An AI application that can chat with very large SQL databases.

Apurv Agarwal · Following
18 min read · Nov 4, 2023

In the last article we created a simple application that can chat with an SQL database. Check it out here before proceeding with this article.

Also get entire repository of this code from here.

The application we built in previous article will be unable to perform well on very large databases because of openAI's token limit. We can't send entire list of columns and tables as context to the prompt if database is too large. We will try to overcome that limitation in this article.

Let's start with the code for the simplest application we already had from the previous article. This below code will start a simple streamlit application through which we can connect with an SQL database and start chatting.

```
import streamlit as st
import requests
import os
import pandas as pd
from uuid import uuid4
```

```
import psycopg2

from langchain.prompts import ChatPromptTemplate
from langchain.prompts.chat import SystemMessage, HumanMessagePromptTemplate

from langchain.llms import OpenAI, AzureOpenAI
from langchain.chat_models import ChatOpenAI, AzureChatOpenAI
from langchain.embeddings import OpenAIEmbeddings
from dotenv import load_dotenv




folders_to_create = ['csvs']
# Check and create folders if they don't exist
for folder_name in folders_to_create:
    if not os.path.exists(folder_name):
        os.makedirs(folder_name)
        print(f"Folder '{folder_name}' created.")
    else:
        print(f"Folder '{folder_name}' already exists.")




## load the API key from the environment variable
load_dotenv()
openai_api_key = os.getenv("OPENAI_API_KEY")


llm = OpenAI(openai_api_key=openai_api_key)
chat_llm = ChatOpenAI(openai_api_key=openai_api_key, temperature=0.4)
embeddings = OpenAIEmbeddings(openai_api_key=openai_api_key)





def get_basic_table_details(cursor):
    cursor.execute("""SELECT
            c.table_name,
            c.column_name,
            c.data_type
        FROM
            information_schema.columns c
        WHERE
            c.table_name IN (
                SELECT tablename
                FROM pg_tables
                WHERE schemaname = 'public'
    );""")
    tables_and_columns = cursor.fetchall()
    return tables_and_columns
```

```
def save_db_details(db_uri):

    unique_id = str(uuid4()).replace("-", "_")
    connection = psycopg2.connect(db_uri)
    cursor = connection.cursor()

    tables_and_columns = get_basic_table_details(cursor)

    ## Get all the tables and columns and enter them in a pandas dataframe
    df = pd.DataFrame(tables_and_columns, columns=['table_name', 'column_name',
    filename_t = 'csvs/tables_' + unique_id + '.csv'
    df.to_csv(filename_t, index=False)

    cursor.close()
    connection.close()

    return unique_id







def generate_template_for_sql(query, table_info, db_uri):
    template = ChatPromptTemplate.from_messages(
            [
                SystemMessage(
                    content=(
                        f"You are an assistant that can write SQL Queries."
                        f"Given the text below, write a SQL query that answers t
                        f"DB connection string is {db_uri}"
                        f"Here is a detailed description of the table(s): "
                        f"{table_info}"
                        "Prepend and append the SQL query with three backticks '


                    )
                ),
                HumanMessagePromptTemplate.from_template("{text}"),

            ]
        )

    answer = chat_llm(template.format_messages(text=query))
    return answer.content




def get_the_output_from_llm(query, unique_id, db_uri):
    ## Load the tables csv
    filename_t = 'csvs/tables_' + unique_id + '.csv'
    df = pd.read_csv(filename_t)
    ## For each relevant table create a string that list down all the columns an
    table_info = ''
    for table in df['table_name']:
        table_info += 'Information about table' + table + ':\n'
        table_info += df[df['table_name'] == table].to_string(index=False) + '\n

    return generate_template_for_sql(query, table_info, db_uri)





def execute_the_solution(solution, db_uri):
    connection = psycopg2.connect(db_uri)
    cursor = connection.cursor()
    _,final_query,_ = solution.split("```")
    final_query = final_query.strip('sql')
    cursor.execute(final_query)
    result = cursor.fetchall()
    return str(result)





# Function to establish connection and read metadata for the database
def connect_with_db(uri):
    st.session_state.db_uri = uri
    st.session_state.unique_id = save_db_details(uri)

    return {"message": "Connection established to Database!"}

# Function to call the API with the provided URI
def send_message(message):
    solution = get_the_output_from_llm(message, st.session_state.unique_id, st.s
    result = execute_the_solution(solution, st.session_state.db_uri)
    return {"message": solution + "\n\n" + "Result:\n" + result}



# ## Instructions
st.subheader("Instructions")
st.markdown(
    """
    1. Enter the URI of your RDS Database in the text box below.
    2. Click the **Start Chat** button to start the chat.
    3. Enter your message in the text box below and press **Enter** to send the
    """
)

# Initialize the chat history list
chat_history = []

# Input for the database URI
uri = st.text_input("Enter the RDS Database URI")
```

```
    if st.button("Start Chat"):
        if not uri:
            st.warning("Please enter a valid database URI.")
        else:
            st.info("Connecting to the API and starting the chat...")
            chat_response = connect_with_db(uri)
            if "error" in chat_response:
                st.error("Error: Failed to start the chat. Please check the URI and
            else:
                st.success("Chat started successfully!")

    # Chat with the API (a mock example)
    st.subheader("Chat with the API")

    # Initialize chat history
    if "messages" not in st.session_state:
        st.session_state.messages = []

    # Display chat messages from history on app rerun
    for message in st.session_state.messages:
        with st.chat_message(message["role"]):
            st.markdown(message["content"])

    # React to user input
    if prompt := st.chat_input("What is up?"):
        # Display user message in chat message container
        st.chat_message("user").markdown(prompt)
        # Add user message to chat history
        st.session_state.messages.append({"role": "user", "content": prompt})

        # response = f"Echo: {prompt}"
        response = send_message(prompt)["message"]
        # Display assistant response in chat message container
        with st.chat_message("assistant"):
            st.markdown(response)
        # Add assistant response to chat history
        st.session_state.messages.append({"role": "assistant", "content": response})

    # Run the Streamlit app
    if __name__ == "__main__":
        st.write("This is a simple Streamlit app for starting a chat with an RDS Dat
```

The basic idea to shorten the prompt is to send only those tables and column names in the prompt that are relevant to user's query. For that we can create embeddings of table and column names, retrieve the ones most relevant to user's message on the fly and pass those to the prompt.

For this article we will use ChromaDB as our vector database, but you can use Pinecone, Milvus or any other. So let's install chromadb

```
pip install chromadb
```

First we will create another folder named vectors along with csvs where we will store embeddings of table and column names, we can also store other information about the database, like what are the foreign keys joining different tables, and also some of the values that can go in where clause.

```
def create_vectors(filename, persist_directory):
    loader = CSVLoader(file_path=filename, encoding="utf8")
    data = loader.load()
    vectordb = Chroma.from_documents(data, embedding=embeddings, persist_directo
    vectordb.persist()
```

We will also first check whether user's query need any information about tables or instead user is asking about just the general schema of database.

```
def check_if_users_query_want_general_schema_information_or_sql(query):
    template = ChatPromptTemplate.from_messages(
            [
                SystemMessage(
                    content=(

                        f"In the text given text user is asking a question about
                        f"Figure out whether user wants information about databa
                        f"Answer 'yes' if user wants information about database

                    )
                ),
                HumanMessagePromptTemplate.from_template("{text}"),

            ]
```

```
        )

    answer = chat_llm(template.format_messages(text=query))
    print(answer.content)
    return answer.content
```

This will answer with yes or no depending on what user want. If the answer her is yes, we will create the prompt

```
def prompt_when_user_want_general_db_information(query, db_uri):
    template = ChatPromptTemplate.from_messages(
            [
                SystemMessage(
                    content=(
                        "You are an assistant who writes SQL queries."
                        "Given the text below, write a SQL query that answers th
                        "Prepend and append the SQL query with three backticks '
                        "Write select query whenever possible"
                        f"Connection string to this database is {db_uri}"
                    )
                ),
                HumanMessagePromptTemplate.from_template("{text}"),

            ]
        )

    answer = chat_llm(template.format_messages(text=query))
    print(answer.content)
    return answer.content
```

Next if the answer is no that means user's query very specifically will need names of tables and columns in the table.

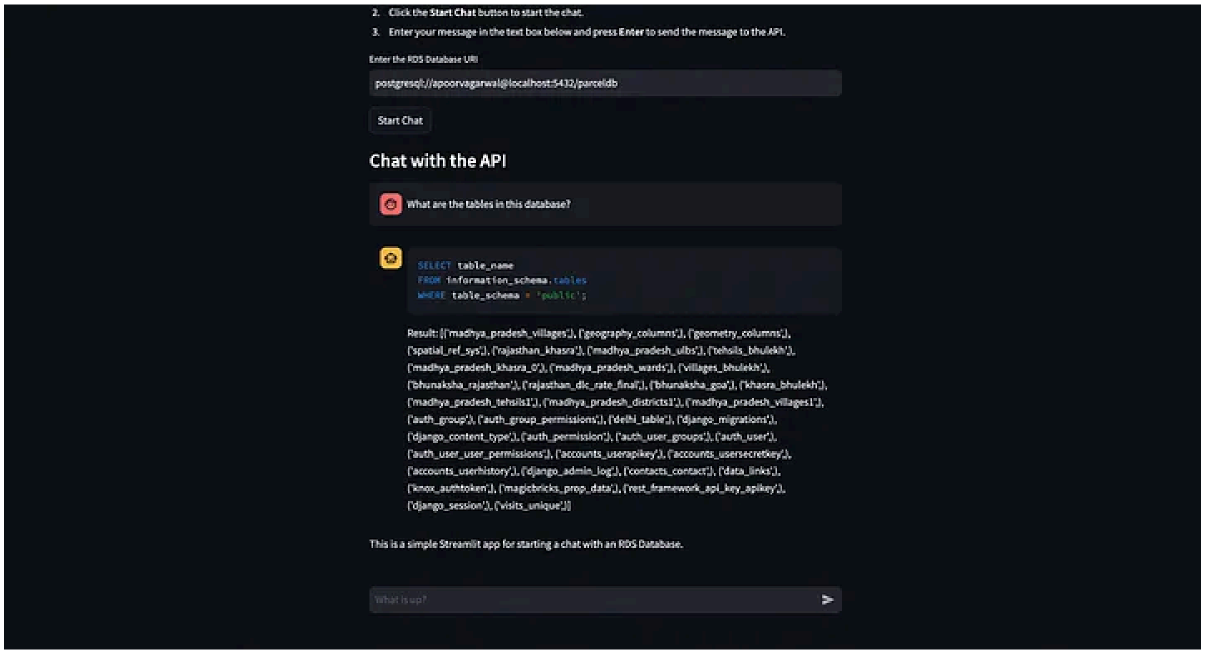
For that we will first retrieve the most relevant tables and columns, and create a string from them to add in our prompt.

Let's check if our vectors are created and everything else is running fine. Here is complete code till now.

```
import streamlit as st
import requests
import os
import pandas as pd
from uuid import uuid4
import psycopg2

from langchain.prompts import ChatPromptTemplate
from langchain.prompts.chat import SystemMessage, HumanMessagePromptTemplate

from langchain.llms import OpenAI, AzureOpenAI
from langchain.chat_models import ChatOpenAI, AzureChatOpenAI
from langchain.embeddings import OpenAIEmbeddings
from dotenv import load_dotenv
from langchain.vectorstores import Chroma
from langchain.document_loaders.csv_loader import CSVLoader




folders_to_create = ['csvs', 'vectors']
# Check and create folders if they don't exist
for folder_name in folders_to_create:
    if not os.path.exists(folder_name):
        os.makedirs(folder_name)
        print(f"Folder '{folder_name}' created.")
    else:
        print(f"Folder '{folder_name}' already exists.")




## load the API key from the environment variable
load_dotenv()
openai_api_key = os.getenv("OPENAI_API_KEY")


llm = OpenAI(openai_api_key=openai_api_key)
chat_llm = ChatOpenAI(openai_api_key=openai_api_key, temperature=0.4)
embeddings = OpenAIEmbeddings(openai_api_key=openai_api_key)





def get_basic_table_details(cursor):
    cursor.execute("""SELECT
            c.table_name,
            c.column_name,
            c.data_type
        FROM
            information_schema.columns c
        WHERE
            c.table_name IN (
                SELECT tablename
                FROM pg_tables
                WHERE schemaname = 'public'
    );""")
    tables_and_columns = cursor.fetchall()
    return tables_and_columns

def create_vectors(filename, persist_directory):
    loader = CSVLoader(file_path=filename, encoding="utf8")
    data = loader.load()
    vectordb = Chroma.from_documents(data, embedding=embeddings, persist_directo
    vectordb.persist()




def save_db_details(db_uri):

    unique_id = str(uuid4()).replace("-", "_")
    connection = psycopg2.connect(db_uri)
    cursor = connection.cursor()

    tables_and_columns = get_basic_table_details(cursor)

    ## Get all the tables and columns and enter them in a pandas dataframe
    df = pd.DataFrame(tables_and_columns, columns=['table_name', 'column_name',
    filename_t = 'csvs/tables_' + unique_id + '.csv'
    df.to_csv(filename_t, index=False)

    create_vectors(filename_t, "./vectors/tables_"+ unique_id)

    cursor.close()
    connection.close()

    return unique_id






def generate_template_for_sql(query, table_info, db_uri):
    template = ChatPromptTemplate.from_messages(
            [
                SystemMessage(
                    content=(
                        f"You are an assistant that can write SQL Queries."
                        f"Given the text below, write a SQL query that answers t
                        f"DB connection string is {db_uri}"
                        f"Here is a detailed description of the table(s): "
                        f"{table_info}"
                        "Prepend and append the SQL query with three backticks '


                    )
                ),
                HumanMessagePromptTemplate.from_template("{text}"),

            ]
        )

    answer = chat_llm(template.format_messages(text=query))
    return answer.content


def check_if_users_query_want_general_schema_information_or_sql(query):
    template = ChatPromptTemplate.from_messages(
            [
                SystemMessage(
                    content=(

                        f"In the text given text user is asking a question about
                        f"Figure out whether user wants information about databa
                        f"Answer 'yes' if user wants information about database 

                    )
                ),
                HumanMessagePromptTemplate.from_template("{text}"),

            ]
        )

    answer = chat_llm(template.format_messages(text=query))
    print(answer.content)
    return answer.content


def prompt_when_user_want_general_db_information(query, db_uri):
    template = ChatPromptTemplate.from_messages(
            [
                SystemMessage(
                    content=(
                        "You are an assistant who writes SQL queries."
                        "Given the text below, write a SQL query that answers th
                        "Prepend and append the SQL query with three backticks '
                        "Write select query whenever possible"
                        f"Connection string to this database is {db_uri}"
                    )
                ),
                HumanMessagePromptTemplate.from_template("{text}"),

            ]
        )

    answer = chat_llm(template.format_messages(text=query))
    print(answer.content)
    return answer.content
```

```
def get_the_output_from_llm(query, unique_id, db_uri):
    ## Load the tables csv
    filename_t = 'csvs/tables_' + unique_id + '.csv'
    df = pd.read_csv(filename_t)

    ## For each relevant table create a string that list down all the columns an
    table_info = ''
    for table in df['table_name']:
        table_info += 'Information about table' + table + ':\n'
        table_info += df[df['table_name'] == table].to_string(index=False) + '\n

    answer_to_question_general_schema = check_if_users_query_want_general_schema
    if answer_to_question_general_schema == "yes":
        return prompt_when_user_want_general_db_information(query, db_uri)

    return generate_template_for_sql(query, table_info, db_uri)




def execute_the_solution(solution, db_uri):
    connection = psycopg2.connect(db_uri)
    cursor = connection.cursor()
    _,final_query,_ = solution.split("```")
    final_query = final_query.strip('sql')
    cursor.execute(final_query)
    result = cursor.fetchall()
    return str(result)




# Function to establish connection and read metadata for the database
def connect_with_db(uri):
    st.session_state.db_uri = uri
    st.session_state.unique_id = save_db_details(uri)

    return {"message": "Connection established to Database!"}

# Function to call the API with the provided URI
def send_message(message):
    solution = get_the_output_from_llm(message, st.session_state.unique_id, st.s
    result = execute_the_solution(solution, st.session_state.db_uri)
    return {"message": solution + "\n\n" + "Result:\n" + result}



# ## Instructions
st.subheader("Instructions")
st.markdown(
```

```
        """
        1. Enter the URI of your RDS Database in the text box below.
        2. Click the **Start Chat** button to start the chat.
        3. Enter your message in the text box below and press **Enter** to send the
        """
    )

    # Initialize the chat history list
    chat_history = []

    # Input for the database URI
    uri = st.text_input("Enter the RDS Database URI")

    if st.button("Start Chat"):
        if not uri:
            st.warning("Please enter a valid database URI.")
        else:
            st.info("Connecting to the API and starting the chat...")
            chat_response = connect_with_db(uri)
            if "error" in chat_response:
                st.error("Error: Failed to start the chat. Please check the URI and
            else:
                st.success("Chat started successfully!")

    # Chat with the API (a mock example)
    st.subheader("Chat with the API")

    # Initialize chat history
    if "messages" not in st.session_state:
        st.session_state.messages = []

    # Display chat messages from history on app rerun
    for message in st.session_state.messages:
        with st.chat_message(message["role"]):
            st.markdown(message["content"])

    # React to user input
    if prompt := st.chat_input("What is up?"):
        # Display user message in chat message container
        st.chat_message("user").markdown(prompt)
        # Add user message to chat history
        st.session_state.messages.append({"role": "user", "content": prompt})

        # response = f"Echo: {prompt}"
        response = send_message(prompt)["message"]
        # Display assistant response in chat message container
        with st.chat_message("assistant"):
            st.markdown(response)
        # Add assistant response to chat history
        st.session_state.messages.append({"role": "assistant", "content": response})

    # Run the Streamlit app
    if __name__ == "__main__":
        st.write("This is a simple Streamlit app for starting a chat with an RDS Dat
```


2. Click the **Start Chat** button to start the chat.
3. Enter your message in the text box below and press **Enter** to send the message to the API.

Enter the RDS Database URI

postgresql://apoorvagarwal@localhost:5432/parceldb

Chat with the API

What are the tables in this database?

```
SELECT table_name
FROM information_schema.tables
WHERE table_schema = 'public';
```

Result: [('madhya_pradesh_villages',), ('geography_columns',), ('geometry_columns',), ('spatial_ref_sys',), ('rajasthan_khasra',), ('madhya_pradesh_ulbs',), ('tehsils_bhulekh',), ('madhya_pradesh_khasra_0',), ('madhya_pradesh_wards',), ('villages_bhulekh',), ('bhunaksha_rajasthan',), ('rajasthan_dlc_rate_final',), ('bhunaksha_goa',), ('khasra_bhulekh',), ('madhya_pradesh_tehsils1',), ('madhya_pradesh_districts1',), ('madhya_pradesh_villages1',), ('auth_group',), ('auth_group_permissions',), ('delhi_table',), ('django_migrations',), ('django_content_type',), ('auth_permission',), ('auth_user_groups',), ('auth_user',), ('auth_user_user_permissions',), ('accounts_userapikey',), ('accounts_usersecretkey',), ('accounts_userhistory',), ('django_admin_log',), ('contacts_contact',), ('data_links',), ('knox_authtoken',), ('magicbricks_prop_data',), ('rest_framework_api_key_apikey',), ('django_session',), ('visits_unique',)]

This is a simple Streamlit app for starting a chat with an RDS Database.

Now in next step we'll finally do vector retrieval of the most relevant tables. For the the relevant tables, we will get all the columns just to give our prompt more context. Finally we create a string from this information to pass in the prompt.

```
vectordb = Chroma(embedding_function=embeddings, persist_directory="./vectors/ta
retriever = vectordb.as_retriever()
docs = retriever.get_relevant_documents(query)
print(docs)

relevant_tables = []
relevant_tables_and_columns = []


for doc in docs:
    table_name, column_name, data_type = doc.page_content.split("\n")
    table_name= table_name.split(":")[1].strip()
    relevant_tables.append(table_name)
    column_name = column_name.split(":")[1].strip()
    data_type = data_type.split(":")[1].strip()
    relevant_tables_and_columns.append((table_name, column_name, data_type))


## Load the tables csv
filename_t = 'csvs/tables_' + unique_id + '.csv'
df = pd.read_csv(filename_t)

## For each relevant table create a string that list down all the columns and th
table_info = ''
for table in relevant_tables:
    table_info += 'Information about table' + table + ':\n'
    table_info += df[df['table_name'] == table].to_string(index=False) + '\n\n\n
```

```
def generate_template_for_sql(query, relevant_tables, table_info):
    tables = ",".join(relevant_tables)
    template = ChatPromptTemplate.from_messages(
            [
                SystemMessage(
                    content=(
                        f"You are an assistant that can write SQL Queries."
                        f"Given the text below, write a SQL query that answers t
                        f"Assume that there is/are SQL table(s) named '{tables}'
                        f"Here is a more detailed description of the table(s): "
                        f"{table_info}"
                        "Prepend and append the SQL query with three backticks '
                    )
                ),
                HumanMessagePromptTemplate.from_template("{text}"),

            ]
        )

    answer = chat_llm(template.format_messages(text=query))
    print(answer.content)
    return answer.content
```

Here is the complete code.

```
import streamlit as st
import requests
import os
import pandas as pd
from uuid import uuid4
import psycopg2

from langchain.prompts import ChatPromptTemplate
from langchain.prompts.chat import SystemMessage, HumanMessagePromptTemplate

from langchain.llms import OpenAI, AzureOpenAI
from langchain.chat_models import ChatOpenAI, AzureChatOpenAI
from langchain.embeddings import OpenAIEmbeddings
from dotenv import load_dotenv
from langchain.vectorstores import Chroma
from langchain.document_loaders.csv_loader import CSVLoader
```

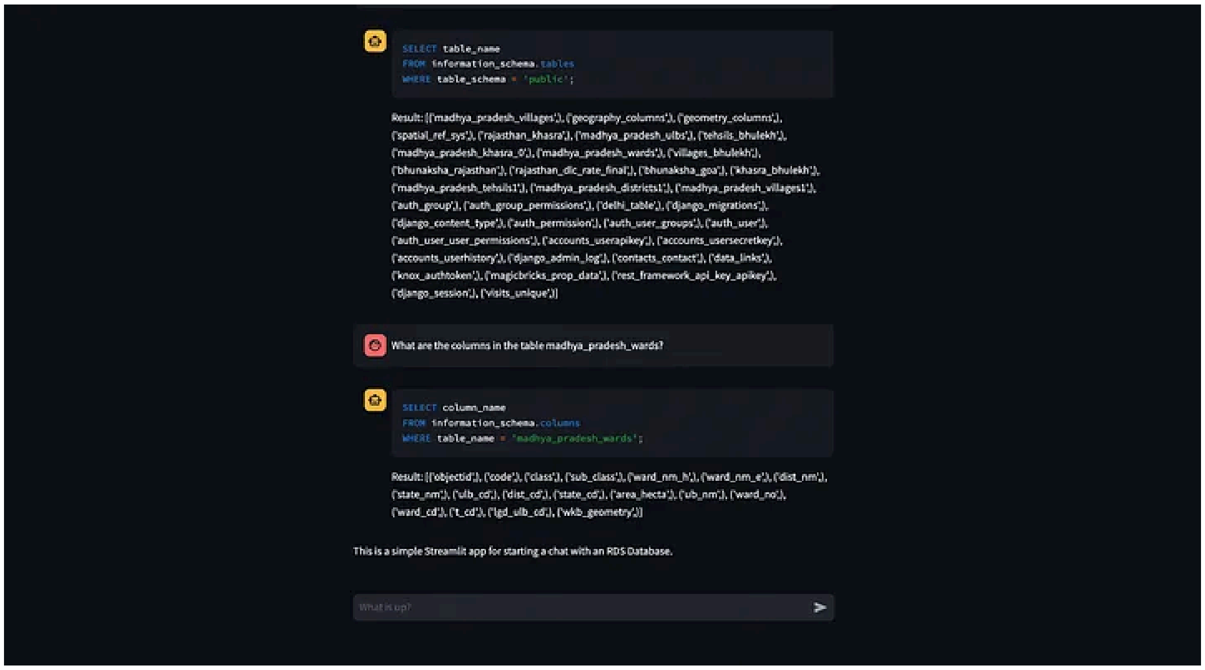
```
folders_to_create = ['csvs', 'vectors']
# Check and create folders if they don't exist
for folder_name in folders_to_create:
    if not os.path.exists(folder_name):
        os.makedirs(folder_name)
        print(f"Folder '{folder_name}' created.")
    else:
        print(f"Folder '{folder_name}' already exists.")




## load the API key from the environment variable
load_dotenv()
openai_api_key = os.getenv("OPENAI_API_KEY")


llm = OpenAI(openai_api_key=openai_api_key)
chat_llm = ChatOpenAI(openai_api_key=openai_api_key, temperature=0.4)
embeddings = OpenAIEmbeddings(openai_api_key=openai_api_key)





def get_basic_table_details(cursor):
    cursor.execute("""SELECT
            c.table_name,
            c.column_name,
            c.data_type
        FROM
            information_schema.columns c
        WHERE
            c.table_name IN (
                SELECT tablename
                FROM pg_tables
                WHERE schemaname = 'public'
    );""")
    tables_and_columns = cursor.fetchall()
    return tables_and_columns

def create_vectors(filename, persist_directory):
    loader = CSVLoader(file_path=filename, encoding="utf8")
    data = loader.load()
    vectordb = Chroma.from_documents(data, embedding=embeddings, persist_directo
    vectordb.persist()




def save_db_details(db_uri):

    unique_id = str(uuid4()).replace("-", "_")
    connection = psycopg2.connect(db_uri)
    cursor = connection.cursor()

    tables_and_columns = get_basic_table_details(cursor)

    ## Get all the tables and columns and enter them in a pandas dataframe
    df = pd.DataFrame(tables_and_columns, columns=['table_name', 'column_name',
    filename_t = 'csvs/tables_' + unique_id + '.csv'
    df.to_csv(filename_t, index=False)

    create_vectors(filename_t, "./vectors/tables_"+ unique_id)

    cursor.close()
    connection.close()

    return unique_id


# def generate_template_for_sql(query, table_info, db_uri):
#     template = ChatPromptTemplate.from_messages(
#             [
#                 SystemMessage(
#                     content=(
#                         f"You are an assistant that can write SQL Queries."
#                         f"Given the text below, write a SQL query that answers
#                         f"DB connection string is {db_uri}"
#                         f"Here is a detailed description of the table(s): "
#                         f"{table_info}"
#                         "Prepend and append the SQL query with three backticks


#                     )
#                 ),
#                 HumanMessagePromptTemplate.from_template("{text}"),

#             ]
#         )

#     answer = chat_llm(template.format_messages(text=query))
#     return answer.content



def generate_template_for_sql(query, relevant_tables, table_info):
    tables = ",".join(relevant_tables)
    template = ChatPromptTemplate.from_messages(
            [
                SystemMessage(
                    content=(
                        f"You are an assistant that can write SQL Queries."
                        f"Given the text below, write a SQL query that answers t
                        f"Assume that there is/are SQL table(s) named '{tables}'
                        f"Here is a more detailed description of the table(s): "
                        f"{table_info}"
                        "Prepend and append the SQL query with three backticks '
                    )
                ),
                HumanMessagePromptTemplate.from_template("{text}"),

            ]
        )

    answer = chat_llm(template.format_messages(text=query))
    print(answer.content)
    return answer.content



def check_if_users_query_want_general_schema_information_or_sql(query):
    template = ChatPromptTemplate.from_messages(
            [
                SystemMessage(
                    content=(

                        f"In the text given text user is asking a question about
                        f"Figure out whether user wants information about databa
                        f"Answer 'yes' if user wants information about database

                    )
                ),
                HumanMessagePromptTemplate.from_template("{text}"),

            ]
        )

    answer = chat_llm(template.format_messages(text=query))
    print(answer.content)
    return answer.content


def prompt_when_user_want_general_db_information(query, db_uri):
    template = ChatPromptTemplate.from_messages(
            [
                SystemMessage(
                    content=(
                        "You are an assistant who writes SQL queries."
                        "Given the text below, write a SQL query that answers th
                        "Prepend and append the SQL query with three backticks '
                        "Write select query whenever possible"
                        f"Connection string to this database is {db_uri}"
                    )
                ),
                HumanMessagePromptTemplate.from_template("{text}"),

            ]
        )

    answer = chat_llm(template.format_messages(text=query))
    print(answer.content)
    return answer.content
```

```
def get_the_output_from_llm(query, unique_id, db_uri):
    ## Load the tables csv
    filename_t = 'csvs/tables_' + unique_id + '.csv'
    df = pd.read_csv(filename_t)

    ## For each relevant table create a string that list down all the columns an
    table_info = ''
    for table in df['table_name']:
        table_info += 'Information about table' + table + ':\n'
        table_info += df[df['table_name'] == table].to_string(index=False) + '\n

    answer_to_question_general_schema = check_if_users_query_want_general_schema
    if answer_to_question_general_schema == "yes":
        return prompt_when_user_want_general_db_information(query, db_uri)
    else:
        vectordb = Chroma(embedding_function=embeddings, persist_directory="./ve
        retriever = vectordb.as_retriever()
        docs = retriever.get_relevant_documents(query)
        print(docs)

        relevant_tables = []
        relevant_tables_and_columns = []

        for doc in docs:
            table_name, column_name, data_type = doc.page_content.split("\n")
            table_name= table_name.split(":")[1].strip()
            relevant_tables.append(table_name)
            column_name = column_name.split(":")[1].strip()
            data_type = data_type.split(":")[1].strip()
            relevant_tables_and_columns.append((table_name, column_name, data_ty

        ## Load the tables csv
        filename_t = 'csvs/tables_' + unique_id + '.csv'
        df = pd.read_csv(filename_t)

        ## For each relevant table create a string that list down all the column
        table_info = ''
        for table in relevant_tables:
            table_info += 'Information about table' + table + ':\n'
            table_info += df[df['table_name'] == table].to_string(index=False) +
        return generate_template_for_sql(query, relevant_tables, table_info)
```

```
def execute_the_solution(solution, db_uri):
    connection = psycopg2.connect(db_uri)
    cursor = connection.cursor()
    _,final_query,_ = solution.split("```")
    final_query = final_query.strip('sql')
    cursor.execute(final_query)
```

```
    result = cursor.fetchall()
    return str(result)




# Function to establish connection and read metadata for the database
def connect_with_db(uri):
    st.session_state.db_uri = uri
    st.session_state.unique_id = save_db_details(uri)

    return {"message": "Connection established to Database!"}

# Function to call the API with the provided URI
def send_message(message):
    solution = get_the_output_from_llm(message, st.session_state.unique_id, st.s
    result = execute_the_solution(solution, st.session_state.db_uri)
    return {"message": solution + "\n\n" + "Result:\n" + result}



# ## Instructions
st.subheader("Instructions")
st.markdown(
    """
    1. Enter the URI of your RDS Database in the text box below.
    2. Click the **Start Chat** button to start the chat.
    3. Enter your message in the text box below and press **Enter** to send the
    """
)

# Initialize the chat history list
chat_history = []

# Input for the database URI
uri = st.text_input("Enter the RDS Database URI")

if st.button("Start Chat"):
    if not uri:
        st.warning("Please enter a valid database URI.")
    else:
        st.info("Connecting to the API and starting the chat...")
        chat_response = connect_with_db(uri)
        if "error" in chat_response:
            st.error("Error: Failed to start the chat. Please check the URI and
        else:
            st.success("Chat started successfully!")

# Chat with the API (a mock example)
st.subheader("Chat with the API")

# Initialize chat history
if "messages" not in st.session_state:
    st.session_state.messages = []
```

```
# Display chat messages from history on app rerun
for message in st.session_state.messages:
    with st.chat_message(message["role"]):
        st.markdown(message["content"])

# React to user input
if prompt := st.chat_input("What is up?"):
    # Display user message in chat message container
    st.chat_message("user").markdown(prompt)
    # Add user message to chat history
    st.session_state.messages.append({"role": "user", "content": prompt})

    # response = f"Echo: {prompt}"
    response = send_message(prompt)["message"]
    # Display assistant response in chat message container
    with st.chat_message("assistant"):
        st.markdown(response)
    # Add assistant response to chat history
    st.session_state.messages.append({"role": "assistant", "content": response})

# Run the Streamlit app
if __name__ == "__main__":
    st.write("This is a simple Streamlit app for starting a chat with an RDS Dat
```



```
SELECT table_name
FROM information_schema.tables
WHERE table_schema = 'public';
```

Result: [('madhya_pradesh_villages',), ('geography_columns',), ('geometry_columns',), ('spatial_ref_sys',), ('rajasthan_khasra',), ('madhya_pradesh_ulbs',), ('tehsils_bhulekh',), ('madhya_pradesh_khasra_0',), ('madhya_pradesh_wards',), ('villages_bhulekh',), ('bhunaksha_rajasthan',), ('rajasthan_dlc_rate_final',), ('bhunaksha_goa',), ('khasra_bhulekh',), ('madhya_pradesh_tehsils1',), ('madhya_pradesh_districts1',), ('madhya_pradesh_villages1',), ('auth_group',), ('auth_group_permissions',), ('delhi_table',), ('django_migrations',), ('django_content_type',), ('auth_permission',), ('auth_user_groups',), ('auth_user',), ('auth_user_user_permissions',), ('accounts_userapikey',), ('accounts_usersecretkey',), ('accounts_userhistory',), ('django_admin_log',), ('contacts_contact',), ('data_links',), ('knox_authtoken',), ('magicbricks_prop_data',), ('rest_framework_api_key_apikey',), ('django_session',), ('visits_unique',)]

What are the columns in the table madhya_pradesh_wards?

```
SELECT column_name
FROM information_schema.columns
WHERE table_name = 'madhya_pradesh_wards';
```

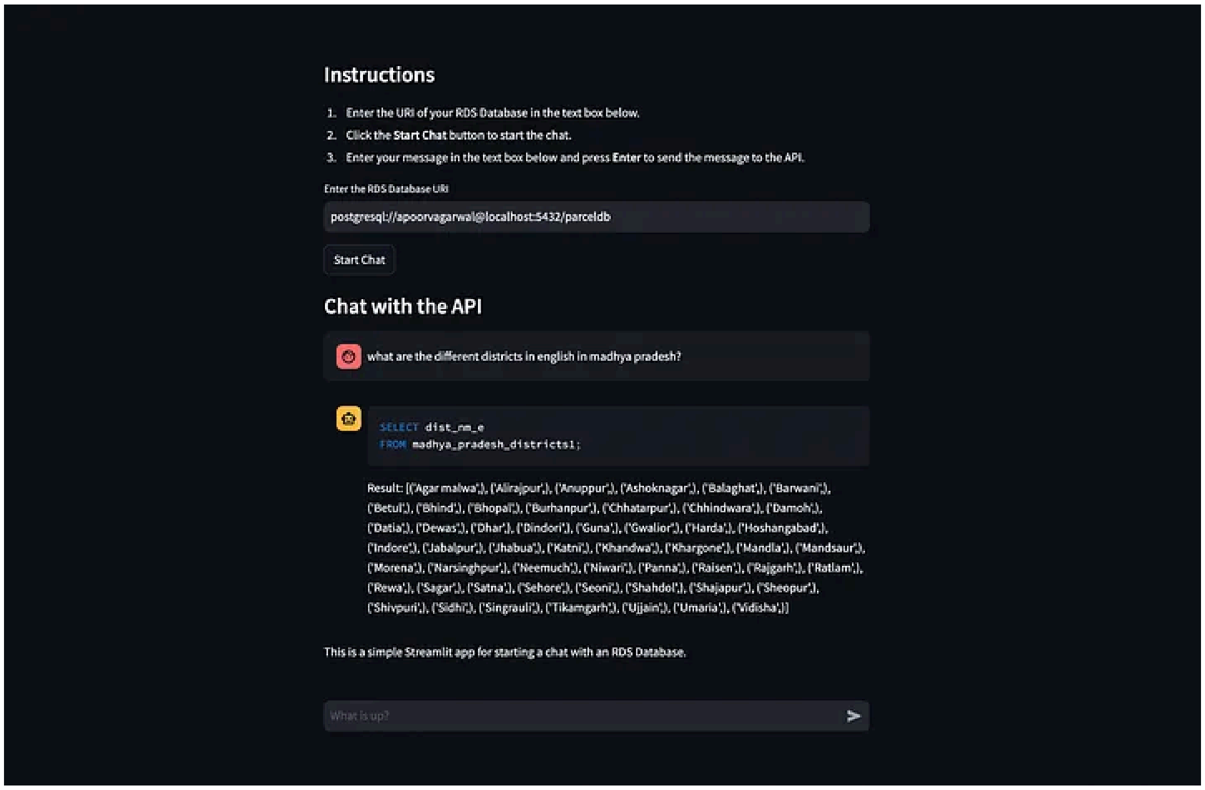
Result: [('objectid',), ('code',), ('class',), ('sub_class',), ('ward_nm_h',), ('ward_nm_e',), ('dist_nm',), ('state_nm',), ('ulb_cd',), ('dist_cd',), ('state_cd',), ('area_hecta',), ('ulb_nm',), ('ward_no',), ('ward_cd',), ('t_cd',), ('lgd_ulb_cd',), ('wkb_geometry',)]

This is a simple Streamlit app for starting a chat with an RDS Database.

One final thing we can do is to give information about foreign keys to the prompt.

```
import streamlit as st
import requests
import os
import pandas as pd
from uuid import uuid4
import psycopg2

from langchain.prompts import ChatPromptTemplate
from langchain.prompts.chat import SystemMessage, HumanMessagePromptTemplate

from langchain.llms import OpenAI, AzureOpenAI
from langchain.chat_models import ChatOpenAI, AzureChatOpenAI
from langchain.embeddings import OpenAIEmbeddings
from dotenv import load_dotenv
from langchain.vectorstores import Chroma
from langchain.document_loaders.csv_loader import CSVLoader




folders_to_create = ['csvs', 'vectors']
# Check and create folders if they don't exist
for folder_name in folders_to_create:
    if not os.path.exists(folder_name):
        os.makedirs(folder_name)
        print(f"Folder '{folder_name}' created.")
    else:
        print(f"Folder '{folder_name}' already exists.")




## load the API key from the environment variable
load_dotenv()
openai_api_key = os.getenv("OPENAI_API_KEY")


llm = OpenAI(openai_api_key=openai_api_key)
chat_llm = ChatOpenAI(openai_api_key=openai_api_key, temperature=0.4)
embeddings = OpenAIEmbeddings(openai_api_key=openai_api_key)





def get_basic_table_details(cursor):
    cursor.execute("""SELECT
            c.table_name,
            c.column_name,
            c.data_type
        FROM
            information_schema.columns c
        WHERE
            c.table_name IN (
                SELECT tablename
                FROM pg_tables
                WHERE schemaname = 'public'
    );""")
    tables_and_columns = cursor.fetchall()
    return tables_and_columns



def get_foreign_key_info(cursor):
    query_for_foreign_keys = """SELECT
    conrelid::regclass AS table_name,
    conname AS foreign_key,
    pg_get_constraintdef(oid) AS constraint_definition,
    confrelid::regclass AS referred_table,
    array_agg(a2.attname) AS referred_columns
    FROM
        pg_constraint
    JOIN
        pg_attribute AS a1 ON conrelid = a1.attrelid AND a1.attnum = ANY(conkey)
    JOIN
        pg_attribute AS a2 ON confrelid = a2.attrelid AND a2.attnum = ANY(confke
    WHERE
        contype = 'f'
        AND connamespace = 'public'::regnamespace
    GROUP BY
        conrelid, conname, oid, confrelid
    ORDER BY
        conrelid::regclass::text, contype DESC;
    """

    cursor.execute(query_for_foreign_keys)
    foreign_keys = cursor.fetchall()

    return foreign_keys




def create_vectors(filename, persist_directory):
    loader = CSVLoader(file_path=filename, encoding="utf8")
    data = loader.load()
    vectordb = Chroma.from_documents(data, embedding=embeddings, persist_directo
    vectordb.persist()





def save_db_details(db_uri):

    unique_id = str(uuid4()).replace("-", "_")
    connection = psycopg2.connect(db_uri)
    cursor = connection.cursor()

    tables_and_columns = get_basic_table_details(cursor)
    ## Get all the tables and columns and enter them in a pandas dataframe
    df = pd.DataFrame(tables_and_columns, columns=['table_name', 'column_name',
    filename_t = 'csvs/tables_' + unique_id + '.csv'
    df.to_csv(filename_t, index=False)

    create_vectors(filename_t, "./vectors/tables_"+ unique_id)

    ## Get all the foreign keys and enter them in a pandas dataframe
    foreign_keys = get_foreign_key_info(cursor)
    df = pd.DataFrame(foreign_keys, columns=['table_name', 'foreign_key', 'forei
    filename_fk = 'csvs/foreign_keys_' + unique_id + '.csv'
    df.to_csv(filename_fk, index=False)

    cursor.close()
    connection.close()

    return unique_id


# def generate_template_for_sql(query, table_info, db_uri):
#     template = ChatPromptTemplate.from_messages(
#             [
#                 SystemMessage(
#                     content=(
#                         f"You are an assistant that can write SQL Queries."
#                         f"Given the text below, write a SQL query that answers
#                         f"DB connection string is {db_uri}"
#                         f"Here is a detailed description of the table(s): "
#                         f"{table_info}"
#                         "Prepend and append the SQL query with three backticks


#                     )
#                 ),
#                 HumanMessagePromptTemplate.from_template("{text}"),

#             ]
#         )

#     answer = chat_llm(template.format_messages(text=query))
#     return answer.content



def generate_template_for_sql(query, relevant_tables, table_info, foreign_key_in
    tables = ",".join(relevant_tables)
    template = ChatPromptTemplate.from_messages(
            [
                SystemMessage(
                    content=(
                        f"You are an assistant that can write SQL Queries."
                        f"Given the text below, write a SQL query that answers t
                        f"Assume that there is/are SQL table(s) named '{tables}'
                        f"Here is a more detailed description of the table(s): "
                        f"{table_info}"
                        "Here is some information about some relevant foreign ke
                        f"{foreign_key_info}"
                        "Prepend and append the SQL query with three backticks '
                    )
                ),
                HumanMessagePromptTemplate.from_template("{text}"),

            ]
        )

    answer = chat_llm(template.format_messages(text=query))
    print(answer.content)
    return answer.content



def check_if_users_query_want_general_schema_information_or_sql(query):
    template = ChatPromptTemplate.from_messages(
            [
                SystemMessage(
                    content=(

                        f"In the text given text user is asking a question about
                        f"Figure out whether user wants information about databa
                        f"Answer 'yes' if user wants information about database

                    )
                ),
                HumanMessagePromptTemplate.from_template("{text}"),

            ]
        )

    answer = chat_llm(template.format_messages(text=query))
    print(answer.content)
    return answer.content


def prompt_when_user_want_general_db_information(query, db_uri):
    template = ChatPromptTemplate.from_messages(
            [
                SystemMessage(
                    content=(
                        "You are an assistant who writes SQL queries."
                        "Given the text below, write a SQL query that answers th
                        "Prepend and append the SQL query with three backticks '
                        "Write select query whenever possible"
                        f"Connection string to this database is {db_uri}"
                    )
                ),
                HumanMessagePromptTemplate.from_template("{text}"),

            ]
        )

    answer = chat_llm(template.format_messages(text=query))
    print(answer.content)
    return answer.content







def get_the_output_from_llm(query, unique_id, db_uri):
    ## Load the tables csv
    filename_t = 'csvs/tables_' + unique_id + '.csv'
    df = pd.read_csv(filename_t)

    ## For each relevant table create a string that list down all the columns an
    table_info = ''
    for table in df['table_name']:
        table_info += 'Information about table' + table + ':\n'
        table_info += df[df['table_name'] == table].to_string(index=False) + '\n

    answer_to_question_general_schema = check_if_users_query_want_general_schema
    if answer_to_question_general_schema == "yes":
        return prompt_when_user_want_general_db_information(query, db_uri)
    else:
        vectordb = Chroma(embedding_function=embeddings, persist_directory="./ve
        retriever = vectordb.as_retriever()
        docs = retriever.get_relevant_documents(query)
        print(docs)

        relevant_tables = []
        relevant_tables_and_columns = []

        for doc in docs:
            table_name, column_name, data_type = doc.page_content.split("\n")
            table_name= table_name.split(":")[1].strip()
            relevant_tables.append(table_name)
            column_name = column_name.split(":")[1].strip()
            data_type = data_type.split(":")[1].strip()
            relevant_tables_and_columns.append((table_name, column_name, data_ty

        ## Load the tables csv
        filename_t = 'csvs/tables_' + unique_id + '.csv'
        df = pd.read_csv(filename_t)

        ## For each relevant table create a string that list down all the column
        table_info = ''
        for table in relevant_tables:
            table_info += 'Information about table' + table + ':\n'
            table_info += df[df['table_name'] == table].to_string(index=False) +



        ## Load the foreign keys csv
        filename_fk = 'csvs/foreign_keys_' + unique_id + '.csv'
        df_fk = pd.read_csv(filename_fk)
        ## If table from relevant_tables above lies in refered_table or table_na
        foreign_key_info = ''
        for i, series in df_fk.iterrows():
            if series['table_name'] in relevant_tables:
                text = table + ' has a foreign key ' + series['foreign_key'] + '
                foreign_key_info += text + '\n\n'
            if series['referred_table'] in relevant_tables:
                text = table + ' is referred to by table ' + series['table_name'
                foreign_key_info += text + '\n\n'


        return generate_template_for_sql(query, relevant_tables, table_info, for





def execute_the_solution(solution, db_uri):
    connection = psycopg2.connect(db_uri)
    cursor = connection.cursor()
    _,final_query,_ = solution.split("```")
    final_query = final_query.strip('sql')
    cursor.execute(final_query)
    result = cursor.fetchall()
    return str(result)





# Function to establish connection and read metadata for the database
def connect_with_db(uri):
    st.session_state.db_uri = uri
    st.session_state.unique_id = save_db_details(uri)

    return {"message": "Connection established to Database!"}

# Function to call the API with the provided URI
def send_message(message):
    solution = get_the_output_from_llm(message, st.session_state.unique_id, st.s
    result = execute_the_solution(solution, st.session_state.db_uri)
    return {"message": solution + "\n\n" + "Result:\n" + result}



# ## Instructions
st.subheader("Instructions")
st.markdown(
    """
    1. Enter the URI of your RDS Database in the text box below.
    2. Click the **Start Chat** button to start the chat.
    3. Enter your message in the text box below and press **Enter** to send the
    """
)

# Initialize the chat history list
chat_history = []

# Input for the database URI
```

```
uri = st.text_input("Enter the RDS Database URI")

if st.button("Start Chat"):
    if not uri:
        st.warning("Please enter a valid database URI.")
    else:
        st.info("Connecting to the API and starting the chat...")
        chat_response = connect_with_db(uri)
        if "error" in chat_response:
            st.error("Error: Failed to start the chat. Please check the URI and
        else:
            st.success("Chat started successfully!")

# Chat with the API (a mock example)
st.subheader("Chat with the API")

# Initialize chat history
if "messages" not in st.session_state:
    st.session_state.messages = []

# Display chat messages from history on app rerun
for message in st.session_state.messages:
    with st.chat_message(message["role"]):
        st.markdown(message["content"])

# React to user input
if prompt := st.chat_input("What is up?"):
    # Display user message in chat message container
    st.chat_message("user").markdown(prompt)
    # Add user message to chat history
    st.session_state.messages.append({"role": "user", "content": prompt})

    # response = f"Echo: {prompt}"
    response = send_message(prompt)["message"]
    # Display assistant response in chat message container
    with st.chat_message("assistant"):
        st.markdown(response)
    # Add assistant response to chat history
    st.session_state.messages.append({"role": "assistant", "content": response})

# Run the Streamlit app
if __name__ == "__main__":
    st.write("This is a simple Streamlit app for starting a chat with an RDS Dat
```



Instructions

1. Enter the URI of your RDS Database in the text box below.
2. Click the **Start Chat** button to start the chat.
3. Enter your message in the text box below and press **Enter** to send the message to the API.

Enter the RDS Database URI

postgresql://apoorvagarwal@localhost:5432/parceldb

Chat with the API

what are the different districts in english in madhya pradesh?

```sql
SELECT dist_nm_e
FROM madhya_pradesh_districts1;
```

```
Result: [('Agar malwa',), ('Alirajpur',), ('Anuppur',), ('Ashoknagar',), ('Balaghat',), ('Barwani',),
('Betul',), ('Bhind',), ('Bhopal',), ('Burhanpur',), ('Chhatarpur',), ('Chhindwara',), ('Damoh',),
('Datia',), ('Dewas',), ('Dhar',), ('Dindori',), ('Guna',), ('Gwalior',), ('Harda',), ('Hoshangabad',),
('Indore',), ('Jabalpur',), ('Jhabua',), ('Katni',), ('Khandwa',), ('Khargone',), ('Mandla',), ('Mandsaur',),
('Morena',), ('Narsinghpur',), ('Neemuch',), ('Niwari',), ('Panna',), ('Raisen',), ('Rajgarh',), ('Ratlam',),
('Rewa',), ('Sagar',), ('Satna',), ('Sehore',), ('Seoni',), ('Shahdol',), ('Shajapur',), ('Sheopur',),
('Shivpuri',), ('Sidhi',), ('Singrauli',), ('Tikamgarh',), ('Ujjain',), ('Umaria',), ('Vidisha',)]
```

This is a simple Streamlit app for starting a chat with an RDS Database.

In similar ways we can keep enhancing this application by adding fallbacks. In each fallback we can keep adding additional information.

AI Langchian Sql Databases Llm

Written by Apurv Agarwal

Following

233 Followers

Fullstack Developer

More from Apurv Agarwal

Apurv Agarwal

What the future of solar autonomous vehicles might look…

If 2020 has taught humanity anything, it is that you can't ignore science. If science is…

2 min read · Jan 20, 2021

3

Apurv Agarwal

How to build AI agents to automate web browsing with human level…

One step towards building AGI is creating agents that can browse the internet, talk to…

3 min read · Feb 12, 2024

140 1

Apurv Agarwal

Aryans in Ancient India

According to Hindu beliefs, during the period of Tretayug, which was possibly the period…

5 min read · Nov 29, 2021

14 1

Apurv Agarwal

Mirage that leads to depression

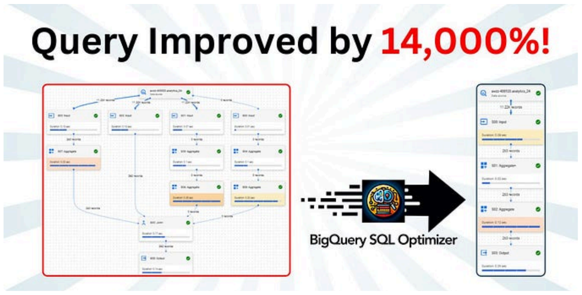
Why are we in depression? What is it about the supposed wonders of modern life that…

1 min read · Nov 2, 2023

3

See all from Apurv Agarwal

Recommended from Medium



ali izadi

AI Tool Improved My SQL Query by 14,000%! BigQuery SQL Optimizer

Maximize SQL query efficiency with the BigQuery SQL Optimizer. deal for both Flat-...

9 min read · Jan 26, 2024

179 3

Uzman Ali

This Guy Makes $1M+ per Year With 0 Employees

Ivan Kutskir—the world's most underrated solopreneur

· 6 min read · May 11, 2024

2.4K 39

Lists

Generative AI Recommended Reading
52 stories · 1120 saves

What is ChatGPT?
9 stories · 371 saves

The New Chatbots: ChatGPT, Bard, and Beyond
12 stories · 398 saves

Natural Language Processing
1499 stories · 1027 saves

Fateh Ali Aamir

A Multi-Layer RAG Chatbot with LangChain's SQL Agent and MySQL

Chatbots are all the craze these days and RAG is a popular mechanism that is being thrown...

5 min read · May 7, 2024

17 1

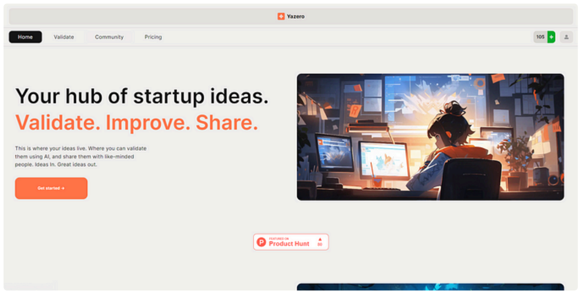
Andreas Stöckl in DataDrivenInvestor

New Chunking Method for RAG-Systems

Enhanced Document Splitting

7 min read · Jun 2, 2024

208 6



Your hub of startup ideas.
Validate. Improve. Share.

Artem Shelamanov in Python in Plain English

How I Built My First AI Startup (With No Experience)

My detailed journey with advices on building startups.

9 min read · Apr 30, 2024

2K 47

Terence Luca... in Government Digital Services, Si...

From Conventional RAG to Graph RAG

When Large Language Models Meet Knowledge Graphs

13 min read · Mar 16, 2024

591 9

See more recommendations