Graph-based QSS: A Graph-based Approach to Quantifying Semantic Similarity for Automated Linear SQL Grading

Leo Köberlein[1], Dominik Probst [1], and Richard Lenz [1]

Abstract: Determining the Quantified Semantic Similarity (QSS) between database queries is a critical challenge with broad applications, from query log analysis to automated SQL skill assessment. Traditional methods often rely solely on syntactic comparisons or are limited to checking for equivalence.

This paper introduces Graph-based QSS, a novel graph-based approach to measure the semantic dissimilarity between SQL queries. Queries are represented as nodes in an implicit graph, while the transitions between nodes are called edits, which are weighted by semantic dissimilarity. We employ shortest path algorithms to identify the lowest-cost edit sequence between two given queries, thereby defining a quantifiable measure of semantic distance.

An empirical study of our prototype suggests that our method provides more accurate and comprehensible grading compared to existing techniques. Moreover, the results indicate that our approach comes close to the quality of manual grading, making it a robust tool for diverse database query comparison tasks.

Keywords: SQL, SQL query comparison, SQL query grading, Linear grading

1 Introduction

The problem of comparing SQL queries has received some attention in the past, both theoretical and practical [ASU79; Ch18; Ch21; Zh19]. These studies generally focus on trying to answer the question of whether two given queries are equivalent or, more generally, whether the result of one is a superset of the other, the so-called containment problem. However, especially in the area of teaching SQL, there is a need to answer a different kind of comparison problem.

To allow for partial grading, a finer differentiation than just a binary scale is required. Not only is it of interest whether a student's query and a reference solution are completely equivalent, but if they are not, their Quantified Semantic Similarity (QSS) is needed.

It is also beneficial for a student's learning process if meaningful feedback is provided. This means explaining why their solution is not (fully) correct, e.g., what steps would be necessary to make it correct. And if it is correct but looks different from the reference solution, it is desirable to be able to explain why it is still correct.

[1] Friedrich-Alexander-Universität Erlangen-Nürnberg, Computer Science 6, Martensstraße 3, 91058 Erlangen, Germany, leo.koeberlein@fau.de; dominik.probst@fau.de, https://orcid.org/0009-0002-1887-4374; richard.lenz@fau.de, https://orcid.org/0000-0003-1551-4824

These tasks are still usually done by hand, especially for exams. This is not only time-consuming, but also error-prone, as the subjective severity of errors can vary between correctors or subtle mistakes can be missed. An automated solution to these problems is therefore required.

It is likely that an approach for measuring QSS between queries would also be beneficial in other application areas. For example, the search for similar queries in query logs as described in [Wa17] or in [BMT19] could be significantly improved.

2 Goals

The goals of this paper can be summarized as follows:

Quantified Semantic Similarity (QSS): Two SQL queries are to be compared. The result of the comparison should not be equivalence, but instead similarity, quantified on a linear scale. The similarity should express how much the queries differ semantically, not just syntactically, to make it more useful for grading.

Meaningful feedback: There should be explanations as to either why two queries were found to be equivalent, or why they were not equivalent and what would be required to change that. Such feedback is helpful for students to understand their mistakes and to justify the reasoning behind grades.

Guaranteed result: The method of comparison should always produce a result. Given that query equivalence is generally undecidable [AHV95; Ch17], this is not trivial.

Unrestricted input: Arbitrary SQL queries should be processable. Some, mostly theoretical, approaches try to circumvent this undecidability by restricting themselves to small subsets of SQL. Our comparison technique should not be restricted in this way.

Configurability and extensibility: The system should be configurable and extensible. Different exercises or problems require different points of focus. It should be easy to take this into account and to adapt the comparison.

3 Related Works

Other publications on comparing SQL queries can be roughly categorized into three groups: Those that do not compare the queries themselves, but rather the tuples returned when they are executed on a sample database, which is called dynamic analysis. Others, that compare the queries themselves using static analysis, in order to determine whether or not they are semantically equivalent. And finally those that also use static analysis, but continue to check how similar they are in case they are not completely equivalent.

3.1 Dynamic analysis

Dynamic analysis is a popular way of comparing SQL queries [Bh15; Ch15; Do19; KTH13; NAJ17; PL04; So06]. For example, given a reference solution R and a student-devised solution S, both are considered semantically equivalent if the combined query in Listing 1 does not return any tuples.

```sql
(S UNION R) EXCEPT (S INTERSECT R)
```

List. 1: Basic SQL statement for comparing queries via their execution results.

The advantage is the conceptual simplicity: The reference solution is an executable query, i.e., it can be executed on the database. For the student's query to be a correct solution, it must be executable and (always) return the same tuples. Since it is executed on a real Database Management System (DBMS), parsing and error checking is automatically handled.

Currently the most advanced take on this is the XData system by Bhangdiya et al. [Bh15] and Chandra et al. [Ch15]. Their goal is to improve the precision of dynamic analysis by generating special test databases for each individual query. These will produce differing results in the presence of typical errors associated with certain query constructs.

The disadvantages of this approach are significant: Dynamic analysis can only prove that queries *are not* equivalent, but not that they *are*. For example, there may be subtle differences that only lead to different results for very specific combinations of data. If these are missing from the test databases, they will go undetected, leading to false positives. The risk of missing such cases is quite high. Even Chandra et al. [Ch15] acknowledge this problem and only strive to detect common errors, while some query constructs are not even fully supported by the data generation [Ch19b]. Secondly, dynamic analysis itself only gives a binary result. Trying to estimate similarity based on the fraction of matching output tuples works poorly because tiny differences, e.g., in the selection, can lead to greatly diverging results. Finally, there is no meaningful feedback. When queries are found not to be equivalent, there is no explicit explanation as to why they produced different outputs.

3.2 Static analysis for equivalence

A promising approach is static analysis, which compares the queries themselves. Given that query equivalence is generally undecidable [AHV95; Ch17], there are two possible avenues to pursue in order to uphold theoretical correctness:

One way is to consider only a subset of SQL, for example conjunctive queries [AHV95, chapter 4], on which equivalence is decidable. Possible decision techniques are those of Aho et al. [ASU79], Chu et al. [Ch18] and Zhou et al. [Zh19]. This is mainly of theoretical relevance, since in practice, a wider range of queries is required.

The other way is to have fewer restrictions on the queries, but this runs the risk of not being able to prove or disprove their equivalence in some cases. Cosette by Chu et al. [Ch17] tries to strike a balance. It is a powerful tool and has been used to prove and disprove many equivalence transformations used in execution plan optimizers. However, there are still some features that are not yet supported, such as the aggregation functions `AVG`, `MIN` and `MAX`, or the `ORDER-BY` clause.

Also in this category resides an approach by Dollinger; Melville [DM11]. It is similar to ours, in that it uses equivalence transformation patterns to try and turn one query into the other. The main differences are that it only checks for equivalence and focuses on high-level transformations involving subqueries, while seemingly neglecting lower-level equivalences, such as the distributive law.

All these techniques have major drawbacks: Either there are restrictions on the query constructs supported, or there is no guaranteed answer. Moreover, they still only determine equivalence, not quantified similarity.

3.3 Static analysis for similarity

A simple approach for quantifying query similarity is to use string similarity metrics to compare the descriptions of the reference solution and the student-devised query. Stajduhar; Mausa [SM15] combine multiple different such metrics (including absolute length difference, Levenshtein distance and Euclidean word frequency distance) and feed them into a logistic regression model, which is trained on a large number of manually assigned scores. This has been tested with some success, but it is obvious that it will only give approximate results.

The problem of partial correctness is addressed by Fabijanic et al. [FDF20] by decomposing the query into clauses (e.g. `SELECT`, `FROM`, `WHERE`, ...). Then, a series of successive scoring rules is applied, each of which checks whether each clause is the same as the corresponding clause in the reference, and deducts a fixed score if it is not. As a measure of similarity, this is better than a binary result, but still a rather rough distribution.

A more detailed and mathematical method is presented by Panni; Hoque [PH20], combining the two ideas to some extent. The queries are split into clauses, but for each clause, an individual score is calculated using mathematical formulas. These formulas are tailored to each clause and take into account the exact number of individual matching and mismatching elements. The individual scores are then combined into a weighted sum.

All mentioned approaches have the disadvantage that, apart from the basic structure of a query, they completely ignore the semantics of SQL. But the location of an error is not as important as the logical step it represents, so weighting by component is of little use. Moreover, there are numerous possibilities of having two semantically equivalent queries, that are syntactically different. The only way to produce correct results would be to provide a comprehensive set of semantically equivalent reference queries, which is unrealistic.

Wang et al. [Wa20] first check all student-developed queries for semantic equivalence to the reference solution, and then use these as if they were reference solutions in calculating the partial scores for all non-equivalent queries. The initial equivalence check is performed using dynamic analysis (see Section 3.1). Partial marks are generally assigned by parsing the query into an Abstract Syntax Tree (AST) and determining the tree edit distance. If the query description cannot be parsed into an AST, string similarity is used instead, as in [SM15]. The calculation of partial marks using the tree-edit distance or string similarity ignores the semantics of SQL and is only approximate. In addition, dynamic analysis is not sufficient as an initial test for semantic equivalence, mainly due to the risk of false positives (see Section 3.1).

Wanjiru et al. [WBH24] achieve partial grades by combining the results of multiple of the aforementioned comparison techniques. Queries have properties, which in turn have outcomes. The outcome of the property "result" is determined via dynamic analysis, "syntax" via string comparison (specifically Levenshtein distance, as in [SM15]) and "semantics" via tree edit distance on the AST (as in [Wa20]). The number of outcomes per property is fixed to either two or three (correct, minor incorrect, incorrect), giving a fixed granularity of either 8 or 27 grades. "Minor incorrect" for dynamic analyis is defined as the reference queries' result tuples being a subset of the student query's, for the string comparison the arbitrary number of 2 differing characters is chosen, and distance 1 for the tree edit distance. Like in [Wa20], student queries deemed "correct" by the dynamic analysis are treated as reference queries, introducing the same risk of false positives (see Section 3.1). Similarly, all other problems of the individual techniques combined by this approach apply as well.

The XData system (see Section 3.1) has been extended by Chandra et al. [Ch16; Ch19a; Ch19b; Ch21; CS22] to be able to generate partial marks. As in the previous two approaches, a student-developed query is first checked against the reference solution by comparing their execution results. Then an edit distance is calculated. However, unlike a basic tree edit distance, the semantics of SQL are taken into account: Small edits, e.g. adding/removing/replacing an attribute or changing a `JOIN`-type, are applied successively to the student's query. These edits only add elements that are actually present in the sample solution and remove those that are not. After each edit, equivalence to the sample solution is checked by applying a normalisation, called canonicalisation, which consists of a series of confluent equivalence transformations. As a result, if the query is semantically equivalent despite syntactic differences, it is correctly identified as such. This approach still carries the risk of false positives during the initial equivalence check (see Section 3.1) and can only process queries that are executable. Since the idea behind the partial marking in this system is quite similar to our Graph-based QSS, further comparisons will be introduced in Section 4.

Most approaches introduced in this section suffer from a lack of meaningful feedback. The last three approaches are able to provide such feedback if the queries aren't equivalent, but, due to the preceding dynamic analyis-step, unable if they are.

4 Concept

We present Graph-based QSS, a new method of comparing SQL queries, that achieves the goals set in Section 2 without the shortcomings of previous approaches shown in Section 3.

4.1 Definitions

The following definitions clarify the terminology used throughout this paper:

Parsable: A query is parsable if and only if (iff) it follows the SQL syntax and can therefore be parsed into an Abstract Syntax Tree (AST) representation.

Executable: A query is executable in the context of a given database schema, iff it is parsable and can be executed on a database of said schema (without throwing errors).

Syntactically equivalent: Two queries are syntactically equivalent iff their AST representations are exactly the same.

Semantically equivalent: Two queries are semantically equivalent in the context of a given database schema, iff they return exactly the same result when executed on an arbitrary but fixed database of said schema. Syntactically equivalent queries are always semantically equivalent, but not vice versa.

Note that in order to determine semantic equivalence, and hence similarity, (parts of) queries cannot simply be compared without context. For example, suppose the two tables `students` and `teachers` both have a column `id`. Then the queries in Listing 2 and Listing 3 are not semantically equivalent. The second one is not even executable, because the column name `id` is ambiguous. But if only the table `students` had a column called `id`, the queries would be semantically equivalent. This means that semantic equivalence depends not only on the queries, but also the context of the database schema the queries are executed on.

```sql
SELECT students.id
FROM students, teachers
```

List. 2: Query 1 of possibly equivalent pair.

```sql
SELECT id
FROM students, teachers
```

List. 3: Query 2 of possibly equivalent pair.

4.2 Idea

Queries are nodes in a graph. The graph is implicit and edits on these queries are neighbor functions which, when applied to a query/node, generate neighboring nodes. Each edit has a (non-negative) cost associated with it, representing the error or dissimilarity it creates. Given this graph, a shortest path algorithm can be used to find the lowest-cost edit sequence from a start node to a destination node. The combined weight of this edit sequence quantifies, how dissimilar the start and target query are.

Suppose we have a table `students` with columns `id`, `name`, and `age`, and the left query in Figure 1. Assume there is an edit called `setDistinct` with a cost of 2, which sets the `DISTINCT` declaration. Applying this edit to the left query creates a neighbor. Attempting to apply the same edit to this neighbor will not create a new node, as it is already `DISTINCT`.

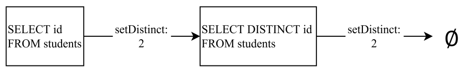
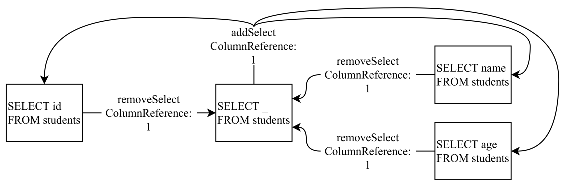


Fig. 1: Two subsequent applications of an edit with cost 2 called `setDistinct` on a query.

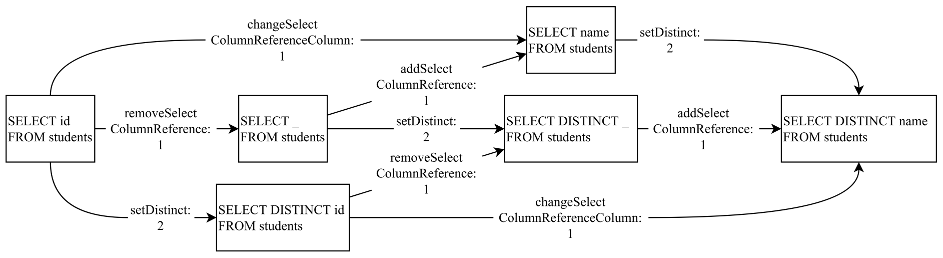
Consider an edit `addSelectColumnReference` with cost 1, which adds a column reference to the expression of a `SELECT` element, and another edit `removeSelectColumnReference` with cost 1, which removes one. Applying the former to the query from Figure 1 will not work because it has only one `SELECT` element, which is already occupied by a column reference to `id`. Applying the latter will produce a new neighbor that still has a `SELECT` element, but now with an empty expression, symbolised as `_`. Doing the same on the new neighbor does not work because there are no more column references to remove. But now `addSelectColumnReference` can be applied, resulting in three neighbors, one for each available column. One is the start query again. And on the other two, `removeSelectColumnReference` can be applied in the same way. This is visualized in Figure 2.



addSelect ColumnReference: 1
SELECT id FROM students
removeSelect ColumnReference: 1
SELECT _ FROM students
removeSelect ColumnReference: 1
SELECT name FROM students
removeSelect ColumnReference: 1
SELECT age FROM students

Fig. 2: Multiple neighbors from one application of `addSelectColumnReference`.

The outer three nodes can reach each other by deleting and re-adding a column reference. This is unnecessarily expensive, so there is another edit `changeSelectColumnReferenceColumn` with cost 1, which changes the referenced column. Finally, suppose we want to compare the right-most query in Figure 3 with the previous start query. Combining all of the above, there are several possible paths between them. The relevant ones are shown in Figure 3, but for the sake of simplicity nothing strongly divergent.



changeSelect ColumnReferenceColumn: 1
SELECT name FROM students
setDistinct: 2
addSelect ColumnReference: 1
SELECT id FROM students
removeSelect ColumnReference: 1
SELECT _ FROM students
setDistinct: 2
SELECT DISTINCT _ FROM students
addSelect ColumnReference: 1
SELECT DISTINCT name FROM students
removeSelect ColumnReference: 1
setDistinct: 2
SELECT DISTINCT id FROM students
changeSelect ColumnReferenceColumn: 1

Fig. 3: Possible paths from the start query to the destination query.

Based on this graph, a shortest path algorithm can be used to find the cheapest edit sequence between the start and destination. In Figure 3, there are two equally cheap ones, consisting of `changeSelectColumnReferenceColumn` and `setDistinct` in any order. The sum of their costs is 3. This is the quantified semantic dissimilarity between start and target.

Iff there is a path with a cumulative cost of 0 between two given queries, then they are **QSS-equivalent**. This concept is an approximation of pure theoretical query equivalence, but the undecidability of the latter means there is no set of edits for which they are the same.

4.3 Nodes

In order to define what exactly a node in this graph is, syntactic and semantic distinctions and the matter of including non-executable queries are relevant.

4.3.1 Syntactic and semantic differences

The relevance of certain syntactic and semantic differences varies depending on the focus of teaching, or whether the comparison context is related to teaching at all.

Suppose a table `students` has a column `id` as its primary key. Then the `DISTINCT` declaration could be removed from the query in Listing 4 and the query would still be semantically equivalent to before. From an objective point of view, this is an equivalence transformation and should have no cost associated with it. From a pedagogical point of view, it could be argued that a student should be penalised for using an unnecessary declaration, so there should be a cost associated with it.

```sql
SELECT DISTINCT id
FROM students
```

List. 4: SQL query with unnecessary `DISTINCT`-declaration.

This becomes even more important when an assignment is specifically intended to teach a particular syntactic construct. For example, the `WITH` clause can be eliminated without changing the semantics by replacing all references to it within the query with its contents. Normally there should be no cost associated with this change. However, if the task description explicitly asks students to use a `WITH` clause, it will be necessary to penalise a student who does not use it.

4.3.2 Non-executability

An important question is how to handle queries that are not executable. It is sensible to allow non-executable queries as intermediate steps along the path. For example, assuming that the tables `students` and `teachers` both have a column `id` and the former also has a column `name`, the query in Listing 6 is non-executable, but is a possible intermediate step between the queries in Listing 5 and Listing 7. This step could be skipped by having the edit detect that the column reference is ambiguous without specifying the table, therefore adding it immediately, but it would make the edit's logic unnecessarily complex.

```sql
SELECT students.name

FROM   students,
       teachers
```

List. 5: Query before.

```sql
SELECT students.name,
       id
FROM   students,
       teachers
```

List. 6: Intermediate step.

```sql
SELECT students.name,
       students.id
FROM   students,
       teachers
```

List. 7: Query after.

If students produce faulty queries, they could not be processed if non-executable queries were not allowed as nodes. The same holds true for incomplete queries, like the one in Listing 8, where students grasp the general structure but are only missing minor parts of it.

```sql
SELECT AVG(   )
FROM students
```

List. 8: Incomplete query missing an expression inside the aggregation function.

The graph should be able to contain non-executable and even incomplete queries, making for less complicated edits and higher flexibility internally, as well as allowing for more freedom in the input.

4.3.3 Node definition and comparison

Based on the observations above, we define a node as follows: A node is one particular AST, which may have missing subcomponents or unset attributes. Nodes are compared directly, i.e., the position and type of each (sub)component and the value of each attribute must match exactly.

Our definition of nodes differs from Chandra et al. [Ch16; Ch19a; Ch19b; Ch21; CS22], who treat a node as a class of semantically equivalent queries. This keeps them from distinguishing between these queries, which we found desirable to support (see Section 4.3.1). Moreover, Chandra et al. only consider executable queries [Ch19b], limiting their system's ability to process many student-generated queries (see Section 4.3.2).

4.4 Edits

In our Graph-based QSS, edges are implicitly represented by edits. We distinguish between the fundamental atomic edits and the semantics-aware shortcut edits.

4.4.1 Edits vs. edges

In our implicit graph, edges are not explicitly given, but there are neighbor functions, called edits, that return the neighbors of a given node. They can conceptually be combined into the single neighbor function an implicit graph is typically defined with by returning the union of all their results.

An edit is a 1-to-n function that takes an AST as input and returns a set of different ASTs. This set can be empty if the conditions for applying the edit are not met by the input. For example, a query cannot be made `DISTINCT` if it already is. An edit can represent multiple outgoing edges. This is visualized by Figure 4. The graph is generally directed. It is possible to make it undirected, but this requires a lot of work to ensure that each edit has a corresponding counter-edit.

An edit might need the database schema as context to preserve semantic equivalence (see Section 4.1). Also, certain information about the destination query is extracted and passed as meta-info (needed in Section 4.6.2). These extra inputs can be seen in Figure 4 as well.

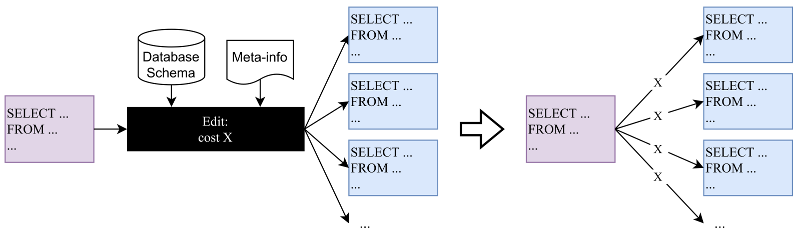


Fig. 4: One edit representing multiple outgoing edges.

Each edit has a fixed, non-negative cost. This cost does not change at runtime. If it is necessary to weight an edit differently in different parts of the query, it must be replaced by multiple part-specific edits. The same principle can be applied to other desired distinctions.

4.4.2 Types of edits

There are several types of edits:

Atomic edits are the fundamental edits that make a small, atomic change to the AST of a query. For example, for every (sub)component, there are respective add- and remove-edits, and for every attribute, there are respective set- and unset-edits. They are responsible for connecting every node in the graph, so they are inherently important to ensure termination. They do not have any conditions, apart from there actually being an empty space to fill, existing component to remove or variable to set/unset. Generally, they change the query semantics and therefore have a cost greater 0. This cost must represent the greatest semantic difference they can cause, even if this over-estimates the distance in other cases.

Shortcut edits rely on certain conditions in order to be applicable but in turn perform transformations, that in many cases would require a combination of multiple atomic operations, for a cost, that is lower than the cumulated cost of the corresponding atomic edits. This is visualized in Figure 5.

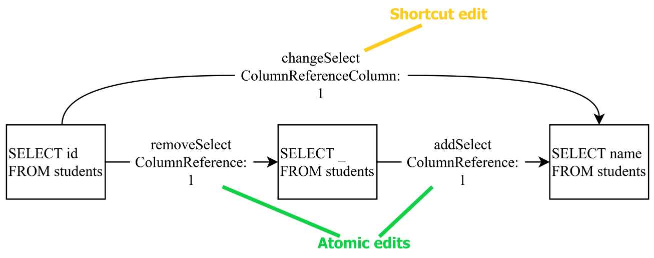

Shortcut edit
changeSelect ColumnReferenceColumn: 1
SELECT id FROM students
removeSelect ColumnReference: 1
SELECT _ FROM students
addSelect ColumnReference: 1
SELECT name FROM students
Atomic edits

Fig. 5: Two atomic edits with combined cost 2 vs. one shortcut edit with cost 1.

Horizontal edits are an intuitive subset of the shortcut edits. Their defining characteristic is that they only swap (sub)components or change values, but don't add/remove or set/unset. Examples include the shortcut edit from Figure 5, or swapping the positions of two `SELECT`-elements, or the associative law.

Equivalence edits are edits with a cost of 0. They connect those queries that are to be considered QSS-equivalent. They are mostly shortcut edits, but there might also be certain atomic edits among them, depending on the exact implementation of the AST and comparison goals. These edits can never be fully exhaustive, because otherwise, this approach would be able to decide query equivalence, which is impossible [AHV95; Ch17].

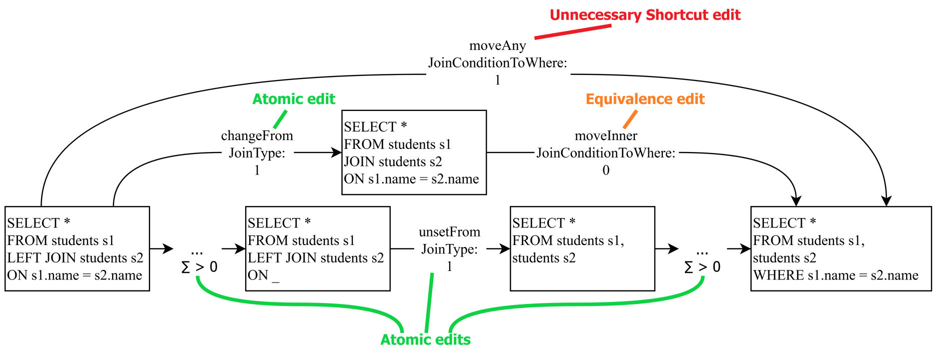
The exact costs of edits depend on the goals of the comparison, e.g., different priorities regarding certain syntactic or semantic differences (see Section 4.3.1).

4.4.3 Finetuning of edits

A base set of edits can be expanded for each use case with task-specific shortcut edits. But when introducing new shortcut edits, attention should be paid to the intended purpose:

If we, for example, consider resolving an `INNER JOIN` by creating a cross product with an associated `WHERE` clause, it's quite clear that a semantically equivalent query arises. A shortcut edit with a cost of 0, i.e., an equivalence edit, could be introduced. Such an edit should not apply to every `JOIN`, though. For example, in the case of a `LEFT JOIN`, there is no equivalence, and the costs for a potential shortcut edit would need to be higher than 0.

Introducing a separate shortcut edit without this precondition and in turn a non-zero cost may not be necessary if there already exists an atomic edit for converting a `LEFT JOIN` to an `INNER JOIN`, and a shortcut for relocating the `INNER JOIN`. Figure 6 illustrates this.



Unnecessary Shortcut edit
moveAny JoinConditionToWhere: 1
Atomic edit
changeFrom JoinType: 1
Equivalence edit
moveInner JoinConditionToWhere: 0
SELECT * FROM students s1 JOIN students s2 ON s1.name = s2.name
SELECT * FROM students s1 LEFT JOIN students s2 ON s1.name = s2.name
... Σ > 0
SELECT * FROM students s1 LEFT JOIN students s2 ON _
unsetFrom JoinType: 1
SELECT * FROM students s1, students s2
... Σ > 0
SELECT * FROM students s1, students s2 WHERE s1.name = s2.name
Atomic edits

Fig. 6: Shortcut edit, that is unnecessary because an equivalence edit combined with an atomic edit already account for it.

4.5 Algorithm

We use a shortest path algorithm to find the cheapest edit sequence between a start and target node. Dijkstra's algorithm, despite being a popular option, cannot be applied to implicit graphs in its commonly taught form [Fe11]. This is why we use uniform cost search as a basis. (See [Fe11] for basics.)

4.5.1 Search direction

Since our graph is directed, it is necessary to define a fixed direction for the shortest path algorithm. We can start with either the student-devised query or the reference solution.

Advantages for choosing the reference solution as the start and searching towards the student-devised solutions are: Uniform cost search builds a tree of shortest paths with the start node as the root. In theory, one run of the shortest path algorithm would be enough to find all the shortest paths to all student-devised solutions. A newly visited node could be compared with every (not yet found) student query until there are none left. If there is a maximum search distance after which a destination is simply considered “too dissimilar”, the single execution could even be run in advance to getting student-devised solutions: For a given start, all visited nodes and edges up to this distance are stored. Later, a given target can be compared to these nodes. If none match, it is marked as too dissimilar, otherwise the shortest path can be found by following the tree back to its root.

The disadvantages, however, outweigh these advantages: It seems unrealistic to try to generate every possible student solution, since no assumptions or even requirements can be made. The presented approach is designed to be able to handle as diverse input as possible, with the only requirement being that it can be parsed into a (possibly incomplete) AST. The reference solution, on the other hand, can reasonably be required to be executable.

This reduces the search space substantially, since no unreasonable values, references or constructs need to be generated, and existing ones can simply be replaced/removed. This includes, for example, references to tables or columns that do not exist in the schema, or illegal expressions such as an aggregation function in the `WHERE` clause. For this reason, we start with the student-devised input and search towards the reference solution.

4.5.2 Custom algorithm

The custom algorithm we use in Graph-based QSS is a variant of uniform cost search that better adapts to and exploits certain properties of the problem.

The most important property is the fixed set of edits and their fixed costs. Due to this, the distances to all possible neighbors of a node are known in advance. This information is used to lazily postpone the generation of these neighbors (= execution of the neighbor function) as long as possible:

When a node `v` becomes visited, because its distance `dist[v]` is the smallest of all unvisited nodes, uniform cost search would generate all of its neighbors, filter for those not visited yet, and then either insert them into the priority queue using `dist[v]` plus the edge cost as a tentative distance or update their existing tentative distance. Instead, `v` itself is inserted into the priority queue at `dist[v]` plus the cost of the lowest-cost edit, and `next_edit[v]` is set to this edit. When a node `u` is removed from the queue, because `dist[u]` plus the cost of `next_edit[u]` is the smallest queued distance, only `next_edit[u]` is applied and each generated neighbor `n` becomes visited with `dist[n]` set to the distance that `u` was queued at. **Effectively, not the neighbors are queued, but tuples of a node and edit that will eventually generate them.** Afterwards, `next_edit[u]` is set to the next-higher-cost edit and `u` is put back into the queue at `dist[u]` plus the cost of the updated `next_edit[u]`.

This way, neighbors are created just before being marked as visited, and `dist[]` values are always final, but nodes are queued multiple times. In the worst case, nodes will be visited as often as there are edits, but most will be visited less often. This is the big advantage of this algorithm, because not doing unnecessary edits also means not having to generate, process, and store unnecessary neighbors.

It is also desirable to only search for the destination within a certain distance. When trying to determine equivalence, it is only of interest whether the destination is within a distance of 0. For grading, it might be unnecessary to keep searching if start and destination are so dissimilar that no points are awarded anyway (see Section 4.7). This can be done by aborting as soon as the current distance has become greater than the maximum search distance.

Chandra et al. [Ch16; Ch19a; Ch19b; Ch21; CS22] propose using a greedy heuristic algorithm and so-called guided edits. While this reduces the search space and thus improves performance significantly, it can also make the result incorrect, so we decided against it.

4.6 Finite termination

It is already proven that if edge weights are greater than or equal to 0, and the target is reached (i.e., visited) by uniform cost search, it is reached via the shortest possible path [Fe11]. Since our algorithm behaves like uniform cost search, except that neighbors are generated as late as possible (see Section 4.5.2), it is trivial to apply the same invariants from the proof.

The first condition is that the edge weights are non-negative. This was specified in Section 4.4.1. The second condition is that the target is reached. This, in turn, requires that there is a path to find and that the algorithm finds it, i.e., it terminates in finite time.

4.6.1 Existence of a path

When considering whether a path exists, the atomic edits are essential. Half of the atomic edits are responsible for unsetting attributes, removing (sub)components, etc., and there is such an edit for every possible attribute/(sub)component. Combined with the support for arbitrary incomplete ASTs, by gradually unsetting and removing everything from the start, the fully incomplete AST, called "empty AST", can eventually be reached. And since any given start only has a finite size, this takes a finite number of steps.

The other half of the atomic edits is responsible for setting attributes, adding (sub)components, etc., and there is also an edit for each possible attribute/(sub)component. Combined with the support for arbitrary incomplete ASTs, by gradually setting and adding everything to the empty AST, the target can eventually be reached. And since any given target has a finite size, this takes a finite number of steps.

Therefore, there exists a path of finite length from any given start to any given destination.

4.6.2 Termination of the algorithm

If there is at least one path, the algorithm also has to find it in finite time. The main problem is the graph being implicit and infinitely large.

There are two points with potential for non-termination: The "main loop" that processes queued nodes and the generation of neighbors/execution of edits. Other loops, such as iterating over the set of edits, depend on the size of the input, which must be finite. Otherwise, passing arguments without even starting the algorithm would take infinitely long.

Termination-related considerations can be reduced to only the main loop: If the main loop has finite iterations/queue elements to process, then the edits have produced only a finite number of nodes. Because in order for an edit to produce infinitely many nodes, but the main loop then only having to process a finite subset of them, there are two options: either the edit would have to generate and discard infinitely many different nodes through the duplicate check, which implies infinitely many nodes have been visited through endless iterations of the main loop, contradicting the assumption; or, the same node would have to be created multiple times by the same edit, which defies the definition of edits returning a set of nodes. It is therefore sufficient to show that the main loop terminates.

The main loop can only terminate by reaching the goal (excluding the option of terminating early due to exceeding a maximum distance). This is because there always is at least one path, so it is impossible to run out of neighbors before reaching the goal. The danger of non-termination comes from having to process infinitely many nodes before this happens. For example, if there are infinitely many nodes with distance 0, but the destination has a distance of 1, it will never be reached. Since nodes are processed in the order of increasing distance from the start, this condition can be weakened to there being a finite number of nodes at all distances smaller than or equal to the distance to the destination.

This is ensured individually for certain portions of edits:

Atomic unset- and remove-edits: The number n of nodes possibly generated by atomic unset- and remove-edits is finite, because it is limited by the components in the AST.

Atomic set-edits: For atomic set-edits, the number n is upper-bounded by the number of possible values to the power of the number of attribute-places: $n \leq values^{places}$. The places are limited because of the finite AST size. But the number of values, especially for literals/constants, is not. A finite set of values must be determined based on those in the destination and the current node. Such information is passed to the edit as part of the meta-info argument.

Atomic add-edits: For atomic add-edits, the number n is upper-bound by the number of different components to the power of the number of places to add them times the number of executions: $n \leq components^{(places \times 2^{executions})}$. The power of 2 is due to components being nestable, i.e. adding a binary-expression creates 2 new places to add expressions. The number of different components is limited by the syntax. The places to add them are limited because of the finite input size. And because atomic add-edits generally have a cost greater than 0, the number of executions up to a limited distance is also limited, making n finite. However, the cost of any edit is configurable to 0. This allows for unlimited executions by growing the AST infinitely large. To solve this, the size of the AST is limited as described further below.

Horizontal equivalence edits: Equivalence edits, that don't change the components but only their positions, e.g., $p \land q \Leftrightarrow q \land p$, are innately limited by the number of possible combinations. Those, that change attributes, are limited by the same finite set of values as for atomic set-edits.

Equivalence edits, that don't increase the AST size: Equivalence edits, that eliminate or change components, but don't increase the AST size, are limited in the same way the atomic remove-, unset-, and set-edits are.

Equivalence edits, that do increase the AST size: Equivalence edits that do increase the AST size are critical. In particular if their output contains the input again, e.g., the right direction of $p \Leftrightarrow p \wedge p$, then they can be chained infinitely, like $p \Leftrightarrow p \wedge p \Leftrightarrow p \wedge p \wedge p \Leftrightarrow p \wedge p \wedge p \wedge p \Leftrightarrow \dots$.

Applying these equivalences only in the simplifying direction, e.g., only the left direction of $p \Leftrightarrow p \wedge p$, would break correctness. For example, assume the destination contains $p \wedge p'$, with p being similar to p', such that transforming the former into the latter is cheaper than creating p' from scratch. If the start now only contains p, then using the right direction of $p \Leftrightarrow p \wedge p$ to turn it into $p \wedge p$, followed by said transformation to get $p \wedge p'$, is cheaper than the alternative (see Figure 7). Applying transformations on the destination instead, e.g., p' to p and then $p \wedge p$ to p, to cover such cases, has been ruled out in Section 4.5.1.

Equivalence edits, that increase the AST size, are instead limited by limiting the AST size itself. This can be done via the number of components, the height and degree of the tree, or both. The information is passed to edits as part of the meta-info argument. The exact values depend on the destination and all edits, because it has to be assured that no shortest path is obstructed. Since custom edits might be involved, this cannot be further determined here.

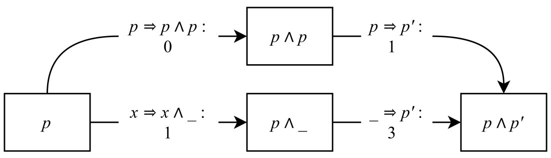


Fig. 7: The equivalence edit "$p \Rightarrow p \wedge p$" and transformation "$p \Rightarrow p'$" being cheaper than introducing the binary expression "$\wedge$" and creating "p'" from scratch.

This way, the number of nodes with a distance smaller than or equal to that of the destination is finite, so the algorithm reaches the destination and terminates in finite time.

4.7 Grading

The result of the shortest path algorithm is the smallest total distance between the start and destination query. To use it for grading, this value is subtracted from the maximum possible points for that task. Since the distance can be large, the points are lower-bounded to 0. If the cost of the edits, and therefore the total distance, is of a different order of magnitude than the maximum points for the tasks, a constant scaling factor can be applied before subtraction. The grading formula is:

$$points = max(maxPoints - distance * scale,\ 0)$$

This approach can also be used to determine a reasonable value for this maximum number of points for an assignment. Since incomplete ASTs can be processed, the "empty AST" can be given as a start. The total distance to the reference solution quantifies the work to build the target from scratch, and is therefore a good “difficulty” measure.

5 Evaluation

Our approach has been evaluated to see if it theoretically fulfills the requirements of Section 2. In addition, a prototype implementation was created and a survey was conducted to see if it could be successfully implemented in practice with respect to fairness and comprehensibility, as well.

5.1 Conceptual evaluation

The concept from Section 4 delivers on all goals set in Section 2:

Quantified Semantic Similarity (QSS): The primary goal is satisfied by the underlying idea specified in Section 4.2. Queries are treated like nodes in a graph. The edges between them are implicitly given by edits to those queries. Edits, and therefore edges, have costs associated with them that represent the semantic error or dissimilarity caused by them. On this graph, a shortest path algorithm can be used to find the lowest cost edit sequence between any two nodes (see Section 4.6). The sum of the costs of these edits represents how semantically dissimilar the queries are. A cost of 0 means that the queries are QSS-equivalent. Otherwise, the higher the value, the less similar they are.

Meaningful feedback: Instead of combining all edits by their cost, they are considered individually. Each edit can be described in natural language. Listing the descriptions of the edges along the path serves as the required feedback, because they either prove why the queries are equivalent or they explain what would be needed in order to make them equivalent.

Guaranteed result: We prove in Section 4.6 that the approach meets the goal of always yielding a result. The atomic edits ensure that there is a path, while the number of nodes to be visited before reaching the destination is finite.

Unrestricted input: The ASTs, which form the nodes of the graph, can represent any SQL construct. In addition, they can represent construct or value combinations that are non-executable or even incomplete. While it makes sense to restrict the target to executable queries to limit the search space (see Section 4.5.1), the start is not restricted in this way.

Configurability and extensibility: The cost of any edit can be configured individually. This enables the approach to adapt to any teaching style or priorities. Further, even custom edits can be added to account for task-specific cases.

5.2 Prototype implementation

In order to prove that Graph-based QSS works in practice, a prototype implementation has been created (https://github.com/FAU-CS6/sql-query-distance).

It was written in TypeScript to eventually be used in a browser-based plugin. At the time of writing, it features 181 edits, 94 of which are the atomic edits. The rest of them are the shortcut edits that enable detection of semantical equivalence and improve the quality of the calculated distance in general. All of them have descriptions in natural language used to provide meaningful feedback. Also, they have adjustable costs and are extensible.

5.3 Survey

We conducted a survey to evaluate Graph-based QSS by comparing it against two other techniques, dynamic analysis and manual grading, in terms of two metrics, fairness and comprehensibility.

The survey consisted of 11 scenarios, taken from our own online SQL grading tool that is based on dynamic analysis. Like in a typical exam question, students were asked to formulate a textual query description based on a provided database schema as an SQL query.

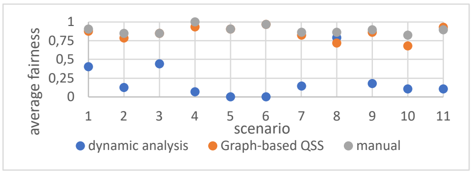
To be as realistic as possible, we took real answers from real students. In addition to the dynamic analysis-based grading provided by the the tool, we asked experienced tutors to manually grade the student answers and calculated grades with our prototype implementation. This gave us three different gradings for each student answer.

In the survey, we asked 20 participants (independent of the students and tutors mentioned above, but also with good knowledge of SQL) to rate the gradings in each scenario with respect to fairness and comprehensibility. The possible answers “low”, “medium”, and “high” were linearly mapped to a scale from 0 to 1 to facilitate quantitative analysis. The results can be found in Table 1 and Figure 8.

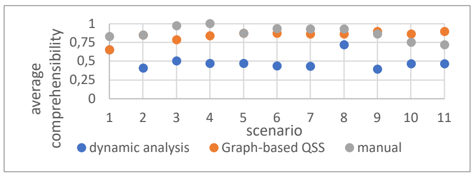
technique	$\bar{f}$	s_f	$\bar{c}$	s_c
dynamic analysis	$0,2186$	$0,3611$	$0,4940$	$0,4238$
Graph-based QSS	$0,8473$	$0,2608$	$0,8293$	$0,2935$
manual	$0,8892$	$0,2283$	$0,8743$	$0,2608$

Tab. 1: The overall sample means of fairness $\bar{f}$ and comprehensibility $\bar{c}$, as well as their (uncorrected) sample standard deviations s_f and s_c, respectively, for each technique.

Table 1 summarizes the answers from all 20 participants: Overall, the fairness of Graph-based QSS was rated 287.7% higher than dynamic analysis and just 4.2% lower than manual grading. Also, it is shown to be 1.68 times as comprehensible as dynamic analysis, or 94.9% as much as manual grading.



(a) The average fairness of each technique.



(b) The average comprehensibility of each technique.

Fig. 8: The summarized survey results per scenario.

Figure 8a shows that, across all scenarios, dynamic analysis mostly performs quite poorly when it comes to fairness, while Graph-based QSS consistently scores almost as well as manual grading, if not slightly better. This means that even in a prototypical stage, the implementation from Section 5.2 can already compete with manual grading in terms of perceived fairness, indicating a high quality of query comparison.

Figure 8b paints a similar picture regarding comprehensibility, but with Graph-based QSS even significantly outperforming manual grading in the last two scenarios. This stands testament to the powerful semantic feedback from Section 5.1.

Scenario 8 being an outlier in both metrics stems from an ambiguous task description, which later turned out to even divide our team as to what should be considered a “correct” answer. Since this is an issue separate from query comparison techniques, it will not be investigated further.

6 Discussion

Our prototypical implementation provides only a subset of edits and does not yet support all SQL features. While the set of equivalence edits can never be fully exhaustive, it makes sense to extend them as much as possible, as this will automatically improve the accuracy for non-equivalent queries as well. Adding support for the missing SQL features is just a matter of time and effort.

Another problem with the prototypical implementation is the performance for complex queries, especially if they have a large semantic distance, since the search space grows exponentially. We avoided canonicalization as part of the node comparison because of the risk of removing important syntactic distinctions, but didn’t introduce an equally good way to save performance. There is great potential for parallelization, especially when applying edits, which is untapped in our TypeScript-based implementation.

A possible performance improvement might be to use a different shortest path algorithm. For example, one could use only the atomic edits to quickly build an in initial path, which gives an upper bound for the distance. Then, shortcut edits could be used to continuously search for cheaper variations. This converges towards the correct result, but can be aborted early to get an approximated, but still valid, path.

While Graph-based QSS can handle non-executable queries, it cannot handle unparsable ones. However, this could be achieved by combining it with other comparison techniques. For example, string similarity, as suggested by Wang et al. [Wa20], could be used as a backup solution for query descriptions that cannot even be parsed into an incomplete AST. In this case, however, care must be taken to ensure that such backup solutions never produce better grades than the base comparison technique, as this might incentivize students to deliberately produce "broken" answers in order to get a better grade.

A comparison with Chandra et al. [Ch16; Ch19a; Ch19b; Ch21; CS22] as part of our survey was planned, but couldn't be implemented due to technical problems with their implementation. As they are the closest to our approach, this would have been a very interesting comparison. Also, we have to admit that the number of 20 participants in our survey is comparatively small and there is no way to prove the representativeness of the study group. However, due to the very clear trends across all scenarios, we believe that this survey is at least a strong indicator in favor of the presented approach.

7 Conclusion

Our goal in this paper was to develop a method for determining the QSS of two queries in a way that is configurable and extensible, while always yielding a result and not limiting input to a small subset of the query language. If needed, the method also provides meaningful feedback to the user.

We proposed Graph-based QSS, an approach that treats queries as nodes and edits as edges. The lowest-cost sequence of edits required to transform one query into another is then determined by applying a shortest path algorithm. The sum of the costs of these edits is the dissimilarity of the queries. The edits are configurable and extensible, making the approach adjustable to different contexts and points of focus. The approach is not limited to a subset of SQL and can even be extended by incomplete ASTs. In turn, QSS-equivalence can only approximate pure theoretical query equivalence, but never fully reach it.

We proved finite termination of the approach and showed its feasibility and practicality by implementing a prototype for the comparison of student SQL queries against a reference solution and conducting a survey on its perceived fairness and comprehensibility. We are confident that Graph-based QSS can be applied to other domains as well, and that it can be extended to support more complex queries and other query languages.

Although the results of the survey have already been very promising, we plan to refine the prototype for enhanced SQL support and robustness against incomplete queries. Additionally, the prototype will be integrated into an automated grading system, providing a long-term test beyond our initial survey.

References

[AHV95] Abiteboul, S.; Hull, R.; Vianu, V.: Foundations of Databases. Addison-Wesley, 1995, ISBN: 0-201-53771-0.

[ASU79] Aho, A. V.; Sagiv, Y.; Ullman, J. D.: Equivalences Among Relational Expressions. SIAM J. Comput. 8 (2), pp. 218–246, 1979, DOI: 10.1137/0208017, URL: https://doi.org/10.1137/0208017.

[Bh15] Bhangdiya, A.; Chandra, B.; Kar, B.; Radhakrishnan, B.; Reddy, K. V. M.; Shah, S.; Sudarshan, S.: The XDa-TA system for automated grading of SQL query assignments. In (Gehrke, J.; Lehner, W.; Shim, K.; Cha, S. K.; Lohman, G. M., eds.): 31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015. IEEE Computer Society, pp. 1468–1471, 2015, DOI: 10.1109/ICDE.2015.7113403, URL: https://doi.org/10.1109/ICDE.2015.7113403.

[BMT19] Bonifati, A.; Martens, W.; Timm, T.: Navigating the Maze of Wikidata Query Logs. In (Liu, L.; White, R. W.; Mantrach, A.; Silvestri, F.; McAuley, J. J.; Baeza-Yates, R.; Zia, L., eds.): The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019. ACM, pp. 127–138, 2019, DOI: 10.1145/3308558.3313472, URL: https://doi.org/10.1145/3308558.3313472.

[Ch15] Chandra, B.; Chawda, B.; Kar, B.; Reddy, K. V. M.; Shah, S.; Sudarshan, S.: Data generation for testing and grading SQL queries. VLDB J. 24 (6), pp. 731–755, 2015, DOI: 10.1007/s00778-015-0395-0, URL: https://doi.org/10.1007/s00778-015-0395-0.

[Ch16] Chandra, B.; Joseph, M.; Radhakrishnan, B.; Acharya, S.; Sudarshan, S.: Partial Marking for Automated Grading of SQL Queries. Proc. VLDB Endow. 9 (13), pp. 1541–1544, 2016, DOI: 10.14778/3007263.3007304, URL: http://www.vldb.org/pvldb/vol9/p1541-chandra.pdf.

[Ch17] Chu, S.; Wang, C.; Weitz, K.; Cheung, A.: Cosette: An Automated Prover for SQL. In: 8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings. www.cidrdb.org, 2017, URL: http://cidrdb.org/cidr2017/papers/p51-chu-cidr17.pdf.

[Ch18] Chu, S.; Murphy, B.; Roesch, J.; Cheung, A.; Suciu, D.: Axiomatic Foundations and Algorithms for Deciding Semantic Equivalences of SQL Queries. Proc. VLDB Endow. 11 (11), pp. 1482–1495, 2018, DOI: 10.14778/3236187.3236200, URL: http://www.vldb.org/pvldb/vol11/p1482-chu.pdf.

[Ch19a] Chandra, B.; Banerjee, A.; Hazra, U.; Joseph, M.; Sudarshan, S.: Automated Grading of SQL Queries. In: 35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019. IEEE, pp. 1630–1633, 2019, DOI: 10.1109/ICDE.2019.00159, URL: https://doi.org/10.1109/ICDE.2019.00159.

[Ch19b] Chandra, B.; Banerjee, A.; Hazra, U.; Joseph, M.; Sudarshan, S.: Edit Based Grading of SQL Queries, 2019, arXiv: 1912.09019, URL: http://arxiv.org/abs/1912.09019.

[Ch21] Chandra, B.; Banerjee, A.; Hazra, U.; Joseph, M.; Sudarshan, S.: Edit Based Grading of SQL Queries. In (Haritsa, J. R.; Roy, S.; Gupta, M.; Mehrotra, S.; Srinivasan, B. V.; Simmhan, Y., eds.): CODS-COMAD 2021: 8th ACM IKDD CODS and 26th COMAD, Virtual Event, Bangalore, India, January 2-4, 2021. ACM, pp. 56–64, 2021, DOI: 10.1145/3430984.3431012, URL: https://doi.org/10.1145/3430984.3431012.

[CS22] Chandra, B.; Sudarshan, S.: Automated Grading of SQL Queries. IEEE Data Eng. Bull. 45 (3), pp. 17–28, 2022, URL: http://sites.computer.org/debull/A22sept/p17.pdf.

[DM11] Dollinger, R.; Melville, N. A.: Semantic evaluation of SQL queries. In: IEEE International Conference on Intelligent Computer Communication and Processing, ICCP 2011, Cluj-Napoca, Romania, August 25-27, 2011. IEEE, pp. 57–64, 2011, DOI: 10.1109/ICCP.2011.6047844, URL: https://doi.org/10.1109/ICCP.2011.6047844.

[Do19] Domínguez, C.; Elizondo, A. J.; Heras, J.; Izquierdo, F. J. G.: The Effects of Adding Non-Compulsory Exercises to an Online Learning Tool on Student Performance and Code Copying. ACM Trans. Comput. Educ. 19 (3), 16:1–16:22, 2019, DOI: 10.1145/3264507, URL: https://doi.org/10.1145/3264507.

[FDF20] Fabijanic, M.; Dambic, G.; Fulanovic, B.: A Novel System for Automatic, Configurable and Partial Assessment of Student SQL Queries. In (Koricic, M.; Skala, K.; Car, Z.; Cicin-Sain, M.; Sruk, V.; Skvorc, D.; Ribaric, S.; Jerbic, B.; Gros, S.; Vrdoljak, B.; Mauher, M.; Tijan, E.; Katulic, T.; Pale, P.; Grbac, T. G.; Fijan, N. F.; Boukalov, A.; Cisic, D.; Gradisnik, V., eds.): 43rd International Convention on Information, Communication and Electronic Technology, MIPRO 2020, Opatija, Croatia, September 28 - October 2, 2020. IEEE, pp. 832–837, 2020, DOI: 10.23919/MIPRO48935.2020.9245264, URL: https://doi.org/10.23919/MIPRO48935.2020.9245264.

[Fe11] Felner, A.: Position Paper: Dijkstra's Algorithm versus Uniform Cost Search or a Case Against Dijkstra's Algorithm. In (Borrajo, D.; Likhachev, M.; López, C. L., eds.): Proceedings of the Fourth Annual Symposium on Combinatorial Search, SOCS 2011, Castell de Cardona, Barcelona, Spain, July 15.16, 2011. AAAI Press, pp. 47–51, 2011, DOI: 10.1609/SOCS.V2I1.18191, URL: https://doi.org/10.1609/socs.v2i1.18191.

[KTH13] Kleiner, C.; Tebbe, C.; Heine, F.: Automated grading and tutoring of SQL statements to improve student learning. In (Laakso, M.; Simon, eds.): 13th Koli Calling International Conference on Computing Education Research, Koli Calling '13, Koli, Finland, November 14-17, 2013. ACM, pp. 161–168, 2013, DOI: 10.1145/2526968.2526986, URL: https://doi.org/10.1145/2526968.2526986.

[NAJ17] Nalintippayawong, S.; Atchariyachanvanich, K.; Julavanich, T.: DBLearn: Adaptive e-learning for practical database course - An integrated architecture approach. In (Hochin, T.; Hirata, H.; Nomiya, H., eds.): 18th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, SNPD 2017, Kanazawa, Japan, June 26-28, 2017. IEEE Computer Society, pp. 109–114, 2017, DOI: 10.1109/SNPD.2017.8022708, URL: https://doi.org/10.1109/SNPD.2017.8022708.

[PH20] Panni, F. A. K.; Hoque, A. S. M. L.: A Model for Automatic Partial Evaluation of SQL Queries. In: 2020 2nd International Conference on Advanced Information and Communication Technology (ICAICT). IEEE, pp. 240–245, 2020, DOI: 10.1109/ICAICT51780.2020.9333475.

[PL04] Prior, J. C.; Lister, R.: The backwash effect on SQL skills grading. Ed. by Boyle, R. D.; Clark, M.; Kumar, A. N., pp. 32–36, 2004, DOI: 10.1145/1007996.1008008, URL: https://doi.org/10.1145/1007996.1008008.

[SM15] Stajduhar, I.; Mausa, G.: Using string similarity metrics for automated grading of SQL statements. In (Biljanovic, P.; Butkovic, Z.; Skala, K.; Mikac, B.; Cicin-Sain, M.; Sruk, V.; Ribaric, S.; Gros, S.; Vrdoljak, B.; Mauher, M.; Sokolic, A., eds.): 38th International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2015, Opatija, Croatia, May 25-29, 2015. IEEE, pp. 1250–1255, 2015, DOI: 10.1109/MIPRO.2015.7160467, URL: https://doi.org/10.1109/MIPRO.2015.7160467.

[So06] Soler, J.; Prados, F.; Boada, I.; Poch, J.: A Web-based tool for teaching and learning SQL. In: International Conference on Information Technology Based Higher Education and Training, ITHET. 2006.

[Wa17] Wahl, A. M.; Endler, G.; Schwab, P. K.; Herbst, S.; Lenz, R.: Query-Driven Knowledge-Sharing for Data Integration and Collaborative Data Science. In (Kirikova, M.; Nørvåg, K.; Papadopoulos, G. A.; Gamper, J.; Wrembel, R.; Darmont, J.; Rizzi, S., eds.): New Trends in Databases and Information Systems - ADBIS 2017 Short Papers and Workshops, AMSD, BigNovelTI, DAS, SW4CH, DC, Nicosia, Cyprus, September 24-27, 2017, Proceedings. Vol. 767. Communications in Computer and Information Science, Springer, pp. 63–72, 2017, doi: 10.1007/978-3-319-67162-8_8.

[Wa20] Wang, J.; Zhao, Y.; Tang, Z.; Xing, Z.: Combining Dynamic and Static Analysis for Automated Grading SQL Statements. Journal of Network Intelligence 5 (4), pp. 179–190, 2020.

[WBH24] Wanjiru, B.; van Bommel, P.; Hiemstra, D.: Sensitivity of Automated SQL Grading in Computer Science Courses. In (Daimi, K.; Al Sadoon, A., eds.): Proceedings of the Third International Conference on Innovations in Computing Research (ICR'24). Springer Nature Switzerland, Cham, pp. 283–299, 2024, isbn: 978-3-031-65522-7.

[Zh19] Zhou, Q.; Arulraj, J.; Navathe, S. B.; Harris, W.; Xu, D.: Automated Verification of Query Equivalence Using Satisfiability Modulo Theories. Proc. VLDB Endow. 12 (11), pp. 1276–1288, 2019, doi: 10.14778/3342263.3342267, url: http://www.vldb.org/pvldb/vol12/p1276-zhou.pdf.