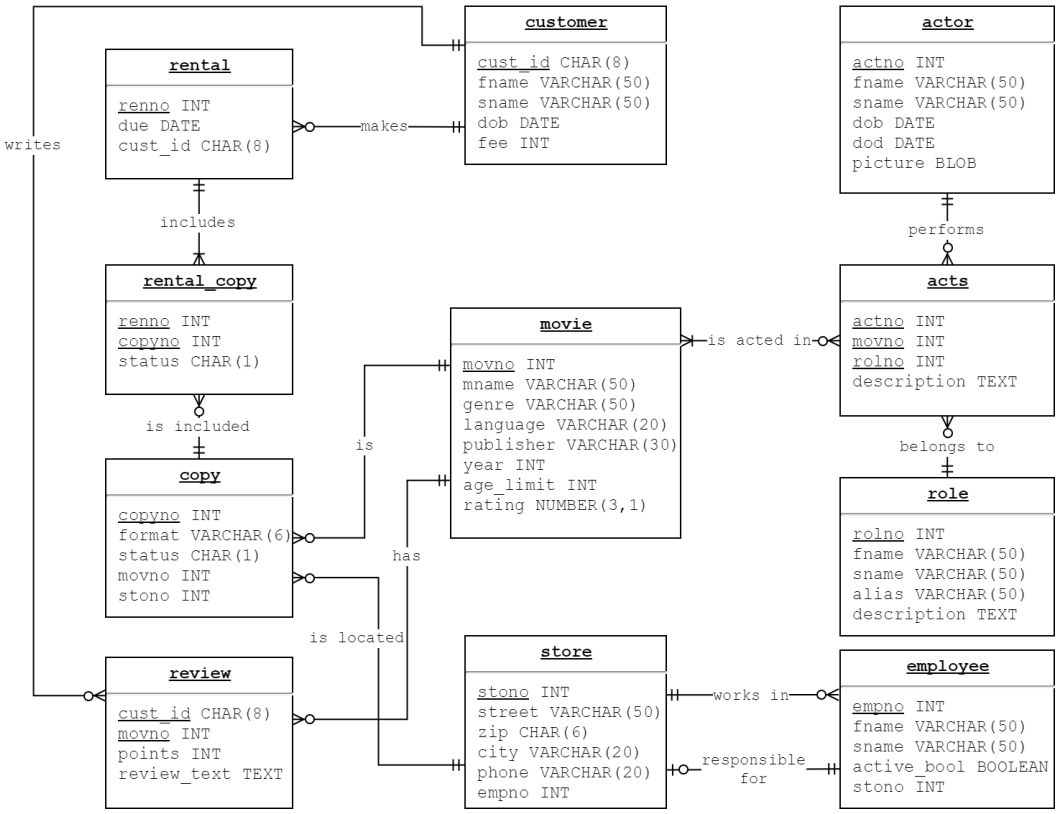
Teaching SQL with a DBMS-independent approach benefits the students in understanding what standard SQL is and what it is not [31]. When identifying different errors, the implementation

used should be secondary, and the SQL standard the primary reference. As more NewSQL systems emerge, it would be interesting to see an experimental research on how vigorously the query languages of those systems conform to the SQL standard.

A DATABASE SCHEMA AND DESCRIPTION

The following database schema diagram and domain description were given to the students. The students could also use the DBMS to acquire more information about the database objects, such as CREATE TABLE statements.

The domain in the SQL exercises is a movie rental service that has stores all over the world. The customers can use a website to search for movies, their favorite actors and the formats (e.g., BluRay) in which the physical copies of the movies can be rented. The customers can write reviews for the movies so that other customers can see whether or not a certain movie is popular. The database also contains information on the company employees that is not visible to customers.



B EXERCISES

The following table lists the 15 exercises used in this study, namely the data demands presented to the students, and example answers formulated by us. The example answers were also provided to the students after the weekly exercise deadlines had passed.

#	Data demand	Example answer
A1	List all information regarding stores in Helsinki and Tampere.	SELECT * FROM store WHERE city IN ('Helsinki', 'Tampere');
A2	List the names, age limits and years of movies that are in English but are not published by Goldeneye BC. Sort the results according to the name of the movie in ascending order.	SELECT mname, age_limit, year FROM movie WHERE language = 'English' AND publisher <>'Goldeneye BC' ORDER BY mname ASC;
A3	List the names, dates of birth and death of actors whose surname starts with an F or an S, and whose date of birth is unknown, or who have a date of death.	SELECT fname, sname, dob, dod FROM actor WHERE (sname LIKE 'F%' OR sname LIKE 'S%') AND (dob IS NULL OR dod IS NOT NULL);
B4	List the city and phone number of the store in which Jaakko Mattila works.	SELECT s.city, s.phone FROM store s, employee e WHERE s.stono = e.stono AND e.fname = 'Jaakko' AND e.sname = 'Mattila';
B5	List the names of actors whose date of death is known and who have acted in at least one movie released after 2010. Sort the results according to surname in descending order.	SELECT a.fname, a.sname FROM actor a INNER JOIN acts ac ON (a.actno = ac.actno) INNER JOIN movie m ON (ac.movno = m.movno) WHERE a.dod IS NOT NULL AND m.year >2010 ORDER BY a.sname DESC;
B6	List the names of actors who have acted a role as himself or herself. Sort the results according to surname, and then according to first name, both in ascending order.	SELECT a.fname, a.sname FROM actor a WHERE EXISTS (SELECT * FROM acts ac WHERE a.actno = ac.actno AND EXISTS (SELECT * FROM role r WHERE ac.rolno = r.rolno AND (r.alias = 'Himself' OR r.alias = 'Herself'))) ORDER BY a.sname ASC, a.fname ASC;
B7	List the movie numbers, names and years of movies that have been released in the first decade of the 2000s, but of which there exists no copy in BluRay format.	SELECT m.movno, m.mname, m.year FROM movie m WHERE m.year BETWEEN 2000 AND 2009 AND NOT EXISTS (SELECT * FROM copy c WHERE m.movno = c.movno AND c.format = 'BluRay');

B8	List the names and dates of birth of customers who have never rented a movie but who have given at least one review.	SELECT c.fname, c.sname, c.dob FROM customer c WHERE NOT EXISTS (SELECT * FROM rental rt WHERE c.cust_id = rt.cust_id) AND EXISTS (SELECT * FROM review rv WHERE c.cust_id = rv.cust_id);
B9	List the number of movies released between the years 1970-2000. Rename the column in the result table descriptively.	SELECT COUNT(*) AS "movies released in 1970-2000" FROM movie WHERE year BETWEEN 1970 AND 2000;
B10	List the names of actors who have acted in the movie Physics 101 and list the names of the roles they have played in that movie. Rename the columns in the result table descriptively.	SELECT a.fname AS "actor's first name", a.sname AS "actor's surname", r.fname AS "character's first name", r.sname AS "character's surname", r.alias AS "character's alias" FROM movie m, actor a, acts ac, role r WHERE m.movno = ac.movno AND ac.rolno = r.rolno AND a.actno = ac.actno AND m.mname = 'Physics 101';
B11	List the name, year and genre of the oldest movie published by Goldeneye BC.	SELECT mname, year, genre FROM movie WHERE publisher = 'Goldeneye BC' AND year = (SELECT MIN(year) FROM movie WHERE publisher = 'Goldeneye BC');
B12	List the actor numbers and full names of actors who have acted in at least five different movies.	SELECT a.actno, a.fname, a.sname FROM actor a WHERE 4 < (SELECT COUNT(DISTINCT ac.movno) FROM acts ac WHERE a.actno = ac.actno);
B13	List the names of customers who have rented exactly the same movie copy that Robert Butler (rbutler1) has rented, whenever.	SELECT c.fname, c.sname FROM customer c, rental r1, rental_copy rc1, rental_copy rc2, rental r2 WHERE c.cust_id = r1.cust_id AND r1.renno = rc1.renno AND rc1.copyno = rc2.copyno AND rc2.renno = r2.renno AND r2.cust_id = 'rbutler1' AND c.cust_id <>'rbutler1';

C14	List the numbers of movie copies located in stores by city and status of the copy. Sort the results by city in ascending order. Make sure that the structure of the result table is as below [example given].	SELECT s.city, c.status, COUNT(c.copyno) AS total FROM store s, copy c WHERE c.stono = s.stono GROUP BY s.city, c.status ORDER BY s.city ASC;
C15	List the numbers of movie copies by movie number and movie name. Disregard movies of which there are less than six copies, regardless of the status of the copy. Sort the results according to the number of the copies in descending order.	SELECT m.movno, m.mname, COUNT(c.movno) AS total FROM movie m, copy c WHERE m.movno = c.movno GROUP BY m.movno, m.mname HAVING COUNT(c.movno) >5 ORDER BY total DESC;

REFERENCES

[1] Alireza Ahadi, Vahid Behbood, Arto Vihavainen, Julia Prior, and Raymond Lister. 2016. 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 (SIGCSE '16)*. ACM Press, New York, New York, USA, 401–406. https://doi.org/10.1145/2839509.2844640

[2] Alireza Ahadi, Julia Prior, Vahid Behbood, and Raymond Lister. 2015. A Quantitative Study of the Relative Difficulty for Novices of Writing Seven Different Types of SQL Queries. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '15)*. ACM Press, New York, New York, USA, 201–206. https://doi.org/10.1145/2729094.2742620

[3] Alireza Ahadi, Julia Prior, Vahid Behbood, and Raymond Lister. 2016. 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 (SIGCSE '16)*. 272–277. https://doi.org/10.1145/2899415.2899464

[4] Stefan Brass and Christian Goldberg. 2005. Semantic errors in SQL queries: A quite complete list. *J. Syst. Softw.* 79, 5 (2005), 630–644. https://doi.org/10.1016/j.jss.2005.06.028

[5] Peter Brusilovsky, Sergey Sosnovsky, Michael V. Uydelson, Danielle H. Lee, Vladimir Zadorozhny, and Xin Zhou. 2010. Learning SQL Programming with Interactive Tools: From Integration to Personalization. *ACM Trans. Comput. Educ.* 9, 4 (2010), 367–376. https://doi.org/10.1145.1656255.1656257

[6] R. B. Buitendijk. 1988. Logical errors in database SQL retrieval queries. *Comput. Sci. Econ. Manag.* 1, 2 (1988), 79–96. https://doi.org/10.1007/BF00427157

[7] Gretchen Irwin Casterella and Leo Vijayasarathy. 2013. An Experimental Investigation of Complexity in Database Query Formulation Tasks. *J. Inf. Syst. Educ.* 24, 3 (2013), 211–221. http://jise.org/Volume24/24-3/pdf/Vol24-3pg211.pdf

[8] Ugur Cetintemel, Nesime Tatbul, Kristin Tufte, Hao Wang, Stanley Zdonik, Jiang Du, Tim Kraska, Samuel Madden, David Maier, John Meehan, Andrew Pavlo, Michael Stonebraker, and Erik Sutherland. 2014. S-Store: a streaming NewSQL system for big velocity applications. *Proceedings of the VLDB Endowment* 7, 13 (2014), 1633–1636. https://doi.org/10.14778/2733004.2733048

[9] Anthony Cleve, Maxime Gobert, Loup Meurice, Jerome Maes, and Jens Weber. 2015. Understanding database schema evolution: A case study. *Sci. Comput. Program.* 97, P1 (2015), 113–121. https://doi.org/10.1016/j.scico.2013.11.025

[10] Christopher J. Date. 1983. Critique of the SQL database language. *SIGMOD Rec.* 14, 3 (Nov 1983). https://doi.org/10.1145/984549.984551

[11] Alireza Ebrahimi. 1994. Novice programmer errors: language constructs and plan composition. *Int. J. Hum. Comput. Stud.* 41 (1994), 457–480. https://doi.org/10.1006/ijhc.1994.1069

[12] Ramez Elmasri and Shamkant B. Navathe. 2016. *Fundamentals of Database Systems (7th. ed.).* Pearson.

[13] Sally Fincher, Josh Tenenberg, and Anthony Robins. 2011. Research Design : Necessary Bricolage. *Comput. Sci. Educ.* (2011), 27–32. https://doi.org/10.1145/2016911.2016919

[14] Michael M. Gorman. 2002. Is SQL A Real Standard Anymore. *The Data Administration Newsletter* (2002), 2–4.

[15] Linda Grandell, Mia Peltomäki, Ralph Johan Back, and Tapio Salakoski. 2006. Why complicate things? Introducing programming in high school using Python. *Conf. Res. Pract. Inf. Technol. Ser.* 52 (2006), 71–80.

[16] Sharon L. Greene, Susan J. Devlin, Philip E. Cannata, and Louis M. Gomez. 1990. No IFs, ANDs, or ORs: A study of database querying. *Int. J. Man. Mach. Stud.* 32, 3 (Mar 1990), 303–326. https://doi.org/10.1016/S0020-7373(08)80005-3

[17] Eric Gregoire, Richard Ostrowski, Bertrand Mazure, and Lahkdar Sais. 2005. Automatic extraction of functional dependencies. *Theory Appl. Satisf. Test.* 3542 (2005), 122–132.

[18] Maria Hristova, Ananya Misra, Megan Rutter, and Rebecca Mercuri. 2003. Identifying and correcting Java programming errors for introductory computer science students. *ACM SIGCSE Bulletin* 35, 1 (Jan 2003), 153. https://doi.org/10.1145/792548.611956

[19] Hsiu-Fang Hsieh and Sarah E Shannon. 2005. Three Approaches to Qualitative Content Analysis. *Qual. Health Res.* 15, 9 (2005), 1277–1288. https://doi.org/10.1177/1049732305276687

[20] Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. 1999. Tane: An Efficient Algorithm for Discovering Functional and Approximate Dependencies. *Comput. J.* 42, 2 (Feb 1999), 100–111. https://doi.org/10.1093/comjnl/42.2.100

[21] Ihab F. Ilyas, Volker Markl, Peter Haas, Paul Brown, and Ashraf Aboulnaga. 2004. CORDS: Automatic Discovery of Correlations and Soft Functional Dependencies. In *Proceedings of the 2004 ACM International Conference on Management of Data (SIGMOD '04)*. ACM Press, New York, New York, USA, 647. https://doi.org/10.1145/1007568.1007641

[22] ISO/IEC. 2003. ISO/IEC 9075-2:2003, "SQL - Part 2: Foundation". (2003).

[23] Eranki L.N. Kiran and Kannan M. Moudgalya. 2015. Evaluation of Programming Competency Using Student Error Patterns. In *2015 International Conference on Learning and Teaching in Computing and Engineering*. IEEE, 34–41. https://doi.org/10.1109/LaTiCE.2015.16

[24] A.J. Ko and B.A. Myers. 2003. Development and evaluation of a model of programming errors. In *Proceedings of the IEEE Symposium on Human Centric Computing Languages and Environments*. IEEE, 7–14. https://doi.org/10.1109/HCC.2003.1260196

[25] Charles R. Litecky and Gordon B. Davis. 1976. A study of errors, error-proneness, and error diagnosis in Cobol. *Commun. ACM* 19, 1 (Jan 1976), 33–38. https://doi.org/10.1145/359970.359991

[26] Victor M. Matos and Rebecca Grasser. 2002. Teaching Tip A Simpler (and Better) SQL Approach to Relational Division. *J. Inf. Syst. Educ.* 13, 2 (2002), 85–88. http://jise.org/Volume13/Pdf/085.pdf

[27] Jim Melton. 2002. *SQL:1999: Understanding Relational Language Components*. Morgan Kaufman.

[28] Antonija Mitrovic. 1998. Learning SQL with a computerized tutor. *ACM SIGCSE Bulletin* 30, 1 (1998), 307–311. https://doi.org/10.1145/274790.274318

[29] Thomas H. Park, Brian Dorn, and Andrea Forte. 2015. An Analysis of HTML and CSS Syntax Errors in a Web Development Course. *ACM Trans. Comput. Educ.* 15, 1 (2015), 4:1–4:21. https://doi.org/10.1145/2700514

[30] Julia Prior. 2003. Online Assessment of SQL Query Formulation Skills. *Proceedings of the Fifth Australasian Computing Education Conference* 20 (2003), 247–256. http://dl.acm.org/citation.cfm?id=858403.858433

[31] Gary B Randolph. 2003. The Forest and the Trees: Using Oracle and SQL Server Together to Teach ANSI-Standard SQL. *Design* (2003), 234–236.

[32] Julian Rith, Philipp S. Lehmayr, and Klaus Meyer-Wegener. 2014. Speaking in tongues: SQL access to NoSQL systems. *Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC '14)*, 855–857. https://doi.org/10.1145/2554850.2555099

[33] Yasin N Silva, Isadora Almeida, and Michell Queiroz. 2016. SQL: From Traditional Databases to Big Data. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16)*. ACM Press, New York, New York, USA, 413–418. https://doi.org/10.1145/2839509.2844560

[34] John B Smelcer. 1995. User errors in database query composition. *Int. J. Hum. Comput. Stud.* 42, 4 (Apr 1995), 353–381. https://doi.org/10.1006/ijhc.1995.1017

[35] Andreas Stefik and Susanna Siebert. 2013. An Empirical Investigation into Programming Language Syntax. *ACM Trans. Comput. Educ.* 13, 4 (2013), 1–40. https://doi.org/10.1145/2534973

[36] Josh Tenenberg and Robert McCartney. 2010. Why Discipline Matters in Computing Education Scholarship. *ACM Trans. Comput. Educ.* 9, 4 (2010), 1–7. https://doi.org/10.1145/1656255.1656256

[37] Heikki Topi, Kate M. Kaiser, Janice C. Sipior, Joseph S. Valacich, J. F. Nunamaker, Jr., G. J. de Vreede, and Ryan Wright. 2010. *Curriculum Guidelines for Undergraduate Degree Programs in Information Systems*. Technical Report. New York, NY, USA.

[38] Geoff Walsham. 2006. Doing interpretive research. *Eur. J. Inf. Syst.* 15, 3 (2006), 320–330. https://doi.org/10.1057/palgrave.ejis.3000589

[39] Charles Welty. 1985. Correcting user errors in SQL. *Int. J. Man. Mach. Stud.* 22, 4 (1985), 463–477. https://doi.org/10.1016/S0020-7373(85)80051-1

[40] Charles Welty and David Stemple. 1981. Human factors comparison of a procedural and a nonprocedural query language. *ACM Trans. Database Syst.* 6, 4 (1981), 626–649. https://doi.org/10.1145/319628.319656

[41] Li-Yan Yuan, Lengdong Wu, Jia-Huai You, and Yan Chi. [n. d.]. A Demonstration of Rubato DB: A Highly Scalable NewSQL Database System for OLTP and Big Data Applications. *Proceedings of the 2015 ACM International Conference on Management of Data (SIGMOD '15)* ([n. d.]), 907–912. https://doi.org/10.1145/2723372.2735380

Received September 2016; revised October 2017; accepted December 2017