SLQ: Lightweight-Detection & LLM-Generation SQL SelfOptimization Framework

Zhu Tianyou*, Qi Yaru, Jiang Kongchen, Sang Yanting, and Yang Chao

State Grid Information & Telecommunication center (Big Data center), Beijing 100052, P. R. China

Corresponding author. E-mail: tianyou-zhu@sgcc.com.cn

Received: October 13, 2025; Accepted: November 16, 2025

In real-world database applications, SQL statements written by users often create performance bottlenecks because they violate best-practice rules. Traditional rule-based detectors have limited ability to recognize diverse and increasingly irregular statements and are costly to maintain. To address this, we propose SLQ, a two-stage intelligent SQL optimization framework. First, a lightweight stacked-LSTM module pinpoints problematic statements; then a pre-trained large language model, Qwen3, automatically generates explanations for each flaw and offers targeted rewrite suggestions, helping users quickly improve query quality. Evaluated on a standard dataset, SLQ achieves accuracy, precision, recall and F1 of 0.9841, 0.9974, 0.9702 and 0.9836 respectively, demonstrating superior detection and optimization capability and markedly enhancing SQL compliance and execution efficiency.

Keywords: SQL Query Optimization; Large Language Model (LLM); Long Short-Term Memory(LSTM)

http://dx.doi.org/10.6180/jase.202607_30.025

1. Introduction

In the widespread practice of database applications, ensuring that user-submitted query statements adhere to specifications and possess good execution efficiency constitutes one of the core challenges for guaranteeing system performance and stability. Non-compliant SQL statements (e.g., full table scans without indexes, redundant subqueries, or excessive nesting) frequently occur in online business scenarios, often leading to query response delays, surging resource consumption, and even triggering systemic performance fluctuations [1]. According to public operations reports, over 60% of online slow queries can be attributed to non-standard writing practices at the statement level, with a significant performance degradation event occurring on average every minute [2]. Such problematic statements typically stem from developers' insufficient understanding of database execution mechanisms or a lack of unified coding standards, causing query plans to fail in utilizing indexes and generating substantial intermediate results, ultimately impeding critical business processes.

Traditionally, preventing such non-compliant statements primarily relies on pre-defined signature matching (e.g., regular expressions), static code analysis, among other methods. While capable of intercepting known problematic patterns, these approaches exhibit significant limitations: they struggle to comprehensively adapt to the highly varied forms of user input (including complex, variant, or unintentionally malformed constructions); maintaining extensive predefined rule libraries is costly; furthermore, when confronted with structurally complex or emerging non-compliant query patterns, their detection accuracy and efficiency are often inadequate. This is particularly evident in complex modern web application interaction scenarios, where updates to predefined rules frequently lag behind newly emerging non-compliant input patterns.

In recent years, machine learning and deep learning technologies have demonstrated substantial potential in

the fields of pattern recognition and anomaly detection [3–6]. Their core value lies in the ability to automatically learn normative interaction patterns and identify deep-seated deviation regularities directly from massive volumes of raw user query data, without relying entirely on manually predefined exhaustive rule sets. This presents a new opportunity to address the limitations of traditional methods in identifying non-compliant input statements for databases.

Ensuring the normative quality of SQL query statements submitted by users (or applications) represents a core challenge in safeguarding database performance and stability. Numerous techniques have been developed to identify and intercept non-compliant or malformed SQL input statements, which can be broadly categorized into two classes: rule-based syntactic normative checking methods, and machine learning-based and deep learning-based normative identification methods.

The core of rule-based syntactic normative checking methods lies in pre-defining a set of rules to determine whether a user-input SQL statement conforms to the expected correct format. Typically, these rules are derived from manual analysis of historically problematic non-compliant statements [7]. As one of the early representative solutions, AMNESIA [8] pioneered a hybrid detection mechanism combining static source code analysis with dynamic runtime monitoring (e.g., SQL query interception) to counter non-compliant SQL input. Modern static analysis-based detection approaches often leverage regular expressions [9] to perform real-time inspection of traffic at the proxy layer, identifying queries with structural anomalies or those containing known non-compliant patterns. Similarly, Kini et al. [10] proposed a framework integrating the BCRYPT hash algorithm with the efficient Aho-Corasick multi-pattern matching algorithm. Herein, the Aho-Corasick algorithm, by pre-storing a vast number of known, typical non-compliant patterns (including common SQL non-standard structures) within a tree data structure, significantly enhances the efficiency of rapidly screening input streams for these non-compliant statement features. Gu et al. [11] proposed a hierarchical detection method that, targeting different types of structural anomalies, utilizes database response information to construct a multi-level regular expression rule set, identifying severely non-compliant input statements through a layered filtering approach. However, such rule-based methods suffer from fundamental limitations: they heavily rely on predefined pattern libraries, leading to insufficient capability in identifying novel, complex, or inadvertently generated non-compliant statements that users continuously create and evolve [12]; the maintenance burden of the rule library is heavy, and these methods struggle to adaptively understand the root cause of statement anomalies to provide repair guidance.

By contrast, Machine Learning (ML) and Deep Learning (DL) techniques can automatically learn deep-seated features and pattern differences from massive volumes of user input query samples (including both compliant and non-compliant statements), offering better generalization potential and the ability to identify unknown or variant non-compliant structures. These methods learn the intrinsic characteristics of non-compliant statements from samples and generalize this knowledge to unseen SQL queries. Gandhi et al. [13] proposed a hybrid CNN-BiLSTM model for learning to identify potentially non-compliant queries, which provided an accuracy of approximately 98% compared to other machine learning algorithms. Ahmed et al. [14] utilized Natural Language Processing (NLP) techniques (such as Bag-of-Words) to process query text, combined with ensemble learning algorithms (like Random Forest) to train models for normative classification of queries, and their study also compared the classification effectiveness of models including Decision Trees, Naive Bayes, Support Vector Machines (SVM), and KNN. Tripathy et al. [15] created a dataset by collecting and combining numerous smaller datasets and trained seven machine learning models (Decision Tree, AdaBoost, Random Forest, Optimized Linear, Linear Neural Network, Deep Artificial Neural Network, and Boosted Tree Classifier), with results indicating that the Random Forest classifier outperformed all others in distinguishing compliant from non-compliant queries. Kasim et al. [16] designed a method using the Decision Tree algorithm to detect normative queries, which achieved 98% accuracy in the binary classification task and 92% accuracy in a three-class task categorizing the severity level of non-compliance into 'simple', 'unified', and 'lateral'. Tang et al. [17] effectively identified statement structural deviations by extracting high-dimensional features from query samples and using an MLP as the classifier. Sommervoll et al. [18] innovatively modeled query compliance checking as a Markov Decision Process (MDP), employing reinforcement learning to train an agent for automated optimization of non-compliant statement identification. FAOOQ [19] also extracts key features from text statistics: it encodes a query statement into a 21-dimensional feature vector by calculating the total frequency of key features like special characters and applies ensemble learning techniques for detection. Zhou et al. [20] proposed a method integrating an LSTM network with syntactic parse trees, which first traverses the syntactic parse tree using a depth-first search algorithm to convert its structure into a numerical sequence,

then leverages the LSTM to extract deep temporal features from these sequences for classification.

Despite the respective advancements of the two classes of methods—with rule-based methods having clear interpretability and ML/DL-based methods achieving continuous progress in recognition accuracy and generalization capability—the primary focus of existing research remains centered on determining whether an input statement is "non-compliant". A critical component is lacking: intervention for non-compliant statements. This significantly limits the system's capacity for "autonomous optimization" of user input queries, leaving the repair process heavily reliant on manual analysis and diagnosis. To address this gap, we propose the SLQ (Stacked-LSTM Qwen) framework, a two-stage framework specifically designed for inspecting user input, identifying non-compliant SQL statements, and providing autonomous optimization solutions. Its two-stage design (detection + autonomous optimization) not only identifies non-compliant statements but also focuses on endowing the system with the ability to "understand the root cause of problems" and "autonomously repair and optimize", thereby addressing the limitations of existing methods.

Faced with massive volumes of user-generated, often highly homogeneous non-compliant queries, the ability to promptly identify and understand the root causes of non-compliance, thereby automatically providing optimization or remediation suggestions, would significantly reduce potential losses and enhance system self-healing capabilities. Many existing methods primarily focus on passively determining compliance status, lacking in-depth analysis of why a specific query statement is non-compliant and corresponding guidance on how to rectify it. To address this gap, we propose the SLQ (Stacked-LSTM Qwen) framework, a two-stage framework specifically designed for inspecting user input, identifying non-compliant SQL statements, and providing autonomous optimization solutions. The SLQ framework comprises a Detector and a Repairer. We designed a lightweight Detector based on a Stacked Long Short-Term Memory (Stacked-LSTM) network, which achieves high recognition accuracy for non-compliant statements while efficiently processing massive query data. The Repairer integrates Keras' Tokenizer with the pre-trained large language model Qwen3. When the Detector identifies a user-input query statement as non-compliant, the Repairer automatically conducts an in-depth analysis of the underlying reasons for its non-compliance. It then intelligently generates the rationale for classifying the statement as non-compliant, along with specific, actionable optimization suggestions and vulnerability remediation plans, assisting developers in swiftly implementing corrections and achieving autonomous optimization.

In summary, the key contributions of this paper are as follows:

This paper proposes a two-stage autonomous SQL optimization framework, SLQ, which achieves efficient and accurate identification of non-compliant queries while providing autonomous optimization solutions, thereby promptly mitigating potential losses.

This paper proposes a high-precision, lightweight method for detecting standard SQL statements, effectively enhancing detection performance.

Compared to existing SQL detection methods, the SLQ framework achieves a higher detection rate on public datasets and automatically provides specific and reasonable repair suggestions.

2. Methodology

2.1. System Architecture

The proposed SLQ framework adopts a modular design, comprising two core modules: the Detector and the Repairer. As illustrated in Fig. 1, the overall workflow of the SLQ framework is as follows: Upon receiving a query statement, SLQ first utilizes the LSTM-based Detector to screen the statement. If the statement is classified as compliant, the Qwen3-based Repairer is not activated. If the statement is identified as a non-compliant query, the built-in Keras Tokenizer class is employed to generate corresponding tokens from the original SQL query string. Subsequently, the Qwen3 model leverages these tokens to generate a detailed rationale for the non-compliant classification, alongside actionable suggestions for statement optimization, thereby assisting the user in promptly identifying and rectifying deficiencies.

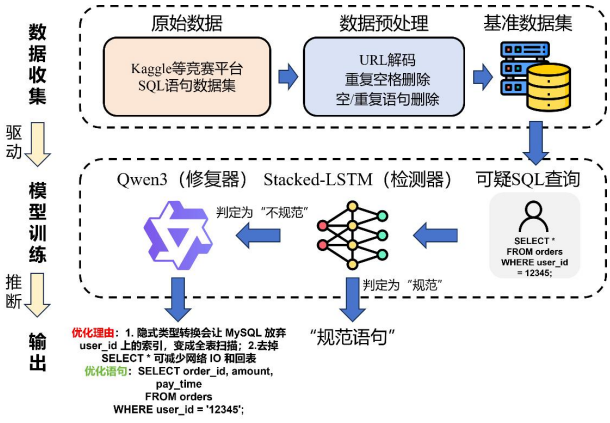

Fig. 1. Schematic Diagram of the SLQ Framework.

2.2. Detector Design

This paper proposes a normative SQL query detector based on an improved Stacked Long Short-Term Memory (LSTM) network architecture. The design objective of this detector is to efficiently and accurately identify non-compliant SQL queries within database operations, thereby providing support for database performance optimization. The input to the entire model is a preprocessed sequence representing the SQL query statement. This sequence first passes through an embedding layer, then through a multi-layer LSTM structure to extract semantic features from the statement, and finally through a fully connected layer to perform binary classification of the query statement—determining whether the statement is a normal query or a non-compliant query.

Assume an input sequence $X = [x_1, x_2, \cdots, x_T]$ where T is the maximum sequence length and x_i is the word index. A pre-trained word embedding matrix $W_e \in \mathbb{R}^{(V+1)\times d}$ (where V is the vocabulary size and d is the embedding dimension) maps the discrete symbols in X to dense vector representations.Specifically, the model input is a sequence representing the SQL query statement, containing the maximum sequence length and word indices. To transform these discrete symbols into dense vectors processable by the model, we utilize a pre-trained word embedding matrix. This matrix, with dimensions of vocabulary size multiplied by embedding dimension, maps each word index into a fixed-dimensional vector space, thereby capturing semantic relationships between words:

$$E = \text{Embedding}(X) = [w_{x_1}, w_{x_2}, \ldots, w_{x_T}] \tag{1}$$

Following the embedding layer, we introduce a SpatialDropout1D mechanism. This serves as a regularization technique that randomly masks entire feature channels with a certain probability to reduce the model's reliance on specific input features, thereby mitigating the risk of overfitting. This method is particularly well-suited for processing sequential data, as it effectively weakens co-occurrence noise between features and enhances the model's generalization capability:

$$E' = \text{SpatialDropout1D}(E, 0.3) \tag{2}$$

Subsequently, a three-layer bidirectional LSTM architecture is employed to progressively extract contextual features. Bidirectional LSTM is a specific variant of the LSTM structure that incorporates both forward and backward LSTM layers, enabling the simultaneous consideration of both preceding and subsequent context for each element in the sequence. Following each LSTM layer, a Batch Normalization layer and a Dropout layer are inserted to accelerate model convergence and further suppress overfitting. The output dimensionality of the first bidirectional LSTM layer is 256, and similarly, the output dimensionality of each subsequent layer is also 256, ensuring consistent information flow between layers. The hidden state calculation formula for the LSTM layer can be expressed as:

$$\overrightarrow{h_t^{(k)}}, \overleftarrow{h_t^{(k)}} = \text{BiLSTM}^{(k)}\left(H_t^{(k-1)}\right) \tag{3}$$

During the computation of the first bidirectional LSTM layer, the output dimensionality for each subsequent layer is 256. Batch Normalization and Dropout layers (Dropout rate = 0.5) are inserted between the layers to accelerate convergence and suppress overfitting.

$$H^{(k)} = \text{Dropout}\left(\begin{pmatrix}\text{BatchNorm}\\ \left(\text{operatornameBiLSTM}^{(k)}\right)\\ (H^{(k-1)})\end{pmatrix}\right) \tag{4}$$

Following processing by the LSTM layers, the output features are passed through a fully connected layer with 256 dimensions. This layer utilizes the ReLU (Rectified Linear Unit) activation function to introduce non-linearity, enabling the model to learn more complex feature representations.The LSTM output features h_{final} are processed by a 256-dimensional fully connected layer followed by ReLU activation:

$$z = \text{ReLU}(W_d h_{\text{final}} + b_d) \tag{5}$$

Finally, unnormalized probabilities are obtained through a Sigmoid output layer:

$$\hat{y} = \sigma(w_o^T z + b_o) \tag{6}$$

The binary cross-entropy loss is minimized based on the Adam optimizer:

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)] \tag{7}$$

Where N is the number of samples, and $y_i \in 0, 1$ is the true label.

2.3. Repairer Design

When the detector of the SLQ framework identifies a non-standard SQL query, these queries are passed to the Text Tokenizer module for further processing. In this process, we utilize the built-in Tokenizer class from the Keras library, which is a powerful tool specifically designed for converting textual data into a format processable by models. The Tokenizer works by splitting the input string based on spaces and other delimiters, thereby generating an ordered token sequence denoted as , where represents a fundamental semantic unit in the input. A token is a basic semantic

unit in the text, which can be a word, a punctuation mark, or a special character. The Tokenizer is modified to accommodate SQL's structural features. Key configuration parameters and rules are defined as follows:

- Sequence Truncation/Padding: We calculated the length distribution of all SQL statements (mean: 32 tokens, 95th percentile: 58 tokens) and set the maximum sequence length to 58. Statements longer than 58 tokens are truncated from the end (to preserve the initial syntax structure like SELECT ... FROM), while shorter ones are padded with $\langle PAD \rangle$ at the end.
- Case Sensitivity: The lower parameter is set to False (SQL keywords are case-insensitive in practice, but we unify them to uppercase during preprocessing to ensure consistent tokenization—e.g., select and SELECT are both converted to SELECT before tokenization).

After the token sequence is generated, these tokens are fed into the Qwen3 large language model. Qwen3 is a pre-trained large language model that has been trained on massive text data, equipping it with profound language understanding and generation capabilities. At this stage, the core task of Qwen3 is to perform detailed analysis and reasoning on the non-standard SQL query to accurately diagnose the underlying causes of its non-conformance.

By thoroughly parsing the semantic structure and intent of the input token sequence, Qwen3 can identify the issues within the query statement. For instance, it can detect implicit type conversion issues in the SQL query, which might cause the database management system to forgo using indexes, thereby increasing query time. Furthermore, Qwen3 can identify full table scan problems caused by the use of SELECT * statements. This operation leads the database to return unnecessary columns, increasing network I/O and table lookup operations, further reducing query efficiency.

Taking the example in the diagram, the original query SELECT * FROM orders WHERE user_id = 12345 exhibits the two aforementioned issues. Firstly, the value for the user_id field is not enclosed in quotes, potentially leading to implicit type conversion. Secondly, the SELECT * statement causes the database to return all columns from the orders table, rather than only the required ones.

After completing the diagnosis of the statement, Qwen3 generates specific optimization suggestions based on its built-in structured knowledge base. In the example above, Qwen3 suggests rewriting the query as SELECT order_id, amount, pay_time FROM orders WHERE user_id = '12345'. This revision offers two main benefits: firstly, by specifying explicit column names, it reduces the reading of unnecessary columns, thereby lowering network I/O and table lookup operations; secondly, by enclosing the user_id value in quotes, it avoids implicit type conversion, allowing the database to utilize indexes and consequently improving query efficiency.

In summary, through the collaborative work of the Text Tokenizer module and the Qwen3 large language model, the SLQ framework can accurately diagnose non-standard SQL queries and provide concrete, actionable optimization solutions, thereby significantly enhancing the efficiency and performance of database queries.

3. Results and discussion

3.1. Dataset Introduction

We collected publicly available datasets for non-standard SQL statement detection from competition platforms such as Kaggle and aggregated them to create a benchmark dataset. This dataset contains a total of 148,326 SQL statements, comprising both non-standard SQL queries (such as full table scans without indexes and redundant subqueries) and standard SQL queries. It is divided into 92,165 non-standard statements (62.1%) and 56,161 standard statements (37.9%). Each statement has only one classification label (yes or no). We categorized non-standard statements into 3 types (syntax-level, logic-level, performance-level) based on their root causes, with specific examples for each type (see Table 1). This classification helps clarify the dataset's coverage and ensures alignment with common SQL optimization pain points.

We split the benchmark dataset into training and validation sets in an 8:2 ratio. Examples of one non-standard and one standard statement are shown below:

Non-standard statement: select * from users where id = 1 or ",." or 1 = 1 – 1

Standard statement: SELECT * FROM depend WHERE escape = mice

Furthermore, during dataset construction, we performed a comprehensive data preprocessing pipeline including "URL Decoding - Repetitive Space Removal - Empty/Duplicate Statement Removal". The specifics are as follows:

URL Decoding: As we collected public datasets from multiple different creators, some datasets contained SQL statements that had undergone URL encoding. Directly mixing URL-encoded statements with unprocessed statements for the detector to learn from could impair the detector's normal operation by causing it to learn incorrect lexical semantics. Therefore, we used Qwen3 to filter all statements, identifying those potentially containing URL encoding. Qwen3 provided the decoded results, which were then manually reviewed.

Table 1. Categories and Examples of Non-Standard SQL Statements.

Category	Description	Example SQL Statement
Syntax-Level	No quotes for string-type conditions, causing index failure, or retrieves unnecessary columns, increasing I/O	SELECT * FROM products WHERE price $>$ 100
Logic-Level	Unnecessary nested subqueries, wasting resources, or prevents index usage, leading to full-table scan	SELECT * FROM customers WHERE age $>$ 30 OR city = 'Beijing'
Performance -Level	No index on filtered columns, slowing query, or risk of Cartesian product, increasing data transfer	SELECT * FROM users, orders WHERE users.id = orders.user_id

Repetitive Space Removal: The initially preprocessed dataset contained numerous SQL query statements with repeated spaces. For instance, the aforementioned non-standard example statement, before processing, was select * from users where id = 1 or ",." or 1 = 1 – 1 and contained excessive repeated spaces. Although repeated spaces do not affect visual observation and judgment by humans, they could introduce additional noise into the detector's decision-making process and the repairer's analysis and suggestion process. We used Python scripts to automatically replace repeated spaces with a single space, thereby achieving repetitive space removal.

Empty/Duplicate Statement Removal: Some statements might become duplicates of others in the dataset after the aforementioned preprocessing steps. Therefore, we placed the "Empty/Duplicate Statement Removal" preprocessing step as the final stage and used Python scripts to automatically remove duplicate and empty statements.

3.2. Experimental Platform and Metrics

The deep learning server platform used in this study is equipped with an Intel Xeon Platinum 8336C CPU @ 2.30GHz, 128 GB of memory, and an Nvidia RTX 4090 graphics card. The experiments were conducted on an Ubuntu 22.04 operating system and relied on Anaconda (version 24.5.0) to build the Python (version 3.9.21) experimental environment. The open-source tool llama.cpp was used to quantize and accelerate the locally deployed Qwen3-8B model. This study employs Accuracy, Precision, Recall, and F1-score as evaluation metrics. The specific calculation formulas are as follows:

$$\text{Accuracy} = \frac{TP + TN}{N} \tag{8}$$

$$\text{Precision} = \frac{TP}{TP + FP} \tag{9}$$

$$\text{Recall} = \frac{TP}{TP + FN} \tag{10}$$

3.3. Comparative Experiments

We conducted a systematic comparison between Stacked-LSTM (the detector of the proposed SLQ framework) and eight open-source algorithms from the Kaggle platform on the collected SQL benchmark dataset, aiming to comprehensively evaluate the performance of different models in detecting non-standard SQL queries. The experimental results are shown in Table 2, where the best performance for each evaluation metric is indicated in bold.

The data in Table 2 provides a comprehensive analysis of the Stacked-LSTM model's performance on the benchmark dataset, revealing its significant advantages across multiple key metrics. Firstly, the Stacked-LSTM model achieved an accuracy of 98.41%. This figure is not only impressive but also 0.69 percentage points higher than the 97.72% accuracy of the next best model, XGBoost. This indicates that Stacked-LSTM is more reliable in identifying samples of the correct category and can more effectively distinguish between non-standard and standard SQL queries.

Secondly, the Precision of Stacked-LSTM reached 99.74%, which is 1.00 percentage point higher than the 98.74% of the traditional CNN model. This improvement in Precision significantly reduces the risk of false positives, meaning fewer instances of standard statements being incorrectly identified as non-standard. This is crucial for database management systems as it reduces unnecessary query optimizations, thereby saving resources and improving efficiency.

Recall is another important metric for assessing model performance, and Stacked-LSTM also excelled in this aspect, achieving 97.02%. This surpasses the Recall of Random Forest (96.02%) and AdaBoost (95.23%). This result verifies that Stacked-LSTM has a stronger capability in capturing target samples, meaning it can more effectively identify all non-standard queries without missing too many.

It is particularly noteworthy that the F1-score of Stacked-LSTM reached 98.36%, which is 1.07 percentage points higher than the 97.29% of the single-layer LSTM. The F1-score is the harmonic mean of Precision and Recall, and this improvement demonstrates that the stacked architecture of Stacked-LSTM is more effective in extracting sequential features. Compared to the baseline model Naive Bayes, Stacked-LSTM's F1-score represents a performance improvement of nearly 10 percentage points.

Furthermore, while achieving ultra-high Precision

Table 2. Performance Comparison of the SLQ Framework Detector (Stacked-LSTM) and Other Models on the Benchmark Dataset.

Model	Accuracy	Precision	Recall	F1-Score
Naive Bayes	0.8814	0.8802	0.8951	0.8876
Decision Trees	0.9215	0.9183	0.9205	0.9187
SVM	0.9438	0.9412	0.9395	0.9402
CNN	0.9662	0.9874	0.9475	0.9671
AdaBoost	0.9681	0.9863	0.9523	0.9689
LSTM	0.9701	0.9891	0.9573	0.9729
Random Forest	0.9774	0.9964	0.9602	0.9779
XGBoost	0.9772	0.9933	0.9629	0.9779
Stacked-LSTM	**0.9841**	**0.9974**	**0.9702**	**0.9836**

(99.74%), the Stacked-LSTM model also maintained a high Recall (97.02%). This balance is critically important for practical applications. It ensures an extremely low false positive rate (only 0.26% false positives) while maintaining a true positive coverage rate of over 97%. This means the model can reduce false alarms while simultaneously ensuring that the majority of truly non-standard queries are correctly identified.

In summary, the Stacked-LSTM model comprehensively surpasses current mainstream models across key performance metrics. This overall superiority highlights the practical value and deployment potential of Stacked-LSTM for the task of identifying non-standard SQL queries. These advantages position Stacked-LSTM as a powerful tool that can assist database administrators and developers in more effectively identifying and optimizing query statements, thereby enhancing the overall performance and stability of databases.

Fig. 2a illustrates the changes in the accuracy of the Stacked-LSTM model on both the training and validation sets as the number of training epochs increases. The graph shows that within the first 5 epochs, the accuracy for both sets exhibits a rapid upward trend. This indicates that during the initial training phase, the Stacked-LSTM model effectively learns and captures the underlying patterns of non-standard features in the SQL statements, leading to a significant performance improvement. However, a critical turning point occurs at the 5th epoch. The validation accuracy experiences a sharp decline, dropping to a range around 0.75, which represents a decrease of over 15 percentage points compared to the previous peak. This significant drop suggests that the model begins to suffer from a degradation in generalization capability at this stage, meaning it overfits the training data while its adaptability to unseen validation data deteriorates.

Fig. 2b depicts the changes in the loss values for the training and validation sets throughout the training process. Corresponding to the accuracy trend, the loss values

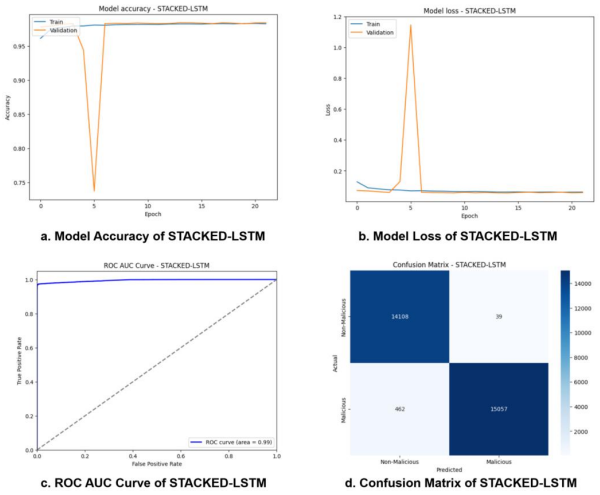

Fig. 2. a. Accuracy performance of the Stacked-LSTM model during training, showing the variation of training and validation accuracy across epochs. b. Loss performance of the Stacked-LSTM model during training, showing the variation of training and validation loss across epochs. c. ROC AUC curve of the Stacked-LSTM model, demonstrating the classification performance of the model, with an AUC value of 0.99. d. Confusion matrix of the Stacked-LSTM model, presenting the classification results of the model on the test set.

for both sets show a rapid decrease in the initial epochs. This demonstrates the model gradually adapting to the training data and effectively identifying non-standard patterns in the SQL statements. However, at the 5th epoch, coinciding with the turning point in the accuracy curve, the validation loss suddenly exhibits severe oscillations. These pronounced oscillations further reveal issues with the model's generalization ability at this phase. While the model continues to optimize on the training set, its performance on the validation set becomes unstable, indicating potential overfitting to the training data and an inability to generalize effectively to new, unseen data, thereby impact-

ing the model's overall performance and stability.

This abrupt fluctuation potentially stems from two causes: Firstly, specific batches within the validation set might contain adversarial samples or novel types of non-standard SQL queries that exposed vulnerabilities in the model at that specific training stage. Secondly, a temporary distribution shift might have occurred between the training data and the validation set at that particular epoch. It is noteworthy that the model demonstrated excellent self-correction capability: after the fluctuation, it required only 2-3 epochs to rapidly recover and converge again. Ultimately, both the training and validation accuracy stabilized at a high plateau above 0.95, while the loss values concurrently converged to a low-risk range near 0.1. Collectively, this indicates that although the Stacked-LSTM exhibited a brief period of vulnerability during mid-training, it ultimately demonstrated robust feature learning capability and stability.

Fig. 2c clearly displays the Receiver Operating Characteristic (ROC) curve of the Stacked-LSTM model. The shape of this curve is near-perfect, and the Area Under the Curve (AUC) value reaches 0.99. This high AUC value indicates that the Stacked-LSTM model demonstrates exceptional accuracy in distinguishing between standard and non-standard SQL queries. In the field of machine learning, the ROC curve is a crucial tool for evaluating the performance of binary classification models. It reflects the model's classification capability by plotting the relationship between the True Positive Rate and the False Positive Rate. The high AUC value of the Stacked-LSTM model signifies that, regardless of the threshold setting, the model maintains an excellent True Positive Rate while keeping the False Positive Rate at a very low level. This demonstrates that the model can very effectively identify non-standard SQL statements in practical applications, thereby providing strong support for database security and performance optimization.

Fig. 2d presents the confusion matrix of the Stacked-LSTM model. This matrix provides a detailed view of the model's classification performance on the test set. A confusion matrix is a tool used to evaluate the performance of classification models; it visually reflects the model's accuracy by showing the relationship between the model's predictions and the actual labels. In the confusion matrix of the Stacked-LSTM model, the counts for both True Positives (correctly identified non-standard statements) and True Negatives (correctly identified standard statements) are very high. This indicates that the model can accurately classify SQL statements in the vast majority of cases. Concurrently, the counts for False Positives (incorrectly identifying standard statements as non-standard) and False Negatives (incorrectly identifying non-standard statements as standard) are relatively low. This low misjudgment rate further validates the high efficiency and reliability of the Stacked-LSTM model in handling the task of SQL statement optimization. These results show that the model not only accurately identifies non-standard SQL statements but also effectively avoids misjudgments while maintaining high accuracy, thereby providing reliable assurance for the stable operation and performance optimization of databases.

3.4. Case Study

Fig. 3 demonstrates the application effectiveness of the SLQ framework in optimizing SQL queries. This framework can identify non-standard SQL statements that lead to low query efficiency and provide targeted optimization suggestions.

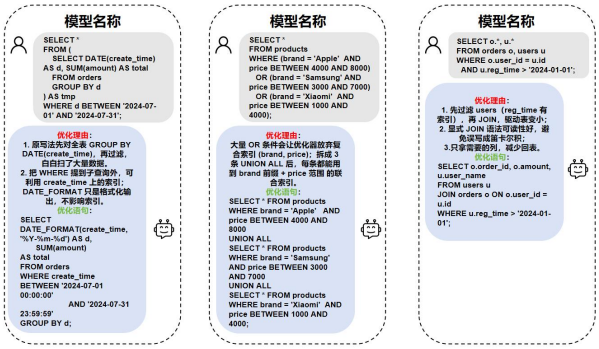

Fig. 3. SLQ Framework Case Study.

In the first case, the SLQ framework processed a complex subquery statement involving a time-range query. The original query performed a grouping operation on the entire orders table before applying the time filter condition. This approach is inefficient because it scans the entire table before applying the filter, resulting in the processing of a large amount of unnecessary data.

The SLQ framework identified this issue and proposed an optimization. It recommended moving the time filter condition outside the subquery, thereby allowing the index on the create_time field to be utilized before the grouping operation. Furthermore, the framework suggested using the DATE_FORMAT function for time formatting instead of performing the grouping within the subquery, as the formatting operation does not interfere with index usage. Through these optimizations, the query can significantly reduce the amount of data scanned and accelerate the query process.

The second case involved a query statement with multiple conditions. The original query used multiple OR con-

ditions, which can prevent the database optimizer from effectively utilizing composite indexes, as OR conditions often cause the optimizer to abandon index usage in favor of a full table scan.

The SLQ framework identified this problem and suggested splitting the original query into multiple subqueries, using UNION ALL to combine the results. This allows each subquery to independently leverage the indexes on the brand and price fields. In this way, the query can utilize indexes more effectively, thereby improving query efficiency.

The final case was a query involving multi-table joins. The original query used implicit joins, which can lead to performance issues because implicit joins might be interpreted as Cartesian products, resulting in significant data transfer and table lookup operations.

The SLQ framework highlighted this issue and recommended replacing the implicit joins with explicit JOIN syntax. This not only improves the readability of the query but also avoids accidental Cartesian products. Additionally, the framework suggested selecting only the necessary columns instead of using SELECT *, which reduces the amount of data transferred and the number of table lookup operations, further enhancing query efficiency.

Through these cases, we can observe the effectiveness of the SLQ framework in optimizing SQL queries. It can not only identify non-standard SQL statements that cause low query efficiency but also provide specific optimization suggestions. These suggestions include moving time filters to utilize indexes, splitting queries to leverage indexes, using explicit JOIN syntax, and selecting necessary columns. By implementing these optimizations, the SLQ framework significantly improves query efficiency and database performance, thereby providing strong support for the stable operation and performance optimization of databases.

3.5. Hyperparameter Optimization

To justify the hyperparameter settings of the Stacked-LSTM Detector, we conducted experiments and visualized the results in Fig. 4, evaluating three key hyperparameters:

- Number of Layers: Tested values were 2, 3, and 4. As shown in the left subplot, accuracy peaks at 3 layers (98.41%), indicating 3 layers are optimal for capturing hierarchical features of SQL queries.
- Dropout Rate: Tested values were 0.1, 0.3, and 0.5. The middle subplot shows a Dropout rate of 0.3 balances regularization and performance, maintaining high accuracy while preventing overfitting.
- Learning Rate: Tested values were 0.0001, 0.001, and 0.01. The right subplot demonstrates 0.001 is optimal, as it achieves the highest accuracy without causing instability or slow convergence.

The current setting (3 layers, Dropout rate 0.3, learning rate 0.001) is thus validated as the optimal combination through this grid search and visual analysis."

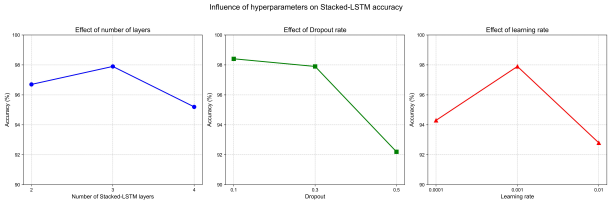

Fig. 4. Influence of hyperparameters on Stacked-LSTM accuracy.

3.6. Evaluations on small database

Due to the fact that the benchmark dataset only contains query statements, it is not possible to conduct actual optimization quality assessment. To quantify the effectiveness of the optimization suggestions, we conducted experiments on a small-scale database with 8 tables (user_info, order_details, product_catalog, etc.) containing 5,000-10,000 records each. From the original dataset (Section 3.1), 100 non-compliant queries were selected and adapted: their table/column names were modified to align with the small database's schema, while structural issues (e.g., full table scans, implicit type conversion) were retained.

The experiment was conducted on a hardware platform equipped with an Intel Core i7-12700H processor, 32GB of random-access memory (RAM), and a 1TB solid-state drive (SSD); for the software environment, the MySQL 8.0 database management system (with InnoDB as the storage engine) was adopted, and the entire experimental process was monitored using MySQL Workbench. Performance tests conducted on the adapted queries demonstrated significant improvements across key performance metrics following optimization: the average execution time of the adapted queries decreased from 420 ms in the pre-optimization phase to 256 ms in the post-optimization phase, achieving a 39% reduction in latency, while CPU utilization dropped from 58% (pre-optimization) to 32% (post-optimization), effectively reducing the consumption of computing resources.These results collectively validate the practical value of the proposed optimization recommendations, providing a feasible empirical basis for enhancing the performance of database queries.

Performance gain analysis of model integration were conducted on the same small-scale database. The test set

Table 3. Performance Gain Analysis of Model Integration.

Scenario	Detection	Suggestion	AvgTime(ms)	Suggestion Accuracy
Only Qwen3	No	Yes	277	97.3%
Only Stacked-LSTM	Yes	No	59	-
SLQ	Yes	Yes	227.6	98.7%

included 1,00 adapted queries (40% compliant, 60% non-compliant) from the small database. The scenarios and results are shown in Table 3. The average processing time per query for SLQ is 227.6ms. Although it is inferior to Stacked-LSTM, it has achieved effective SQL statement optimization functionality. Only Qwen3 lacks statement filtering, which tends to cause over-generation of suggestions, so the accuracy of its suggestions is not as high as that of SLQ.

4. Conclusion

This study proposes a modular SQL autonomous optimization framework named SLQ, whose core involves the collaborative operation of a Stacked-LSTM detector and a Qwen3-based repairer. The workflow of the SLQ framework is designed as follows: the detector first identifies the input query statements, triggering the repair process only for suspicious statements; at this point, the Tokenizer converts the original statement into tokens, based on which Qwen3 intelligently generates the rationale for classifying the statement as non-standard, along with optimization and repair suggestions, to assist users in reinforcing defenses. Experimental analysis shows that SLQ provides a lightweight and responsive solution for the stable operation of databases. Its capabilities in accurate statement identification and intelligent repair suggestions significantly enhance the autonomous optimization level of database systems when dealing with non-standard query inputs.

In the future, we will focus on optimizing the efficiency of suggestion generation. We will integrate a lightweight prefix tuning module in Qwen3 to reduce the number of parameter updates during the recommendation generation process. The goal is to reduce the average generation time from 150ms while maintaining a suggestion error rate of less than 1.5%.

Acknowledgment

This work is supported by the Science and Technology Project of State Grid Corporation of China (Project No.: SGSJ0000YWJS2400086).

References

[1] J. Yuanli, (2005) *"Optimization Methods for Database SQL Query Statements"* **Ordnance Automation 24**(6): 113–114. DOI: 10.3969/j.issn.1006-1576.2005.06.055.

[2] L. Jiajun and G. Mei, (2022) *"Research and Application of Oracle Query Optimization"* **Information Technology and Informatization** (1): 57–60. DOI: 10.3969/j.issn.1672-9528.2022.01.016.

[3] Z. Zheng, X. Yukun, J. Chao, et al., (2024) *"Research on Anomaly Detection in Electric Energy Metering Based on Improved Random Forest Algorithm"* **Electric Measurement and Instrumentation**: 1–8.

[4] L. Wei, C. Dongsheng, F. Fuyong, et al., (2024) *"Research on Short-Term Power Load Forecasting Based on DCT-CNN-GRU"* **Electric Measurement and Instrumentation**: 1–11.

[5] S. Jinwei, L. Junni, X. Donghai, et al., (2024) *"Research on Big Data Diagnosis Method for Transformer Condition Abnormality Based on Improved k-medoids Clustering and Carbon Constraint"* **Electrical Measurement and Instrumentation**: 1–10.

[6] H. Dan, Z. Yonggang, L. Shanhua, et al., (2024) *"Calculation and Analysis of Theoretical Line Losses in Low-Voltage Substations Based on AdaBoost Ensemble Learning Algorithm"* **Electrical Measurement and Instrumentation**: 1–9.

[7] S. Maesaroh, H. Gunawan, A. Lestari, et al., (2022) *"Query optimization in mysql database using index"* **International Journal of Cyber and IT Service Management 2**(2): 104–110.

[8] S. Palanisamy and P. SuvithaVani. "A survey on RDBMS and NoSQL Databases MySQL vs MongoDB". In: *2020 international conference on computer communication and informatics (ICCCI)*. IEEE, 2020, 1–7.

[9] M. Malekpour, N. Shaheen, F. Khomh, et al., (2024) *"Towards optimizing sql generation via llm routing"* **arXiv preprint arXiv:2411.04319**:

[10] M. M. Rahman, S. Islam, M. Kamruzzaman, et al., (2024) *"Advanced query optimization in SQL databases for real-time big data analytics"* **Academic Journal on Business Administration, Innovation & Sustainability *4***(3): 1–14.

[11] Y. Du, Z. Cai, and Z. Ding, (2024) *"Query Optimization in Distributed Database Based on Improved Artificial Bee Colony Algorithm"* **Applied Sciences *14***(2): 846.

[12] A. Uzzaman, M. M. I. Jim, N. Nishat, et al., (2024) *"Optimizing SQL databases for big data workloads: techniques and best practices"* **Academic Journal on Business Administration, Innovation & Sustainability *4***(3): 15–29.

[13] V. B. Ramu, (2023) *"Optimizing database performance: Strategies for efficient query execution and resource utilization"* **International Journal of Computer Trends and Technology *71***(7): 15–21.

[14] J. Shao, X. Liu, Y. Li, et al., (2015) *"Database performance optimization for SQL Server based on hierarchical queuing network model"* **International Journal of Database Theory and Application *8***(1): 187–196.

[15] K. S. Maabreh. "Optimizing Database Query Performance Using Table Partitioning Techniques". In: *2018 International Arab Conference on Information Technology (ACIT)*. IEEE, 2018, 1–4.

[16] J. Zhang. "Research on database application performance optimization method". In: *2016 6th International Conference on Machinery, Materials, Environment, Biotechnology and Computer*. Atlantis Press, 2016, 2236–2239.

[17] V. K. Myalapalli, T. P. Totakura, and S. Geloth. "Augmenting database performance via SQL tuning". In: *2015 International Conference on Energy Systems and Applications*. IEEE, 2015, 13–18.

[18] S. J. Kamatkar, A. Kamble, A. Viloria, et al. "Database performance tuning and query optimization". In: *International Conference on Data Mining and Big Data*. Springer International Publishing, 2018, 3–11.

[19] X. Sun, B. Jiang, and X. He. "Database query optimization based on distributed photovoltaic power generation". In: *2018 2nd IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*. IEEE, 2018, 2382–2386.

[20] C. Anneser, N. Tatbul, D. Cohen, et al., (2023) *"Autosteer: Learned query optimization for any sql database"* **Proceedings of the VLDB Endowment *16***(12): 3515–3527.