



Tutorials



Build a Question/Answering system over SQL data

Build a Question/Answering system over SQL data

⚠️ PREREQUISITES

This guide assumes familiarity with the following concepts:

- [Chaining runnables](#)
- [Chat models](#)
- [Tools](#)
- [Agents](#)

Enabling a LLM system to query structured data can be qualitatively different from unstructured text data. Whereas in the latter it is common to generate text that can be searched against a vector database, the approach for structured data is often for the LLM to write and execute queries in a DSL, such as SQL. In this guide we'll go over the basic ways to create a Q&A system over tabular data in databases. We will cover implementations using both chains and agents. These systems will allow us to ask a question about the data in a database and get back a natural language answer. The main difference between the two is that our agent can query the database in a loop as many times as it needs to answer the question.

⚠️ **Security note** ⚠️

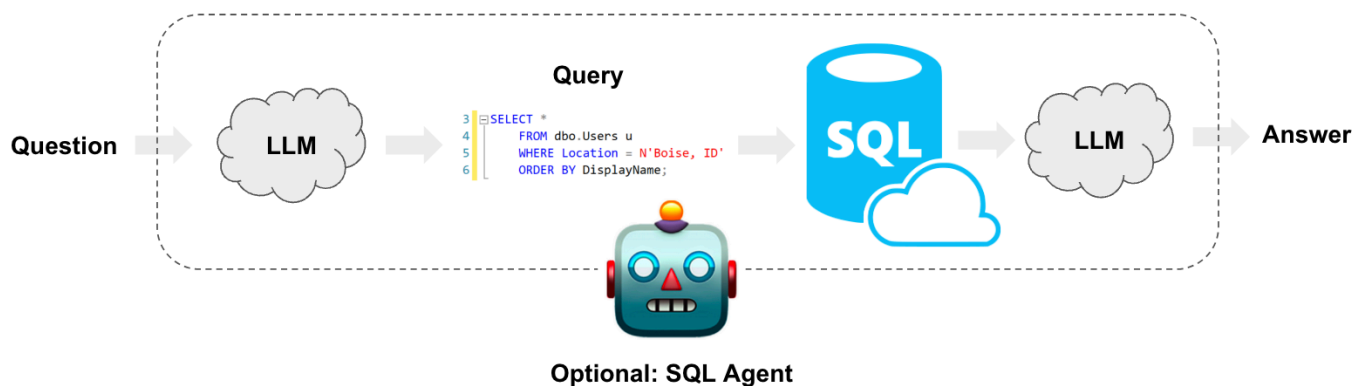
Building Q&A systems of SQL databases requires executing model-generated SQL queries. There are inherent risks in doing this. Make sure that your database connection permissions are always scoped as narrowly as possible for your chain/agent's needs. This will mitigate though not eliminate the risks of building a model-driven system. For more on general security best practices, [see here](#).

Architecture

At a high-level, the steps of these systems are:

1. **Convert question to DSL query:** Model converts user input to a SQL query.
2. **Execute SQL query:** Execute the query.
3. **Answer the question:** Model responds to user input using the query results.

Note that querying data in CSVs can follow a similar approach. See our [how-to guide](#) on question-answering over CSV data for more detail.



Setup

First, get required packages and set environment variables:

```
%capture --no-stderr
%pip install --upgrade --quiet langchain langchain-community
```

langchain-openai faiss-cpu

We will use an OpenAI model and a **FAISS-powered vector store** in this guide.

```
import getpass
import os

if not os.environ.get("OPENAI_API_KEY"):
    os.environ["OPENAI_API_KEY"] = getpass.getpass()

# Comment out the below to opt-out of using LangSmith in this
# notebook. Not required.
if not os.environ.get("LANGCHAIN_API_KEY"):
    os.environ["LANGCHAIN_API_KEY"] = getpass.getpass()
    os.environ["LANGCHAIN_TRACING_V2"] = "true"
```

The below example will use a SQLite connection with Chinook database. Follow [these installation steps](#) to create `Chinook.db` in the same directory as this notebook:

- Save [this file](#) as `Chinook.sql`
- Run `sqlite3 Chinook.db`
- Run `.read Chinook.sql`
- Test `SELECT * FROM Artist LIMIT 10;`

Now, `Chinook.db` is in our directory and we can interface with it using the SQLAlchemy-driven `SQLDatabase` class:

```
from langchain_community.utilities import SQLDatabase

db = SQLDatabase.from_uri("sqlite:///Chinook.db")
print(db.dialect)
print(db.get_usable_table_names())
db.run("SELECT * FROM Artist LIMIT 10;")
```

API Reference: SQLDatabase

```
sqlite  
['Album', 'Artist', 'Customer', 'Employee', 'Genre', 'Invoice',  
'InvoiceLine', 'MediaType', 'Playlist', 'PlaylistTrack',  
'Track']
```

```
"[(1, 'AC/DC'), (2, 'Accept'), (3, 'Aerosmith'), (4, 'Alanis  
Morissette'), (5, 'Alice In Chains'), (6, 'Antônio Carlos  
Jobim'), (7, 'Apocalyptica'), (8, 'Audioslave'), (9,  
'BackBeat'), (10, 'Billy Cobham')]"
```

Great! We've got a SQL database that we can query. Now let's try hooking it up to an LLM.

Chains

Chains (i.e., compositions of LangChain **Runnable**s) support applications whose steps are predictable. We can create a simple chain that takes a question and does the following:

- convert the question into a SQL query;
- execute the query;
- use the result to answer the original question.

There are scenarios not supported by this arrangement. For example, this system will execute a SQL query for any user input--even "hello". Importantly, as we'll see below, some questions require more than one query to answer. We will address these scenarios in the Agents section.

Convert question to SQL query

The first step in a SQL chain or agent is to take the user input and convert it to a SQL query. LangChain comes with a built-in chain for this: `create_sql_query_chain`.

[OpenAI](#)[Anthropic](#)[Azure](#)[Google](#)[Cohere](#)[NVIDIA](#)[Fireworks](#)

```
pip install -qU langchain-openai
```

```
import getpass
import os

os.environ["OPENAI_API_KEY"] = getpass.getpass()

from langchain_openai import ChatOpenAI

llm = ChatOpenAI(model="gpt-4o-mini")
```

```
from langchain.chains import create_sql_query_chain

chain = create_sql_query_chain(llm, db)
response = chain.invoke({"question": "How many employees are there"})
response
```

API Reference: `create_sql_query_chain`

```
'SELECT COUNT("EmployeeId") AS "TotalEmployees" FROM
"Employee"\nLIMIT 1;'
```

We can execute the query to make sure it's valid:

```
db.run(response)
```

```
'[(8,)]'
```

We can look at the [LangSmith trace](#) to get a better understanding of what this chain is doing. We can also inspect the chain directly for its prompts. Looking at the prompt (below), we can see that it is:

- Dialect-specific. In this case it references SQLite explicitly.
- Has definitions for all the available tables.
- Has three examples rows for each table.

This technique is inspired by papers like [this](#), which suggest showing examples rows and being explicit about tables improves performance. We can also inspect the full prompt like so:

```
chain.get_prompts()[0].pretty_print()
```

```
You are a SQLite expert. Given an input question, first create a syntactically correct SQLite query to run, then look at the results of the query and return the answer to the input question.
```

```
Unless the user specifies in the question a specific number of examples to obtain, query for at most 5 results using the LIMIT clause as per SQLite. You can order the results to return the most informative data in the database.
```

```
Never query for all columns from a table. You must query only the columns that are needed to answer the question. Wrap each column name in double quotes (") to denote them as delimited identifiers.
```

```
Pay attention to use only the column names you can see in the tables below. Be careful to not query for columns that do not exist. Also, pay attention to which column is in which table. Pay attention to use date('now') function to get the current date, if the question involves "today".
```

```
Use the following format:
```

```
Question: Question here
SQLQuery: SQL Query to run
SQLResult: Result of the SQLQuery
Answer: Final answer here
```

```
Only use the following tables:
[33;1m [1;3m{table_info} [0m
```

```
Question: [33;1m [1;3m{input} [0m
```

Execute SQL query

Now that we've generated a SQL query, we'll want to execute it. **This is the most dangerous part of creating a SQL chain.** Consider carefully if it is OK to run automated queries over your data. Minimize the database connection permissions as much as possible. Consider adding a human approval step to you chains before query execution (see below).

We can use the `QuerySQLDatabaseTool` to easily add query execution to our chain:

```
from langchain_community.tools.sql_database.tool import
QuerySQLDataBaseTool

execute_query = QuerySQLDataBaseTool(db=db)
write_query = create_sql_query_chain(llm, db)
chain = write_query | execute_query
chain.invoke({"question": "How many employees are there"})
```

API Reference: `QuerySQLDataBaseTool`

```
'[(8,)]'
```

Answer the question

Now that we've got a way to automatically generate and execute queries, we just need to combine the original question and SQL query result to generate a final answer. We can do this by passing question and result to the LLM once more:

```
from operator import itemgetter

from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import PromptTemplate
from langchain_core.runnables import RunnablePassthrough

answer_prompt = PromptTemplate.from_template(
    """Given the following user question, corresponding SQL
    query, and SQL result, answer the user question.

    Question: {question}
    SQL Query: {query}
    SQL Result: {result}
    Answer: """"
)

chain = (
    RunnablePassthrough.assign(query=write_query).assign(
        result=itemgetter("query") | execute_query
    )
    | answer_prompt
    | llm
    | StrOutputParser()
)

chain.invoke({"question": "How many employees are there"})
```

API Reference: [StrOutputParser](#) | [PromptTemplate](#) | [RunnablePassthrough](#)

```
'There are a total of 8 employees.'
```

Let's review what is happening in the above LCEL. Suppose this chain is invoked.

- After the first `RunnablePassthrough.assign`, we have a runnable with two elements:

```
{"question": question, "query": write_query.invoke(question)}
```

Where `write_query` will generate a SQL query in service of answering the question.
- After the second `RunnablePassthrough.assign`, we have add a third element `"result"` that contains `execute_query.invoke(query)`, where `query` was computed in the previous step.
- These three inputs are formatted into the prompt and passed into the LLM.
- The `StrOutputParser()` plucks out the string content of the output message.

Note that we are composing LLMs, tools, prompts, and other chains together, but because each implements the Runnable interface, their inputs and outputs can be tied together in a reasonable way.

Next steps

For more complex query-generation, we may want to create few-shot prompts or add query-checking steps. For advanced techniques like this and more check out:

- **Prompting strategies:** Advanced prompt engineering techniques.
- **Query checking:** Add query validation and error handling.
- **Large databses:** Techniques for working with large databases.

Agents

LangChain has a SQL Agent which provides a more flexible way of interacting with SQL Databases than a chain. The main advantages of using the SQL Agent are:

- It can answer questions based on the databases' schema as well as on the databases' content (like describing a specific table).
- It can recover from errors by running a generated query, catching the traceback and regenerating it correctly.
- It can query the database as many times as needed to answer the user question.
- It will save tokens by only retrieving the schema from relevant tables.

To initialize the agent we'll use the `SQLDatabaseToolkit` to create a bunch of tools:

- Create and execute queries
- Check query syntax
- Retrieve table descriptions
- ... and more

```
from langchain_community.agent_toolkits import
SQLDatabaseToolkit

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

tools = toolkit.get_tools()

tools
```

API Reference: `SQLDatabaseToolkit`

```
[QuerySQLDataBaseTool(description="Input to this tool is a
detailed and correct SQL query, output is a result from the
database. If the query is not correct, an error message will be
returned. If an error is returned, rewrite the query, check the
query, and try again. If you encounter an issue with Unknown
column 'xxxx' in 'field list', use sql_db_schema to query the
correct table fields.", db=
```

```

<langchain_community.utilities.sql_database.SQLDatabase object
at 0x113403b50>),
    InfoSQLDatabaseTool(description='Input to this tool is a comma-
separated list of tables, output is the schema and sample rows
for those tables. Be sure that the tables actually exist by
calling sql_db_list_tables first! Example Input: table1, table2,
table3', db=
<langchain_community.utilities.sql_database.SQLDatabase object
at 0x113403b50>),
    ListSQLDatabaseTool(db=
<langchain_community.utilities.sql_database.SQLDatabase object
at 0x113403b50>),
    QuerySQLCheckerTool(description='Use this tool to double check
if your query is correct before executing it. Always use this
tool before executing a query with sql_db_query!', db=
<langchain_community.utilities.sql_database.SQLDatabase object
at 0x113403b50>, llm=ChatOpenAI(client=
<openai.resources.chat.completions.Completions object at
0x115b7e890>, async_client=
<openai.resources.chat.completions.AsyncCompletions object at
0x115457e10>, temperature=0.0,
openai_api_key=SecretStr('*****'), openai_proxy=''),
    llm_chain=LLMChain(prompt=PromptTemplate(input_variables=
['dialect', 'query'], template='\n{query}\nDouble check the
{dialect} query above for common mistakes, including:\n- Using
NOT IN with NULL values\n- Using UNION when UNION ALL should
have been used\n- Using BETWEEN for exclusive ranges\n- Data
type mismatch in predicates\n- Properly quoting identifiers\n-
Using the correct number of arguments for functions\n- Casting
to the correct data type\n- Using the proper columns for
joins\n\nIf there are any of the above mistakes, rewrite the
query. If there are no mistakes, just reproduce the original
query.\n\nOutput the final SQL query only.\n\nSQL Query: '),
    llm=ChatOpenAI(client=
<openai.resources.chat.completions.Completions object at
0x115b7e890>, async_client=
<openai.resources.chat.completions.AsyncCompletions object at
0x115457e10>, temperature=0.0,
openai_api_key=SecretStr('*****'), openai_proxy='')))]

```

System Prompt

We will also want to create a system prompt for our agent. This will consist of instructions for how to behave.

```
from langchain_core.messages import SystemMessage

SQL_PREFIX = """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 the results of the query and return
the answer.
Unless the user specifies a specific number of examples they
wish to obtain, always limit your query to at most 5 results.
You can order the results by a relevant column to return the
most interesting examples in the database.
Never query for all the columns from a specific table, only ask
for the relevant columns given the question.
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 construct your final answer.
You MUST double check your query before executing it. If you
get an error while executing a query, rewrite the query and try
again.

DO NOT make any DML statements (INSERT, UPDATE, DELETE, DROP
etc.) to the database.

To start you should ALWAYS look at the tables in the database
to see what you can query.
Do NOT skip this step.
Then you should query the schema of the most relevant
tables."""

system_message = SystemMessage(content=SQL_PREFIX)
```

API Reference: [SystemMessage](#)

Initializing agent

First, get required package **LangGraph**

```
%capture --no-stderr
%pip install --upgrade --quiet langgraph
```

We will use a prebuilt **LangGraph** agent to build our agent

```
from langchain_core.messages import HumanMessage
from langgraph.prebuilt import create_react_agent

agent_executor = create_react_agent(llm, tools,
messages_modifier=system_message)
```

API Reference: [HumanMessage](#) | [create_react_agent](#)

Consider how the agent responds to the below question:

```
for s in agent_executor.stream(
    {"messages": [HumanMessage(content="Which country's
customers spent the most?")]}
):
    print(s)
    print("-----")
```

```
{'agent': {'messages': [AIMessage(content='', additional_kwargs=
{'tool_calls': [{'id': 'call_vnHKe3oul1xbpX0Vrb2vsamZ',
'function': {'arguments': '{"query":"SELECT c.Country,
SUM(i.Total) AS Total_Spent FROM customers c JOIN invoices i ON
c.CustomerId = i.CustomerId GROUP BY c.Country ORDER BY
Total_Spent DESC LIMIT 1"}', 'name': 'sql_db_query'}, 'type':
'function'}]]}, response_metadata={'token_usage':
{'completion_tokens': 53, 'prompt_tokens': 557, 'total_tokens':
610}, 'model_name': 'gpt-3.5-turbo', 'system_fingerprint':
'fp_3b956da36b', 'finish_reason': 'tool_calls', 'logprobs':
None}, id='run-da250593-06b5-414c-a9d9-3fc77036dd9c-0',
tool_calls=[{'name': 'sql_db_query', 'args': {'query': 'SELECT
c.Country, SUM(i.Total) AS Total_Spent FROM customers c JOIN
invoices i ON c.CustomerId = i.CustomerId GROUP BY c.Country
ORDER BY Total_Spent DESC LIMIT 1'}, 'id':
```

```
'call_vnHKe3oul1xbpX0Vrb2vsamZ'}})]])}]}
```

```
-----
{'action': {'messages': [ToolMessage(content='Error:
(sqlite3.OperationalError) no such table: customers\n[SQL:
SELECT c.Country, SUM(i.Total) AS Total_Spent FROM customers c
JOIN invoices i ON c.CustomerId = i.CustomerId GROUP BY
c.Country ORDER BY Total_Spent DESC LIMIT 1]\n(Background on
this error at: https://sqlalche.me/e/20/e3q8)',
name='sql_db_query', id='1a5c85d4-1b30-4af3-ab9b-325cbce3b2b4',
tool_call_id='call_vnHKe3oul1xbpX0Vrb2vsamZ'))]}
-----
```

```

{'agent': {'messages': [AIMessage(content='', additional_kwargs=
{'tool_calls': [{'id': 'call_pp3BBD1hwpdwskJ63G3tgaQ',
'function': {'arguments': '{}', 'name': 'sql_db_list_tables'},
'type': 'function'}]}], response_metadata={'token_usage':
{'completion_tokens': 12, 'prompt_tokens': 699, 'total_tokens':
711}, 'model_name': 'gpt-3.5-turbo', 'system_fingerprint':
'fp_3b956da36b', 'finish_reason': 'tool_calls', 'logprobs':
None}, id='run-04cf0e05-61d0-4673-b5dc-1a9b5fd71fff-0',
tool_calls=[{'name': 'sql_db_list_tables', 'args': {}, 'id':
'call_pp3BBD1hwpdwskJ63G3tgaQ'}])]}
-----
```

```

{'action': {'messages': [ToolMessage(content='Album, Artist,
Customer, Employee, Genre, Invoice, InvoiceLine, MediaType,
Playlist, PlaylistTrack, Track', name='sql_db_list_tables',
id='c2668450-4d73-4d32-8d75-8aac8fa153fd',
tool_call_id='call_pp3BBD1hwpdwskJ63G3tgaQ'))]}
-----
```

```

{'agent': {'messages': [AIMessage(content='', additional_kwargs=
{'tool_calls': [{'id': 'call_22Asbqgdx26YyEvJxBuANVdY',
'function': {'arguments': '{"query": "SELECT c.Country,
SUM(i.Total) AS Total_Spent FROM Customer c JOIN Invoice i ON
c.CustomerId = i.CustomerId GROUP BY c.Country ORDER BY
Total_Spent DESC LIMIT 1"}', 'name': 'sql_db_query'}, 'type':
'function'}]}], response_metadata={'token_usage':
{'completion_tokens': 53, 'prompt_tokens': 744, 'total_tokens':
797}, 'model_name': 'gpt-3.5-turbo', 'system_fingerprint':
'fp_3b956da36b', 'finish_reason': 'tool_calls', 'logprobs':
None}, id='run-bdd94241-ca49-4f15-b31a-b7c728a34ea8-0',
tool_calls=[{'name': 'sql_db_query', 'args': {'query': 'SELECT
c.Country, SUM(i.Total) AS Total_Spent FROM Customer c JOIN
Invoice i ON c.CustomerId = i.CustomerId GROUP BY c.Country
ORDER BY Total_Spent DESC LIMIT 1'}, 'id':

```

```
'call_22Asbqgdx26YyEvJxBuANVdY'}})]])}]}
```

```
-----
```

```
{'action': {'messages': [ToolMessage(content="(['USA',
523.06000000000003)]", name='sql_db_query', id='f647e606-8362-
40ab-8d34-612ff166dbe1',
tool_call_id='call_22Asbqgdx26YyEvJxBuANVdY')]]}}
```

```
-----
```

```
{'agent': {'messages': [AIMessage(content='Customers from the
USA spent the most, with a total amount spent of $523.06.',
response_metadata={'token_usage': {'completion_tokens': 20,
'prompt_tokens': 819, 'total_tokens': 839}, 'model_name': 'gpt-
3.5-turbo', 'system_fingerprint': 'fp_3b956da36b',
'finish_reason': 'stop', 'logprobs': None}, id='run-92e88de0-
ff62-41da-8181-053fb5632af4-0')]]}}
```

```
-----
```

Note that the agent executes multiple queries until it has the information it needs:

1. List available tables;
2. Retrieves the schema for three tables;
3. Queries multiple of the tables via a join operation.

The agent is then able to use the result of the final query to generate an answer to the original question.

The agent can similarly handle qualitative questions:

```
for s in agent_executor.stream(
    {"messages": [HumanMessage(content="Describe the
playlisttrack table")]}
):
    print(s)
    print("-----")
```

```
{'agent': {'messages': [AIMessage(content='', additional_kwargs=
{'tool_calls': [{'id': 'call_WN0N3mm8WFvPXYlK9P7KvIEr',
'function': {'arguments': '{"table_names": "playlisttrack"}',
```

```
'name': 'sql_db_schema'}, 'type': 'function']]],
response_metadata={'token_usage': {'completion_tokens': 17,
'prompt_tokens': 554, 'total_tokens': 571}, 'model_name': 'gpt-
3.5-turbo', 'system_fingerprint': 'fp_3b956da36b',
'finish_reason': 'tool_calls', 'logprobs': None}, id='run-
be278326-4115-4c67-91a0-6dc97e7bffa4-0', tool_calls=[{'name':
'sql_db_schema', 'args': {'table_names': 'playlisttrack'}, 'id':
'call_WN0N3mm8WFvPXylK9P7KvIEr'}]]])}]}
```

```
----
{'action': {'messages': [ToolMessage(content="Error: table_names
{'playlisttrack'} not found in database", name='sql_db_schema',
id='fe32b3d3-a40f-4802-a6b8-87a2453af8c2',
tool_call_id='call_WN0N3mm8WFvPXylK9P7KvIEr')]]}}
```

```
----
{'agent': {'messages': [AIMessage(content='I apologize for the
error. Let me first check the available tables in the
database.', additional_kwargs={'tool_calls': [{'id':
'call_CzHt30847ql2MmnGxgYeVSL2', 'function': {'arguments': '{}',
'name': 'sql_db_list_tables'}, 'type': 'function'}]],
response_metadata={'token_usage': {'completion_tokens': 30,
'prompt_tokens': 592, 'total_tokens': 622}, 'model_name': 'gpt-
3.5-turbo', 'system_fingerprint': 'fp_3b956da36b',
'finish_reason': 'tool_calls', 'logprobs': None}, id='run-
f6c107bb-e945-4848-a83c-f57daec1144e-0', tool_calls=[{'name':
'sql_db_list_tables', 'args': {}, 'id':
'call_CzHt30847ql2MmnGxgYeVSL2'}]]])}]}
```

```
----
{'action': {'messages': [ToolMessage(content='Album, Artist,
Customer, Employee, Genre, Invoice, InvoiceLine, MediaType,
Playlist, PlaylistTrack, Track', name='sql_db_list_tables',
id='a4950f74-a0ad-4558-ba54-7bcf99539a02',
tool_call_id='call_CzHt30847ql2MmnGxgYeVSL2')]]}}
```

```
----
{'agent': {'messages': [AIMessage(content='The database contains
a table named "PlaylistTrack". Let me retrieve the schema and
sample rows from the "PlaylistTrack" table.', additional_kwargs=
{'tool_calls': [{'id': 'call_wX9IjHLgRBumxlfCthprABR0',
'function': {'arguments': '{"table_names": "PlaylistTrack"}',
'name': 'sql_db_schema'}, 'type': 'function'}]],
response_metadata={'token_usage': {'completion_tokens': 44,
'prompt_tokens': 658, 'total_tokens': 702}, 'model_name': 'gpt-
3.5-turbo', 'system_fingerprint': 'fp_3b956da36b',
'finish_reason': 'tool_calls', 'logprobs': None}, id='run-
```



```
e8d34372-1159-4654-a185-1e7d0cb70269-0', tool_calls=[{'name':
'sql_db_schema', 'args': {'table_names': 'PlaylistTrack'}, 'id':
'call_wX9IjHLgRBUmxfCthprABR0'}}]]}}
----
{'action': {'messages': [ToolMessage(content='\nCREATE TABLE
"PlaylistTrack" (\n\t"PlaylistId" INTEGER NOT NULL,
\n\t"TrackId" INTEGER NOT NULL, \n\tPRIMARY KEY ("PlaylistId",
"TrackId"), \n\tFOREIGN KEY("TrackId") REFERENCES "Track"
("TrackId"), \n\tFOREIGN KEY("PlaylistId") REFERENCES "Playlist"
("PlaylistId")\n)\n\n/*\n3 rows from PlaylistTrack
table:\nPlaylistId\tTrackId\n1\t3402\n1\t3389\n1\t3390\n*/',
name='sql_db_schema', id='f6ffc37a-188a-4690-b84e-c9f2c78b1e49',
tool_call_id='call_wX9IjHLgRBUmxfCthprABR0')}}
----
{'agent': {'messages': [AIMessage(content='The "PlaylistTrack"
table has the following schema:\n- PlaylistId: INTEGER (NOT
NULL)\n- TrackId: INTEGER (NOT NULL)\n- Primary Key:
(PlaylistId, TrackId)\n- Foreign Key: TrackId references
Track(TrackId)\n- Foreign Key: PlaylistId references
Playlist(PlaylistId)\n\nHere are 3 sample rows from the
"PlaylistTrack" table:\n1. PlaylistId: 1, TrackId: 3402\n2.
PlaylistId: 1, TrackId: 3389\n3. PlaylistId: 1, TrackId:
3390\n\nIf you have any specific questions or queries regarding
the "PlaylistTrack" table, feel free to ask!',
response_metadata={'token_usage': {'completion_tokens': 145,
'prompt_tokens': 818, 'total_tokens': 963}, 'model_name': 'gpt-
3.5-turbo', 'system_fingerprint': 'fp_3b956da36b',
'finish_reason': 'stop', 'logprobs': None}, id='run-961a4552-
3cbd-4d28-b338-4d2f1ac40ea0-0')}}
----
```

Dealing with high-cardinality columns

In order to filter columns that contain proper nouns such as addresses, song names or artists, we first need to double-check the spelling in order to filter the data correctly.

We can achieve this by creating a vector store with all the distinct proper nouns that exist in the database. We can then have the agent query that vector store each time

the user includes a proper noun in their question, to find the correct spelling for that word. In this way, the agent can make sure it understands which entity the user is referring to before building the target query.

First we need the unique values for each entity we want, for which we define a function that parses the result into a list of elements:

```
import ast
import re

def query_as_list(db, query):
    res = db.run(query)
    res = [el for sub in ast.literal_eval(res) for el in sub if
    el]
    res = [re.sub(r"\b\d+\b", "", string).strip() for string in
    res]
    return list(set(res))

artists = query_as_list(db, "SELECT Name FROM Artist")
albums = query_as_list(db, "SELECT Title FROM Album")
albums[:5]
```

```
['Big Ones',
 'Cidade Negra - Hits',
 'In Step',
 'Use Your Illusion I',
 'Voodoo Lounge']
```

Using this function, we can create a **retriever tool** that the agent can execute at its discretion.

```
from langchain.agents.agent_toolkits import
create_retriever_tool
from langchain_community.vectorstores import FAISS
```

```
from langchain_openai import OpenAIEmbeddings

vector_db = FAISS.from_texts(artists + albums,
OpenAIEmbeddings())
retriever = vector_db.as_retriever(search_kwargs={"k": 5})
description = """Use to look up values to filter on. Input is
an approximate spelling of the proper noun, output is \
valid proper nouns. Use the noun most similar to the search."""
retriever_tool = create_retriever_tool(
    retriever,
    name="search_proper_nouns",
    description=description,
)
```

API Reference: [create_retriever_tool](#) | [FAISS](#) | [OpenAIEmbeddings](#)

Let's try it out:

```
print(retriever_tool.invoke("Alice Chains"))
```

Alice In Chains

Alanis Morissette

Pearl Jam

Pearl Jam

Audioslave

This way, if the agent determines it needs to write a filter based on an artist along the lines of "Alice Chains", it can first use the retriever tool to observe relevant values of a column.

Putting this together:

```
system = """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 the results of the query and return the answer.

Unless the user specifies a specific number of examples they wish to obtain, always limit your query to at most 5 results. You can order the results by a relevant column to return the most interesting examples in the database.

Never query for all the columns from a specific table, only ask for the relevant columns given the question.

You have access to tools for interacting with the database. Only use the given tools. Only use the information returned by the tools to construct your final answer.

You MUST double check your query before executing it. If you get an error while executing a query, rewrite the query and try again.

DO NOT make any DML statements (INSERT, UPDATE, DELETE, DROP etc.) to the database.

You have access to the following tables: {table_names}

If you need to filter on a proper noun, you must ALWAYS first look up the filter value using the "search_proper_nouns" tool! Do not try to guess at the proper name – use this function to find similar ones.""".format(

```
    table_names=db.get_usable_table_names()
)
```

```
system_message = SystemMessage(content=system)
```

```
tools.append(retriever_tool)
```

```
agent = create_react_agent(llm, tools,
messages_modifier=system_message)
```

```
for s in agent.stream(
    {"messages": [HumanMessage(content="How many albums does
    alis in chain have?")]}
):
```

```
print(s)
print("-----")
```

```
{'agent': {'messages': [AIMessage(content='', additional_kwargs=
{'tool_calls': [{'id': 'call_r5UlSwHKQcWDHx6LrttnqE56',
'function': {'arguments': '{"query": "SELECT COUNT(*) AS
album_count FROM Album WHERE ArtistId IN (SELECT ArtistId FROM
Artist WHERE Name = \'Alice In Chains\')"}', 'name':
'sql_db_query'}], 'type': 'function'}}], response_metadata=
{'token_usage': {'completion_tokens': 40, 'prompt_tokens': 612,
'total_tokens': 652}, 'model_name': 'gpt-3.5-turbo',
'system_fingerprint': 'fp_3b956da36b', 'finish_reason':
'tool_calls', 'logprobs': None}, id='run-548353fd-b06c-45bf-
beab-46f81eb434df-0', tool_calls=[{'name': 'sql_db_query',
'args': {'query': "SELECT COUNT(*) AS album_count FROM Album
WHERE ArtistId IN (SELECT ArtistId FROM Artist WHERE Name =
'Alice In Chains')"}], 'id':
'call_r5UlSwHKQcWDHx6LrttnqE56'}]]]}
-----
{'action': {'messages': [ToolMessage(content='[(1,)]',
name='sql_db_query', id='093058a9-f013-4be1-8e7a-ed839b0c90cd',
tool_call_id='call_r5UlSwHKQcWDHx6LrttnqE56')]}]}
-----
{'agent': {'messages': [AIMessage(content='Alice In Chains has
11 albums.', response_metadata={'token_usage':
{'completion_tokens': 9, 'prompt_tokens': 665, 'total_tokens':
674}, 'model_name': 'gpt-3.5-turbo', 'system_fingerprint':
'fp_3b956da36b', 'finish_reason': 'stop', 'logprobs': None},
id='run-f804eaab-9812-4fb3-ae8b-280af8594ac6-0')]}]}
-----
```

As we can see, the agent used the `search_proper_nouns` tool in order to check how to correctly query the database for this specific artist.

 [Edit this page](#)