8
Using LLMs with Structured Data

In this chapter, we are going to cover yet another great capability of **large language models** (**LLMs**): the ability to handle structured, tabular data. We will see how, thanks to plugins and an agentic approach, we can use LLMs as a natural language interface between us and our structured data, reducing the gap between the business user and the structured information.

During this chapter, we will cover the following topics:

- Introduction to the main structured data systems
- Using tools and plugins to connect LLMs to tabular data
- Building a database copilot with LangChain

By the end of this chapter, you will be able to build your own natural language interface for your data estate and be able to combine unstructured with structured sources.

Technical requirements

To complete the tasks in this chapter, you will need the following:

- A Hugging Face account and user access token.
- An OpenAI account and user access token.
- Python 3.7.1 or later version.
- Python packages: Make sure to have the following Python packages installed: `langchain`, `python-dotenv`, `huggingface_hub`, `streamlit`, and `sqlite3`. Those can be easily installed via `pip install` in your terminal.

You can find all the code and examples in the book's GitHub repository at https://github.com/PacktPublishing/Building-LLM-Powered-Applications.

What is structured data?

In previous chapters, we focused on how LLMs can handle textual data. In fact, those models are, as the name suggests, "language" models, meaning that they have been trained and are able to handle unstructured text data.

Nevertheless, unstructured data only refers to a portion of the overall data realm that applications can handle. Generally, data can be categorized into three types, which are as follows:

- **Unstructured data**: This refers to data that doesn't have a specific or predefined format. It lacks a consistent structure, making it challenging to organize and analyze using traditional databases. Examples of unstructured data include:
  - Text documents: Emails, social media posts, articles, and reports.
  - Multimedia: Images, videos, audio recordings.

    - Natural language text: Chat logs, transcriptions of spoken conversations.
    - Binary data: Files without a specific data format, such as proprietary file formats.

Note

When it comes to storing unstructured data, NoSQL databases play a crucial role, due to their flexible schema-less design, which allows them to handle various data types like text, images, and videos efficiently. The term "NoSQL" originally stood for "non-SQL" or "not only SQL" to emphasize that these databases don't rely solely on the traditional **Structured Query Language** (**SQL**) to manage and query data. NoSQL databases emerged as a response to the limitations of relational databases, particularly their rigid schema requirements and difficulties in scaling horizontally.

An example of a NoSQL database is MongoDB, a document-oriented NoSQL database, which stores data in JSON-like documents, making it highly effective for managing diverse unstructured content; similarly, Cassandra, with its wide-column store model, excels at handling large volumes of data across many commodity servers, providing high availability without compromising performance. This flexibility enables NoSQL databases to adapt to the volume, variety, and velocity of unstructured data, accommodating rapid changes and scaling easily. Traditional relational databases, with their rigid schema requirements, struggle to manage such diversity and volume efficiently.

- **Structured data**: This type of data is organized and formatted with a clear structure, typically into rows and columns. It follows a fixed schema, making it easy to store, retrieve, and analyze using relational databases. Examples of structured data include:
    - Relational databases: Data stored in tables with predefined columns and data types.
    - Spreadsheets: Data organized in rows and columns in software like Microsoft Excel.
    - Sensor data: Recorded measurements like temperature, pressure, and time in a structured format.
    - Financial data: Transaction records, balance sheets, and income statements.
- **Semi-structured data**: This falls between the two categories. While it doesn't adhere to a rigid structure like structured data, it has some level of organization and may contain tags or other markers that provide context. Examples of semi-structured data include:
    - **eXtensible Markup Language** (**XML**) files: They use tags to structure data, but the specific tags and their arrangement can vary.
    - **JavaScript Object Notation** (**JSON**): This is used for data interchange and allows for nested structures and key-value pairs.
    - NoSQL databases: Storing data in a format that doesn't require a fixed schema, allowing for flexibility.

In summary, unstructured data lacks a defined format, structured data follows a strict format, and semi-structured data has some level of structure but is more flexible than structured data. The distinction between these types of data is important as it impacts how they are stored, processed, and analyzed in various applications.

However, regardless of its nature, querying structured data involves using a query language or methods specific to that database technology. For example, for SQL databases, SQL is used to interact with relational databases. Henceforth, to extract data from tables, you need to know this specific language.

But what if we want to ask questions in natural language to our structured data? What if our application could provide us not only with a sterile numeric answer but rather with a conversational answer, which also gives us context about the number? This is exactly what we will try to achieve in the next sections with our LLM-powered applications. More specifically, we are going build something that we've already defined in *Chapter 2*: a **copilot**. Since we are going to mount our copilot to a relational database, we will name our application **DBCopilot**. First, let's look at what relational databases are.

Getting started with relational databases

The concept of relational databases was first proposed by E.F. Codd, an IBM researcher, in 1970. He defined the rules and principles of the relational model, which aimed to provide a simple and consistent way of accessing and manipulating data. He also introduced SQL, which became the standard language for querying and manipulating relational databases. Relational databases have become widely used in various domains and applications, such as e-commerce, inventory management, payroll, **customer relationship management** (**CRM**), and **business intelligence** (**BI**).

In this section, we are going to cover the main aspects of a relational database. Then, we will start working with the sample database we will use in our DBCopilot, the Chinook database. We will inspect this database and explore how to connect to remote tables using Python.

Introduction to relational databases

A relational database is a type of database that stores and organizes data in structured tables with rows and columns. Each row represents a record, and each column represents a field or attribute. The relationships between tables are established through keys, primarily the primary key and foreign key. This allows for efficient querying and manipulation of data using SQL. These databases are commonly used for various applications like websites and business management systems, due to their ability to manage structured data effectively.

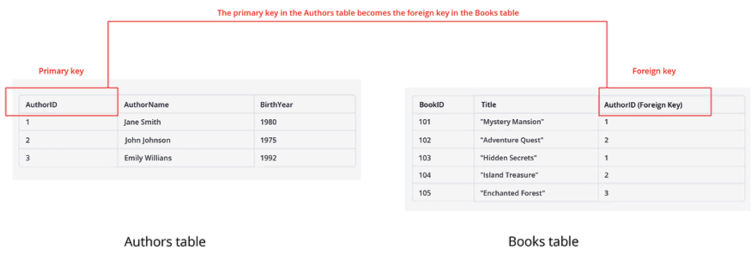
To have a better understanding of relational databases, let's consider an example of a database of a library. We'll have two tables: one for books and another for authors. The relationship between them will be established using primary and foreign keys.

Definition

A primary key is like the unique fingerprint of each record in a table. It's a special column that holds a value that's distinct for each row in that table. Think of it as the "identity" of a record. Having a primary key is important because it guarantees that no two records in the same table will share the same key. This uniqueness makes it easy to locate, modify, and manage individual records in the table.

A foreign key is a bridge between two tables. It's a column in one table that references the primary key column in another table. This reference creates a link between the data in the two tables, establishing a relationship. The purpose of the foreign key is to maintain data consistency and integrity across related tables. It ensures that if a change is made in the primary key table, the related data in the other table remains accurate. By using foreign keys, you can retrieve information from multiple tables that are connected, enabling you to understand how different pieces of data are related to each other.

Let's take a closer look at our example, as shown in the following image:


The primary key in the Authors table becomes the foreign key in the Books table
Primary key
Foreign key
Authors table
Books table

AuthorID	AuthorName	BirthYear
1	Jane Smith	1980
2	John Johnson	1975
3	Emily Willians	1992

BookID	Title	AuthorID (Foreign Key)
101	"Mystery Mansion"	1
102	"Adventure Quest"	2
103	"Hidden Secrets"	1
104	"Island Treasure"	2
105	"Enchanted Forest"	3

Figure 8.1: An example of the relationship between two tables in a database

🔍**Quick tip**: Need to see a high-resolution version of this image? Open this book in the next-gen Packt Reader or view it in the PDF/ePub copy.

🔒 **The next-gen Packt Reader** and a **free PDF/ePub copy** of this book are included with your purchase. Unlock them by scanning the QR code below or visiting https://www.packtpub.com/unlock/9781835462317.

In this example, the Authors table contains information about authors, including their ID, name, and birth year. The Books table includes details about books, including the book's ID, title, and a foreign key called AuthorID, which references the corresponding author in the Authors table (with AuthorID as the primary key). This way, you can use SQL queries to retrieve information like finding all books written by a specific author or the birth year of an author based on the book they wrote. The relational structure allows for efficient management and retrieval of data in a structured manner.

Some of the main database systems in the market include:

- **SQL databases**: These are **relational database management systems (RDBMS)** that use SQL for data manipulation and querying. Examples include MySQL, PostgreSQL, and Microsoft SQL Server.
- **Oracle Database**: A widely-used RDBMS that offers advanced features and scalability for large-scale applications.
- **SQLite:** A self-contained, serverless, and zero-configuration SQL database engine commonly used in embedded systems and mobile applications.
- **IBM Db2**: A family of data management products, including relational database servers, developed by IBM.
- **Amazon Web Services (AWS) RDS**: A managed relational database service offered by Amazon, providing options for various databases like MySQL, PostgreSQL, SQL Server, and more.
- **Google Cloud SQL**: A managed database service by Google Cloud Platform, supporting MySQL, PostgreSQL, and SQL Server.
- **Redis**: An open-source, in-memory data structure store that can be used as a database, cache, and message broker.

In this chapter, we are going to use SQLite database, which also offers a seamless integration with Python. But before we do that, let's understand the database we'll be using.

Overview of the Chinook database

The Chinook database is a sample database that can be used for learning and practicing SQL. It is based on a fictional digital media store and contains data about artists, albums, tracks, customers, invoices, and more. The Chinook database is available for various database management systems, such as SQL Server, Oracle, MySQL, PostgreSQL, SQLite, and DB2.

Here are some features of this database:

- It uses real data from an iTunes library, which makes it more realistic and interesting.
- It has a clear and simple data model, which makes it easy to understand and query.
- It covers more features of SQL, such as subqueries, joins, views, and triggers.
- It is compatible with multiple database servers, which makes it more versatile and portable.

You can find the configuration instructions at `https://database.guide/2-sample-databases-sqlite/`.

You can see an illustration of the relationship among the database's tables here:

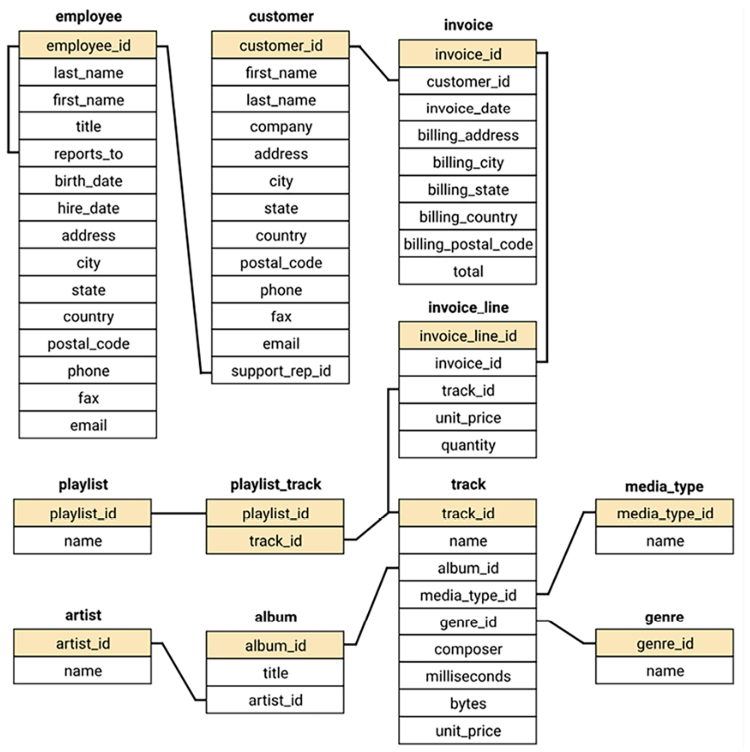


Figure 8.2: Diagram of Chinook Database (source: https://github.com/arjunchndr/Analyzing-Chinook-Database-using-SQL-and-Python)

As you can see, there are 11 tables, all related to each other with primary and foreign keys. In the upcoming paragraph, we will see how LLMs will be able to navigate among those tables, capturing their relationships and gathering relevant information. But before jumping to LLMs, let's first inspect the Chinook database a bit more by setting up the connection with Python.

How to work with relational databases in Python

To work with relational databases in Python, you need to use a library that can connect to the database and execute SQL queries. Some of these libraries are as follows:

- `SQLAlchemy`: This is an open-source SQL toolkit and **object-relational mapper** (**ORM**) for Python. It allows you to create, read, update, and delete data from relational databases using Python objects and methods. It supports many database engines, such as SQLite, MySQL, PostgreSQL, and Oracle.
- `Psycopg`: This is a popular database connector for PostgreSQL. It enables you to execute SQL queries and access PostgreSQL features from Python. It is fast, reliable, and thread-safe.
- `MySQLdb`: This is a database connector for MySQL. It allows you to interact with MySQL databases from Python using the DB-API 2.0 specification. It is one of the oldest and most widely used Python libraries for MySQL, but its development is mostly frozen.
- `cx_Oracle`: This is a database connector for Oracle Database. It enables you to connect to Oracle databases and use SQL and PL/SQL features from Python. It supports advanced features such as object types, **Large Objects** (**LOBs**), and arrays.
- `sqlite3`: This is a database connector for SQLite3, a widely used, lightweight, serverless, self-contained, and open-source relational

database management system. You can use sqlite3 to create, query, update, and delete data from `SQLite` databases in your Python programs

Since we are going to work with SQLite, we will use the `sqlite3` module, which you will need to install via `pip install sqlite3`. Some of the features of sqlite3 are as follows:

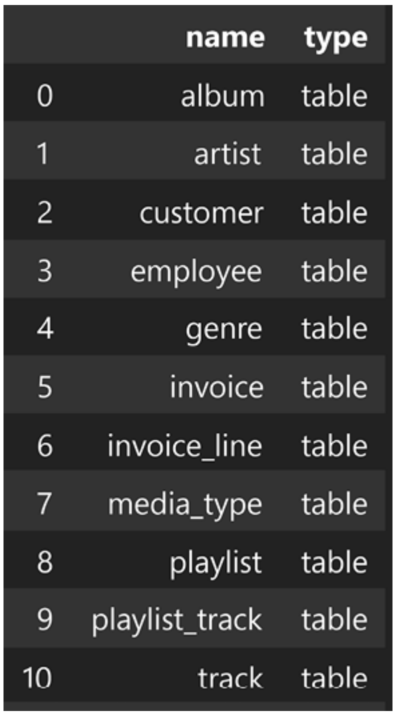
- It follows the DB-API 2.0 specification, which defines a standard interface for Python database access modules.
- It supports transactions, which allow you to execute multiple SQL statements as a single unit of work and roll back in case of errors.
- It allows you to use Python objects as parameters and results for SQL queries, using various adapters and converters.
- It supports user-defined functions, aggregates, collations, and authorizers, which enable you to extend the functionality of SQLite with Python code.
- It has a built-in row factory, which returns query results as named tuples or dictionaries instead of plain tuples.

Let's see an example of this connection using our Chinook database:

1. The database can be downloaded locally from **https://www.sqlitetutorial.net/wp-content/uploads/2018/03/chinook.zip**. You will only need to unzip the `chinook.db` file and it will be ready to be consumed. In the following code, we are initializing a connection (`conn`) to our `chinook.db`, which will be used to interact with the database. Then, we will save our tables in a pandas object with the `read_sql` module, which allows you to run SQL queries against your database:

   ```
   import sqlite3
   import pandas as pd
   ## creating a connection
   database = 'chinook.db'
   conn = sqlite3.connect(database)
   ## importing tables
   tables = pd.read_sql("""SELECT name, type
                           FROM sqlite_master
                            WHERE type IN ("table", "view");""", conn)
   ```

Here is the output that we can see:



	name	type
0	album	table
1	artist	table
2	customer	table
3	employee	table
4	genre	table
5	invoice	table
6	invoice_line	table
7	media_type	table
8	playlist	table
9	playlist_track	table
10	track	table

Figure 8.3: A list of tables within the Chinook database

Note

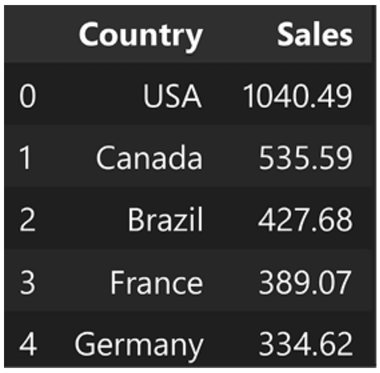
Column names might be slightly different as the online database is updated over time. To get up-to-date columns' naming conventions, you can run the following command:

```
pd.read_sql("PRAGMA table_info(customers);", conn)
print(customer_columns)
```

2. We can also inspect the single table to gather some relevant data. For example, let's say we want to see the top five countries per album sales:

```
pd.read_sql("""
SELECT c.country AS Country, SUM(i.total) AS Sales
FROM customer c
JOIN invoice i ON c.customer_id = i.customer_id
GROUP BY Country
ORDER BY Sales DESC
LIMIT 5;
""", conn)
```

Here is the corresponding output:



	Country	Sales
0	USA	1040.49
1	Canada	535.59
2	Brazil	427.68
3	France	389.07
4	Germany	334.62

Figure 8.4: Top 5 countries with highest sales

3. Finally, we can also use the `matplotlib` Python library to create useful diagrams about the database's statistics. In the following Python snippet, we are going to run an SQL query to extract the number of tracks grouped by genre, and then plot the result using `matplotlib` as follows:

```python
import matplotlib.pyplot as plt
# Define the SQL query
sql = """
SELECT g.Name AS Genre, COUNT(t.track_id) AS Tracks
FROM genre g
JOIN track t ON g.genre_id = t.genre_id
GROUP BY Genre
ORDER BY Tracks DESC;
"""
# Read the data into a dataframe
data = pd.read_sql(sql, conn)
# Plot the data as a bar chart
plt.bar(data.Genre, data.Tracks)
plt.title("Number of Tracks by Genre")
plt.xlabel("Genre")
plt.ylabel("Tracks")
plt.xticks(rotation=90)
plt.show()
```

We'll see the following output:

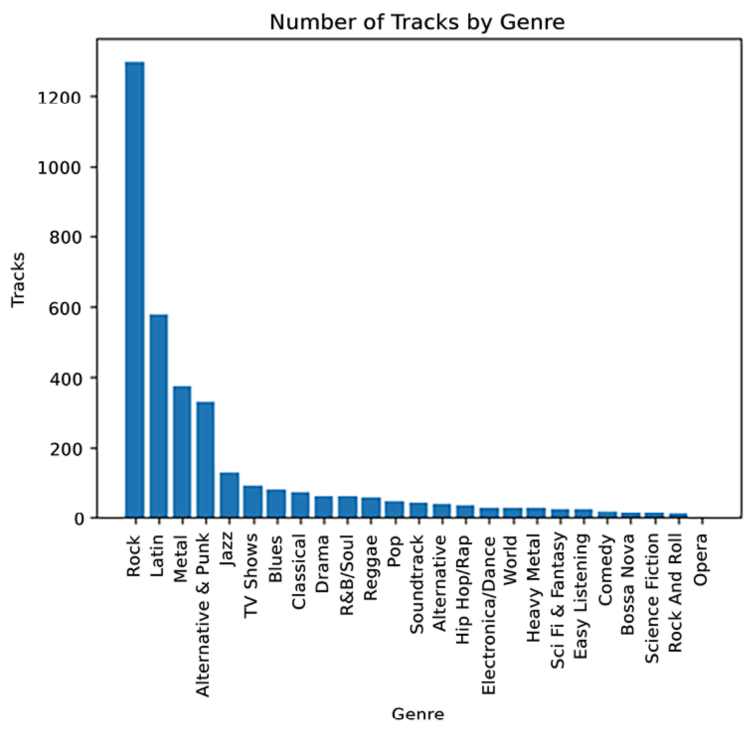


Figure 8.5: Number of tracks by genre

As you can see, in order to gather relevant information from our database, we used the syntax of SQL. Our goal is to gather information by simply asking in natural language, and we are going to do so starting in the next section.

Implementing the DBCopilot with LangChain

In this section, we are going to cover the architecture and implementation steps behind a DBCopilot application, a natural language interface to chat with database-structured data. In the upcoming sections, we will explore how to achieve that by leveraging a powerful LangChain component called SQL Agent.

LangChain agents and SQL Agent

In *Chapter 4*, we introduced the concept of LangChain agents, defining them as entities that drive decision making within LLM-powered applications.

Agents have access to a suite of tools and can decide which tool to call based on the user input and the context. Agents are dynamic and adaptive, meaning that they can change or adjust their actions based on the situation or the goal.

In this chapter, we will see agents in action, using the following LangChain components:

- `create_sql_agent`: An agent designed to interact with relational databases
- `SQLDatabaseToolkit`: A toolkit to provide the agent with the required non-parametric knowledge

- `OpenAI`: An LLM to act as the reasoning engine behind the agent, as well as the generative engine to produce conversational results

Let's start with our implementation by following these steps:

1. We'll first initialize all the components and establish the connection to the Chinook database, using the `SQLDatabase` LangChain component (which uses `SQLAlchemy` under the hood and is used to connect to our database):

```
from langchain.agents import create_sql_agent
from langchain.llms import OpenAI
from langchain.chat_models import ChatOpenAI
from langchain.agents.agent_toolkits import SQLDatabaseToolkit
from langchain.sql_database import SQLDatabase
from langchain.llms.openai import OpenAI
from langchain.agents import AgentExecutor
from langchain.agents.agent_types import AgentType
from langchain.chat_models import ChatOpenAI
llm = OpenAI()
db = SQLDatabase.from_uri('sqlite:///chinook.db')
toolkit = SQLDatabaseToolkit(db=db, llm=llm)
agent_executor = create_sql_agent(
    llm=llm,
    toolkit=toolkit,
    verbose=True,
    agent_type=AgentType.ZERO_SHOT_REACT_DESCRIPTION,
)
```

2. Before running the agent, let's first inspect its available tools:

```
[tool.name for tool in toolkit.get_tools()]
```

Here is the output:

```
['sql_db_query', 'sql_db_schema', 'sql_db_list_tables', 'sql_db_query_checker']
```

Those tools have the following capabilities:

- `sql_db_query`: This takes as input a detailed and correct SQL query, and it outputs a result from the database. If the query is not correct, an error message will be returned.
- `sql_db_schema`: This takes as input a comma-separated list of tables, and it outputs the schema and sample rows for those tables.
- `sql_db_list_tables`: This takes as input an empty string, and it outputs a comma-separated list of tables in the database.
- `sql_db_query_checker`: This tool double-checks whether the query is correct before executing it.

3. Let's now execute our agent with a simple query to describe the `playlisttrack` table:

```
agent_executor.run("Describe the playlisttrack table")
```

The following output is then obtained (the output is truncated – you can find the full output in the book's GitHub repository):

```
> Entering new AgentExecutor chain...
Action: sql_db_list_tables
Action Input:
Observation: album, artist, customer, employee, genre, invoice, invoice_line, media_type,
Thought: The table I need is playlist_track
Action: sql_db_schema
Action Input: playlist_track
Observation:
CREATE TABLE playlist_track (
[...]
> Finished chain.
'The playlist_track table contains the playlist_id and track_id columns. It has a primary
```

As you can see, with a simple question in natural language, our agent was able to understand its semantics, translate it into an SQL query, extract the relevant information, and use it as context to generate the response.

But how was it able to do all of that? Under the hood, the SQL agent comes with a default prompt template, which makes it tailored to this type of activity. Let's see the default template of the LangChain component:

```
print(agent_executor.agent.llm_chain.prompt.template)
```

Here is the output obtained:

```
You are an agent designed to interact with a SQL database.
Given an input question, create a syntactically correct sqlite query to run, then look at
Unless the user specifies a specific number of examples they wish to obtain, always limit
You can order the results by a relevant column to return the most interesting examples in
Never query for all the columns from a specific table, only ask for the relevant columns g
You have access to tools for interacting with the database.
Only use the below tools. Only use the information returned by the below tools to construc
You MUST double check your query before executing it. If you get an error while executing
DO NOT make any DML statements (INSERT, UPDATE, DELETE, DROP etc.) to the database.
If the question does not seem related to the database, just return "I don't know" as the a
sql_db_query: Input to this tool is a detailed and correct SQL query, output is a result f
sql_db_schema: Input to this tool is a comma-separated list of tables, output is the schem
Be sure that the tables actually exist by calling sql_db_list_tables first! Example Input:
sql_db_list_tables: Input is an empty string, output is a comma separated list of tables i
sql_db_query_checker: Use this tool to double check if your query is correct before execut
Use the following format:
Question: the input question you must answer
Thought: you should always think about what to do
Action: the action to take, should be one of [sql_db_query, sql_db_schema, sql_db_list_tab
Action Input: the input to the action
...
Question: {input}
Thought: I should look at the tables in the database to see what I can query.  Then I shou
{agent_scratchpad}
```

Thanks to this prompt template, the agent is able to use the proper tools and generate a SQL query, without modifying the underlying database (you can see the explicit rule not to run any **data manipulation language (DML)** statements).

Definition

DML is a class of SQL statements that are used to query, edit, add, and delete row-level data from database tables or views. The main DML statements are as follows:

- `SELECT` : This is used to retrieve data from one or more tables or views based on specified criteria.
- `INSERT` : This is used to insert new data records or rows into a table.
- `UPDATE` : This is used to modify the values of existing data records or rows in a table.
- `DELETE` : This is used to remove one or more data records or rows from a table.
- `MERGE` : This is used to combine the data from two tables into one based on a common column.
- DML statements are used to store, modify, retrieve, delete, and update data in a database.

We can also see how the agent is able to correlate more than one table within the database:

```
agent_executor.run('what is the total number of tracks and the average length of tracks by
```

From the first lines of the chain, you can see that `Action Input` invokes two tables – track and genre:

```
> Entering new AgentExecutor chain...
Action: sql_db_list_tables
Action Input:
Observation: album, artist, customer, employee, genre, invoice, invoice_line, media_type,
Thought: I should look at the schema of the track and genre tables.
Action: sql_db_schema
Action Input: track, genre
[…]
```

The following is the output:

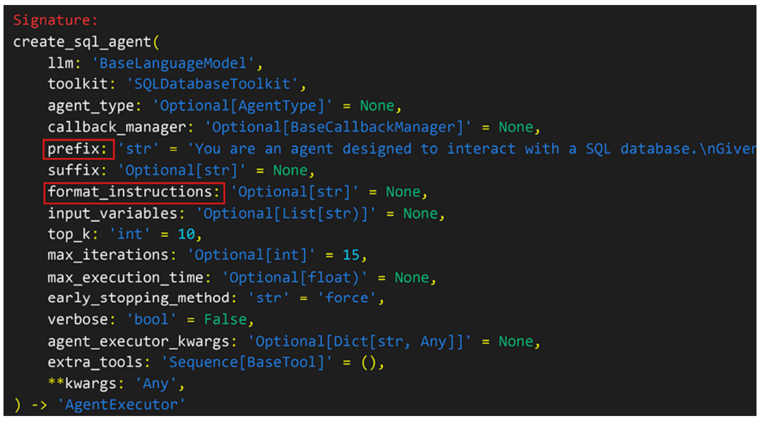
```
'The top 10 genres by track count and average track length are Rock (1297 tracks with an a
```

Now, the question is as follows: are we sure that we are getting the proper result? A nice way to double-check this would be to print the SQL query that the agent ran against the database. To do so, we can modify the default prompt to ask the agent to explicitly show us the reasoning behind its result.

Prompt engineering

As we saw in the previous chapter, pre-built LangChain agents and chains come with default prompts, which make it easier to tailor them toward their goals. Nevertheless, we can customize that prompt and pass it as a parameter to our component. For example, let's say that we want our SQL agent to print the SQL query it used to return the result.

First of all, we have to understand which kind of prompt chunks the SQL Agent is able to take as parameters. To do so, we can simply inspect the objects running `create_sql_agent`.



```
Signature:
create_sql_agent(
    llm: 'BaseLanguageModel',
    toolkit: 'SQLDatabaseToolkit',
    agent_type: 'Optional[AgentType]' = None,
    callback_manager: 'Optional[BaseCallbackManager]' = None,
    prefix: 'str' = 'You are an agent designed to interact with a SQL database.\nGiver
    suffix: 'Optional[str]' = None,
    format_instructions: 'Optional[str]' = None,
    input_variables: 'Optional[List[str]]' = None,
    top_k: 'int' = 10,
    max_iterations: 'Optional[int]' = 15,
    max_execution_time: 'Optional[float]' = None,
    early_stopping_method: 'str' = 'force',
    verbose: 'bool' = False,
    agent_executor_kwargs: 'Optional[Dict[str, Any]]' = None,
    extra_tools: 'Sequence[BaseTool]' = (),
    **kwargs: 'Any',
) -> 'AgentExecutor'
```

Figure 8.6: A screenshot of the description of the SQL agent

The Agent takes a prompt prefix and a format instruction, which are merged and constitute the default prompt we inspected in the previous section. To make our agent more self-explanatory, we will create two variables, `prefix` and `format_instructions`, which will be passed as parameters and that slightly modify the default prompt as follows (you can find the whole prompts in the GitHub repository at **https://github.com/PacktPublishing/Building-LLM-Powered-Applications**:

- We have the `prompt_prefix`, which is already configured as follows:

```
prefix: 'str' = 'You are an agent designed to interact with a SQL database.\nGiven an i
```

To this, we will add the following line of instruction:

```
As part of your final answer, ALWAYS include an explanation of how to got to the final ans
```

- In `prompt_format_instructions`, we will add the following example of explanation using few-shot learning, which we covered in *Chapter 1*:

~~~
Explanation:
<===Beginning of an Example of Explanation:
I joined the invoices and customers tables on the customer_id column, which is the comm
```sql
SELECT c.country AS Country, SUM(i.total) AS Sales
FROM customer c
JOIN invoice i ON c.customer_id = i.customer_id
GROUP BY Country
ORDER BY Sales DESC
LIMIT 5;
```sql
===>End of an Example of Explanation
~~~

Now, let's pass those prompt chunks as parameters to our agent and print the result (I will omit the whole chain here, but you can see it in the GitHub repository):

```
agent_executor = create_sql_agent(
    prefix=prompt_prefix,
    format_instructions = prompt_format_instructions,
    llm=llm,
    toolkit=toolkit,
    verbose=True,
    top_k=10
)
result = agent_executor.run("What are the top 5 best-selling albums and their artists?")
print(result)
```

Here is the obtained output:

~~~
The top 5 best-selling albums and their artists are 'A Matter of Life and Death' by Iron M
Explanation: I joined the album and invoice tables on the album_id column and joined the a
```sql
SELECT al.title AS Album, ar.name AS Artist, SUM(i.total) AS Sales
FROM album al
JOIN invoice i ON al.album_id = i.invoice_id
JOIN artist ar ON al.artist_id = ar.artist_id
GROUP BY ar.name
ORDER BY Sales
~~~

Now, in our result, we have a clear explanation of the thought process as well as the printed query our agent made for us. This is key if we want to double-check the correctness of the reasoning procedure happening in the backend of our agent.

This is already extremely useful, but we want to bring it to the next level: we want our DBCopilot to also be able to generate graphs and save results in our local file system. To achieve this goal, we need to add tools to our agent, and we are going to do so in the next section.

Adding further tools

In order to make our DBCopilot more versatile, there are two further capabilities we need to add:

- **PythonREPLTool**: This tool allows you to interact with the Python programming language using natural language. You can use this tool to write, run, and debug Python code without having to use a script file or an IDE. You can also use this tool to access and manipulate various Python modules, libraries, and data structures. **We will need this tool to produce the matplotlib graphs from the SQL query's results.**

Definition

REPL is an acronym for read-eval-print loop, which is a term that describes an interactive shell or environment that allows you to execute code and see the results immediately. REPL is a common feature of many programming languages, such as Python, Ruby, and Lisp.

In the context of LangChain, REPL is a feature that allows you to interact with LangChain agents and tools using natural language. You can use REPL in LangChain to test, debug, or experiment with different agents and tools without having to write and run a script file. You can also use

REPL in LangChain to access and manipulate various data sources, such as databases, APIs, and web pages.

- **FileManagementToolkit**: This is a set of tools, or toolkit, that allows you to interact with the file system of your computer or device using natural language. You can use this toolkit to perform various operations on files and directories, such as creating, deleting, renaming, copying, moving, searching, reading, and writing. You can also use this toolkit to access and manipulate the metadata and attributes of files and directories, such as name, size, type, date, and permissions.

We will need this toolkit to save the graphs generated by our agent in our working directory.

Now, let's see how we can add these tools to our DBCopilot:

1. First, we define the list of tools for our agent:

```
from  langchain_experimental.tools.python.tool import PythonREPLTool
from  langchain_experimental.python import PythonREPL
from langchain.agents.agent_toolkits import FileManagementToolkit
working_directory  = os.getcwd()
tools = FileManagementToolkit(
    root_dir=str(working_directory),
    selected_tools=["read_file", "write_file", "list_directory"],).get_tools()
tools.append(
    PythonREPLTool())
tools.extend(SQLDatabaseToolkit(db=db, llm=llm).get_tools())
```

2. In order to leverage that heterogeneous set of tools – SQL Database, Python REPL, and File System (**https://python.langchain.com/v0.1/docs/integrations/tools/filesystem/**) – we cannot work anymore with the SQL Database-specific agent, since its default configurations are meant to only accept SQL-related contents. Henceforth, we need to set up an agnostic agent that is able to use all of the tools that we provide it with. For this purpose, we are going to use the `STRUCTURED_CHAT_ZERO_SHOT_REACT_DESCRIPTION` agent type, which is able to use a multi-tool input.

Let's first start with initializing the agent and asking it to produce a bar chart and save it in the current working directory for the top five countries for sales (note that, for this purpose, I've used a chat model as best suited for the type of agent in use):

```
from langchain.chat_models import ChatOpenAI
from langchain.agents import initialize_agent, Tool
from langchain.agents import AgentType
model = ChatOpenAI()
agent = initialize_agent(
    tools, model, agent= AgentType.STRUCTURED_CHAT_ZERO_SHOT_REACT_DESCRIPTION, verbose=Tr
)
agent.run("generate a matplotlib bar chart of the top 5 countries for sales from the chino
```

We then receive the following output, showing how, in this case, the agent was also able to dynamically orchestrate the available tools to generate the final answer (I will report here just the main actions of the chain – you can see the whole code in the GitHub repository of the book):

```
> Entering new AgentExecutor chain...
Action:
```
{
  "action": "sql_db_query",
  "action_input": "SELECT billing_country as Country, SUM(total) as Sales FROM invoices GR
}
```
[…]
Observation: [('USA', 10405.889999999912), ('Canada', 5489.549999999994), ('Brazil', 4058.
[…]
We have successfully retrieved the top 5 countries for sales. We can now use matplotlib to
Action:
```
{
  "action": "Python_REPL",
  "action_input": "import matplotlib.pyplot as plt\nsales_data = [('USA', 10405.89), ('Can
}
```
[…]
> Finished chain.
'Here is the bar chart of the top 5 countries for sales from the Chinook database. It has
```

The following is the generated chart of the top five countries by sales, as requested:

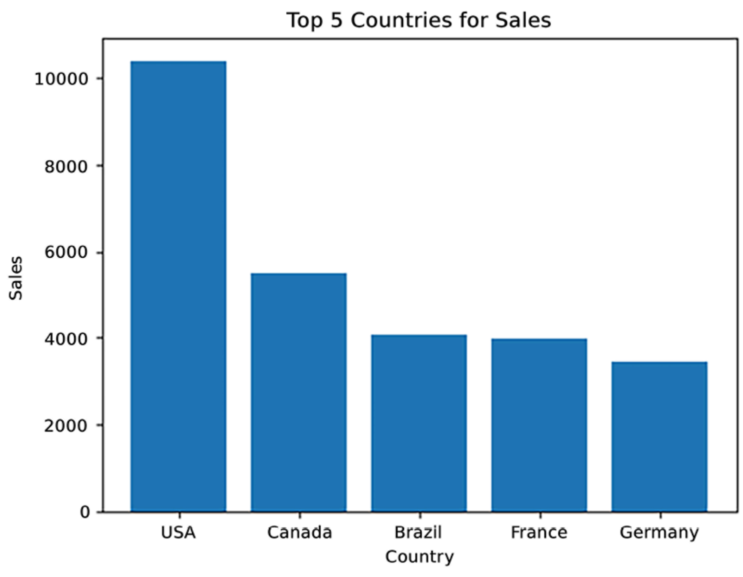


Figure 8.7: Bar chart of top five countries by sales

Great! The agent was able to first invoke the SQL tool to retrieve the relevant information, then it used the Python tool to generate the `matplotlib` bar chart. Then, it used the file system tool to save the result as PNG.

Also, in this case, we can modify the prompt of the agent. For example, we might want the agent to provide an explanation not only of the SQL query but also of the Python code. To do so, we need to define the `prompt_prefix` and `prompt_format_instructions` variables to be passed as `kgwargs` to the agent as follows:

```
prompt_prefix = """ Your prefix here
"""
prompt_format_instructions= """
Your instructions here.
"""
agent = initialize_agent(tools, model, agent=AgentType.STRUCTURED_CHAT_ZERO_SHOT_REACT_DES
                         agent_kwargs={
                            'prefix':prompt_prefix,
                            'format_instructions': prompt_format_instructions })
```

Thanks to LangChain's tools components, we were able to extend our DBCopilot capabilities and make it more versatile, depending upon the user's query.

With the same logic, we can tailor our agents to any domain, adding or removing tools so that we can control its perimeter of actions. Plus, thanks to the prompt customization, we can always refine the agent's backend logic to make it more customized.

Developing the front-end with Streamlit

Now that we have seen the logic behind an LLM-powered DBCopilot, it is time to give a GUI to our application. To do so, we will once again leverage Streamlit. As always, you can find the whole Python code in the GitHub book repository at **https://github.com/PacktPublishing/Building-LLM-Powered-Applications**.

As per the previous sections, you need to create a `.py` file to run in your terminal via `streamlit run file.py`. In our case, the file will be named `dbcopilot.py`.

Here are the main steps to set up the frontend:

1. Configure the application web page:

```
import streamlit as st
st.set_page_config(page_title="DBCopilot", page_icon="📊")
st.header('📊 Welcome to DBCopilot, your copilot for structured databases.')
```

2. Import the credentials and establish the connection with the Chinook database:

```
load_dotenv()
#os.environ["HUGGINGFACEHUB_API_TOKEN"]
openai_api_key = os.environ['OPENAI_API_KEY']
db = SQLDatabase.from_uri('sqlite:///chinook.db')
```

3. Initialize the LLM and the toolkit:

```
llm = OpenAI()
toolkit = SQLDatabaseToolkit(db=db, llm=llm)
```

4. Initialize the Agent using the prompt variables defined in the previous sections:

```
agent_executor = create_sql_agent(
    prefix=prompt_prefix,
    format_instructions = prompt_format_instructions,
    llm=llm,
    toolkit=toolkit,
    verbose=True,
    top_k=10
)
```

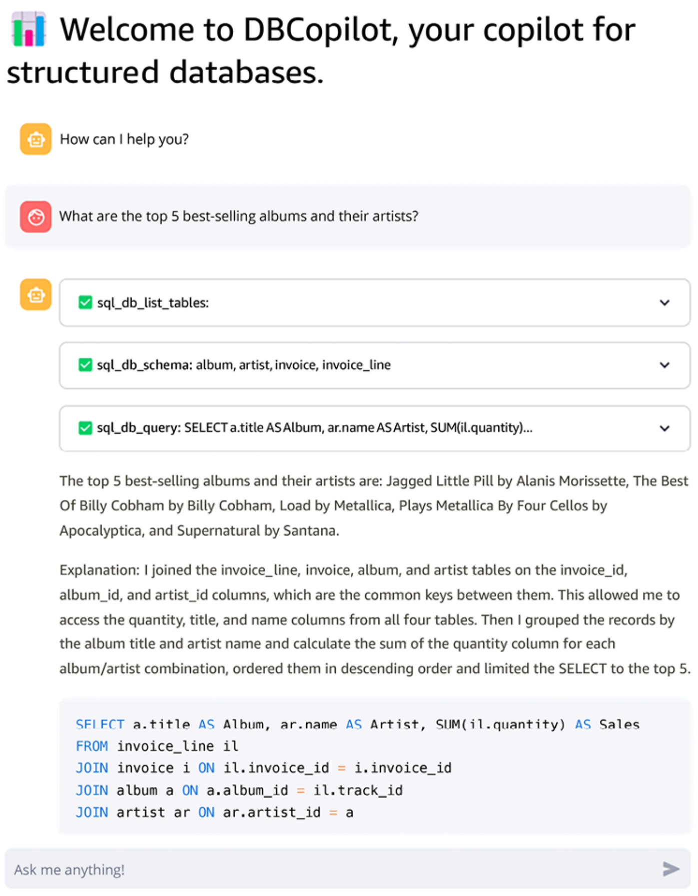
5. Define Streamlit's session states to make it conversational and memory aware:

```
if "messages" not in st.session_state or st.sidebar.button("Clear message history"):
    st.session_state["messages"] = [{"role": "assistant", "content": "How can I help yo
for msg in st.session_state.messages:
    st.chat_message(msg["role"]).write(msg["content"])
```

6. Finally, define the logic of the application whenever a user makes a query:

```
if user_query:
    st.session_state.messages.append({"role": "user", "content": user_query})
    st.chat_message("user").write(user_query)
    with st.chat_message("assistant"):
        st_cb = StreamlitCallbackHandler(st.container())
        response = agent_executor.run(user_query, callbacks = [st_cb], handle_parsing_e
        st.session_state.messages.append({"role": "assistant", "content": response})
        st.write(response)
```

You can run your application in the terminal with the `streamlit run copilot.py` command. The final web page looks as follows:



📊 Welcome to DBCopilot, your copilot for structured databases.

How can I help you?

What are the top 5 best-selling albums and their artists?

sql_db_list_tables:

sql_db_schema: album, artist, invoice, invoice_line

sql_db_query: SELECT a.title AS Album, ar.name AS Artist, SUM(il.quantity)...

The top 5 best-selling albums and their artists are: Jagged Little Pill by Alanis Morissette, The Best Of Billy Cobham by Billy Cobham, Load by Metallica, Plays Metallica By Four Cellos by Apocalyptica, and Supernatural by Santana.

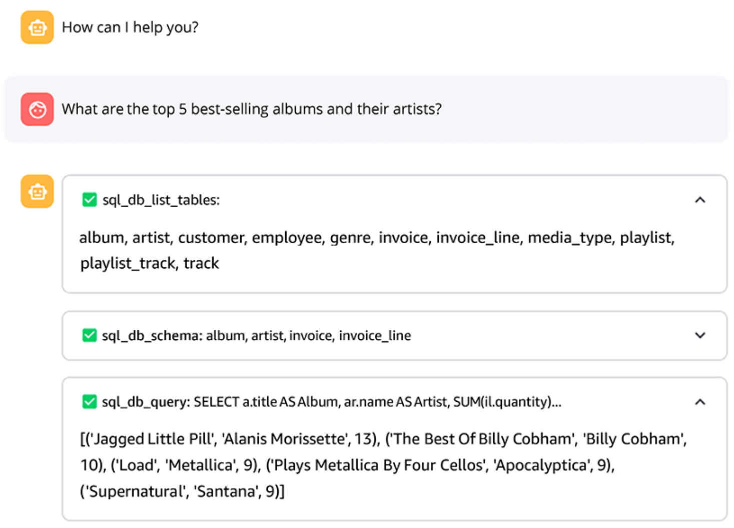
Explanation: I joined the invoice_line, invoice, album, and artist tables on the invoice_id, album_id, and artist_id columns, which are the common keys between them. This allowed me to access the quantity, title, and name columns from all four tables. Then I grouped the records by the album title and artist name and calculate the sum of the quantity column for each album/artist combination, ordered them in descending order and limited the SELECT to the top 5.

```
SELECT a.title AS Album, ar.name AS Artist, SUM(il.quantity) AS Sales
FROM invoice_line il
JOIN invoice i ON il.invoice_id = i.invoice_id
JOIN album a ON a.album_id = il.track_id
JOIN artist ar ON ar.artist_id = a
```

Ask me anything!

Figure 8.8: Screenshot of the front-end of DBCopilot

Thanks to the `StreamlitCallbackHandler` module, we can also expand each action the agent took, for example:



How can I help you?

What are the top 5 best-selling albums and their artists?

sql_db_list_tables:

album, artist, customer, employee, genre, invoice, invoice_line, media_type, playlist, playlist_track, track

sql_db_schema: album, artist, invoice, invoice_line

sql_db_query: SELECT a.title AS Album, ar.name AS Artist, SUM(il.quantity)...

```
[('Jagged Little Pill', 'Alanis Morissette', 13), ('The Best Of Billy Cobham', 'Billy Cobham', 10), ('Load', 'Metallica', 9), ('Plays Metallica By Four Cellos', 'Apocalyptica', 9), ('Supernatural', 'Santana', 9)]
```

Figure 8.9: Illustration of the agent's actions during the chain

With just a few lines of code, we were able to set up a simple front-end for our DBCopilot with a conversational user interface.

Summary

In this chapter, we saw how LLMs are not only capable of interacting with textual and unstructured data, but also with structured and numeric data. This is made possible because of two main elements: the natural capabilities of LLMs and, more generally, LFMs for understanding a problem's statement, planning a resolution, and acting as reasoning engines, as well as a set of tools that extend LLMs' capabilities with domain-specific skills.

In this case, we mainly relied upon LangChain's SQL Database toolkit, which connects the Agent to an SQL database with a curated prompt. Furthermore, we extended the Agent's capabilities even further, making it able to generate matplotlib graphs, with the Python REPL tool, and save the output to our local file system with the File Management tool.

In the next chapter, we are going to delve even deeper into the analytical capabilities of LLMs. More specifically, we are going to cover their capabilities of working with code.

References

- Chinook Database: https://github.com/lerocha/chinook-database/tree/master/ChinookDatabase/DataSources
- LangChain File system tool: https://python.langchain.com/docs/integrations/tools/filesystem
- LangChain Python REPL tool: https://python.langchain.com/docs/integrations/toolkits/python

Unlock this book's exclusive benefits now

This book comes with additional benefits designed to elevate your learning experience.

Note: Have your purchase invoice ready before you begin.

https://www.packtpub.com/unlock/9781835462317