JSON Processing and Error Correction in SQL Functions

Andrea Meleková
University of Zilina
Žilina, Slovakia
melekova2@stud.uniza.sk

Michal Kvet
University of Zilina
Žilina, Slovakia
Michal.Kvet@uniza.sk

***Abstract*— This paper presents a tool specifically designed for relational databases, particularly Oracle, to improve SQL query accuracy by automatically detecting and correcting typographical errors, focusing on SQL functions. By extracting schema information (e.g., table structures, column attributes, and function signatures) and converting it into JSON format, the tool automates the correction of errors in SQL functions and parameters using algorithms such as Levenshtein distance. By analyzing metadata retrieved through SQL queries, such as SELECT * FROM USER_TAB_COLUMNS and SELECT * FROM ALL_OBJECTS WHERE OBJECT_TYPE = 'FUNCTION', the system reduces manual correction efforts while improving query reliability. This paper explores how the tool automates function signature correction and parameter validation, contributing to higher SQL query accuracy for both educational and practical applications.**

Keywords*— *SQL function parameters, relational databases, JSON validation, Oracle, automated SQL correction

I. INTRODUCTION

In database management, SQL is the most widely used query language for interacting with relational databases. However, even small typographical errors—such as misspelled function names or incorrect parameter orders—can cause SQL queries to fail or produce inaccurate results, especially in educational environments where students are learning SQL with complex databases like Oracle.

This work presents an automated tool specifically designed for relational databases to correct SQL functions and parameters, enhancing query accuracy. By extracting metadata from an Oracle database, such as tables, columns, and functions, and converting it into JSON format, the tool automates the detection and correction of errors in SQL queries. This significantly reduces manual intervention, which is particularly valuable in educational and production environments where debugging can impact performance and learning outcomes.

The tool uses algorithms like Levenshtein distance for error detection in function names and parameters. Levenshtein's string-matching capabilities allow for the automatic correction of typographical errors, improving SQL query reliability without manual correction efforts. This paper demonstrates how the system automates the validation and repair of SQL functions and explores how it handles errors related to them, which are crucial in complex query execution.

Process automation is essential for minimizing manual work and improving query processing efficiency. Existing tools like XDa-TA [4][5] evaluate SQL statements for correctness, but often lack automatic error correction capabilities, limiting their usefulness in larger projects. Our approach combines the flexibility of the JSON format with automated error correction, focusing on how metadata about tables and functions is stored and validated, particularly in Oracle databases. The use of Levenshtein distance allows the system to efficiently detect and correct function-related errors in SQL queries [6], improving accuracy and reliability.

This system aligns with trends in automation, reducing human error and simplifying the handling of large databases. By automating the correction of typographical errors, it improves both the accuracy and reliability of SQL queries in relational database systems.

II. STATE OF THE ART

Automated error correction in SQL statements is a key challenge in big data processing and cloud systems. Errors in SQL statements, such as typos or incorrect parameters, can lead to query failures and cause unnecessary burden on developers who must manually correct these errors. A number of solutions have been developed to address this issue using various text processing and autocorrection techniques. These solutions can be divided into several categories.

A. Using distance methods to correct errors

Distance methods such as Levenshtein distance are widely used in the field of text processing for error detection and correction. These algorithms measure the differences between two strings based on operations such as insertion, deletion or replacement of characters. Levenshtein distance is effective in correcting typographical errors in text strings, such as function names in SQL statements, but also in other applications, including genetic sequences or search algorithms.

Levenshtein distance has proven useful in applications where it is important to correct typographical errors in text strings, including SQL statements, programming languages, search algorithms, and biological sequences. This algorithm identifies minimal differences between strings and enables efficient automatic error correction without human intervention [6].

The Damerau-Levenshtein distance is an extension of the basic Levenshtein distance and includes transpositions of two adjacent features. This method has proven successful in detecting more complex errors, such as the swapping of two characters in text data, which is particularly useful when correcting typos in SQL function names and parameters. Damerau-Levenshtein distance has been used in many applications where not only simple typos need to be corrected, but also errors in character order, for example, when correcting misspelled identifiers or variables [7] .

Jaccard distance aims at comparing two sets of data and measures the similarity between them. Jaccard distance is often used in text analysis and retrieval algorithms, where it is necessary to determine to what extent two sets overlap. This approach is particularly useful for comparing large datasets and detecting similarities between two SQL queries, where it may be necessary to verify that the parameters or values of the

XXX-X-XXXX-XXXX-X/XX/$XX.00 ©20XX IEEE

SQL queries are consistent [8]. Nevertheless, Jaccard distance is not as accurate in detecting errors at the single-character level and is therefore not ideal for correcting typos.

Hamming distance is a technique that measures the differences between two strings of the same length. Hamming distance is mainly used in coding where it is important to detect errors in fixed formats such as codes or identifiers. In SQL statements, Hamming distance is of limited use because table or function names can be of different lengths, so it is more commonly used in applications where strings are of fixed length [9].

B. Comparison of Distance Algorithms in SQL Error Detection

AfterDifferent distance algorithms offer unique advantages depending on the type of error in SQL queries. Levenshtein distance provides efficient error correction for SQL queries by detecting minimal differences in function names and parameters, making it well-suited for typographical error correction.. The formula for Levenshtein distance is:

$$\operatorname{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j)=0, \\ \min \begin{cases} \operatorname{lev}_{a,b}(i-1,j)+1 \\ \operatorname{lev}_{a,b}(i,j-1)+1 \\ \operatorname{lev}_{a,b}(i-1,j-1)+1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

Fig. 1. Lavenstein distance formule

The Damerau-Levenshtein distance adds the ability to detect adjacent transpositions, such as when SEVR is typed instead of SERV. This makes it more effective in situations where character order is frequently mistyped. However, it is computationally more intensive than Levenshtein distance for longer strings.

Jaccard distance is useful for comparing larger sets of parameters or values in SQL queries, rather than detecting individual character errors

Hamming distance is limited to strings of the same length, which reduces its applicability in SQL error correction, where function names and parameters often vary in length.

C. Using JSON files for data storage and validation

In the case of validating SQL statements and their parameters, The use of JSON for data validation adds flexibility and reduces the need for manual intervention, streamlining SQL query corrections. For example, if there is a change in the database structure, the data can be updated in a JSON file and then used to validate SQL statements in real time [2].

In practice, JSON is also used to store configurations in programming languages such as Java or Python, where it enables dynamic data management. For example, in NoSQL databases such as MongoDB, JSON serves as a native format for storing documents, allowing high flexibility when working with dynamic data [4]. Compared to traditional formats such as XML, JSON offers easier data serialization and deserialization, making it a suitable tool for validating SQL statement parameters and for storing metadata in database systems [2] [4].

D. Automated systems for SQL statement correction

Automated SQL statement repair systems are concerned with detecting, validating, and correcting syntax errors and logical incorrectness in SQL queries. The most prominent tools in this area include XDa-TA, SQLTutor, and AutoGrader. These systems offer different approaches for error detection, statement validation, and user feedback, while differing in the degree of error correction automation and usability.

1) XDa-TA

XDa-TA is an advanced system that deals with the analysis and correction of SQL statements based on advanced syntactic and logical rules. XDa-TA uses predefined patterns and syntactic analysis rules to identify misused functions, missing tables or incorrectly assigned parameters. The main advantage of XDa-TA is its ability to analyze the structure of SQL queries and provide contextual feedback.

XDa-TA uses syntax detection algorithms that rely on syntax tree generation rules. This tree represents the structure of an SQL statement and allows the system to identify incorrect ordering of functions, incorrectly specified function or parameter names, and other SQL query syntax errors. Unlike XDa-TA, which focuses on feedback and query structure analysis, our tool provides fully automated correction, reducing the need for manual intervention. [4].

2) SQLTutor

Unlike SQLTutor or AutoGrader, which focus on feedback, our tool offers fully automated correction, making it more suitable for complex database environments.This system was designed to compare the results of student queries against correct solutions and provide basic feedback on syntax and logic errors [11].

3) AutoGrader

AutoGrader is another system that is designed to automatically evaluate SQL queries, especially in an academic environment. AutoGrader analyzes SQL queries based on syntactic and logical rules and provides immediate feedback to students on their results [12] [13].

Compared to XDa-TA, AutoGrader does not have advanced tools for detecting structural errors in SQL queries or automatic repair. The main advantage of AutoGrader is its fast feedback on the correctness of query outputs, but its use is limited where automatic syntax and function corrections are needed.

4) Comparison of Automated Error Correction Methods and Tools

Distance algorithms, such as Levenshtein distance, enable fully automated correction of textual errors at the individual character level. This is particularly advantageous when SQL statements are structured incorrectly or contain typos in function or table names. For instance, Levenshtein distance can detect and correct small but significant differences, such as mistyping TOCAR instead of TO_CHAR, by measuring the minimum number of edits needed to match the correct function name [6].

In addition to distance algorithms, the use of JSON for data validation adds another layer of flexibility. JSON files store

dynamic information about database structures, such as function names and parameters, which the system uses to validate SQL queries. This allows for efficient parameter checking without needing to repeatedly access the database [1], [2].

While systems like XDa-TA, SQLTutor, and AutoGrader provide feedback on the correctness of SQL queries, they do not offer automated correction. In contrast, the integration of distance algorithms like Levenshtein distance in this system automates the correction of typographical errors in SQL function names and parameters at the character level. This approach ensures a higher degree of automation compared to traditional tools, which only provide validation feedback.

The combination of distance algorithms with JSON validation enhances the automation process even further. JSON dynamically stores database structure and function information, making it easier to check parameters and validate SQL queries without manual intervention. This provides significant improvements in efficiency and accuracy over traditional validation systems, which rely heavily on user input for error correction [5], [6].

III. Working with JSON in the DatabaseInfoToJson

The DatabaseInfoToJSON class is a core component of the system, written in Java, that handles various operations involving database schema information. This class includes methods that communicate with the database to extract schema details, such as table structures, function names, and parameters, and return the relevant data. It also saves the extracted information into JSON format for validation and correction purposes. Additionally, the class reads from JSON files and loads the data into hashmaps, enabling efficient processing during SQL query validation.

A. Use of JSON files in a database system

In the context of SQL statements, JSON is used to store metadata about function names and parameters. This metadata is stored in a JSON file, which serves as a reference source for validating SQL statements. The system compares the entered SQL statements with the reference data stored in the hashmap, allowing immediate validation of the statements without needing to repeatedly access the database.

B. Validation and correction of SQL statements

Each SQL statement contains associated parameters and functions stored in a JSON file. This data can be manually entered or automatically generated based on the database structure. If the database structure changes, the JSON file is updated to include the new parameter and function information.

When an SQL statement is executed, the system parses the statement and compares its components—such as table names, functions, and parameters—with the information stored in the JSON files. This process allows the system to detect errors, such as typographical errors in function names or incorrectly specified parameters. If errors are detected, the system suggests a correction and updates the SQL statement based on the data in the JSON files, thereby improving its accuracy and reliability.

C. Detection Mechanism and Algorithm

The error detection process begins by retrieving the relevant schema information from the database using SQL queries, such as:

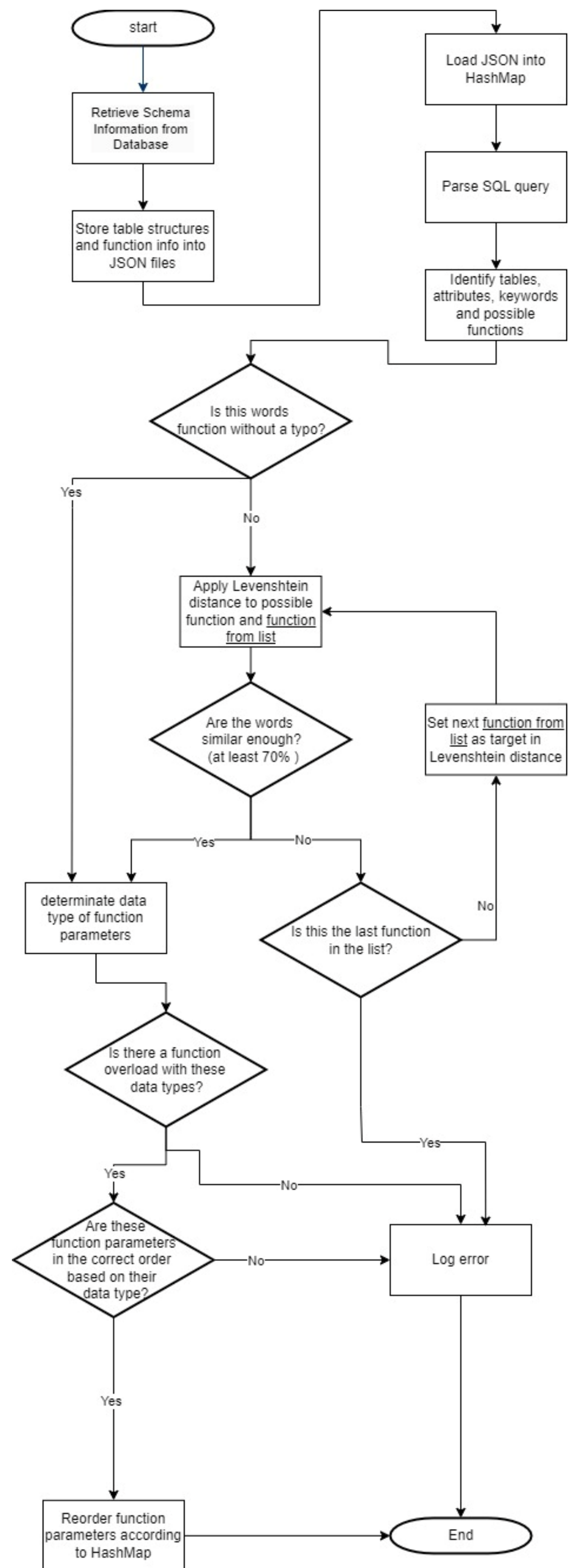

Fig. 2. Flow diagram of function processing

```sql
SELECT
    object_name,
    overload,
    data_type,
    POSITION
FROM
    all_arguments
WHERE
    object_type = 'FUNCTION'
    AND in_out = 'IN'
ORDER BY
    object_name,
    overload,
    POSITION;
```

Fig. 3 Select to get function and parameters from database

The condition AND in_out = 'IN' in this SQL query filters the results to show only the input parameters of the functions. In database systems like Oracle, functions and procedures can have parameters defined as input (IN), output (OUT), or both (IN OUT).

This information is stored in JSON format and loaded into hashmaps. When processing an SQL query, the system parses each word in the query, determining whether the word is a table name, function, alias, or something else. If the word is not recognized, the system applies a string distance algorithm, such as the Levenshtein distance algorithm, to determine the closest match with a similarity threshold of 70%. This ensures that even minor typographical errors in function names or table attributes are automatically corrected.

For example, if a function is mistyped as ROUNDD instead of ROUND, the system compares the function name with the stored values from the JSON file, calculates the similarity using the Levenshtein distance, and corrects it to ROUND.

Once functions are identified and corrected, the system checks the parameters passed to these functions. It first tries to determine the parameter type. If the parameter refers to a column, the system retrieves the data type from the hashmap. If the parameter is not a column, the system attempts to parse it and determine the data type. It then checks if the parameter matches any valid variation of the function and, if so, corrects the order of parameters. If the parameters do not match any known variation, the system logs the issue for manual review.

This step-by-step approach ensures that the system can automatically detect and correct errors in SQL statements, improving both the efficiency and accuracy of database queries.

D. *Updating JSON Files and Dynamic Checking*

A key feature of the system is its ability to dynamically update JSON files. If there are changes to the database structure or if new functions or parameters are added, the JSON files are updated to reflect these changes. An automated script tracks database versions and modifies the JSON files accordingly. This ensures that the reference data used for validation remains up-to-date and accurate at all times.

IV. Correcting Typos in Functions and Parameters

Typo correction in functions is implemented using a Levenshtein distance based algorithm. This algorithm compares the given function names with the correct names stored in the database and determines the minimum distance between them. If a similarity is detected, the system automatically suggests a typo correction.

Functions in SQL statements are compared against the database of functions stored in a JSON file to detect errors. When incorrect names are identified, the system suggests corrections based on the Levenshtein distance. When the similarity level is high (e.g. 70%), the system corrects the specified function name. This threshold can be adjusted based on specific requirements. Words in quotation marks, column or table names are excluded from this process to avoid incorrect corrections.This step focuses only on the functions themselves, regardless of attributes.

V. Processing Funtion Parameters

Processing function parameters is an important step in parsing SQL statements. This process ensures that functions use correct and valid parameters that are validated against the values stored in the JSON file.

Initially, all parameters in function calls need to be identified. This step involves parsing the SQL statements looking for all occurrences of the functions. For each function in the statement, we extract all the values specified in parentheses that represent parameters. In this process, we need to pay attention to the different formats and types of data that can be specified in the parameters.

Once the parameters are identified, a validation step follows to ensure that the parameters are valid and match the expected values. We verify that the parameters are entered in the correct format and that they are compatible with the expected data types.

The processing of parameters is different for different types of functions, so it is important to ensure that the system can handle different cases. Special processing is required for functions that use asterisk (for example, aggregation functions), where parameters may take different forms and require special validation. These functions are checked separately to see if they can contain asterisk. Checking and processing functions that allow optional parameters or a variable number of parameters ensures that all variants are correctly processed. All overloading types are always stored in a JSON file, so we cover all combinations and options of parameters

All parameter issues are documented within the logs to ensure transparency and allow tracking of errors and their solutions.

VI. Example of JSON Processing and Function Correction

This section focuses in detail on the processing of data from JSON files and the subsequent correction of typos in function names and parameters in SQL statements. The information from JSON files is used for automated correction and optimization of SQL code.

A. *JSON file processing*

JSON files store important metadata about the functions in the database, including function names and their parameters. This metadata is essential for automated validation and error correction in SQL statements because it allows you to compare the specified functions and parameters against predefined values. For example, a JSON object may contain different variations of the ROUND function and allowed parameter combinations, giving systems accurate data to correct incorrect function calls.

This JSON object displays the different variants of the ROUND function, where each object in the parameters field represents different combinations of parameters that the function can accept. This overview allows us to better understand what parameters are accepted for a given function and how to apply them correctly.

```
"ROUND": [
 {
  "parameters": ["NUMBER", "NUMBER", "NUMBER"]
 },
 {
  "parameters": ["DATE", "DATE"]
 },
 {
  "parameters": ["DATE", "DATE", "VARCHAR2"]
 },
 {
  "parameters": ["NUMBER", "NUMBER"]
 }
]
```

Fig. 4 Example of JSON file

B. *Fixing typos in function names*

Typos in function names are corrected using the Levenshtein distance algorithm, which compares the specified function name with the correct names stored in JSON files. For example, if the user enters the function name "ROUNDD" instead of "ROUND", the algorithm recognizes that the distance between the two is small and suggests the correct function name.

C. *Correcting typos in function parameters*

After correcting the function names, the system will focus on the parameters. The parameters are checked based on the data types and the correct order. If the parameters are in the wrong order or incorrect data types are entered, the system will suggest correcting them. For example, if the parameters for the ROUND function are entered incorrectly, the system will swap or correct the values based on the data from the JSON files. Example of SQL statement with errors:

```sql
SELECT *
FROM drivers
JOIN licenses USING (license_number)
WHERE lower(driver_name) = 'john'
  AND lenth(driver_surname) <= 8
  AND substr(1, license_type, 1) = 'B'
  AND extract(YEAR
              FROM license_issue_date) = '2020';
```

Fig. 5 Select with mistakes in function part

After correcting the function names with the help of Lavenstein distance :

```sql
SELECT *
FROM drivers
JOIN licenses USING (license_number)
WHERE lower(driver_name) = 'john'
  AND lenth(driver_surname) <= 8
  AND substr(1, license_type, 1) = 'B'
  AND extract(YEAR
              FROM license_issue_date) = '2020';
```

Fig. 6 Select with corrected function's name

And after the last correction the final correct statement will be created.

```sql
SELECT *
FROM drivers
JOIN licenses USING (license_number)
WHERE lower(driver_name) = 'john'
  AND length(driver_surname) <= 8
  AND substr(license_type, 1, 1) = 'B'
  AND extract(YEAR
              FROM license_issue_date) = '2020';
```

Fig. 7 Select with correct parameters

D. *Checking the correctness of the parameters*

The final check ensures that all the parameters are correctly entered, in the correct order and have the required data types. This validation ensures that SQL statements will be executed efficiently and without errors. If any parameters do not match, the system flags them for manual checking.

VII. Experiments

To validate the functionality and efficiency of the proposed system, we conducted a series of experiments to detect and correct errors in SQL statements and process data stored in JSON files. These experiments were designed to simulate real-world conditions in a database environment with large volumes of data. The tests focused on two main areas: correcting typos in SQL statements using Levenshtein distance and validating parameters and functions in statements using data stored in JSON files.

A. *Scenario experiments*

The experiments were designed to test the effectiveness of the Levenshtein distance algorithm in detecting and correcting errors in SQL statements. The algorithm showed high accuracy especially for longer function names where the differences between the erroneous and correct names were clear. However, for shorter names, there were problems with error detection, indicating the need for further optimization. For very short names, however, the algorithm exhibited lower accuracy, suggesting the need to add mechanisms to better handle corrections for short names. The problem could be addressed by dynamically adjusting the desired similarity depending on the length of the string, which would improve the flexibility of the algorithm.

B. *Tests for feature repair*

One of the key areas of experimentation was the real-time correction of typos in SQL statements. The advantage of this approach is its applicability in dynamic database systems where commands are often executed in a data-intensive environment. The system was able to quickly identify and correct incorrect function and parameter names, which contributed to the reliability of the results.

However, we noticed that with more functions in the commands, the time required to analyze the distances increased, which could negatively affect the performance of the system when processing very large datasets. In addition, functions that were not frequently used or were at the end of JSON files were processed more slowly, suggesting a challenge for deploying the system in real-world applications with large volumes of data.

C. *Parameter validation tests*

The next phase of experiments focused on parameter validation using data stored in JSON files. The automated validation was successful in cases where the parameters were unambiguous and correctly defined, allowing the system to

correctly identify and correct incorrect values. However, the problem occurred with more complex parameters such as dates or specific ranges of values, where the system was not always able to correctly identify the parameter data type. This caused problems when validating some functions, such as the TO_CHAR function that works with dates.

The results of these tests showed that the system can work efficiently with simply defined parameters, but manual checking is required for more complex structures, which can be a challenge for fully automating the validation of more complex data types in real-world settings.

D. Summary of Experiments

Overall, the experiments showed that the proposed solution for error correction in SQL statements is effective, especially when working with simply structured data. However, the system has some limitations with more complex data types such as dates and ranges of values, where improved validation mechanisms are needed. The results of the experiments highlighted the need for further research aimed at optimizing the algorithm for short names and improving the processing speed of large datasets.

The automated error correction system was tested using a dataset of approximately 500 different SQL queries, each containing intentional errors in function names and parameters. The performance of the system was measured in terms of time taken, reliability, and percentage of errors corrected.

The function and parameter correction process is extremely fast, relying on a simple for loop for function corrections and parameter checks using a hash table. The entire correction process takes only a few milliseconds to second, even in the worst case scenario, where function is near the end of JSON file.

For function names longer than 5 characters, 90% of the functions were correctly identified and corrected. In contrast, for function names with 4 or fewer characters, the success rate dropped to 70%, due to the increased likelihood incorrect correction of function.

VIII. Discussion

The experimental results demonstrate that the proposed system for correcting SQL queries using the Levenshtein distance algorithm and validating parameters through JSON storage significantly enhances SQL query processing accuracy. The system effectively reduced manual correction efforts by automating the detection and correction of typographical errors in SQL function names and parameters.

One of the main advantages of this system is its ability to handle complex SQL functions automatically, which is particularly relevant in environments that process large datasets. By integrating distance algorithms and automated parameter validation, the tool provides a higher level of automation compared to existing systems like SQLTutor and AutoGrader. These existing systems offer feedback but lack the capability to automatically correct errors.

A. Benefits of solution

One of the biggest advantages of the proposed system is its ability to automate the correction of typos and incorrectly specified functions and variables in SQL statements. Using Levenshtein distance, the system can quickly and efficiently identify minor errors in text strings and automatically suggest a correction based on the metadata stored in JSON files.

The use of JSON files to store metadata and validate parameters is another innovative aspect of this solution. Storing information about correct parameters and functions in a structured format allows the system to dynamically validate commands before they are executed, reducing errors and increasing the reliability of queries.

By automating and validating data, the system significantly reduces the need for manual intervention, which is often the source of further errors. Automated correction and validation allows developers to focus on other tasks, saving time and increasing the efficiency of working with databases.

B. Limitations of the Solution

Although the proposed system has demonstrated its effectiveness in correcting typos and validating simply structured parameters, we have encountered several challenges that need to be addressed in further development.

As experiments have shown, the Levenshtein distance algorithm is less efficient for very short function names, where the differences between correct and incorrect names are minimal. This limitation may lead to incorrect corrections, so the algorithm needs to be supplemented in the future with additional rules to increase its accuracy in these cases.

when validating complex data types such as dates or specific value formats, the system has encountered problems that require manual checking. For example, the system was not always able to correctly identify the data type or correctly validate dates, leading to validation failures in some cases.

IX. Conclusion

The proposed system for automated error correction in SQL queries has proven to be an effective tool in reducing manual correction efforts and improving overall query accuracy. By employing the Levenshtein distance algorithm for detecting and correcting typographical errors in SQL function names and leveraging JSON for parameter validation, the system enhances both educational and practical applications. This automation not only ensures higher reliability in SQL query execution but also minimizes the time spent on manual debugging and correction.

Despite these advantages, there are several opportunities for further refinement. The system's current limitation in handling short function names, where the Levenshtein algorithm shows reduced accuracy, highlights the need for additional rules or dynamic threshold adjustments. By improving the detection of short function name errors, the system could offer more reliable corrections across various query scenarios.

Another area for future research is the validation of complex data types, such as dates and specific value formats, which still require manual intervention. Developing specialized algorithms to automate the detection and validation of these complex data types would significantly expand the system's utility, especially in handling more sophisticated SQL queries.

Furthermore, while the system has been successfully tested in an Oracle environment, its core principles can be extended to support other database management systems, such as PostgreSQL and MySQL. Expanding compatibility to these

systems would make the tool more versatile and applicable in a wider range of database environments.

In conclusion, the system shows great promise in automating error correction in SQL queries, with several avenues for future enhancement. Addressing its current limitations in short function names and complex data types, along with broadening its database support, will contribute to making this tool an even more robust and indispensable resource for database administrators and developers.

ACKNOWLEDGMENT

This paper was supported by the VEGA 1/0192/24 project - Developing and applying advanced techniques for efficient processing of large-scale data in the intelligent transport systems environment.

REFERENCES

[1] D. Crockford, "The application/json Media Type for JavaScript Object Notation (JSON)," *RFC 4627*, IETF, July 2006. [Online]. Available: https://www.rfc-editor.org/rfc/rfc4627

[2] T. Bray, "The JavaScript Object Notation (JSON) Data Interchange Format," *RFC 8259*, IETF, Dec. 2017. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8259

[3] C. L. Boisvert, K. Domdouzis, and J. License, "A Comparative Analysis of Student SQL and Relational Database Knowledge Using Automated Grading Tools," in *Proc. 2019 IEEE Global Engineering Education Conf. (EDUCON)*, April 2019, pp. 1128–1133. [Online]. Available: https://ieeexplore.ieee.org/document/8586684

[4] Bhangdiya et al., "The XDa-TA system for automated grading of SQL query assignments," in *Proc. 2019 34th IEEE/ACM Int. Conf. Automated Software Engineering (ASE)*, Nov. 2019, pp. 1049–1053. [Online]. Available: https://ieeexplore.ieee.org/document/7113403

[5] M. Martin and M. Nathan, "Automated Grading of SQL Queries," in *Proc. 2019 13th IEEE Int. Conf. e-Science (e-Science)*, Dec. 2019, pp. 284–288. [Online]. Available: https://ieeexplore.ieee.org/document/8731495

[6] S. Wu and U. Manber, "Fast text searching: allowing errors," *Commun. ACM*, vol. 35, no. 10, pp. 83–91, Oct. 1992. [Online]. Available: https://www.researchgate.net/publication/228818851_Using_TF-IDF_to_determine_word_relevance_in_document_queries.

[7] S. Liu, "Wi-Fi Energy Detection Testbed (12MTC)," 2023, gitHub repository. [Online]. Available: https://github.com/liustone99/Wi-Fi-Energy-Detection-Testbed-12MTC

[8] J. Ramos, "Using TF-IDF to determine word relevance in document queries," in *Proc. 1st Instructional Conf. Machine Learning*, 2003. [Online]. Available: https://www.cs.odu.edu/~tkennedy/cs595f18/lectures/TF-IDF-Ramos.pdf

[9] R. Hamming, "Error Detecting and Error Correcting Codes," *Bell Syst. Tech. J.*, vol. 29, no. 2, pp. 147–160, April 1950. [Online]. Available: https://ieeexplore.ieee.org/document/6772729

[10] J. Bloch, *Effective Java*, 3rd ed., Addison-Wesley, 2018. [Book]

[11] B. N. Groznik, "SQLTutor: A Tool for SQL Query Evaluation," *J. Comput. Sci. Educ.*, vol. 25, no. 3, pp. 45-52, Sept. 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8992341

[12] P. R. Nair, "A Comparative Study on Automated SQL Grading Tools: SQLTutor and AutoGrader," in *Proc. 2020 Int. Conf. Comput. Sci. Educ.*, Aug. 2020, pp. 305–312. [Online]. Available: https://ieeexplore.ieee.org/document/9205645

[13] S. Mukherjee et al., "AutoGrader: An Automated SQL Query Grading System," in *Proc. 2019 IEEE Frontiers Educ. Conf. (FIE)*, Oct. 2019, pp. 1-7. [Online]. Available: https://ieeexplore.ieee.org/document/9052199