SQL: A Trojan Horse Hiding a Decathlon of Complexities

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

Figure 1: An image of a Trojan horse, original generated with DALL·E with the prompt "a realistic painting of a Trojan horse at the gates of Troy, with silhouettes of people in awe"

ABSTRACT

Despite its age, SQL is still a widely sought skill among software developers and data engineers, which makes learning SQL a tempting prospect. Several online courses and tutorials may even inspire learners by stating that SQL is a simple and easy language to learn. This impression might also be strengthened by looking at simple SQL statements that read close to English, in contrast to most programming languages. In this paper, I will present ten complexities hiding behind SQL's initial appeal, and my experiences and possible solutions in mitigating these complexities in data systems education.

CCS CONCEPTS

• **Applied computing** → **Education**; • **Theory of computation** → **Database query languages (principles)**; *Data structures and algorithms for data management*; *Logic and databases.*

KEYWORDS

SQL, complexity, relational theory, education, is SQL difficult

ACM Reference Format:
Toni Taipalus. 2023. SQL: A Trojan Horse Hiding a Decathlon of Complexities. In *Proceedings of 2nd International Workshop on Data Systems Education (DataEd '23).* ACM, New York, NY, USA, 5 pages. https://doi.org/XXXXXXX.XXXXXXX

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
DataEd '23, June 23, 2023, Seattle, WA
© 2023 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06...$15.00
https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

SQL is a language required for many software development positions to complement programming skills [4]. The language itself appears user-friendly: it is domain-specific, SQL statements look almost English, and there seems to be little syntactical padding such as different parentheses, brackets or, esoteric keywords. However, inside the appealing plumage hides a powerful query language, and more powerful languages have a tendency to also be more complex.

Becker [2] argues against the common convention of stating that programming is difficult. I have said and written several times that SQL is a difficult or challenging language to learn and use. In retrospect, I have to admit that such statements, even when citing a secondary source, were not based on systematic, empirical evidence but rather on personal viewpoints in order to prepare students for something that is expected to take time, or to motivate a scientific study. Rather than arguing in this paper that SQL is particularly difficult or particularly easy, I will instead present ten complexities in SQL that might not be evident from the start of the learning process.

I agree with an educator who might say that calling something difficult is counter-productive. However, I also think it is important to agree if something is going to be challenging (or difficult or hard, etc.) to learn in order for the learner to assume a mindset that learning this particular topic will take time and effort. While agreeing with Becker's arguments regarding the statement about the difficulty of programming, I cannot promise to refrain from saying that SQL is difficult in the future. I do believe that SQL has made querying databases much simpler and easier than any other way of querying, given the nature of relational database design to cater to all possible queries instead of a chosen few. Without SQL, querying databases might be more difficult, or, if querying would be easier, some other aspects such as database design would be more difficult.

In this paper, I consider SQL to be the language the SQL Standard [9, 10] defines, but in practice, it is difficult to consider SQL separately from both the underlying principles such as the relational model, as well as from the practical implementations which are the dialects offered by various DBMSs. Consequently, I discuss the complexities of SQL, but in addition to discussing the properties of the language itself, the discussion inevitably bleeds to the underlying principles and practical implementations.

This paper is not a critique of SQL, but an opposing point to the notion that SQL is a particularly easy language to learn, with the key takeaway that SQL, like many other computer languages, has many intricacies that are not inherently obvious if one simply considers simple tasks. These complexities are the aspects that make SQL *as difficult as it is*, but the reader can personally decide how difficult *difficult* is in this case. Perhaps the old adage "*easy to learn, difficult to master*" applies to SQL as well. In the next sections, the ten complexities are divided under the underlying principles behind SQL, the language itself, and the implementations of the language and the environments in which SQL statements are executed.

2 THE UNDERLYING PRINCIPLES

2.1 The role of relational theory

At first glance, relational theory appears intuitive and straightforward, as we are likely already familiar with representing data in a tabular format. However, upon closer inspection, relational theory can become challenging due to the various concepts involved, such as candidate keys, superkeys, functional dependencies and axioms, and normal forms. What makes it even more challenging is that, although the concepts of relational theory are strictly defined, they are not always followed in practice. As Date [8, p.204ff] points out, relational theory does not require the use of primary keys, yet they are typically defined in tables. Furthermore, as Codd [5] defined that for a relation to be a relation, it must adhere to first normal form, but even the SQL Standard describes non-first normal form data structures such as JSON columns.

It is unclear how a target normal form is selected in industry settings, and with only a few exceptions [1], it is unclear what the effects of database normalization on dataset sizes, occurring anomalies, and query execution performance are in practice. In fact, in environments such as data warehouses, it is not uncommon to sacrifice data redundancy minimization for performance gains. From a learner's perspective, this might seem counter-intuitive or discouraging after familiarizing oneself with complicated relational principles, only to discover that they are not always followed in practice. As educators, we are providing students with formal tools that can be difficult to use, but even more challenging is knowing when to use them.

How to mitigate: I teach formal relational theory and database design only after SQL. This way, SQL is just querying data structures such as tables and columns, which are in my opinion, mostly intuitive without prior knowledge of relational theory. In this setting, my table structures always adhere strictly to at least third normal form, and students do not define the tables themselves, only query and modify data. I can also see the appeal of teaching formal relational theory first, but I have come to see that this order introduces fewer complexities.

2.2 Data demand agnosticism

Closely related to the previous complexity, it may be easy to lull oneself into a sense of security that if one follows relational theory in the logical design of one's database, the database will be able to answer any data demand the business domain requires. And this is almost always true. In contrast, many NoSQL data models follow an opposing principle, according to which the database is not designed to accommodate all possible data demands, but rather only the specific ones required by the business domain.

Aside from other considerations arising from this difference, it is easy to see which approach makes answering complex data demands easier and which makes it more difficult, particularly from a query writing perspective. Writing SQL can be challenging because the underlying database was not designed to fulfill your specific query, but rather all possible queries. Although a general-purpose approach has its advantages, querying relational databases inherently requires a more powerful query language. Therefore, it follows intuitively that a powerful query language has more complexities than a query language that relies on a specific database structure to satisfy data demands.

How to mitigate: I have not found a way to mitigate this complexity. In data warehouses, one can define aggregated tables to make querying easier (and computationally faster), but someone needs to write the complex queries that create these new data structures in the first place. It has also been shown that databases in low normal forms make querying less error-prone [3], but from a pedagogical perspective, teaching querying using only such databases introduces more complexities than it solves.

2.3 Sets and operations

Set theory in relational databases can be described in layperson's terms and does not necessarily require prior mathematical knowledge. The operations on sets described in Codd's [5] application of relational algebra on query languages is, for the most part, intuitive for a layperson as well. One does not necessarily even need to grasp that this is a branch of mathematics, as it is intuitive to, e.g., point out the common items in basket A: (apple, orange, lemon) and basket B: (orange, lemon), or point out the baskets which do not contain any apples. The difficulty arises when one considers the intersection of several sets, each with a seemingly inconceivable number of items.

For a more experienced writer, joining sets (i.e., contents of columns of different tables) becomes almost mechanical, without the need to think about the items themselves. That is, with n sets, one typically needs n - *1* intersections (i.e., joins). If n is, say, 10, one does not even begin to try to put the 9 intersections into natural language. That is, writing is not about verbalizing the logic in a data demand as much as it is about mechanically writing the query. With the data demand agnostic database structures, queries can become complex, and although the principles behind sets and operations are relatively easy, they are sometimes applied in complex contexts. Scholars like Danaparamita and Gatterbauer [6], Miedema and Fletcher [14] and myself [21] have proposed visualizing complex queries. Many of these propositions rely (sometimes implicitly) on graph theory representations of tables (vertices) and joins (edges).

How to mitigate: I utilize a query planning notation [21] to support query writing. I do not teach the notation but rather explain it as I draw. I sometimes only give a query plan and ask students to explain the data demand behind it. It is also a quick way to compare different queries and the logical differences between them without focusing on syntactical nuances.

3 THE LANGUAGE

3.1 Imperative or declarative

The language can be charming to novices due to its simplicity. The first SQL statements presented read deceptively close to plain English. Instead of defining a loop as in programming, one quite straightforwardly writes "*give me the names of products that cost more than 50 euros*". However, this uniformity between SQL and natural language usually starts to wane when table joins are needed.

In a typical scenario, one selects the columns in the `SELECT` clause using table aliases, which are rather counter-intuitively defined in the following `FROM` clause. If tables are joined using subqueries, it might be easier to understand the query by reading inside the innermost parentheses first, which are typically located in the last lines of the query. Furthermore, if derived tables (i.e., subqueries in the `FROM` or `SELECT` clauses) or common table expressions are used, the query can no longer be read as a natural language statement in the sense that text is read from start to finish. Instead, one might need to perform somewhat complex acrobatics, jumping between different parts of the query to understand how it works. The difficulty here is not necessarily the jumping but how imperative elements are incorporated into a declarative language after a threshold of query complexity is reached.

My argument here is not that complex data demands should be expressed as simple SQL statements, but that the seemingly gentle learning curve takes a relatively steep turn. In imperative languages, complex problems are typically divided into smaller ones, but in declarative languages, this division is often more challenging and takes away from the declarative nature of the language.

How to mitigate: I refrain from trying to treat SQL like a natural language. This approach might not help with the complexity of reading queries, but perhaps it manages expectations.

3.2 A myriad of choices

The SQL Standard gives a myriad of choices for formulating queries. Table joins can be achieved *(i)* with subqueries formulated with `EXISTS` or `IN`, with an equals or non-equijoin operator with or without the keywords `ANY`, `SOME` or `ALL`, *(ii)* with an explicit join condition in the `WHERE` clause without using any other join-related keywords, or *(iii)* with the dedicated `JOIN` structure and all its variations such as `LEFT`/`RIGHT`/`FULL OUTER`, `NATURAL` or `CROSS`.

While it is nice to have options, and while many of the table join methods are interchangeable in many cases, it requires care to recognize the situations in which they are not. A result table containing columns from multiple tables typically requires that subqueries are not used in joining the tables from which the columns in the result table are selected. If one does not want to use a common table expression, a subquery (rather than `JOIN`) is typically the way to compare the results output by an aggregate function. A subquery formulated with `EXISTS` operates with two-valued logic while a subquery formulated with `IN` uses three-valued logic. A positive side is that while, for example, relational division using multiple `NOT EXISTS` subqueries has been observed to be "intellectually challenging" [13] and an "overwhelming challenge" [12] for students, division can be written with more readable constructs such as `ALL` or with `GROUP BY` and `HAVING` [12]. There are also other alternatives such as `NOT EXISTS` paired with `EXCEPT`.

How to mitigate: I (sometimes falsely) abstract to create rules of thumb on when to use which query concepts. In basic-level courses, I leave out many query concepts such as derived tables and `LIMIT`/`OFFSET`. I also leave out SQL keywords such as `INTERSECTION`, `CROSS JOIN` and `FULL OUTER JOIN` which are in my experience rarely used.

3.3 Strange conventions

SQL has some syntactical conventions that may seem strange at first. Even after becoming accustomed to these conventions, they can still be challenging to work with in query writing.

For example, a learner quickly discovers that aggregate functions are not allowed in the `WHERE` clause. Such a syntactical feature typically introduces the need for a subquery which is evaluated against a column value or a constant in the query above the subquery. This feature can also introduce the need for a self-join, which may be a difficult concept to understand. Although queries with such constructs are longer, they are not necessarily as difficult to read as they are to write.

Another example is that the SQL Standard defines two ways to implement `GROUP BY`. These ways do not seem to have official names, but I will call them *strict* (which is a core feature in the Standard) and *non-strict* (which is an optional feature). To my understanding, the strict `GROUP BY` dictates that all the grouping columns in the `SELECT` clause must appear in the `GROUP BY` clause and vice versa. The non-strict `GROUP BY` dictates that all the grouping columns in the `SELECT` clause must appear in the `GROUP BY` clause, or be functionally dependent on at least one column in the `GROUP BY` clause. Some DBMSs implement both ways, while others implement only one.

The challenge related to query writing with strict grouping is redundancy. If the grouping columns in the `SELECT` clause must be identical to the columns in the `GROUP BY` clause, it is a strange syntactical convention to require the `GROUP BY` clause at all. On the other hand, the challenge related to non-strict grouping is that most DBMSs are not concerned with functional dependencies. If a table has many candidate keys or violates (perhaps by design) the third normal form, primary keys are not enough to enforce functional dependencies. Consequently, most (if not all) DBMSs that implement non-strict grouping allow grouping on whichever columns, which can lead to spurious rows in result tables. Some DBMSs such as Oracle Database and PostgreSQL keep metadata on functional dependencies but only implement strict grouping. As the existence or absence of functional dependencies is in these cases dictated by database data rather than business logic, a DBMS that hypothetically used non-strict grouping with functional dependencies based only on automatically collected metadata would risk the situation in which adding or modifying data would render one or several `SELECT` statements syntactically incorrect.

How to mitigate: I teach only strict grouping and try to choose a DBMS that implements only strict grouping.

3.4 Three-valued logic

In terms of logic, two worlds collide in SQL. Date and Rubinson's insightful discussions [7, 18] on SQL's logic seem to agree that "three-valued logic is incompatible with database management systems". On one hand, SQL operates using three-valued logic, and learners may be taught what this means in terms of truth tables. On the other hand, three-valued logic is seldom useful in the context of relational databases, and often only adds to how challenging SQL is to learn and use. That is, even though in theory SQL follows three-valued logic, many SQL concepts operate using two-valued logic, and the looming presence of three-valued logic can cause unexpected behavior.

From three-valued logic it follows that `NULL` is `NULL`, that `NOT NULL` is `NULL`, or that the expression on column `c = NULL` is also `NULL`. However, it is often desired that missing values are translated into something else than unknowns (or whatever `NULL` stands for in different cases). For example, the SQL keyword `IS` and functions such as `COALESCE` and `NULLIF` translate the presence or absence of unknowns into `TRUE` or `FALSE`. The presence of `IS` arguably adds to the complexity of needing to remember the keyword instead of using common comparison operators to check for the presence of unknowns. Instead of returning `NULL` for all queries where at least one calculated value is unknown, SQL's aggregate function `SUM` treats `NULL` like a zero instead of an unknown value. `EXISTS` operates using two-valued logic, and a correlated subquery with `EXISTS` results in either `TRUE` or `FALSE` per row regardless of whether there are unknown values present in the joined columns. This discrepancy between what is logically correct under a set of rules, and which rules we need in practice is challenging.

How to mitigate: I merely mention three-valued logic, but otherwise act as if SQL operates on two-valued logic.

4 THE ENVIRONMENTS

4.1 Dialects

After discovering discrepancies between theoretical foundations and the SQL Standard, the final impedance mismatch (before the application code) occurs between the SQL Standard and the various implementations of the SQL language in relational DBMSs. Different DBMSs such as SQL Server, MariaDB, and PostgreSQL implement their own dialects of SQL.

Different dialects may implement different parts of the SQL Standard, features contrary to the SQL Standard, and features not defined in the SQL Standard. The differences between dialects are sometimes effectively non-existent or minor to the point that one can forget that the practical implementations are indeed dialects. However, in some cases, these differences add to the complexity of learning and using SQL.

The differences in grouping behavior, as discussed in Section 3.3, are one of the bigger differences, along with the naming, functionality, and availability of scalar functions. Time manipulation is often complex, and changing from one DBMS to another may mean that none of the familiar functions or keywords (e.g., `OVERLAPS` as defined in the Standard) are available in the new DBMS. The syntax for modifying table structures is usually different in each dialect. Some differences may even fundamentally affect how the logical structures of databases are designed, as Oracle Database does not implement `CASCADE` in foreign keys.

How to mitigate: I teach only one dialect per course, yet mention that all queries are not portable from one DBMS to another. I try to choose a dialect that is to my knowledge close to the SQL Standard's definition to maximize compatibility. I can also see the value in Randolph's [17] way of teaching several dialects, yet in my opinion, teaching several dialects adds unnecessary complexity to learning the language itself. Switching between dialects can come after learning the basics of SQL.

4.2 Error messages

Many popular DBMSs have been around for decades, and their usability has often been neglected, especially when it comes to SQL compilers. Especially for learners, any compiler is generally seen as an unwavering, unerring authority [24]. However, when a query writer commits a syntax error in query formulation, they are faced with error messages that can be redundant, inaccurate, or difficult to read [23].

We recently conducted a mixed-method study (forthcoming) on the qualities of syntax error messages of eight relational DBMSs. While some systems provide clear and helpful error messages, others overwhelm users with irrelevant information or fail to communicate the actual problem. Without naming any culprits, some DBMSs state multiple times in the same error message that the query contains an error, some replicate parts of the erroneous query but not the erroneous part, some begin the error message by stating seemingly incomprehensible environmental variables, and some simply usually instruct to read the manual.

Furthermore, and closely related to the previous complexity of dialects, different DBMSs check SQL syntax differently, meaning that what is a syntax error according to one DBMS may not be a syntax error according to another. Overall, error messages are crucial for fixing queries, but they could be more effective if they simply pointed out even the approximate location of the error. I expect the rising popularity of large language models to tackle much of the current grievances in SQL error messages in DBMSs. Similar works in SQL education [16], natural language processing [11] and hybrid query processing [19] have already appeared.

How to mitigate: I strive to teach with a DBMS that has user-friendly error messages. This, naturally, cannot be the only aspect that matters, and there are often other considerations such as technical constraints and costs. Nevertheless, students need, if not a friendly, at least a non-hostile environment in which to try out erroneous queries.

4.3 Lack of error messages

In addition to syntax errors, many DBMSs identify *semantic* errors. A typical query with a semantic error will always return an empty result table, regardless of what is stored in the database, and the fact that the query will return an empty result table is evident by reading the query rather than executing it. Most likely for query optimization reasons, many DBMSs identify such queries and return an empty result table without even evaluating expressions such as

`WHERE age > 20 AND age < 20;` against the database. There are also other similar "always incorrect" query constructs which are often identified by the DBMS [22].

The complexity here is that DBMSs do not notify the user of such errors. Instead, these valuable pieces of information are hidden in query execution plans, and they must be explicitly requested. If a novice user is even aware of the existence of query execution plans, manually requesting one after each syntactically valid query is arguably an arduous task even though doing so could reveal semantic errors in the query. Additionally, reading query execution plans is not a trivial task for the uninitiated, and query execution plans and physical database operations can vary significantly between DBMSs.

How to mitigate: In basic-level courses I have begun to utilize relatively simple database structures with relatively simple, heterogeneous datasets [15] which help students in manually checking the data to understand why a queries return the datasets they return. I also show correct result tables for each exercise, so that students may check the correctness of their queries themselves, and iterate if necessary.

5 CONCLUSION

In this paper, I presented ten complexities in learning and using SQL, many of them stemming from the discrepancies between the theoretical foundations, the SQL Standard, and the different implementations of SQL. These complexities might not account for too much on their own, but together they can cause a healthy dose of dread to even the most experienced developer [20]. Based on my personal experiences, I also presented ways to mitigate each of these complexities in teaching SQL. Some of these mitigations might even be called *hacks* in a negative sense, as they effectively omit parts of knowledge that the learner will likely encounter later. However, I believe that I am not alone in my view that in education, some aspects of any topic should be omitted or abstracted when learning the basics.

REFERENCES

[1] Mashel Albarak, Rami Bahsoon, Ipek Ozkaya, and Robert Nord. 2022. Managing Technical Debt in Database Normalization. *IEEE Transactions on Software Engineering* 48, 3 (2022), 755–772. https://doi.org/10.1109/TSE.2020.3001339

[2] Brett A Becker. 2021. What does saying that'programming is hard'really say, and about whom? *Commun. ACM* 64, 8 (2021), 27–29.

[3] A.F Borthick, P.L Bowen, S.T Liew, and F.H Rohde. 2001. The effects of normalization on end-user query errors: An experimental evaluation. *International Journal of Accounting Information Systems* 2, 4 (2001), 195–221. https://doi.org/10.1016/S1467-0895(01)00023-9

[4] Stephen Cass. 2022. SQL Should Be Your Second Language. *IEEE Spectrum* 59, 10 (2022), 20–21. https://doi.org/10.1109/MSPEC.2022.9915547

[5] Edgar F. Codd. 1970. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM* 13, 6 (June 1970), 377–387. https://doi.org/10.1145/362384.362685

[6] Jonathan Danaparamita and Wolfgang Gatterbauer. 2011. QueryViz: Helping Users Understand SQL Queries and Their Patterns. In *Proceedings of the 14th International Conference on Extending Database Technology* (Uppsala, Sweden) *(EDBT/ICDT '11)*. Association for Computing Machinery, New York, NY, USA, 558–561. https://doi.org/10.1145/1951365.1951440

[7] C. J. Date. 2008. A Critique of Claude Rubinson's Paper Nulls, Three - Valued Logic, and Ambiguity in SQL: Critiquing Date's Critique. *SIGMOD Rec.* 37, 3 (2008), 20–22. https://doi.org/10.1145/1462571.1462574

[8] C. J. Date. 2012. *Database Design and Relational Theory: Normal Forms and All That Jazz*. O'Reilly Media, Inc.

[9] ISO/IEC. 2016. ISO/IEC 9075-1:2016, "SQL - Part 1: Framework". https://www.iso.org/standard/63555.html

[10] ISO/IEC. 2016. ISO/IEC 9075-2:2016, "SQL - Part 2: Foundation". https://www.iso.org/standard/63556.html

[11] Aiwei Liu, Xuming Hu, Lijie Wen, and Philip S Yu. 2023. A comprehensive evaluation of ChatGPT's zero-shot Text-to-SQL capability. *arXiv preprint arXiv:2303.13547* (2023).

[12] Victor M. Matos and Rebecca Grasser. 2002. Teaching Tip A Simpler (and Better) SQL Approach to Relational Division. *Journal of Information Systems Education* 13, 2 (2002), 85–88. http://jise.org/Volume13/Pdf/085.pdf

[13] L. I. McCann. 2003. On making relational division comprehensible. In *Proceedings of the 2003 33rd Annual Frontiers in Education Conference (FIE)*, Vol. 2. F2C–6. https://doi.org/10.1109/FIE.2003.1264699

[14] Daphne Miedema and George Fletcher. 2021. SQLVis: Visual Query Representations for Supporting SQL Learners. In *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 1–9. https://doi.org/10.1109/VL/HCC51201.2021.9576431

[15] Daphne Miedema, Toni Taipalus, and Efthimia Aivaloglou. 2023. Students' Perceptions on Engaging Database Domains and Structures. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1* (Toronto ON, Canada) *(SIGCSE 2023)*. Association for Computing Machinery, New York, NY, USA, 122–128. https://doi.org/10.1145/3545945.3569727

[16] Rubén Pérez-Mercado, Antonio Balderas, Andrés Muñoz, Juan Francisco Cabrera, Manuel Palomo-Duarte, and Juan Manuel Dodero. 2023. ChatbotSQL: Conversational agent to support relational database query language learning. *SoftwareX* 22 (2023), 101346. https://doi.org/10.1016/j.softx.2023.101346

[17] Gary B. Randolph. 2003. The Forest and the Trees: Using Oracle and SQL Server Together to Teach ANSI-standard SQL. In *Proceedings of the 4th ACM Conference on Information Technology Curriculum (CITC)* (Lafayette, Indiana, USA) *(CITC4 '03)*. ACM, New York, NY, USA, 234–236. https://doi.org/10.1145/947121.947174

[18] Claude Rubinson. 2007. Nulls, Three-Valued Logic, and Ambiguity in SQL: Critiquing Date's Critique. *SIGMOD Rec.* 36, 4 (2007), 13–17. https://doi.org/10.1145/1361348.1361350

[19] Mohammed Saeed, Nicola De Cao, and Paolo Papotti. 2023. Querying Large Language Models with SQL. *arXiv preprint arXiv:2304.00472* (2023).

[20] StackOverflow. 2022. *StackOverflow 2022 Developer Survey*. https://survey.stackoverflow.co/2022/#technology-most-loved-dreaded-and-wanted

[21] Toni Taipalus. 2019. Teaching tip: A notation for planning SQL queries. *Journal of Information Systems Education* 30, 3 (2019), 160–166. http://jise.org/Volume30/n3/JISEv30n3p160.pdf

[22] 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* (Hamburg, Germany) *(CHI EA '23)*. Association for Computing Machinery, New York, NY, USA, Article 239, 6 pages. https://doi.org/10.1145/3544549.3585794

[23] 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

[24] Christopher Watson, Frederick WB Li, and Jamie L Godwin. 2012. Bluefix: Using crowd-sourced feedback to support programming students in error diagnosis and repair. In *Advances in Web-Based Learning-ICWL 2012: 11th International Conference, Sinaia, Romania, September 2-4, 2012. Proceedings 11*. Springer, 228–239.