CODING

Batch 11 - NATURAL LANGUAGE TO SQL QUERY GENERATION USING GEN AI

```
import sys
import os
sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), "../")))
import io
import json
import re
import logging
from typing import Dict, List, Optional, Union, TypedDict
import pandas as pd
import plotly.express as px
import plotly.figure_factory as ff
import plotly.graph_objects as go
from scipy import stats
import statsmodels.api as sm
from statsmodels.stats.outliers_influence import variance_inflation_factor
from statsmodels.tsa.seasonal import seasonal_decompose
import streamlit as st
from dotenv import load_dotenv
from streamlit_extras.colored_header import colored_header
import streamlit_nested_layout
import numpy as np
from streamlit_extras.dataframe_explorer import dataframe_explorer
import src.database.DB_Config as DB_Config
from src.prompts.Base_Prompt import SYSTEM_MESSAGE
from src.api.LLM_Config import get_completion_from_messages
import hashlib
from datetime import datetime
from time import time
from collections import defaultdict
from jsonschema import validate as json_validate, ValidationError

# Configure logging
logging.basicConfig(level=logging.DEBUG)
logger = logging.getLogger(__name__)

SUPPORTED_CHART_TYPES = {
    "Bar Chart": "A chart that presents categorical data with rectangular bars.",
    "Line Chart": "A chart that displays information as a series of data points called 'markers' connected by straight line segments.",
    "Scatter Plot": "A plot that displays values for typically two variables for a set of data.",
    "Area Chart": "A chart that displays quantitative data visually, using the area below the line.",
    "Histogram": "A graphical representation of the distribution of numerical data.",
    "Pie Chart": "A chart that shows proportions of a whole using slices.",
    "Box Plot": "A chart that shows the distribution of data based on quartiles."
}

# Page Configuration with dark theme details
st.set_page_config(
    page_icon="🗃️",
    page_title="NLP2SQL",
    layout="wide"
)

def apply_custom_theme():
    custom_css = f"""
    <style>
    /* Global Styles */
    body, .stApp {{
        background-color: #1e1e1e;
        color: #64ffda;
        font-family: sans-serif;
    }}
    /* Sidebar */
    .css-1d391kg, .stSidebar .sidebar-content {{
        background-color: #333333;
    }}
    /* Buttons */
    .stButton>button {{
        background-color: #00ADB5;
        color: #fff;
        border: none;
    }}
    /* Expander */
    .stExpander {{
        background-color: #333333;
        border: none;
        border-radius: 8px;
        padding: 0.5rem;
    }}
    .stExpander .stExanderHeader, .stExpander .stExanderContent {{
        color: #64ffda;
    }}
    /* Tabs */
    .stTabs [data-baseweb="tab"] {{
        background-color: #333333;
        border-radius: 6px;
        padding: 0.5rem 1rem;
        color: #64ffda;
    }}
    .stTabs [data-baseweb="tab"][aria-selected="true"] {{
        background-color: #00ADB5;
        color: #fff;
    }}
```

```
    /* Code Blocks */
    pre {{
        background-color: #333333;
        color: #64ffda;
    }}
    </style>
    """
    st.markdown(custom_css, unsafe_allow_html=True)

# Apply the custom theme early
apply_custom_theme()

load_dotenv()

@st.cache_resource
def load_system_message(schemas: dict) -> str:
    """Load and format the system message with JSON-serialized schemas."""
    return SYSTEM_MESSAGE.format(schemas=json.dumps(schemas, indent=2))

# Add input validation to prevent SQL injection and other security vulnerabilities

def validate_sql_query(query: str) -> bool:
    """
    Ensure the SQL query is valid and safe (select queries only).

    Parameters:
    - query (str): The SQL query to validate.

    Returns:
    - bool: True if the query is valid and safe, False otherwise.
    """
    if not isinstance(query, str):
        return False

    disallowed_keywords = r'\b(DROP|DELETE|INSERT|UPDATE|ALTER|CREATE|EXEC)\b'

    if re.search(disallowed_keywords, query, re.IGNORECASE):
        return False

    if not query.strip().lower().startswith(('select', 'with')):
        return False

    if query.count('(') != query.count(')'):
        return False

    return True

# --- New helper: Validate that query uses existent tables/columns ---
def validate_query_tables(query: str, schemas: dict) -> bool:
    """
    Very basic check: warn if any known schema table name is missing in the query.
    This is a heuristic check.
    """
    lower_query = query.lower()
    missing = []
    for table in schemas.keys():
        if table.lower() not in lower_query:
            missing.append(table)
    if missing:
        logging.warning(f"LLM query does not mention these tables from the schema: {', '.join(missing)}")
        return False
    return True
```

```
def get_data(query: str, db_name: str, db_type: str, host: Optional[str] = None, user: Optional[str] = None, password: Optional[str] = None) -> pd.DataFrame:
    """Run the specified query and return the complete resulting DataFrame."""
    if not validate_sql_query(query):

        st.dataframe(filtered_stats.style.format("{:.2f}").highlight_max(axis=0, color="lightgreen"))

        # Histograms for meaningful distributions
        for col in numeric_cols:
            if df[col].nunique() > 1:
                st.markdown(f"**Distribution of {col}**")
                st.plotly_chart(px.histogram(df, x=col, nbins=30, title=f"Histogram of {col}"), use_container_width=True)

    # --- CATEGORICAL ANALYSIS ---
    with tab2:
        st.markdown("### Categorical Data Insights")

        for col in categorical_cols:
            value_counts = df[col].value_counts()
            unique_count = value_counts.shape[0]

            # Only show if the column has meaningful variability
            if unique_count < len(df) * 0.8:
                st.markdown(f"**{col}:** {unique_count} unique values")
                freq_table = value_counts.reset_index()
                freq_table.columns = ["Category", "Count"]
                freq_table["Percentage"] = (freq_table["Count"] / len(df) * 100).round(2)
                st.table(freq_table.style.format({"Percentage": "{:.2f}%"}))

                if unique_count <= 10:
                    st.plotly_chart(px.pie(freq_table, names="Category", values="Count", title=f"Pie Chart for {col}"), use_container_width=True)
                else:
                    st.plotly_chart(px.bar(freq_table, x="Category", y="Count", title=f"Bar Chart for {col}"), use_container_width=True)


    # --- MISSING DATA & CORRELATIONS ---
    with tab3:
        st.markdown("### Missing Data Analysis")


        missing_data = df.isnull().sum()
        missing_data = missing_data[missing_data > 0]
        if not missing_data.empty:
            missing_df = missing_data.reset_index()
            missing_df.columns = ["Column", "Missing Values"]
            missing_df["Percentage"] = (missing_df["Missing Values"] / len(df) * 100).round(2)
            st.table(missing_df.style.format({"Percentage": "{:.2f}%"}))
        else:
            st.success("No missing data detected.")


        st.markdown("### Correlation Matrix")
        if len(numeric_cols) >= 2:
            correlation_matrix = df[numeric_cols].corr()
            heat_fig = px.imshow(correlation_matrix, text_auto=True, aspect="auto", title="Correlation Matrix")
            st.plotly_chart(heat_fig, use_container_width=True)
        else:
            st.info("Not enough numeric columns for correlation analysis.")


def perform_advanced_analysis(df: pd.DataFrame) -> None:
    """Perform advanced statistical analysis on the dataset."""
    st.markdown("## 📊 Advanced Statistical Analysis")


    # Create tabs for different analyses
    tabs = st.tabs(["Distribution Analysis", "Outlier Detection", "Time Series Analysis", "Feature Relationships"])

    numeric_cols = df.select_dtypes(include=[np.number]).columns
    datetime_cols = df.select_dtypes(include=['datetime64']).columns

    with tabs[0]:
        st.markdown("### 📈 Distribution Analysis")
        if len(numeric_cols) > 0:
            col = st.selectbox("Select column for distribution analysis", numeric_cols)

            # Calculate statistical measures
            skewness = stats.skew(df[col].dropna())
            kurtosis = stats.kurtosis(df[col].dropna())

            # Create distribution plot
            fig = ff.create_distplot([df[col].dropna()], [col], bin_size=0.2)
            st.plotly_chart(fig, use_container_width=True)

            # Display statistical measures
            col1, col2, col3 = st.columns(3)
            col1.metric("Skewness", f"{skewness:.2f}")
            col2.metric("Kurtosis", f"{kurtosis:.2f}")
            col3.metric("Normality Test p-value", f"{stats.normaltest(df[col].dropna())[1]:.4f}")

    with tabs[1]:
        st.markdown("### 🔍 Outlier Detection")
        if len(numeric_cols) > 0:
            col = st.selectbox("Select column for outlier detection", numeric_cols, key="outlier_col")

            # Calculate outliers using IQR method
            Q1 = df[col].quantile(0.25)
            Q3 = df[col].quantile(0.75)
            IQR = Q3 - Q1
            outliers = df[(df[col] < (Q1 - 1.5 * IQR)) | (df[col] > (Q3 + 1.5 * IQR))][col]

            # Create box plot
            fig = go.Figure()
            fig.add_trace(go.Box(y=df[col], name=col))
            st.plotly_chart(fig, use_container_width=True)

            if not outliers.empty:
                st.markdown(f"**Found {len(outliers)} outliers:**")
                st.dataframe(outliers)

    with tabs[2]:
        st.markdown("### ⏳ Time Series Analysis")
        if len(datetime_cols) > 0:
            date_col = st.selectbox("Select date column", datetime_cols)
            value_col = st.selectbox("Select value column", numeric_cols)

            # Ensure data is sorted by date
            ts_data = df[[date_col, value_col]].sort_values(date_col)
            ts_data = ts_data.set_index(date_col)

            # Automatically detect the period based on the frequency of the date column
            period = st.number_input("Enter the period for seasonal decomposition (default is 12)", min_value=1, value=12)

            # Perform seasonal decomposition
            try:
                decomposition = seasonal_decompose(ts_data[value_col], period=period)

                # Plot components
                fig = go.Figure()
                fig.add_trace(go.Scatter(x=ts_data.index, y=decomposition.trend, name='Trend'))
                fig.add_trace(go.Scatter(x=ts_data.index, y=decomposition.seasonal, name='Seasonal'))
                fig.add_trace(go.Scatter(x=ts_data.index, y=decomposition.resid, name='Residual'))
                fig.update_layout(title='Time Series Decomposition')
                st.plotly_chart(fig, use_container_width=True)
            except Exception as e:
                st.warning("Could not perform seasonal decomposition. Ensure enough data points and regular intervals.")

    with tabs[3]:
        st.markdown("### 🔗 Feature Relationships")
        if len(numeric_cols) >= 2:
            # Correlation analysis
            correlation = df[numeric_cols].corr()

            # Heatmap
            fig = px.imshow(correlation,
                          labels=dict(color="Correlation"),
                          title="Feature Correlation Matrix")
            st.plotly_chart(fig, use_container_width=True)

            # VIF Analysis
            if st.checkbox("Show Variance Inflation Factor (VIF) Analysis"):
                if len(numeric_cols) < 2:
                    st.warning("At least two numeric columns are required to calculate VIF.")
                else:
                    try:
                        X = df[numeric_cols].dropna()
                        vif_data = pd.DataFrame()
                        vif_data["Feature"] = numeric_cols
                        vif_data["VIF"] = [variance_inflation_factor(X.values, i)
                    for i in range(X.shape[1])]
               st.dataframe(vif_data.sort_values('VIF', ascending=False))
            except Exception as e:
               st.warning("Could not calculate VIF. Check for multicollinearity or missing values.")

def assess_data_quality(df: pd.DataFrame) -> None:
   """Assess the quality of the dataset and provide detailed insights."""
   st.markdown("## 🔍 Data Quality Assessment")

   # Create tabs for different quality checks
   tabs = st.tabs(["Overview", "Missing Values", "Duplicates", "Consistency", "Anomalies"])

   with tabs[0]:
      st.markdown("### 📊 Data Quality Overview")

      # Basic statistics
      total_rows = len(df)
      total_cols = len(df.columns)
      memory_usage = df.memory_usage(deep=True).sum() / 1024**2  # in MB

      # Display metrics
      col1, col2, col3, col4 = st.columns(4)
      col1.metric("Total Rows", f"{total_rows:,}")
      col2.metric("Total Columns", total_cols)
      col3.metric("Memory Usage", f"{memory_usage:.2f} MB")
      col4.metric("Data Types", len(df.dtypes.unique()))

      # Data type distribution
      dtype_counts = df.dtypes.value_counts()
      fig = px.pie(values=dtype_counts.values,
            names=dtype_counts.index.astype(str),
            title="Column Data Type Distribution")
```

```
    st.plotly_chart(fig, use_container_width=True)

with tabs[1]:
    st.markdown("### ❌ Missing Values Analysis")

    # Calculate missing values
    missing = df.isnull().sum()
    missing_pct = (missing / len(df) * 100).round(2)
    missing_df = pd.DataFrame({
        'Column': missing.index,
        'Missing Count': missing.values,
        'Missing Percentage': missing_pct.values
    }).sort_values('Missing Percentage', ascending=False)

    # Display missing values
    if missing_df['Missing Count'].sum() > 0:
        st.dataframe(missing_df)

        # Visualize missing values
        fig = px.bar(missing_df,
                    x='Column',
                    y='Missing Percentage',
                    title="Missing Values by Column")
        st.plotly_chart(fig, use_container_width=True)

            if st.button(f"🔄 Re-run Query {i + 1}", key=f"rerun_query_{i}"):
                user_message = row['Query']
                with st.spinner('🔄 Re-running the saved SQL query...'):
                    selected_schemas = {table: schemas[table] for table in selected_tables}
                    response = generate_sql_query(user_message, selected_schemas)
                    handle_query_response(
                        response,
                        db_file if db_type == "SQLite" else postgres_db,
                        db_type,
                        host=postgres_host if db_type == "PostgreSQL" else None,
                        user=postgres_user if db_type == "PostgreSQL" else None,
                        password=postgres_password if db_type == "PostgreSQL" else None
                    )

    st.write(f"Page {current_page} of {num_pages}")

else:
    st.info("📭 No query history available.")
```