# Simplifying Data Analysis: Python Classes and GenAI for RDMS

## Abstract

Title: Simplifying Data Analysis: Python Classes and GenAI for RDMS.  
  
Abstract: In modern data-driven environments, the ability to efficiently retrieve and analyze data from relational database management systems (RDMS) is crucial. Traditionally, SQL has been the primary tool for querying databases. However, advancements in Python programming and generative AI (GenAI) offer new paradigms for data interaction. This paper presents a method to use Python for creating dynamic classes that interact with RDMS tables, fetching data into Pandas DataFrames. These classes are equipped with metadata that describes the table structure and business context, facilitating insightful data analysis through natural language queries.

## Introduction

The management and analysis of large datasets stored in RDMS are foundational to business intelligence and decision-making processes. While SQL remains a powerful tool for querying databases, Python's rich ecosystem, including libraries such as Pandas, offers enhanced flexibility and capabilities for data manipulation and analysis. This paper explores an innovative approach to bridge SQL-based data retrieval with the analytical prowess of Python and GenAI.

## Methodology

1. Class Design for Data Retrieval:

- Structure: Each RDMS table is represented by a Python class. The class includes methods to fetch rows from the table into a Pandas DataFrame.

- Metadata: Classes maintain metadata that includes the table name, a description of the table, and additional context.

2. Using GenAI for Data Analysis:

- Integrating GenAI: Describing how generative AI models can be incorporated to understand and generate natural language queries for data analysis.

- Example Workflow: A step-by-step explanation of how a user can interact with the system using natural language queries to retrieve and analyze data.

## Coding Implementation

1. Setup\_database.py:  
We created a SQL database here example.db using the SQLite3 module and then we made an employees table and filled it with sample data. Now SQLite3 helps in interacting with other SQLite databases.  
  
i. Importing the sqlite3 module:  
import sqlite3  
This line helps in importing the sqlite3 module.  
  
ii. Connecting to the SQLite database:  
conn = sqlite3.connect('example.db')  
cursor = conn.cursor()  
These lines connect to a SQLite database named example.db. If the database file does not exist, it will be created. A cursor object is then created, which allows us to execute SQL commands.  
  
iii. Creating the employees table:  
cursor.execute('''  
CREATE TABLE IF NOT EXISTS employees (  
id INTEGER PRIMARY KEY AUTOINCREMENT,  
name TEXT NOT NULL,  
department TEXT NOT NULL,  
salary REAL NOT NULL  
)  
''')  
This SQL command creates a table named employees with the following columns:  
- id: An integer that is automatically incremented as the primary key.  
- name: A text field for the employee's name, which cannot be NULL.  
- department: A text field for the employee's department, which cannot be NULL.  
- salary: A real number field for the employee's salary, which cannot be NULL.  
The IF NOT EXISTS clause ensures that the table is only created if it does not already exist.  
  
iv. Inserting sample employee data:  
employee\_data = [  
 ('Alice', 'HR', 50000),  
 ('Bob', 'Engineering', 60000),  
 ('Charlie', 'Finance', 55000),  
 ('Harry', 'Finance', 40000),  
 ('Sam', 'Engineering', 70000),  
 ('David', 'HR', 60000)  
]  
cursor.executemany('''  
INSERT INTO employees (name, department, salary) VALUES (?, ?, ?)  
''', employee\_data)  
A list of tuples, employee\_data, is created to hold the sample data. Each tuple contains values for the name, department, and salary columns. The cursor.executemany method is used to execute the INSERT statement for each tuple in the list, adding the sample data to the employees table.  
  
v. Committing the changes and closing the connection:  
conn.commit()  
conn.close()  
These lines commit the changes to the database to ensure that all operations are saved. Finally, the connection to the database is closed.  
  
2. Database\_table.py:  
This code defines a DatabaseTable class that uses the pandas and sqlalchemy libraries to interact with a database. It allows you to fetch data from a specified table in the database. Here’s a step-by-step explanation:  
  
i. Importing required modules:  
import pandas as pd  
from sqlalchemy import create\_engine  
- pandas: A powerful data manipulation and analysis library.  
- sqlalchemy: A SQL toolkit and Object-Relational Mapping (ORM) library for Python, which simplifies database interactions.

ii. Defining the dataBase table class:  
class DatabaseTable:

    def \_\_init\_\_(self, table\_name, connection\_string):

        self.table\_name = table\_name

        self.engine = create\_engine(connection\_string)

The DatabaseTable class is defined with an \_\_init\_\_ method that initializes the class with two parameters:

* table\_name: The name of the table in the database from which data will be fetched.
* connection\_string: The connection string used to connect to the database.

The create\_engine function from sqlalchemy is used to create an engine object, which is responsible for managing the connection to the database.

iii. Defining the fetch\_data method:  
def fetch\_data(self):

    query = f"SELECT \* FROM {self.table\_name}"

    return pd.read\_sql(query, self.engine)   
The fetch\_data method constructs a SQL query to select all rows from the specified table (self.table\_name). It then uses pandas.read\_sql to execute the query and load the result into a pandas DataFrame.   
The DatabaseTable class provides a simple way to connect to a database and fetch data from a specified table using SQLAlchemy for the connection and pandas for data handling. The fetch\_data method returns the data as a DataFrame, making it easy to manipulate and analyze using pandas' powerful tools.

3. Use\_database\_table.py:  
This code snippet extends the functionality of the DatabaseTable class to create a specific EmployeeTable class for handling employee data. It also demonstrates how to use these classes to fetch and display data from an SQLite database.  
Steps:  
i. Define the Table Name and Connection String:  
table\_name = "employees"

connection\_string = "sqlite:///example.db" # SQLite connection string  
•  table\_name: Specifies the name of the table you want to interact with in the database.

•  connection\_string: The connection string for the SQLite database. This tells SQLAlchemy to connect the databse.  
ii. Create an Instance of the DatabaseTable Class:  
db\_table = DatabaseTable(table\_name, connection\_string)  
An instance of the DatabaseTable class is created with the table name and connection string as parameters. This instance (db\_table) is now set up to interact with the employees table in the example.db database.  
iii. Fetch Data from the Table:  
data = db\_table.fetch\_data()  
The fetch\_data method of the DatabaseTable class is called to retrieve all rows from the employees table. The fetched data is stored in the data variable as a pandas DataFrame.  
iv. Display the Data:  
print(data)  
The contents of the data DataFrame are printed to the console, displaying the data fetched from the employees table.  
vi. Define the EmployeeTable Class:  
class EmployeeTable(DatabaseTable):

def \_\_init\_\_(self, connection\_string):

super().\_\_init\_\_('employees', connection\_string)

self.metadata = {

'table\_name': 'employees',

'description': 'Employee details and payroll information',

'columns': {

'id': 'Employee ID',

'name': 'Employee Name',

'department': 'Department Name',

'salary': 'Salary of the Employee'

}

}  
•  The super() function is used to call the constructor of the parent class (DatabaseTable).

•  super().\_\_init\_\_('employees', connection\_string) initializes the DatabaseTable with the table name 'employees' and the connection\_string.

•  This sets up the connection to the employees table in the database and allows the EmployeeTable class to inherit the functionality of the DatabaseTable class.

self.metadata = { ... }

•  This creates an attribute metadata for the EmployeeTable instance and initializes it with a dictionary containing metadata about the table.

•  The metadata dictionary includes:

* **'table\_name': 'employees'**: The name of the table.
* **'description': 'Employee details and payroll information'**: A description of what the table contains.
* **'columns'**: A dictionary describing the columns of the table, where:
* 'id' maps to 'Employee ID'
* 'name' maps to 'Employee Name'
* 'department' maps to 'Department Name'
* 'salary' maps to 'Salary of the Employee'

4. test\_gpt\_neox.py  
This script demonstrates how to use a pre-trained language model (GPT-2) to interact with and generate responses based on the data fetched from the employees table in the database.  
Steps:  
i. Imports:  
  
import pandas as pd

from transformers import pipeline

from use\_database\_table import EmployeeTable  
•  **pandas**: A powerful data manipulation and analysis library.

•  **pipeline from transformers**: This function from the transformers library by Hugging Face allows you to easily use pre-trained language models.

•  **EmployeeTable from use\_database\_table**: This is a custom module where the EmployeeTable class is defined to handle database operations.  
ii. Function to Fetch Data from the Table:  
def fetch\_data\_from\_table(table\_name, connection\_string):

# Initialize the table object and fetch data

table = EmployeeTable(connection\_string)

return table.fetch\_data()  
•  **fetch\_data\_from\_table**: This function takes the name of a table and a connection string as input.

•  It creates an instance of the EmployeeTable class using the provided connection string.

•  It then calls the fetch\_data method of the EmployeeTable instance to fetch data from the specified table.   
iii. Initialising with LLM:  
pipe = pipeline("text-generation", model="gpt2-medium")  
**pipeline("text-generation", model="gpt2-medium")**: This initializes a text-generation pipeline using the GPT-2 medium model. This model will be used to generate text based on the prompts provided to it.  
iv. Function to Ask a Question to the LLM:  
def ask\_llm(question, context):

    # Convert context to string

    context\_str = context.to\_string(index=False)

    prompt = f"{question}\n\nContext:\n{context\_str}"

    response = pipe(prompt, max\_length=150, num\_return\_sequences=1, truncation=True)

    return response[0]['generated\_text']

•  **ask\_llm**: This function takes a question and a context (data from the table) as input.

•  **context.to\_string(index=False)**: Converts the context DataFrame to a string format, excluding the index.

•  **prompt**: Combines the question and the context string into a single prompt.

•  **pipe(prompt, max\_length=150, num\_return\_sequences=1, truncation=True)**: Uses the GPT-2 pipeline to generate a response based on the prompt. The max\_length parameter sets the maximum length of the generated response, and truncation=True ensures that the prompt is truncated if it exceeds the maximum length.

•  **response[0]['generated\_text']**: Extracts the generated text from the response.   
Main Function

def main():

    connection\_string = 'sqlite:///example.db'

    table\_name = 'employees'

# Fetch data from the employees table   
 context\_df = fetch\_data\_from\_table(table\_name, connection\_string)

   # Ask the LLM a question directly

    question = "Find the highest paid employee with names for each department."

    response = ask\_llm(question, context\_df)

    print(response)

if \_\_name\_\_ == "\_\_main\_\_":

    main()

•  **main**: The main function that orchestrates the entire process.

•  **connection\_string = 'sqlite:///example.db'**: Defines the connection string to the SQLite database.

•  **table\_name = 'employees'**: Specifies the name of the table to fetch data from.

•  **context\_df = fetch\_data\_from\_table(table\_name, connection\_string)**: Fetches data from the employees table and stores it in a DataFrame.

•  **question = "Find the highest paid employee with names for each department."**: Defines the question to ask the LLM.

•  **response = ask\_llm(question, context\_df)**: Calls the ask\_llm function with the question and the context data, and stores the response.

•  **print(response)**: Prints the response generated by the LLM.

This code integrates database operations with a language model to perform complex queries:

1. **Fetches data**: Connects to a database, fetches data from the specified table, and converts it to a DataFrame.
2. **Initializes an LLM**: Uses the GPT-2 medium model to generate text responses.
3. **Queries the LLM**: Converts the DataFrame to a string, creates a prompt with the question and context, and uses the LLM to generate an answer.
4. **Displays the result**: Prints the response generated by the LLM.

This setup allows you to dynamically fetch data from a database and use a pre-trained language model to generate insightful responses based on that data.

5. test\_pandasai.py

Import the necessary libraries and set up the API key for PandasAI.

from use\_database\_table import EmployeeTable

import pandasai  
from pandasai import SmartDataframe  
import os  
os.environ['PANDASAI\_API\_KEY'] = '$2a$10$eSCjrm3l2fnZA.VyYjTHo.JtqjsK0PxgpQX2iL2BdRx.g2ESCEdxe'

def fetch\_data\_from\_table(table\_name, connection\_string):

# Initialize the table object and fetch data

table = EmployeeTable(connection\_string)  
 return table.fetch\_data()  
  
The os module is used to set an environment variable for the PandasAI API key. This key is necessary to authenticate our requests to the language model.  
  
Define a helper function ask\_llm2 that takes a question and a context (in this case, a DataFrame) and returns the response from the language model:  
  
def ask\_llm2(question, context):  
 sdf = SmartDataframe(context)  
 response = sdf.chat(question)  
 return response

def main2():  
 connection\_string = 'sqlite:///example.db'  
 table\_name = 'employees'  
 # Fetch data from the employees table  
 context\_df = fetch\_data\_from\_table(table\_name, connection\_string)  
 # Ask the LLM a question directly  
 question = "Find the highest paid employee with names for each department."  
 response = ask\_llm2(question, context\_df)  
 print(response)

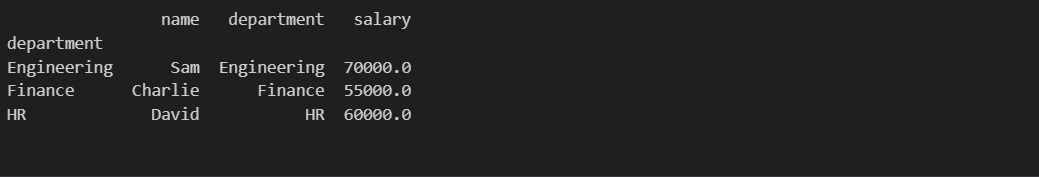
if \_\_name\_\_ == "\_\_main\_\_":  
 main2()

SUMMARY

* **setup\_database.py**: Sets up the SQLite database and populates it with sample data.
* **database\_table.py**: Defines a base class for interacting with the database.
* **use\_database\_table.py**: Extends the base class to include metadata specific to the employees table.
* **test\_gpt\_neox.py**: Fetches data from the database, queries a pre-trained language model (GPT-2), and prints the response based on a specific question.
* test\_pandasai.py: Using PandasAI and SmartDataframe for Data Querying with Language Models

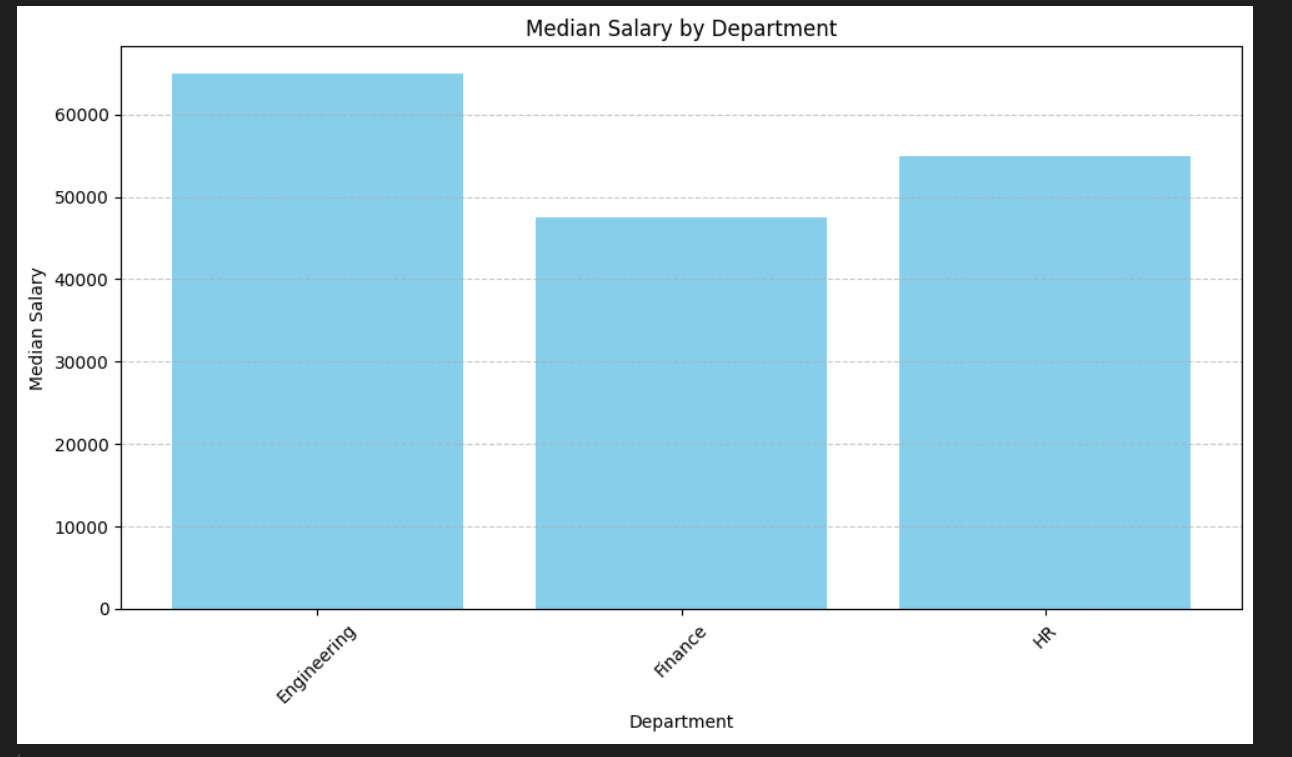
## Results

Analysis using PandasAI SmartdataFrame

Question 1. “Find the highest paid employee with names for each department”  
   
Question 2. “Is the any relationship between department vs salary”



Question 3. “could you give me the graph of median salary vs department”



NOTE: The gpt2-medium model demonstrates limitations in its analytical capabilities, often failing to provide accurate or precise results.

## Conclusion

Integrating Python classes with relational database management systems (RDBMS) enhances the flexibility and depth of data analysis. These classes not only retrieve data but also maintain comprehensive metadata, providing essential business context. Utilizing Pandas AI to query these structured datasets using natural language simplifies data interaction, enabling more intuitive and efficient advanced analytics. This approach empowers analysts to derive insights effectively, leveraging the combined strengths of Python and generative AI.

## Future Work:

Future enhancements could include automated metadata generation, support for more complex queries, and integration with other AI models to further enrich data analysis capabilities. Additionally, expanding this framework to handle real-time data streams and larger datasets can provide even greater value for dynamic business environments.

## Keywords:

Python,SQL, RDMS, SQLAlchemy, Pandas, Generative AI, DataFrame, Data Analysis, Metadata, Business Intelligence.