Query Execution Plans and Semantic Errors: Usability and Educational Opportunities

Toni Taipalus
toni.taipalus@jyu.fi
University of Jyväskylä
Finland

ABSTRACT

Syntax errors are typically separated from semantic errors in query formulation, the former being detected by the database management system (DBMS), and the latter seemingly not. On the other hand, query execution plans are typically utilized in query optimization, and not interconnected with syntax errors, as a syntactically invalid query produces no execution plan. In this study, we show and argue for breaking the confound between execution plans and error messages for better query formulation usability and education. We show how several popular DBMSs detect semantic errors and complications in queries, yet often do not inform the user of such problems. This study is a demonstration of how decades old technology could be used more effectively in novel contexts of usability and software engineering education with little effort by showing query writers not merely syntax errors, but also semantic errors and complications detected by DBMSs.

CCS CONCEPTS

• **Information systems** → *Information retrieval query processing*; • **Human-centered computing** → **Human computer interaction (HCI)**.

KEYWORDS

query execution plan, error, usability, database management system, SQL

ACM Reference Format:

Toni Taipalus. 2023. Query Execution Plans and Semantic Errors: Usability and Educational Opportunities. In *Extended Abstracts of the 2023 CHI Conference on Human Factors in Computing Systems (CHI EA '23), April 23–28, 2023, Hamburg, Germany*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3544549.3585794

1 INTRODUCTION

Database management systems (DBMS) are a multi-billion industry, where some of the industry leaders have been around for over four decades. The DBMS industry has only recently begun to address usability concerns to gain a marketing edge or cater for the needs of ubiquitous computing. These efforts have been realized in more effortless installation, configuration, and load management, yet usability concerns have not been addressed in query language compilers [15]. Although there has been a myriad of scientific efforts in facilitating the development of programming language usability through studying compiler error messages [3], query languages have remained in the sidelines in this regard. In fact, some DBMSs use largely the same error messages now as in the 1990s [15].

When a Structured Query Language (SQL) query is being executed by a DBMS, a DBMS component called the query optimizer formulates one or several query execution plans [20]. An execution plan consists of lower level operations, such as choices concerning the implementation of joins, utilization of indices, and reduction of arithmetic [5, 9]. These operations are consequently used to retrieve the dataset the user requires. If the DBMS deems that the query is not syntactically valid, an error message is returned instead, and no execution plan is formulated. Intuitively, syntax error messages, albeit often lacking in terms of usability, have been shown to facilitate error discovery and successful query formulation [16]. As querying a relational database with SQL is typically text-based activity, in this study by *usability* we refer to the communication of errors via textual prompts, even though *usability* is a much larger concept.

Query execution plans are a treasure trove to increased usability and potential support for effective query formulation. Namely, DBMSs identify certain semantic errors and complications in queries, but since these problems are often syntactically valid SQL, the DBMS does not directly inform the (human) query writer about said problems. For example, the query `SELECT * FROM t WHERE c > 0 AND c < 0;` is syntactically valid SQL and thus produces no errors, but it is obvious that such a query will always produce an empty result table (as the value of `c` cannot be both less than and greater than zero). Rather, these problems are often hidden in query execution plans, which in turn are often both relatively difficult to read [8, 21], as well as something a query writing novice is seldom even aware of, despite the fact that a novice arguably needs more support when compared to a professional. Therefore, it is arguably educationally counter-productive that a query plan is only shown if the user explicitly requests one, even though the query plan contains information on detected problems in query formulation.

In this study, we show how PostgreSQL, Oracle Database and SQL Server identify and handle semantic errors and complications, and argue why and how the identification of a semantic error or a complication should also be communicated to the user writing the query. We also describe a software tool which utilizes query execution plans to communicate problems in queries to the end-user in a more readable form and without the need for the user to consult query execution plans.

CHI EA '23, April 23–28, 2023, Hamburg, Germany

ACM ISBN 978-1-4503-9422-2/23/04.
https://doi.org/10.1145/3544549.3585794

2 RELATED WORK

Previous, tangential works have focused on either errors in query formulation, or query optimization through interpreting query execution plans. As query optimization is recognized as one of the more complex issues in database system research [2, 8], it is not surprising that several tools have been developed to facilitate the interpretation of query execution plans [6, 10, 19, 21]. Furthermore, the errors in query formulation have been studied, but these two lines of research and practice are yet to converge.

Query formulation errors have been divided into syntax, logical, and semantic errors, as well as complications [4, 18]. In short, syntax errors are caused by statements that violate a DBMS's interpretation of the SQL standard, halting the query execution and resulting in an error message instead of a result table. Queries with logical errors are syntactically valid, yet retrieve a result table that is incorrect for a particular data demand, e.g., the query `SELECT * FROM customers;` is logically incorrect if the data demand is something other than "find all information on all customers". In contrast, statements with semantic errors are "always incorrect" [4], regardless of the data demand. Such statements may, for example, always return all data, or always return no data, regardless of the database data, often due to inconsistent or tautological expressions. Finally, complications do not affect the result table, but cause a query to be more complex than the data demand requires. Complications are not concerned with subjective aspects such as code style, but rather complications such as unnecessary joins or unnecessary ordering.

A previous study [4] reported over 40 different semantic errors and complications, and a subsequent study [18] complemented these errors with several new ones. Using these categorizations, several educational studies have shown that both semantic errors and complications are prominent in query formulation [11–13], and particularly difficult to fix when compared to syntax errors [14]. However, previous studies have simply listed and studied semantic errors without connecting the dots between query execution plans and semantic errors, and how these two aspects could be used in facilitating successful query formulation.

Even though previous studies have demonstrated that DBMS error messages often disregard common usability guidelines [15, 16], it seems reasonable to argue that the presence of *any* error message is a clear indication that a query should be fixed. Therefore, it seems intuitive that if a DBMS can recognize an error, this should also be communicated to the user executing the query. However, this is not always the case, as we will show in this study.

3 EVALUATION

3.1 Test Suite

We demonstrate how three popular relational DBMSs (PostgreSQL 9.6, SQL Server 2019 Developer and Oracle Database 19c) handle SQL queries with six different semantic errors or complications [4, 18] using a simple database (Fig. 1) and simple `SELECT` statements. For the evaluation, we inserted 250 customers and 500 orders into the tables. The data were generated with Mockaroo. No secondary indices were created. Among the dozens of semantic errors and complications presented in prior scientific literature, we selected six errors for this preliminary evaluation. This selection of six errors among the dozens on potential candidates was dictated by both brevity as well as selecting errors of different nature to demonstrate the different nature of different DBMSs. We demonstrate the test suite queries and their execution in different DBMSs in the next section.

```
1  CREATE TABLE customers (
2    customer_id INT
3  , fname       VARCHAR(50) NOT NULL
4  , sname       VARCHAR(50) NOT NULL
5  , type        CHAR(1)     NOT NULL
6    CHECK (type IN ('C', 'B')) -- C = private, B = business
7  , PRIMARY KEY (customer_id)
8  );
9
10 CREATE TABLE orders (
11   order_id        INT
12 , order_total_eur DECIMAL(6,2) NOT NULL
13 , customer_id     INT          NOT NULL
14 , PRIMARY KEY (order_id)
15 , FOREIGN KEY (customer_id)
16   REFERENCES customers (customer_id)
17 );
```

Figure 1: Test suite database

3.2 Results

The query execution plans were obtained using `EXPLAIN ANALYZE` in PostgreSQL, `SET SHOWPLAN_TEXT ON` in SQL Server, and by querying the `V$SQL` view in Oracle Database. Table 1 summarizes how the selected DBMSs handle the selected semantic errors and complications. An empty circle represents that the DBMS executes the query and returns a (sometimes empty) dataset without any other output. A half circle means that the DBMS returns a dataset (again, sometimes empty) without any other output, but executes an alternative version of the query. We explain this in more detail in the following subsections. Finally, a full circle means that the semantic error halts the query execution similarly to a syntax error. In this regard, some DBMSs consider some semantic errors or complications as violations of syntax.

3.2.1 Implied expression. An implied expression is something that is already enforced (or alternatively, implied against) by the table, e.g., via a primary or foreign key, or by a `CHECK` constraint. For example, the expression in Fig. 2a, line 3, is already enforced by the test suite table definition described in Fig. 1, lines 5-6. Therefore, the result set is empty regardless of the data, making this a semantic error (cf. e.g., [4, 18]). According to the execution plans, SQL Server executes the query, but both PostgreSQL's (Fig. 3a, line 1: *rows=0*) and Oracle Database's (Fig. 3b, line 12: *filter*(`NULL IS NOT NULL`)) execution plans show that the expression was not evaluated against database data, but rather skipped completely.

3.2.2 Tautological expression. A tautological expression – such as `100 = 100` in Fig. 2b – could simply be replaced with `TRUE`. In the same figure, bound to the logical operator `OR`, the whole `WHERE` clause could be replaced with `TRUE`. Naturally, such expressions are unnecessary complications to anyone reading the query, as well as potential problems in query execution if the DBMS cannot identify the tautology. As can be seen in all the execution plans in Fig. 4a, Fig. 4b and Fig. 4c, none of the DBMSs evaluate either of the expressions, but simply perform a sequential scan on the entire table.

Table 1: How different DBMSs address SQL queries with different semantic errors or complications; ○ = the query is executed, ◐ = the query is seemingly executed, ● = the query execution halts to an error

Semantic error or complication	PostgreSQL	SQL Server	Oracle Database
Implied expression	◐	○	◐
Tautological expression	◐	◐	◐
Inconsistent expression	◐	○	◐
ORDER BY in a subquery	◐	●	●
IN/EXISTS can be replaced by comparison	○	○	○
Join on incorrect column (matches impossible)	●	○	●

```sql
SELECT customer_id, fname, sname
FROM customers
WHERE type = 'A';
```

(a) Query with an implied expression

```sql
SELECT *
FROM customers
WHERE type = 'C' OR 100 = 100;
```

(b) Query with a tautological expression

Figure 2: Queries with semantic errors

```
Seq Scan on customers (cost=0.00..5.12 rows=1 width=18) (
      actual time=0.049..0.049 rows=0 loops=1)
  Filter: (type = 'A'::bpchar)
  Rows Removed by Filter: 250
 Planning Time: 0.125 ms
 Execution Time: 0.057 ms
(5 rows)
```

(a) PostgreSQL query execution plan

```
---------------------------------------------------------
| Id | Operation          | Name      | E-Rows| Cost (%CPU)|
---------------------------------------------------------
|  0 | SELECT STATEMENT   |           |       | 1 (100)    |
|* 1 |  FILTER            |           |       |            |
|* 2 |   TABLE ACCESS FULL| CUSTOMERS| 1      | 3 (0)      |
---------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

  1 - filter(NULL IS NOT NULL)
  2 - filter("TYPE"='A')
```

(b) Oracle query execution plan

Figure 3: Query execution plans produced by the query in Fig. 2a

```
Seq Scan on customers (cost=0.00..4.50 rows=250 width=20) (
      actual time=0.021..0.047 rows=250 loops=1)
Planning Time: 0.073 ms
Execution Time: 0.057 ms
(3 rows)
```

(a) PostgreSQL query execution plan

```
----------------------------------------------------------
| Id | Operation          | Name      | E-Rows | Cost (%CPU)|
----------------------------------------------------------
|  0 | SELECT STATEMENT   |           |        | 3 (100)    |
|  1 |  TABLE ACCESS FULL|CUSTOMERS | 1       | 3 (0)      |
----------------------------------------------------------
```

(b) Oracle query execution plan

```
|--Clustered Index Scan(OBJECT:([TestDB].[dbo].[customers].[
      PK__customer__CD65CB859E0B7BB4]))
(1 rows affected)
```

(c) SQL Server query execution plan

Figure 4: Query execution plans produced by the query in Fig. 2b

```sql
SELECT *
FROM orders
WHERE order_total_eur = 0 AND order_total_eur = 100;
```

(a) Query with an inconsistent expression

```sql
SELECT *
FROM customers c
WHERE EXISTS
 (SELECT *
 FROM orders o
 WHERE c.customer_id = o.customer_id
 ORDER BY o.customer_id);
```

(b) Query with an `ORDER BY` clause in a subquery

Figure 5: Queries with semantic errors

3.2.3 Inconsistent expression. An inconsistent expression is an expression or a set of expressions that could be reduced to `FALSE`, as the two expressions in Fig. 6b connected with the logical operator `AND` can never evaluate to `TRUE`. Both PostgreSQL and Oracle Database show in their query execution plans that while the query is run, the expressions are not evaluated against database data. PostgreSQL (Fig. 6a, lines 2-3) reads *One-Time Filter: false* and *never executed*, describing that PostgreSQL identifies the `WHERE` clause as inconsistent, and the result set empty regardless of the data. Oracle Database (Fig. 6b, line 12) executes the query in a similar fashion.

3.2.4 `ORDER BY` in a subquery. An `ORDER BY` clause in a subquery (Fig. 5b) is considered an unnecessary complication. While such clause should not affect the results of a query, and is syntactically valid in PostgreSQL, PostgreSQL merely performs a grouping operation rather than sorting (Fig. 7a, line 7), and executes the query. In

```
Result (cost=0.00..9.25 rows=1 width=14) (actual time
      =0.001..0.001 rows=0 loops=1)
 One-Time Filter: false
 -> Seq Scan on orders (cost=0.00..9.25 rows=1 width=14) (
      never executed)
    Filter: (order_total_eur = '0'::numeric)
Planning Time: 0.106 ms
Execution Time: 0.008 ms
(6 rows)
```

(a) PostgreSQL query execution plan

```
---------------------------------------------------------
| Id | Operation          | Name   | E-Rows | Cost (%CPU)|
---------------------------------------------------------
| 0  | SELECT STATEMENT  |        |        | 1 (100)    |
|* 1 | FILTER            |        |        |            |
|* 2 |  TABLE ACCESS FULL| ORDERS | 1      | 3 (0)      |
---------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

  1 - filter(NULL IS NOT NULL)
  2 - filter("ORDER_TOTAL_EUR"=0)
```

(b) Oracle query execution plan

Figure 6: Query execution plans produced by the query in Fig. 5a

```
Hash Join (cost=14.22..21.84 rows=221 width=20) (actual time
      =0.274..0.378 rows=221 loops=1)
 Hash Cond: (c.customer_id = o.customer_id)
 -> Seq Scan on customers c (cost=0.00..4.50 rows=250 width
      =20) (actual time=0.004..0.043 rows=250 loops=1)
 -> Hash (cost=11.46..11.46 rows=221 width=4) (actual time
      =0.267..0.267 rows=221 loops=1)
    Buckets: 1024 Batches: 1 Memory Usage: 16kB
    -> HashAggregate (cost=9.25..11.46 rows=221 width=4) (
      actual time=0.233..0.248 rows=221 loops=1)
       Group Key: o.customer_id
       Batches: 1 Memory Usage: 48kB
       -> Seq Scan on orders o (cost=0.00..8.00 rows=500 width
      =4) (actual time=0.004..0.051 rows=500 loops=1)
Planning Time: 0.182 ms
Execution Time: 0.411 ms
(11 rows)
```

(a) PostgreSQL query execution plan

```
ORA-00907: missing right parenthesis
```

(b) Oracle error message

```
Msg 1033, Level 15, State 1, Server testserver, Line 1
The ORDER BY clause is invalid in views, inline functions,
      derived tables, subqueries, and common table expressions
      , unless TOP, OFFSET or FOR XML is also specified.
```

(c) SQL Server error message

Figure 7: Query execution plans and error messages produced by the query in Fig. 5b

contrast, SQL Server (Fig. 7c) and Oracle Database (Fig. 7b) return an error message and refuse to execute the query. SQL Server succeeds in communicating what causes the query execution to halt, but Oracle Database returns a seemingly detached error message.

3.2.5 `IN/EXISTS` can be replaced by comparison. Sometimes, a subquery may be reduced to a simple expression. Such a complication

```
SELECT *
FROM customers c1
WHERE EXISTS
 (SELECT *
 FROM customers c2
 WHERE c2.fname LIKE 'A%'
 AND c1.customer_id = c2.customer_id);
```

(a) Query in which a subquery could be replaced with a comparison

```
SELECT *
FROM customers c
JOIN orders o ON (c.fname = o.order_total_eur);
```

(b) Query with a join on incorrect column with impossible matches

Figure 8: Queries with semantic errors

```
ERROR: operator does not exist: character varying = numeric
LINE 4: ON (c.fname = o.order_total_eur);
                    ^
HINT: No operator matches the given name and argument types.
      You might need to add explicit type casts.
```

(a) PostgreSQL error message

```
ORA-01722: invalid number
```

(b) Oracle error message

Figure 9: Query execution plans produced by the query in Fig. 8b

is demonstrated in Fig. 8a, and the subquery's expression with `LIKE` could be moved to the upper level query while dropping the subquery altogether. Regardless, all three DBMSs performed two sequential scans in total, one for table *c1* and one for table *c2*.

3.2.6 Join on incorrect column (matches impossible). Lastly, joining tables using columns with different data types often results in a situation where none of the rows satisfy the join condition, which in turn causes an empty result set. Naturally, the data types in question play a crucial role, and comparing a column with decimal numbers to a column with integers potentially yields more results than comparing integers to character strings, as some DBMSs implicitly perform type conversions. For this semantic error, we constructed a join using columns in which matches are very likely impossible (Fig. 8b). Again, while the error categorization [18] does not consider this a syntax error, neither PostgreSQL (Fig. 9a) nor Oracle Database (Fig. 9b) execute the query, and return syntax errors. SQL Server, however, executes the query.

4 DISCUSSION AND A SOLUTION PROPOSAL

Previous studies have shown that novices are prone to writing queries with semantic errors and complications [18], and that these errors are relatively easy to miss and be left unfixed, possibly due to the DBMS not informing the user of these problems, as opposed to syntax errors, which produce an error message [17]. Furthermore, it has been shown that as the complexity of the database increases, the number of unfixed complications in queries also increase [13]. Therefore, it seemed reasonable to explore possibilities on how the user could be informed of possible problems in queries. For brevity, we selected three DBMSs and six semantic errors or complications to evaluate in this work, with the intention of demonstrating that DBMSs are able to identify certain problems in queries without communicating these problems to the user.

In summary, the results of this study are presented in Table 1, where white circles represent query formulation problems that are not identified by the DBMS, and therefore cannot be communicated to query writers as of yet. Half circles represent the untapped opportunities for usability considerations and education, as these problems are detected by the DBMS, yet not communicated to the query writers. Finally, black circles represent problems that are detected and communicated, yet some of these communications could be improved in terms of readability (cf. e.g., [7]), as query execution plans are not targeted for novices, and not necessarily intended for query formulation as much as for query optimization.

Given the fact that several DBMSs can detect several problems without communicating them to the user, implementing the propagation of information should not be an arduous task. Execution plan operations such as *One-Time Filter: false* in PostgreSQL, *Constant scan* in SQL Server, or `NULL IS NOT NULL` in Oracle Database would simply need to be rephrased as more readable, and shown to the user. In the case that professional users deem such warnings distracting, or that the generation of warnings is computationally expensive [1] in production environments, DBMS vendors could give users the option to disable such warnings. All in all, it has been argued that ease-of-use of DBMSs in educational contexts benefits both novices and DBMS vendors who can provide experts systems that are relatively easy to learn [15]. Additionally, fixing complications in queries also benefits software industry by making queries more readable and computationally faster to execute, in the case the DBMS does not identify said complications.

To show that such problems in queries could be communicated to the end-user with relative ease, we are currently developing a wrapper for PostgreSQL called *pg4n*, to be available as a free, open-source project. The wrapper analyzes syntactically valid queries and provides hints (Fig. 10) to the end-user of detected problems. The wrapper is text-based to provide cross-application compatibility to different development and learning environments. At the time of writing, the wrapper identifies approximately a dozen common problems in queries.

```
=>
SELECT c.customer_id
FROM customers c
WHERE (
 sname LIKE 'S%'
 AND EXISTS
  (SELECT *
  FROM orders o
  WHERE c.customer_id = o.customer_id
  AND o.order_total_eur > 5000)
)
AND (
 sname LIKE 'T%'
 AND EXISTS
  (SELECT *
  FROM orders o
  WHERE c.customer_id = o.customer_id
  AND o.order_total_eur < 10000)
);
customer_id
-------------
(0 rows)

Hint: you have written a query that will always return an
      empty result table. Please check your expressions.
      Perhaps you have used the logical operator AND instead
      of OR somewhere?
```

Figure 10: A simple example of our wrapper for PostgreSQL which reads the query execution plans of syntactically valid queries, and communicates found problems to the end-user in a more user-friendly wording and without the need to consult query execution plans; the figure consists of an SQL query with a semantic error, an empty result table, and a hint section added by the wrapper

5 CONCLUSION AND FUTURE WORK

In this study, we showed how different relational DBMSs address semantic errors and complications in queries, and proposed a solution for communicating these problems to the end-user in an user-friendly fashion. This work could be extended in several ways. As explained in Section 2, previous studies [4, 18] have identified several dozen semantic errors and complications that are not demonstrated in this study. Additionally, a scientific evaluation of the potential benefits of informing the user of semantic errors or complications in queries is warranted to justify the additional cognitive load to the user presented by such warnings. Furthermore, what constitutes a better SQL error message remains a scientifically unexplored avenue that needs to be investigated before the work can continue, and we plan to test the extension with a simple wrapper first, hoping to acquire knowledge whether such information benefits novice query writers by steering them away from semantic errors and complications. If such information is indeed useful, we plan to develop the extension to account for the future work suggested in the previous points. In summary, we believe that by utilizing already implemented semantic error discovery, several DBMSs would achieve increased usability that would facilitate query formulation in both education and industry.

REFERENCES

[1] Andrei Alexandrescu. 1999. Better template error messages. *C/C++ Users Journal* 17 (03 1999), 37–47.

[2] Brett Allenstein, Andrew Yost, Paul Wagner, and Joline Morrison. 2008. A Query Simulation System to Illustrate Database Query Execution. *SIGCSE Bull.* 40, 1 (2008), 493–497. https://doi.org/10.1145/1352322.1352301

[3] Brett A. Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J. Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, Janice L. Pearce, and James Prather. 2019. Compiler Error Messages Considered Unhelpful. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*. ACM. https://doi.org/10.1145/3344429.3372508

[4] Stefan Brass and Christian Goldberg. 2006. Semantic errors in SQL queries: A quite complete list. *Journal of Systems and Software* 79, 5 (2006), 630–644. https://doi.org/10.1016/j.jss.2005.06.028

[5] Surajit Chaudhuri. 1998. An Overview of Query Optimization in Relational Systems. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Seattle, Washington, USA) *(PODS '98)*. Association for Computing Machinery, New York, NY, USA, 34–43. https://doi.org/10.1145/275487.275492

[6] Peng Chen, Hui Li, Sourav S. Bhowmick, Shafiq R. Joty, and Weiguo Wang. 2022. LANTERN: Boredom-Conscious Natural Language Description Generation of Query Execution Plans for Database Education. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 2413–2416. https://doi.org/10.1145/3514221.3520165

[7] Paul Denny, Andrew Luxton-Reilly, Ewan Tempero, and Jacob Hendrickx. 2011. Understanding the syntax barrier for novices. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science education - ITiCSE '11*. ACM Press. https://doi.org/10.1145/1999747.1999807

[8] Mrunal Gawade and Martin Kersten. 2012. Stethoscope: A Platform for Interactive Visual Analysis of Query Execution Plans. *Proceedings of the VLDB Endowment* 5, 12 (2012), 1926–1929. https://doi.org/10.14778/2367502.2367539

[9] Joseph M. Hellerstein, Michael Stonebraker, and James Hamilton. 2007. Architecture of a Database System. *Foundations and Trends in Databases* 1, 2 (2007), 141–259. https://doi.org/10.1561/1900000002

[10] Siyuan Liu, Sourav S. Bhowmick, Wanlu Zhang, Shu Wang, Wanyi Huang, and Shafiq Joty. 2019. NEURON: Query Execution Plan Meets Natural Language Processing For Augmenting DB Education. In *Proceedings of the 2019 International Conference on Management of Data*. ACM, Amsterdam Netherlands, 1953–1956. https://doi.org/10.1145/3299869.3320213

[11] Daphne Miedema, Efthimia Aivaloglou, and George Fletcher. 2021. *Identifying SQL Misconceptions of Novices: Findings from a Think-Aloud Study.* Association for Computing Machinery, New York, NY, USA, 355–367. https://doi.org/10.1145/3446871.3469759

[12] Daphne Miedema, George Fletcher, and Efthimia Aivaloglou. 2022. Expert Perspectives on Student Errors in SQL. *ACM Transactions on Computing Education* (2022). https://doi.org/10.1145/3551392

[13] Toni Taipalus. 2020. The effects of database complexity on SQL query formulation. *Journal of Systems and Software* 165 (2020), 110576. https://doi.org/10.1016/j.jss.2020.110576

[14] Toni Taipalus. 2020. Explaining Causes Behind SQL Query Formulation Errors. In *2020 IEEE Frontiers in Education Conference (FIE)*. 1–9. https://doi.org/10.1109/FIE44824.2020.9274114

[15] Toni Taipalus and Hilkka Grahn. 2023. NewSQL Database Management System Compiler Errors: Effectiveness and Usefulness. *International Journal of Human–Computer Interaction* (2023), 1–12. https://doi.org/10.1080/10447318.2022.2108648 arXiv:https://doi.org/10.1080/10447318.2022.2108648

[16] Toni Taipalus, Hilkka Grahn, and Hadi Ghanbari. 2021. Error messages in relational database management systems: A comparison of effectiveness, usefulness, and user confidence. *Journal of Systems and Software* 181 (2021), 111034. https://doi.org/10.1016/j.jss.2021.111034

[17] Toni Taipalus and Piia Perälä. 2019. What to Expect and What to Focus on in SQL Query Teaching. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE)* (Minneapolis, MN, USA) *(SIGCSE '19)*. ACM, New York, NY, USA, 198–203. https://doi.org/10.1145/3287324.3287359

[18] Toni Taipalus, Mikko Siponen, and Tero Vartiainen. 2018. Errors and Complications in SQL Query Formulation. *ACM Transactions on Computing Education* 18, 3, Article 15 (August 2018), 29 pages. https://doi.org/10.1145/3231712

[19] Jess Tan, Desmond Yeoh, Rachael Neoh, Huey-Eng Chua, and Sourav S. Bhowmick. 2022. MOCHA: A Tool for Visualizing Impact of Operator Choices in Query Execution Plans for Database Education. *Proc. VLDB Endow.* 15, 12 (2022), 3602–3605. https://www.vldb.org/pvldb/vol15/p3602-bhowmick.pdf

[20] Florian Waas and César Galindo-Legaria. 2000. Counting, enumerating, and sampling of execution plans in a cost-based query optimizer. *ACM SIGMOD Record* 29, 2 (2000), 499–509. https://doi.org/10.1145/335191.335451

[21] Weiguo Wang, Sourav S. Bhowmick, Hui Li, Shafiq Joty, Siyuan Liu, and Peng Chen. 2021. Towards Enhancing Database Education: Natural Language Generation Meets Query Execution Plans. In *Proceedings of the 2021 International Conference on Management of Data*. Association for Computing Machinery, New York, NY, USA, 1933–1945. https://doi.org/10.1145/3448016.3452822