|  |  |
| --- | --- |
|  | GenAI SQL |
|  |  |
|  | HANS ESQUIVEL  [Course title]  5/10/25 |

Table of Contents

[Proposed Features (New and Merged Items) 4](#_Toc197777549)

[SQL Performance Benchmarking and Optimization 6](#_Toc197777550)

[Implementation Plan 6](#_Toc197777551)

[New File: tasks/sql\_performance\_benchmark.py 7](#_Toc197777552)

[Prompt Update: prompts/index.yaml 7](#_Toc197777553)

[Testing 8](#_Toc197777554)

[Steps to Test 8](#_Toc197777555)

[Support for Multiple SQL Dialects (Future State) 9](#_Toc197777556)

[Implementation Plan 9](#_Toc197777557)

[SQL Query Validator 10](#_Toc197777558)

[Implementation Plan 10](#_Toc197777559)

[**1. Define the SQLQueryValidator Task** 10](#_Toc197777560)

[**2. Add Prompts to prompts/index.yaml** 11](#_Toc197777561)

[**3. Integrate the Task into the CLI** 11](#_Toc197777562)

[**4. Write Unit Tests** 11](#_Toc197777563)

[Testing 11](#_Toc197777564)

[**1. Unit Tests** 11](#_Toc197777565)

[**2. CLI Testing** 12](#_Toc197777566)

[**3. Sample SQL Queries** 12](#_Toc197777567)

[Steps to Test 12](#_Toc197777568)

[Overview of Enhanced Security Audits 13](#_Toc197777569)

[Implementation Plan 13](#_Toc197777570)

[Testing 14](#_Toc197777571)

[**1. Unit Tests** 14](#_Toc197777572)

[**2. CLI Testing** 15](#_Toc197777573)

[**3. Sample SQL Queries** 15](#_Toc197777574)

[Steps to Test 16](#_Toc197777575)

[Natural Language Query to SQL Conversion 16](#_Toc197777576)

[Implementation Plan 17](#_Toc197777577)

[Testing 18](#_Toc197777578)

[Steps to Test 19](#_Toc197777579)

# Proposed Features (New and Merged Items)

1. **SQL Performance Benchmarking and Optimization**:
   * Simulates query execution and provides performance metrics (e.g., execution time, index usage).
   * Suggests indexing, partitioning, and tuning options.
   * **Merge Note:** Combined with "SQL Performance Optimization Suggestions."
2. **Support for Multiple SQL Dialects**:
   * Adds compatibility for various SQL dialects (e.g., MySQL, PostgreSQL, Oracle).
   * **Impact:** Updates may be required in all tasks/sql\_\* files and prompts/index.yaml.
3. **Query Simulation and Validation**:
   * Simulates query execution to validate syntax and logic without needing a live database.
   * **Impact:** New utilities for mock database interactions.
4. **Enhanced Security Audits**:
   * Expands existing audits to include:
     + Detection of SQL injection vulnerabilities.
     + Identification of improper permissions and role usage.
   * **Implementation:** Updates to tasks/sql\_security\_auditor.py.
5. **Automated Migration Scripts**:
   * Generates migration scripts for schema and data transformations.
   * **New File:** tasks/sql\_migration\_generator.py.
6. **Interactive Web Interface**:
   * Adds a web-based UI for running tasks, viewing results, and managing configurations.
   * **New Directory:** Web framework integration (e.g., Flask/Django).
7. **Integration with CI/CD Pipelines**:
   * Provides CLI commands and configuration templates for CI/CD tools (e.g., Jenkins, GitHub Actions).
   * **Impact:** May require changes to the CLI and documentation.
8. **Natural Language Query to SQL Conversion**:
   * Converts plain language queries into SQL using LLMs.
   * **New File:** tasks/natural\_language\_to\_sql.py.
9. **AI-Powered Query Explanation (Enhanced)**:
   * Extends the existing explanation tool with richer visualizations (e.g., flow diagrams, execution plans).
   * **Merge Note:** Combined with the existing "AI-Powered SQL Explanation" feature.
10. **Custom Workflow Configuration**:
    * Allows users to define custom workflows by chaining tasks in prompts/index.yaml.
11. **Dynamic SQL Detection**:
    * Identifies and analyzes dynamically generated SQL for risks and optimization.
    * **New Utility:** utils/dynamic\_sql\_detector.py.
12. **Integration with Cloud Platforms**:
    * Adds support for connecting to cloud databases (e.g., AWS RDS, Azure SQL, Google BigQuery).
    * **Impact:** Updates to core/config\_loader.py for cloud-specific configurations.
13. **Data Masking and Anonymization**:
    * Automatically masks sensitive data in SQL queries for testing or sharing purposes.
    * **New File:** tasks/sql\_data\_masker.py.
14. **Visualization and Insights**:
    * Generates visual representations (e.g., query plans, schema diagrams) from SQL code to aid debugging.
    * **New File:** tasks/sql\_visualizer.py.
15. **Error Correction and Debugging**:
    * Provides suggestions and corrections for SQL errors or warnings using LLM-based analysis.
    * **New File:** tasks/sql\_error\_corrector.py.
16. **Automated Query Generation**:
    * Generates SQL queries from natural language descriptions or metadata.
    * **Merge Note:** Related to "Natural Language Query to SQL Conversion."
17. **SQL Style Guide Enforcement**:
    * Enforces SQL coding standards (e.g., naming conventions, formatting rules).
    * **New File:** tasks/sql\_style\_enforcer.py.
18. **SQL Education Mode**:
    * Adds an interactive tutorial mode to guide users through creating and optimizing SQL queries step-by-step.
    * **New Directory:** education/.

# SQL Performance Benchmarking and Optimization

This feature involves simulating query execution and providing performance metrics such as:

* Estimated execution time.
* Index usage.
* Suggestions for optimizations like indexing, partitioning, and query tuning.

## Implementation Plan

**Step 1: Define the Scope**

1. Simulate query execution using mock data or integration with supported databases.
2. Analyze query execution plans and provide insights.
3. Suggest optimizations (e.g., indexing, partitioning).
4. Ensure compatibility with multiple SQL dialects for broader usability.

**Step 2: Create or Update Files**

1. Create a new task file: tasks/sql\_performance\_benchmark.py.
2. Update core/sql\_task\_base.py (if necessary) to include any shared logic for this task.
3. Add new prompts in prompts/index.yaml to retrieve performance insights from the AI model.

### New File: tasks/sql\_performance\_benchmark.py

The following file will define the SQL Performance Benchmarking tool:

**tasks/sql\_performance\_benchmark.py**v2

"""

SQL Performance Benchmark Tool (Async + Modular)

Simulates query execution and provides performance metrics and optimization suggestions.

"""

### Prompt Update: prompts/index.yaml

Add a new entry for performance benchmarking:

**prompts/index.yaml**

performance\_benchmark.simulate:

inline: |

"You are a database performance optimization expert. Analyze the SQL query below and:

1. Estimate its execution time and resource usage.

2. Identify potential performance bottlenecks.

3. Suggest indexing, partitioning, or query tuning strategies to improve performance.

## Testing

1. **Unit Tests**:
   * Create unit tests for SQLPerformanceBenchmark in a new file: tests/test\_sql\_performance\_benchmark.py.
   * Mock responses from the AI client to validate the output.
2. **CLI Integration**:
   * Ensure the task is callable from the CLI with a new flag, e.g., --task=benchmark.
3. **Validation**:
   * Test with SQL queries in various dialects (e.g., MySQL, PostgreSQL, SQL Server).
   * Validate performance suggestions against real query execution plans (optional).

### Steps to Test

1. **Unit Test**:
   * Run the following command to execute the unit test:

bash

pytest tests/test\_sql\_performance\_benchmark.py

1. **CLI Integration**:
   * **Test the CLI by running the benchmark task on an SQL file:**

bash

python app.py --task=benchmark --path="queries/sample.sql" --dry-run

* + **Test recursive directory processing:**

bash

python app.py --task=benchmark --path="queries/" --recursive --dry-run

* + **Test with backup and Git staging:**

bash

python app.py --task=benchmark --path="queries/sample.sql" --backup –git

# Support for Multiple SQL Dialects (Future State)

This feature involves adding compatibility for various SQL dialects, such as MySQL, PostgreSQL, and Oracle. It ensures that tasks like analysis, commenting, and benchmarking work correctly across different SQL dialects.

## Implementation Plan

**Step 1: Define Scope**

1. Ensure all SQL-related tasks (e.g., commenting, analysis, benchmarking) can handle dialect-specific syntax and features.
2. Update the prompts/index.yaml file to include dialect-specific prompts where necessary.
3. Introduce a configuration or detection mechanism to determine the SQL dialect.
4. Validate tasks with real-world queries from supported SQL dialects.

**Step 2: Tasks to Update**

* tasks/sql\_commenter.py
* tasks/sql\_analyzer.py
* tasks/sql\_refactorer.py
* tasks/sql\_explainer.py
* tasks/sql\_security\_auditor.py
* tasks/sql\_performance\_benchmark.py

**Step 3: Configuration**

* Add a dialect argument to the existing tasks. This can either be passed via the CLI or detected automatically based on the SQL query.

**Step 4: Code Changes**

1. Update the base SQL task class (core/sql\_task\_base.py) to include dialect handling.
2. Update individual tasks to use the new dialect functionality.

**Step 5: Testing**

* Test all tasks with queries from multiple SQL dialects to ensure compatibility and correctness.

# SQL Query Validator

The sql\_query\_validator.py script will implement a tool to simulate and validate SQL queries using AI. The key responsibilities of this script include:

1. **Simulating Query Execution**: Provide insights into how the query would execute, including identifying potential bottlenecks or inefficiencies.
2. **Syntax Validation**: Detect syntax errors in the query and provide corrections if needed.
3. **Logical Validation**: Check for logical issues like ambiguous joins, unused clauses, or missing WHERE conditions for updates/deletions.
4. **Feedback and Suggestions**: Provide actionable recommendations to improve the query's performance and correctness.

## Implementation Plan

**1. Define the SQLQueryValidator Task**

* Create a new Python class, SQLQueryValidator, in the tasks/ directory.
* This class will inherit from the SQLTask base class and implement the run method to:
  + Accept an SQL query as input.
  + Use an AI model to simulate and validate the query.
  + Return a clear and concise validation report.

**2. Add Prompts to prompts/index.yaml**

* Add a new prompt for the query validator task in the prompts/index.yaml file.
* The prompt will instruct the AI to simulate the query and validate its syntax, logic, and efficiency.

**3. Integrate the Task into the CLI**

* Register the SQLQueryValidator task in the TASKS dictionary in app.py.
* Allow users to invoke the query validator using the --task=validate flag in the CLI.

**4. Write Unit Tests**

* Create a new test file, tests/test\_sql\_query\_validator.py.
* Write unit tests to validate the behavior of the SQLQueryValidator class and mock AI responses.

## Testing

**1. Unit Tests**

* Run the unit tests to validate the SQLQueryValidator class:

bash

pytest tests/test\_sql\_query\_validator.py

**2. CLI Testing**

* Test the query validator via the CLI:

bash

python app.py --task=validate --path="queries/sample.sql" --dry-run

**3. Sample SQL Queries**

* Use various SQL queries (simple and complex) to test the tool:
  + Valid query: SELECT name, age FROM users WHERE active = true;
  + Invalid query: SELECT name age FROM users WHERE active = true; (missing comma)
  + Logical issue: DELETE FROM users; (missing WHERE clause)

## Steps to Test

1. **Set Up Project**:
   * Ensure the tasks/sql\_query\_validator.py file and prompts/index.yaml entries are properly added.
   * Register the task in app.py.
2. **Run Unit Tests**:
   * Execute the unit test file with pytest to validate the implementation.
3. **Test End-to-End CLI**:
   * Use the CLI to validate SQL queries in single-file or directory modes.
4. **Verify Outputs**:
   * Ensure the output includes syntax validation, logical feedback, and actionable suggestions.

# Overview of Enhanced Security Audits

The Enhanced Security Audit tool will analyze SQL queries for potential security vulnerabilities and compliance risks. It will:

1. **Detect Common Vulnerabilities**:
   * SQL Injection risks.
   * Unsafe dynamic SQL usage.
   * Improper data exposure (PII/PHI, e.g., SSNs, passwords).
   * Absence of access controls.
2. **Highlight Compliance Risks**:
   * Identify risks related to HIPAA, HITECH, or other regulatory requirements.
3. **Severity Classification**:
   * Classify vulnerabilities as Critical, High, Medium, or Low.
4. **Provide Fix Recommendations**:
   * Suggest actionable steps to remediate issues.

The tool should be integrated into the CLI and tested to ensure it provides accurate and actionable security insights.

## Implementation Plan

**1. Define the Enhanced Security Audit Task**

* Create a new class named EnhancedSQLSecurityAuditor in the tasks/sql\_security\_auditor.py file.
* Implement the run method to:
  + Accept an SQL query as input.
  + Analyze the query for vulnerabilities using an AI model.
  + Return a detailed report of findings, severity levels, and recommendations.

**2. Update Prompts in prompts/index.yaml**

* Add a new prompt specifically designed for enhanced security auditing.
* Include merged details from the original prompt (e.g., SQL Injection, PHI/PII risks) and the enhanced version (e.g., severity classification, compliance risks).

**3. Integrate into the CLI**

* Register the Enhanced Security Auditor as a task in the TASKS dictionary in app.py.
* Allow users to invoke the task via the --task=audit flag.

**4. Write Unit Tests**

* Create a new test file, tests/test\_sql\_security\_auditor.py.
* Write tests to:
  + Mock AI responses for various scenarios (e.g., vulnerable queries, safe queries).
  + Verify that the output matches expected results, including severity classification and remediation suggestions.

## Testing

**1. Unit Tests**

* Run the unit test file:

bash

pytest tests/test\_sql\_security\_auditor.py

**2. CLI Testing**

* Test the Enhanced Security Audit via the CLI:

bash

python app.py --task=audit --path="queries/vulnerable.sql" --dry-run

**3. Sample SQL Queries**

* **Vulnerable Query**:

SQL

SELECT \* FROM users WHERE username = 'admin' AND password = 'password123';

* + Expected: Flags SQL injection and sensitive data exposure.
* **Safe Query**:

SQL

SELECT \* FROM users WHERE username = ? AND password = ?;

* + Expected: No vulnerabilities.
* **Sensitive Data Access**:

SQL

SELECT credit\_card\_number FROM transactions;

* + Expected: Flags sensitive data access.

## Steps to Test

1. **Set Up Project**:
   * Ensure the tasks/sql\_security\_auditor.py file, prompts/index.yaml entry, and CLI integration are properly added.
2. **Run Unit Tests**:
   * Validate the functionality of EnhancedSQLSecurityAuditor using mocked responses.
3. **Test End-to-End CLI Functionality**:
   * Use the CLI to audit SQL queries for vulnerabilities.
4. **Verify Outputs**:
   * Confirm that the tool:
     + Detects vulnerabilities.
     + Classifies severity levels.
     + Provides actionable recommendations.

# Natural Language Query to SQL Conversion

The **Natural Language Query to SQL Conversion** feature allows users to input natural language queries (e.g., "Get all users who joined in the last month") and converts these into valid SQL queries. It leverages a predefined JSON schema file that describes the database structure, including tables, columns, and relationships. This schema file is generated using a T-SQL stored procedure and referenced at runtime for query generation.

## Implementation Plan

**1. Define the Natural Language Query to SQL Task**

* Create a task class NaturalLanguageToSQL in tasks/natural\_language\_to\_sql.py.
* Implement the run method:
  + Accept a natural language query and SQL dialect as input.
  + Load the predefined JSON schema file.
  + Use the AI model to convert the query into SQL based on the schema and the provided dialect.
  + Return the generated SQL query.

**2. Create a T-SQL Stored Procedure to Generate the JSON Schema**

* Define the stored procedure sp\_GenerateSchemaJSON:
  + Accept a database name as a parameter.
  + Generate the schema metadata using INFORMATION\_SCHEMA views.
  + Format the metadata into JSON, including table names, columns, and relationships (e.g., primary and foreign keys).
  + Output the JSON schema as a result.
* Ensure the stored procedure is globally accessible by installing it in the master database and prefixing it with sp\_.

**3. Update the Prompt**

* Add a new prompt for converting natural language to SQL:
  + Include the schema JSON in the prompt to provide the AI with context about the database structure.
  + Allow users to specify the SQL dialect (e.g., T-SQL, PostgreSQL).

**4. CLI Integration**

* Register the NaturalLanguageToSQL task in the TASKS dictionary in app.py.
* Enable users to invoke the task via a CLI flag, e.g., --task=nl\_to\_sql.

**5. Unit Tests**

* Write unit tests in tests/test\_natural\_language\_to\_sql.py:
  + Mock the AI response for various natural language inputs.
  + Ensure the outputs match the expected SQL queries.
  + Verify that the schema file is loaded correctly.

## Testing

**1. Unit Tests**

* **Test Case 1**:
  + Input: "Get all users who joined in the last month."
  + Expected Output: SELECT \* FROM users WHERE created\_at >= NOW() - INTERVAL '1 month';
* **Test Case 2**:
  + Input: "List all orders for customer ID 123."
  + Expected Output: SELECT \* FROM orders WHERE customer\_id = 123;
* **Test Case 3**:
  + Input: "Show the total sales grouped by product category."
  + Expected Output: SELECT category, SUM(sales) FROM sales\_data GROUP BY category;

**2. Stored Procedure Validation**

* Run the stored procedure:

SQL

EXEC sp\_GenerateSchemaJSON @DatabaseName = 'YourDatabaseName';

* Verify the JSON output matches the expected schema structure.

**3. CLI Testing**

* Test the feature via the CLI:

bash

python app.py --task=nl\_to\_sql --path="queries/nl\_query.txt" --sql\_dialect="T-SQL"

* Validate the generated SQL against expected outputs.

## Steps to Test

**Step 1: Create the Stored Procedure**

* Deploy the sp\_GenerateSchemaJSON procedure to the database server.

**Step 2: Generate the Schema File**

* Call the stored procedure for the target database:

SQL

EXEC sp\_GenerateSchemaJSON @DatabaseName = 'YourDatabaseName';

* Copy the JSON output to a file, e.g., schema.json.

**Step 3: Configure the Tool**

* Place the schema.json file in the appropriate directory for the NaturalLanguageToSQL task to access.

**Step 4: Run Unit Tests**

* Execute the test file:

bash

pytest tests/test\_natural\_language\_to\_sql.py

**Step 5: Test via CLI**

* Run the CLI tool for various natural language queries:

bash

python app.py --task=nl\_to\_sql --path="queries/nl\_query.txt" --sql\_dialect="T-SQL"

* Verify the generated SQL matches expectations.

**Step 6: Test with Different Databases**

* Repeat the process with other databases by generating new schema files using sp\_GenerateSchemaJSON.