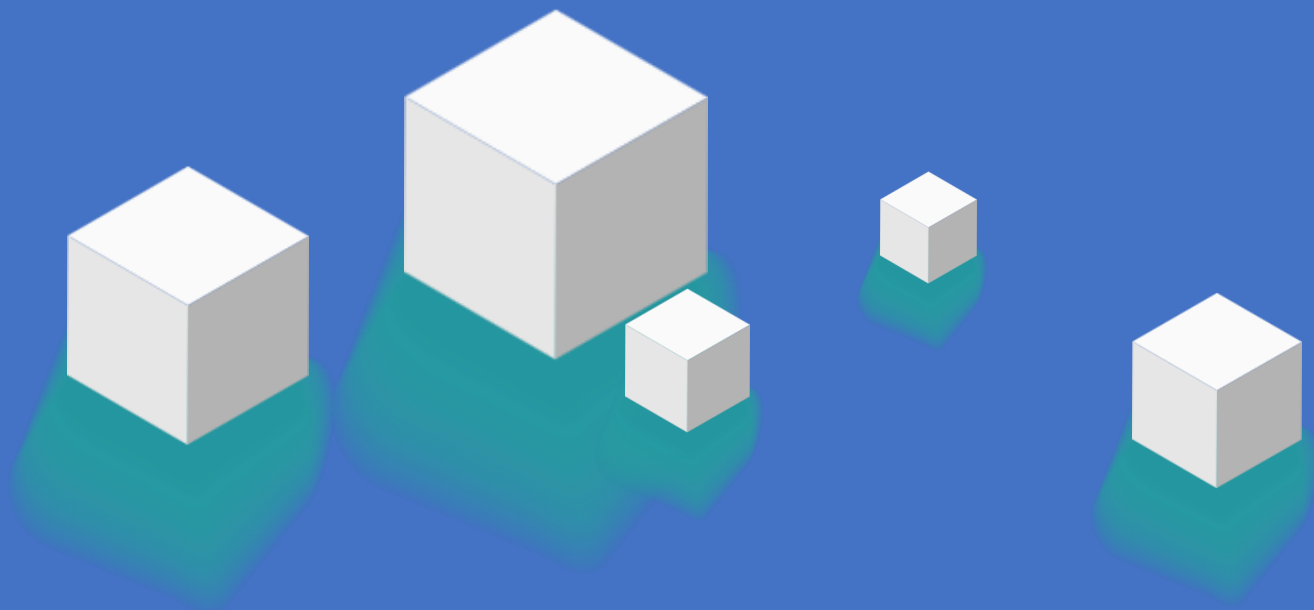


# 自助式数据报表开发 (基于Text2SQL)



# >> 今天的学习目标

---

## 自助式数据报表开发（基于Text2SQL）

- Text to SQL技术
- LLM模型选择
- 大模型API使用
- Function Call
- 搭建SQL Copilot

LangChain中的SQL Agent

自己编写（LLM + Prompt）

- CASE: 保险场景SQL Copilot实战

# CASE：数据查询

TO DO：小张是运营负责人员，要给老板汇报投放数据，现在渠道，统计维度很多，如何响应老板的灵活查询？

比如：社交媒体在哪天的投放费用是最高的？ 2025年2月份一共花了多少钱，带来了多少转化量？ .....

渠道名称	日期	投放费用	展示量	点击量	转化量	次日留存率	7日留存率	转化ROI(%)
社交媒体	2025/2/11	3793.97	29502	479	2	0.3819	0.2571	0.05
行业论坛	2025/2/4	753.85	7889	79	1	0.3618	0.2256	0.13
信息流广告	2024/11/29	1810.53	14437	313	3	0.3814	0.2255	0.17
信息流广告	2025/1/2	4924.54	60417	2248	26	0.3395	0.1707	0.53
短视频平台	2025/2/17	571.85	4861	181	2	0.2722	0.1814	0.35
社交媒体	2025/1/7	3481.35	36267	1155	8	0.154	0.1069	0.23
KOL合作	2024/12/26	4669.96	38226	1254	16	0.4689	0.3246	0.34
行业论坛	2025/1/4	3140.38	32009	2200	12	0.3919	0.2089	0.38
搜索引擎	2024/12/25	4169.58	50293	2054	45	0.4431	0.2533	1.08
.....	.....	.....	.....	.....	.....	.....	.....	.....

channel\_performance.csv

# Text to SQL技术

---

Text-to-SQL (文本转 SQL)

将自然语言问题自动转换为结构化的 SQL 查询语句，可以让用户更直观的与数据库进行交互。

Text-to-SQL的技术演变经历了3个阶段：

- **早期阶段：**依赖于人工编写的规则模板来匹配自然语言和SQL语句之间的对应关系。
- **机器学习阶段：**采用序列到序列模型等机器学习方法来学习自然语言和SQL之间的映射关系。
- **LLM阶段：**借助 LLM 强大的语言理解和代码生成能力，利用提示工程、微调等方法将 Text-to-SQL 性能提升到新的高度。

# Text to SQL技术

---

我们目前已处于LLM阶段，基于 LLM 的 Text-to-SQL 系统会包含以下几个步骤：

- **自然语言理解**: 分析用户输入的自然语言问题，理解其意图和语义。
- **模式链接**: 将问题中的实体与数据库模式中的表和列进行链接。
- **SQL 生成**: 根据理解的语义和模式链接结果，生成相应的 SQL 查询语句。
- **SQL 执行**: 在数据库上执行SQL查询，将结果返回给用户。

# LLM模型选择（闭源模型）

---

- GPT-o1/o3: o1模型开启了新的Scaling Law，更专注于推理阶段，在编程和Text to SQL中能力优于gpt-4o，同时mini模型速度更快，价格更低。

- Claude 3.7-sonnet: Anthropic公司于2025年2月发布，号称迄今为止最智能的模型，首款混合推理模型

Claude 3.7-sonnet 实现了两种思考方式的结合，既能提供接近即时的响应，也能展示分步骤的详细思考过程

- Claude 3.5-sonnet：2024年推出的模型，支持20万tokens上下文，性能超过GPT-4o，在Cursor中使用非常顺滑。
- Gemini 2.0：性能强悍，支持100万token上下文。
- Qwen-Turbo：支持100万token上下文，速度快，价格非常便宜。

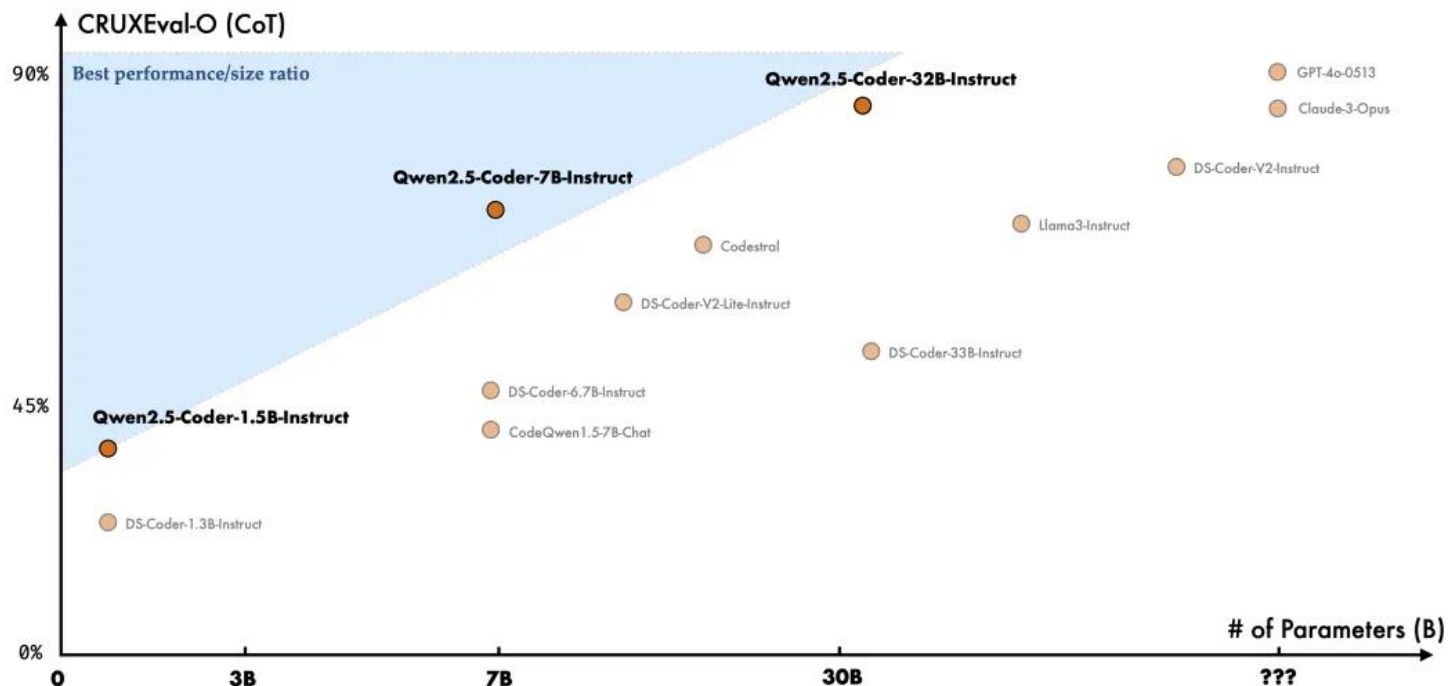
# LLM模型选择（开源模型）

---

- DeepSeek-V3: 在代码生成任务中表现出色，能够快速生成高质量的代码片段。它通过从 DeepSeek-R1 中蒸馏推理能力，显著提升了代码生成的准确性和效率
- DeepSeek-R1: R1 在代码生成和复杂逻辑推理方面表现卓越，特别是在处理复杂的编程任务和数学问题时，准确率更高。例如，在 Codeforces Elo 评分中，R1 达到 2029 分，超越 96.3% 的人类参赛者
- Qwen: Qwen系列从0.5B, 1.5B, 3B, 7B, 14B, 32B, 72B等多种尺寸，性能优于Llama3.1。

# LLM模型选择（代码大模型）

- Qwen-Coder：能力强，接近闭源一线大模型，其中Qwen2.5-Coder-32B能力与GPT-4o持平
- CodeGeeX：智谱开源的代码大模型，基于GLM底座，性能卓越，在vscode等编辑器中可以找到对应的插件。
- SQLCoder：专为 SQL 生成而设计的开源模型，但是维护更新慢。
- DeepSeek-Coder：在多种编程语言中与开源代码模型中实现了先进的性能



# 大模型API 使用

# API使用（情感分析）

TO DO：对用户观点评论进行情感分析，即正向、负向

- 1) 使用openai.Completion
- 2) 使用openai.ChatCompletion

针对提取的用户评论，可以进行批量化分析

	用户评论	情感
0	价格还可以，我是6768拿下的（用了一张500返20的券），第二天就涨回到6999了。送的正...	正向
1	非常好，有品质	正向
2	我是韶音忠实粉丝，从上一款AS800头戴旗舰到这一次OpenFit，从未让我失望，音质有了更...	正向
3	韶音的品质一直没问题。	正向
4	这款音效特别好 给你意想不到的音质。	正向
5	佩戴非常舒服，无感佩戴，随便运动不会掉	正向
6	预装的是Linux,不是xp，给客服打电话说不能换操作系统，要不就不保修了，哪有这样的道理	负向
7	牌子够老，够响亮，冲着牌子去的，结果让人很伤心！唉。。。。。。	负向
8	用了几天，结果系统崩溃了，到同方检测，发现30%坏道，已经退回换货了，不知道换来的如何	负向

商品评论观点.xlsx

# API使用 (ChatCompletion)

---

```
from openai import OpenAI
import os
client = OpenAI(api_key=os.environ.get("OPENAI_API_KEY"))
def get_chat_completion(prompt):
    completion = client.chat.completions.create(
        model="gpt-3.5-turbo",
        messages=[
            {"role": "system", "content": "你是一名舆情分析师，帮我判断产品口碑的正负向，回复请用一个词语：正向 或者 负向"},
            {"role": "user", "content": review}
        ]
    )
    return completion.choices[0].message
```

```
review = '这款音效特别好 给你意想不到的音质。'
```

```
response = get_chat_completion(review)
```

```
response.content
```

正向

# API使用 (dashscope)

```
import json
import os
import dashscope
from dashscope.api_entities.dashscope_response import Role
dashscope.api_key = "sk..."

# 封装模型响应函数
def get_response(messages):
    response = dashscope.Generation.call(
        model='qwen-turbo', # 大模型的名称
        messages=messages,
        result_format='message' # 将输出设置为message形式
    )
    return response
```

```
review = '这款音效特别好 给你意想不到的音质。'
messages=[
    {"role": "system", "content": "你是一名舆情分析师，帮我判断产品口碑的正负向，回复请用一个词语：正向 或者 负向"},
    {"role": "user", "content": review}
]

response = get_response(messages)
response.output.choices[0].message.content
```

正向

Thinking: 不仅是定期执行某个任务,  
还可以给大模型提供专属工具, 比如  
天气服务, 产品查询等

# API使用 (天气Function)

TO DO: 编写一个天气Function, 当LLM要查询天气的时候提供该服务, 比如当前不同城市的气温为:

北京: 35度

上海: 36度

深圳: 37度

天气均为晴天, 微风

1) 使用 model="gpt-3.5-turbo-0613"

2) 编写function get\_current\_weather

对于用户询问指定地点的天气, 可以获取该地当前天气



# API使用 (Function Call)

# 编写你的函数

```
def get_current_weather(location, unit="摄氏度"):
```

```
    # 获取指定地点的天气
```

```
    temperature = -1
```

```
    if location=='北京':
```

```
        temperature = 35
```

```
    if location=='上海':
```

```
        temperature = 36
```

```
    if location=='深圳':
```

```
        temperature = 37
```

```
    weather_info = {
```

```
        "location": location,
```

```
        "temperature": temperature,
```

```
        "unit": unit,
```

```
        "forecast": ["晴天", "微风"],
```

```
    }
```

```
    return json.dumps(weather_info)
```

# 使用function call进行QA

```
def run_conversation():
```

```
    query = "深圳的天气怎样"
```

```
    response = openai.ChatCompletion.create(
```

```
        model="gpt-3.5-turbo-0613",
```

```
        messages=[{"role": "user", "content": query}], # 定义message
```

```
        functions=[{
```

```
            "name": "get_current_weather",
```

```
            "description": "对于用户询问指定地点的天气，可以获取该地当前的天气",
```

```
            "parameters": {"type": "object",
```

```
                "properties": {
```

```
                    "location": {
```

```
                        "type": "string",
```

```
                        "description": "城市名称，比如北京",
```

```
                    }, "unit": {"type": "string", "enum": ["摄氏度", "华氏度"]},
```

```
                }, "required": ["location"],
```

```
            }]. function_call="auto".)
```

# API使用 (Function Call)

```
message = response["choices"][0]["message"]
# Step 2, 判断用户是否要call function
if message.get("function_call"):
    function_name = message["function_call"]["name"]

    # Step 3, 执行function call
    arguments = eval(message['function_call']['arguments'])
    function_response = get_current_weather(
        location=arguments.get("location"),
        unit=arguments.get("unit"),
    )

    # Step 4, 调用ChatCompletion (传入function_response)
    second_response = openai.ChatCompletion.create(
        model="gpt-3.5-turbo-0613",
```

```
messages=[
    {"role": "user", "content": query},
    message,
    {
        "role": "function",
        "name": function_name,
        "content": function_response,
    },
],
)
return second_response
```

```
result = run_conversation()
result['choices'][0]['message']['content']
```

运行结果: '深圳目前的天气是37°C, 晴天, 微风。'

# 搭建SQL Copilot

# SQL Copilot

## 方法1: SQLDatabase

采用LangChain框架

提供了sql chain, prompt, retriever, tools, agent, 让用户通过自然语言, 执行SQL查询

优点: 使用方便, 自动通过数据库连接, 获取数据库的metadata

不足: 执行不灵活, 需要多次判断哪个表适合  
复杂查询很难胜任, 对于复杂查询通过率低

## 方法2: 自己编写

本质是: LLM + RAG

选择适合的LLM, 比如: ChatModel: DeepSeek-V3,  
CodeModel: Qwen2.5-Coder, CodeGeeX2-6B

RAG, 可以分成: 向量数据库检索 + 固定文件 (比如本地数据表说明等)

优点: 重心在于RAG的提供上, 准确性高, 配置灵活

不足: 需要设置的条件规则多

# 方法1: SQLiteDatabaseToolkit使用

---

```
from langchain.agents import create_sql_agent
from langchain.agents.agent_toolkits import SQLiteDatabaseToolkit
from langchain.sql_database import SQLiteDatabase
from langchain.llms.openai import OpenAI
from langchain.agents import AgentExecutor
# 连接到数据库
db_user = "root"
db_password = "your_password"
db_host = "localhost:3306"
db_name = "wucai"
db = SQLiteDatabase.from_uri(f"mysql+pymysql://{db_user}:{db_password}@{db_host}/{db_name}")
```

通过SQLiteDatabase可以访问到数据库的Schema

# 方法1: SQLDatabaseToolkit使用

```
from langchain.chat_models import ChatOpenAI

llm = ChatOpenAI(model_name="deepseek-chat")

# 需要设置llm

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

agent_executor = create_sql_agent(
    llm=llm,
    toolkit=toolkit,
    verbose=True
)

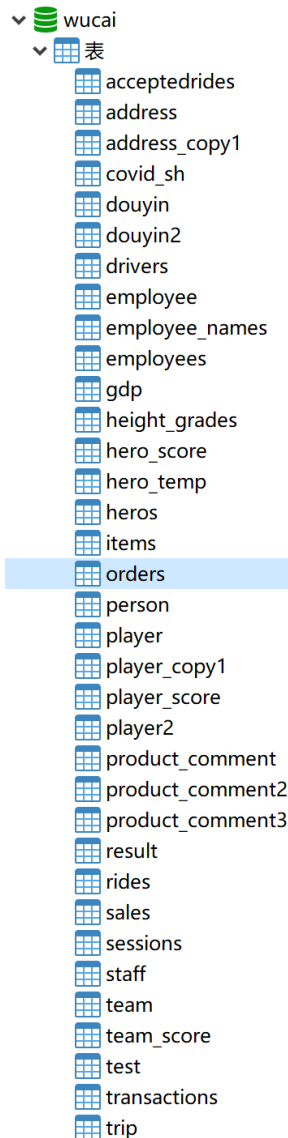
# Task: 描述数据表

agent_executor.run("描述与订单相关的表及其关系")
```

```
> Entering new AgentExecutor chain...
Action: sql_db_list_tables
Action Input: ""
Observation: acceptedrides, address, address_copy1, covid_sh, douyin, douyin2,
e_names, employees, gdp, height_grades, hero_score, hero_temp, heros, items, or
er2, player_copy1, player_score, product_comment, product_comment2, product_con
s, sessions, staff, team, team_score, test, transactions, trip, user, users, we
Thought: There are many tables in the database, so I should query the schema of
rs.
Action: sql_db_schema
Action Input: "orders"
Observation:
CREATE TABLE orders (
  order_id INTEGER,
  customer_id INTEGER,
  order_date DATE,
  item_id VARCHAR(30),
  quantity INTEGER
)DEFAULT CHARSET=utf8mb4 COLLATE utf8mb4_0900_ai_ci ENGINE=InnoDB

/*
3 rows from orders table:
order_id      customer_id  order_date  item_id quantity
1             1           2020-06-01  1         10
2             1           2020-06-08  2         10
3             2           2020-06-02  1         5
*/
```

# 方法1: SQLDatabaseToolkit使用



Thought: The tables related to orders are "orders". I can query the "orders" table to get information about orders.

Action: sql\_db\_query

Action Input: "SELECT \* FROM orders LIMIT 10"

Observation: [(1, 1, datetime.date(2020, 6, 1), '1', 10), (2, 1, datetime.date(2020, 6, 8), '2', 10), (3, 2, datetime.date(2020, 6, 2), '1', 5), (4, 3, datetime.date(2020, 6, 3), '3', 5), (5, 4, datetime.date(2020, 6, 4), '4', 1), (6, 4, datetime.date(2020, 6, 5), '5', 5), (7, 5, datetime.date(2020, 6, 5), '1', 10), (8, 5, datetime.date(2020, 6, 14), '4', 5), (9, 5, datetime.date(2020, 6, 21), '3', 5)]

Thought: The relevant tables related to orders are "orders". From the "orders" table, I obtained the following rows:

order_id	customer_id	order_date	item_id	quantity
1	1	2020-06-01	1	10
2	1	2020-06-08	2	10
3	2	2020-06-02	1	5

Final Answer: The table related to orders is "orders". The columns in the "orders" table are order\_id, customer\_id, order\_date, item\_id, and quantity.

> Finished chain.

'The table related to orders is "orders". The columns in the "orders" table are order\_id, customer\_id, order\_date, item\_id, and quantity.'

# 方法1: SQLDatabaseToolkit使用

# 这个任务, 实际上数据库中 没有HeroDetails表  
agent\_executor.run("描述HeroDetails表")

```
> Entering new AgentExecutor chain...
Action: sql_db_list_tables
Action Input: ""
Observation: acceptedrides, address, address_copy1, covid_sh, douyin, douyin2, drivers, employee,
yee_names, employees, gdp, height_grades, hero_score, hero_temp, heros, items, orders, person, p
player2, player_copy1, player_score, product_comment, product_comment2, product_comment3, result
s, sales, sessions, staff, team, team_score, test, transactions, trip, user, users, weather
Thought: The tables in the database are: acceptedrides, address, address_copy1, covid_sh, douyin,
2, drivers, employee, employee_names, employees, gdp, height_grades, hero_score, hero_temp, hero
s, orders, person, player, player2, player_copy1, player_score, product_comment, product_comment
uct_comment3, result, rides, sales, sessions, staff, team, team_score, test, transactions, trip,
users, weather.
I should query the schema of the HeroDetails table.
Action: sql_db_schema
Action Input: "HeroDetails"
Observation: Error: table_names {'HeroDetails'} not found in database
Thought:
```

如果没有找到对应的表, 会解析报错

OutputParseException

OutputParserException Traceback (most recent call last)

Cell In[6], line 2

```
1 # Task:
----> 2 agent_executor.run("描述HeroDetails表")
```

File ~\AppData\Roaming\Python\Python311\site-packages\langchain\chains\base.py:451, in Chain.run(self, callbacks, tags, metadata, \*args, \*\*kwargs)

```
449     if len(args) != 1:
450         raise ValueError("`run` supports only one positional argument.")
--> 451     return self(args[0], callbacks=callbacks, tags=tags, metadata=metadata)[
452         _output_key
453     ]
455 if kwargs and not args:
456     return self(kwargs, callbacks=callbacks, tags=tags, metadata=metadata)[
457         _output_key
458     ]
```

```
54         observation=MISSING_ACTION_AFTER_THOUGHT_ERROR_MESSAGE,
55         llm_output=text,
56         send_to_llm=True,
57     )
58 elif not re.search(
59     r"[\s]*Action\s*\d*\s*Input\s*\d*\s*: [\s]*(.*)", text, re.DOTALL
60 ):
61     raise OutputParserException(
62         f"Could not parse LLM output: `{text}`",
63         observation=MISSING_ACTION_INPUT_AFTER_ACTION_ERROR_MESSAGE,
64         llm_output=text,
65         send_to_llm=True,
66     )
```

OutputParserException: Could not parse LLM output: `The table "HeroDetails" does not exist in the database. I don't know the schema of the HeroDetails table.`

# 方法1：SQLDatabaseToolkit使用

# 这个任务，数据库中的表实际为 heros

agent\_executor.run("描述Hero表")

```
> Entering new AgentExecutor chain...
Action: sql_db_list_tables
Action Input: ""
Observation: acceptedrides, address, address_copy1, covid_sh, douyin, douyin2, drivers, employee, employe
e_names, employees, gdp, height_grades, hero_score, hero_temp, heros, items, orders, person, player, play
er2, player_copy1, player_score, product_comment, product_comment2, product_comment3, result, rides, sale
s, sessions, staff, team, team_score, test, transactions, trip, user, users, weather
Thought:The relevant table is "heros". I should query the schema of this table.
Action: sql_db_schema
Action Input: "heros"
Observation:
CREATE TABLE heros (
  id INTEGER NOT NULL AUTO_INCREMENT,
  name VARCHAR(255) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL,
  hp_max FLOAT,
  hp_growth FLOAT,
  hp_start FLOAT,
  mp_max FLOAT,
  mp_growth FLOAT,
  mp_start FLOAT,
  attack_max FLOAT,
  attack_growth FLOAT,
  attack_start FLOAT,
  defense_max FLOAT,
  defense_growth FLOAT,
  defense_start FLOAT,
  hp_5s_max FLOAT,
  hp_5s_growth FLOAT,
```

也可以会找到多张表，逐一进行尝试

```
/*
3 rows from heros table:
id      name      hp_max hp_growth      hp_start      mp_max mp_growth      mp_start      attack_max
attack_growth attack_start defense_max defense_growth defense_start hp_5s_max      hp_5s_growth
hp_5s_start mp_5s_max      mp_5s_growth mp_5s_start      attack_speed_max      attack_range      role
_main      role_assist      birthdate
10000 夏侯惇 7350.0 288.8 3307.0 1746.0 94.0 430.0 321.0 11.57 159.0 397.0 21.14 101.
0 98.0 3.357 51.0 37.0 1.571 15.0 0.0 近战 坦克 战士 2016-07-19
10001 钟无艳 7000.0 275.0 3150.0 1760.0 95.0 430.0 318.0 11.0 164.0 409.0 22.07 100.
0 92.0 3.143 48.0 37.0 1.571 15.0 0.0 近战 战士 坦克 None
10002 张飞 8341.0 329.4 3450.0 100.0 0.0 100.0 301.0 10.57 153.0 504.0 27.07 125.
0 115.0 4.143 57.0 5.0 0.0 5.0 0.0 近战 坦克 辅助 None
*/
Thought:The Hero table has the following columns:
- id
- name
- hp_max
- hp_growth
- hp_start
- mp_max
- mp_growth
- mp_start
- attack_max
- attack_growth
- attack_start
- defense_max
- defense_growth
- defense_start
- hp_5s_max
- hp_5s_growth
- hp_5s_start
- mp_5s_max
- mp_5s_growth
- mp_5s_start
- attack_speed_max
- attack_range
- role_main
- role_assist
- birthdate

Final Answer: The Hero table has the above columns.

> Finished chain.

'The Hero table has the above columns.'
```

# 方法1：SQLDatabaseToolkit使用

# 执行指定的任务：编写SQL => 查询结果

agent\_executor.run("找出英雄攻击力最高的前5个英雄")

```
> Entering new AgentExecutor chain...
Action: sql_db_list_tables
Action Input: ""
Observation: acceptedrides, address, address_copy1, covid_sh, douyin, douyin2, drivers, employee, employe
e_names, employees, gdp, height_grades, hero_score, hero_temp, heros, items, orders, person, player, play
er2, player_copy1, player_score, product_comment, product_comment2, product_comment3, result, rides, sale
s, sessions, staff, team, team_score, test, transactions, trip, user, users, weather
Thought:The tables that might contain information about heroes and their attack power are "heros" and "he
ro_score". I should query the schema of these tables to see what columns they have.
Action: sql_db_schema
Action Input: "heros, hero_score"
Observation:
CREATE TABLE hero_score (
    id INTEGER NOT NULL AUTO_INCREMENT,
    score INTEGER NOT NULL DEFAULT '0',
    name VARCHAR(20) CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci,
    PRIMARY KEY (id)
)DEFAULT CHARSET=utf8mb4 COLLATE utf8mb4_0900_ai_ci ENGINE=InnoDB

/*
3 rows from hero_score table:
id      score  name
1        80    张飞
2        95    关羽
3        92    刘备
*/
```

找到了数据库中两个相关的表： heros, hero\_score

分别进行查询，发现heros这个数据表可以找到问题的

答案，最终结果为：阿轲, 孙尚香, 百里守约, 虞姬, 黄忠

```
CREATE TABLE heros (
    id INTEGER NOT NULL AUTO_INCREMENT,
    name VARCHAR(255) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL,
    hp_max FLOAT,
    hp_growth FLOAT,
    hp_start FLOAT,
    mp_max FLOAT,
    mp_growth FLOAT,
    mp_start FLOAT,
    attack_max FLOAT,
    attack_growth FLOAT,
    attack_start FLOAT,
    defense_max FLOAT,
    defense_growth FLOAT,
    defense_start FLOAT,
    hp_5s_max FLOAT,
    hp_5s_growth FLOAT,
    hp_5s_start FLOAT,
    mp_5s_max FLOAT,
    mp_5s_growth FLOAT,
    mp_5s_start FLOAT,
    attack_speed max FLOAT,
    attack_range VARCHAR(255) CHARACTER SET utf8 COLLATE utf8_general_ci,
    role_main VARCHAR(255) CHARACTER SET utf8 COLLATE utf8_general_ci,
    role_assist VARCHAR(255) CHARACTER SET utf8 COLLATE utf8_general_ci,
    birthdate DATE,
    PRIMARY KEY (id)
)DEFAULT CHARSET=utf8 ROW_FORMAT=DYNAMIC ENGINE=InnoDB

/*
3 rows from heros table:
id      name      hp_max  hp_growth      hp_start      mp_max  mp_growth      mp_start      attack_max
attack_growth  attack_start  defense_max  defense_growth  defense_start  hp_5s_max      hp_5s_growth
hp_5s_start  mp_5s_max      mp_5s_growth  mp_5s_start      attack_speed_max      attack_range  role
_main  role_assist  birthdate
10000 夏侯惇 7350.0 288.8 3307.0 1746.0 94.0 430.0 321.0 11.57 159.0 397.0 21.14 101.
0 98.0 3.357 51.0 37.0 1.571 15.0 0.0 近战 坦克 战士 2016-07-19
10001 钟无艳 7000.0 275.0 3150.0 1760.0 95.0 430.0 318.0 11.0 164.0 409.0 22.07 100.
0 92.0 3.143 48.0 37.0 1.571 15.0 0.0 近战 战士 坦克 None
10002 张飞 8341.0 329.4 3450.0 100.0 0.0 100.0 301.0 10.57 153.0 504.0 27.07 125.
0 115.0 4.143 57.0 5.0 0.0 5.0 0.0 近战 坦克 辅助 None
*/
Thought:I can query the "heros" table to find the heroes with the highest attack power. I should order th
e results by the "attack_max" column in descending order and limit the results to 5.
Action: sql_db_query
Action Input: "SELECT name, attack_max FROM heros ORDER BY attack_max DESC LIMIT 5"
Observation: [('阿轲', 427.0), ('孙尚香', 411.0), ('百里守约', 410.0), ('虞姬', 407.0), ('黄忠', 403.0)]
Thought:The heroes with the highest attack power are "阿轲", "孙尚香", "百里守约", "虞姬", and "黄忠".
Final Answer: 阿轲, 孙尚香, 百里守约, 虞姬, 黄忠
```

> Finished chain.

'阿轲, 孙尚香, 百里守约, 虞姬, 黄忠'

# Summary (SQL+LLM)

SQL + LLM 使用:

- 通过SQLDatabase可以访问到数据库的Schema
- agent\_executor 作为SQL Agent, 可以执行用户的各种SQL需求 (通过自然语言 => 编写SQL => 查询结果返回)

如果数据库中没有找到对应的表, 会报

OutputParseException错误

如果有多张表, 会分别执行, 然后判断哪个数据表可以得到结果

```
from langchain.chat_models import ChatOpenAI
llm = ChatOpenAI(model_name="gpt-3.5-turbo")
# 需要设置llm
toolkit = SQLDatabaseToolkit(db=db, llm=llm)

agent_executor = create_sql_agent(
    llm=llm,
    toolkit=toolkit,
    verbose=True
)

# Task: 描述数据表
agent_executor.run("描述与订单相关的表及其关系")
```

Thinking: 直接使用SQL + LLM会有什么问题?

- 1) 多个相似的数据表 => 导致LangChain会尝试多次生成SQL
- 2) 用户Prompt太宽泛 => 生成的结果, 不是用户想要的



给Agent配备专有知识库, 在prompt中动态完善 和query相关的context

# SQL + 向量数据库 + LLM

SQL + 向量数据库 + LLM:

- 向量数据库可以提供领域知识, 当用户检索某个问题的时候 => 从向量数据库中找到相关的内容, 放到 prompt 中 => 提升 SQL 查询的相关性

RAG 技术 (Retrieval Augmented Generation)

- 在 prompt 中增加 few-shot examples
- 专门定制检索工具, 从向量数据库中检索到与用户 query 相近的知识

```
agent.run("How many employees do we have?")
```

```
> Entering new AgentExecutor chain...
```

```
Invoking: `sql_get_similar_examples` with `How many employees do we have?`
```

```
[Document(page_content='How many employees are there',  
metadata={'sql_query': 'SELECT COUNT(*) FROM "employee"'}),  
Document(page_content='Which employee has sold the most?',  
metadata={'sql_query': "SELECT e.FirstName || ' ' || e.LastName AS  
EmployeeName, SUM(i.Total) AS TotalSales\n      FROM Employee  
e\n      JOIN Customer c ON e.EmployeeId = c.SupportRepId\n      JOIN Invoice i ON c.CustomerId = i.CustomerId\n      GROUP BY  
e.EmployeeId\n      ORDER BY TotalSales DESC\n      LIMIT 1;"})]
```

```
Invoking: `sql_db_query` with `SELECT COUNT(*) FROM employee`
```

```
responded: {content}
```

```
[(8,)]We have 8 employees.
```

# SQL + 向量数据库 + LLM

Thinking: 除了对用户query, 补充领域知识外, 针对专门名词 (用户可以拼写错误的), 也可以进行纠正

```
`  
sql_agent("What is 'Francis Trembling's email address?")  
  
Invoking: `name_search` with `Francis Trembling`  
  
[Document(page_content='François Tremblay', metadata={}),  
Document(page_content='Edward Francis', metadata={}),  
Document(page_content='Frank Ralston', metadata={}),  
Document(page_content='Frank Harris', metadata={}),  
Document(page_content='N. Frances Street', metadata={})]  
Invoking: `sql_db_query_checker` with `SELECT Email FROM Customer  
WHERE FirstName = 'François' AND LastName = 'Tremblay' LIMIT 1`  
responded: {content}
```

```
SELECT Email FROM Customer WHERE FirstName = 'François' AND  
LastName = 'Tremblay' LIMIT 1  
  
Invoking: `sql_db_query` with `SELECT Email FROM Customer WHERE  
FirstName = 'François' AND LastName = 'Tremblay' LIMIT 1`  
  
[('ftremblay@gmail.com',)]The email address of 'François Tremblay' is  
'ftremblay@gmail.com'.  
  
> Finished chain.  
{'input': "What is 'Francis Trembling' email address?",  
'output': "The email address of 'François Tremblay' is  
'ftremblay@gmail.com'."}
```

# Summary (SQL+向量数据库+LLM)

---

SQL + 向量数据库 + LLM 使用:

- 如果能让LLM使用tool (可以按照某个顺序, 执行完这个再执行下一个), 比较有效的方式是写在prompt中, 而不是在tool description中进行定义

• 向量数据库的作用:

给Prompt提供更多的context, 用于LLM进行决策

Thinking: 使用向量数据库的时候, 有哪些可以优化的地方?

- 使用similarity threshold, 来决定retrieved examples的质量 (有些example和用户query关系不大)
- 优化few-shot examples的多样性, 让尽可能多的情况展示给prompt

# CASE：保险场景SQL Copilot实战

# 保险场景 SQL查询

数据表：

- 1) 客户信息表 (CustomerInfo) :
- 2) 保单信息表 (PolicyInfo)
- 3) 理赔信息表 (ClaimInfo)
- 4) 受益人信息表 (BeneficiaryInfo)
- 5) 代理人信息表 (AgentInfo)
- 6) 保险产品信息表 (ProductInfo)
- 7) 保险公司内部员工表 (EmployeeInfo)

<div><div>agentinfo</div><div>AgentID: bigint(0) Name: text Gender: text DateOfBirth: datetime(0) Address: text PhoneNumber: bigint(0) EmailAddress: text 更多 ( 4 列 ) ...</div></div>	<div><div>claiminfo</div><div>ClaimNumber: text PolicyNumber: text ClaimDate: datetime(0) ClaimType: text ClaimAmount: bigint(0) ClaimStatus: text ClaimDescription: text 更多 ( 8 列 ) ...</div></div>	<div><div>productinfo</div><div>ProductID: bigint(0) ProductName: text ProductType: text CoverageRange: text CoverageTerm: text Premium: bigint(0) PaymentFrequency: text 更多 ( 7 列 ) ...</div></div>
<div><div>beneficiaryinfo</div><div>BeneficiaryID: bigint(0) Name: text Gender: text DateOfBirth: text Nationality: text Address: text PhoneNumber: bigint(0) EmailAddress: text</div></div>	<div><div>policyinfo</div><div>PolicyNumber: text CustomerID: text ProductID: text PolicyStatus: text Beneficiary: text Relationship: text PolicyStartDate: datetime(...) 更多 ( 5 列 ) ...</div></div>	<div><div>customerinfo</div><div>CustomerID: bigint(0) Name: text Gender: text DateOfBirth: text IDNumber: text Address: text PhoneNumber: bigint(0) 更多 ( 8 列 ) ...</div></div>
<div><div>employeeinfo</div><div>EmployeeID: bigint(0) Name: text Gender: text DateOfBirth: datetime(0) Address: text PhoneNumber: text EmailAddress: text 更多 ( 8 列 ) ...</div></div>		

# 保险场景 SQL查询

---

## TO DO：针对保险场景进行SQL查询

- 1、获取所有客户的姓名和联系电话。
- 2、找出所有已婚客户的保单。
- 3、查询所有未支付保费的保单号和客户姓名。
- 4、找出所有理赔金额大于10000元的理赔记录，并列出相关客户的姓名和联系电话。
- 5、查找代理人的姓名和执照到期日期，按照执照到期日期升序排序。
- 6、获取所有保险产品的产品名称和保费，按照保费降序排序。

7、查询所有在特定销售区域工作的员工的姓名和职位。

8、找出所有年龄在30岁以下的客户，并列出其客户ID、姓名和出生日期。

9、查找所有已审核但尚未支付的理赔记录，包括理赔号、审核人和审核日期。

10、获取每个产品类型下的平均保费，以及该产品类型下的产品数量。

Thinking：如何让LLM进行SQL查询，准确率如何？

方法1：SQLDatabaseToolkit

方法2：自己编写

# SQL数据表

**客户信息表 (CustomerInfo) :** 客户ID (CustomerID)、客户姓名 (Name)、性别 (Gender)、出生日期 (DateOfBirth)、身份证号码 (IDNumber)、联系地址 (Address)、联系电话 (PhoneNumber)、电子邮件地址 (EmailAddress)、婚姻状况 (MaritalStatus)、职业 (Occupation)、健康状况 (HealthStatus)、客户注册日期 (RegistrationDate)、客户类型 (CustomerType)、客户来源 (SourceOfCustomer)、客户状态 (CustomerStatus)

**保单信息表 (PolicyInfo) :** 保单号 (PolicyNumber)、客户ID (CustomerID)、保险产品ID (ProductID)、保单状态 (PolicyStatus)、受益人 (Beneficiary)、受益人关系 (Relationship)、投保日期 (PolicyStartDate)、终止日期 (PolicyEndDate)、保费支付状态 (PremiumPaymentStatus)、保费支付日期 (PaymentDate)、保费支付方式 (PaymentMethod)、代理人ID (AgentID)

**理赔信息表 (ClaimInfo) :** 理赔号 (ClaimNumber)、保单号 (PolicyNumber)、理赔日期 (ClaimDate)、理赔类型 (ClaimType)、理赔金额 (ClaimAmount)、理赔状态 (ClaimStatus)、理赔描述 (ClaimDescription)、受益人ID (BeneficiaryID)、医疗记录 (MedicalRecords)、事故报告 (AccidentReport)、审核人 (ClaimHandler)、审核日期 (ReviewDate)、支付方式 (PaymentMethod)、支付日期 (PaymentDate)、拒赔原因 (DenialReason)

# SQL数据表

**受益人信息表 (BeneficiaryInfo)** : 受益人ID (BeneficiaryID)、姓名 (Name)、性别 (Gender)、出生日期 (DateOfBirth)、国籍 (Nationality)、联系地址 (Address)、电话号码 (PhoneNumber)、电子邮件 (EmailAddress)

**代理人信息表 (AgentInfo)** : 代理人ID (AgentID)、姓名 (Name)、性别 (Gender)、出生日期 (DateOfBirth)、联系地址 (Address)、电话号码 (PhoneNumber)、电子邮件 (EmailAddress)、证书号码 (CertificateNumber)、执照发放日期 (LicenseIssueDate)、执照到期日期 (LicenseExpirationDate)、佣金结构 (CommissionStructure)

**保险产品信息表 (ProductInfo)** : 产品ID (ProductID)、产品名称 (ProductName)、产品类型 (ProductType)、保险金额范围 (CoverageRange)、保险期限 (CoverageTerm)、保费 (Premium)、缴费频率 (PaymentFrequency)、产品特性 (ProductFeatures)、投保年龄限制 (AgeLimit)、保费计算方式 (PremiumCalculation)、理赔流程 (ClaimsProcess)、投保要求 (UnderwritingRequirements)、销售区域 (SalesRegion)、产品状态 (ProductStatus)

**保险公司内部员工表 (EmployeeInfo)** : 员工ID (EmployeeID)、姓名 (Name)、性别 (Gender)、出生日期 (DateOfBirth)、联系地址 (Address)、电话号码 (PhoneNumber)、电子邮件 (EmailAddress)、入职日期 (HireDate)、职位 (Position)、部门 (Department)、工资 (Salary)、工作地点 (Location)、上级主管 (Supervisor)、员工类型 (EmployeeType)、员工状态 (EmployeeStatus)

# SQL数据表

---

CREATE TABLE CustomerInfo (

CustomerID INT PRIMARY KEY,

Name VARCHAR(50),

Gender VARCHAR(10),

DateOfBirth DATE,

IDNumber VARCHAR(18),

Address VARCHAR(100),

PhoneNumber VARCHAR(20),

EmailAddress VARCHAR(50),

MaritalStatus VARCHAR(20),

Occupation VARCHAR(50),

HealthStatus VARCHAR(20),

RegistrationDate DATE,

CustomerType VARCHAR(20),

SourceOfCustomer VARCHAR(50),

CustomerStatus VARCHAR(20)

CREATE TABLE PolicyInfo (

PolicyNumber INT PRIMARY KEY,

CustomerID INT,

ProductID INT,

PolicyStatus VARCHAR(20),

Beneficiary VARCHAR(50),

Relationship VARCHAR(20),

PolicyStartDate DATE,

PolicyEndDate DATE,

PremiumPaymentStatus VARCHAR(20),

PaymentDate DATE,

PaymentMethod VARCHAR(20),

AgentID INT

);

...

# 保险场景 SQL查询

1、获取所有客户的姓名和联系电话。

```
SELECT Name, PhoneNumber  
FROM CustomerInfo
```

2、找出所有已婚客户的保单。

```
SELECT  
    PolicyInfo.PolicyNumber,  
    PolicyInfo.CustomerID,  
    PolicyInfo.ProductID,  
    PolicyInfo.PolicyStatus,  
    PolicyInfo.Beneficiary,  
    PolicyInfo.Relationship,  
    PolicyInfo.PolicyStartDate,  
    PolicyInfo.PolicyEndDate,
```

```
    PolicyInfo.PremiumPaymentStatus,  
    PolicyInfo.PaymentDate,  
    PolicyInfo.PaymentMethod,  
    PolicyInfo.AgentID  
FROM  
    PolicyInfo  
WHERE  
    PolicyInfo.CustomerID IN (  
        SELECT  
            CustomerInfo.CustomerID  
        FROM  
            CustomerInfo  
        WHERE  
            CustomerInfo.MaritalStatus = 'Married'  
    )
```

# 保险场景 SQL查询

3、查询所有未支付保费的保单号和客户姓名。

```
SELECT
    PolicyInfo.PolicyNumber,
    CustomerInfo.Name
FROM
    PolicyInfo
    LEFT JOIN
    CustomerInfo ON PolicyInfo.CustomerID =
CustomerInfo.CustomerID
WHERE
    PolicyInfo.PremiumPaymentStatus = 'Not Paid'
```

4、找出所有理赔金额大于10000元的理赔记录，并列出相关客户的姓名和联系电话。

```
SELECT
    CustomerInfo.Name,
    CustomerInfo.PhoneNumber
FROM
    CustomerInfo
    JOIN
    PolicyInfo ON CustomerInfo.CustomerID =
PolicyInfo.CustomerID
    JOIN
    ClaimInfo ON PolicyInfo.PolicyNumber =
ClaimInfo.PolicyNumber
WHERE
    ClaimInfo.ClaimAmount > 10000;
```

# 保险场景 SQL查询

5、查找代理人的姓名和执照到期日期，按照执照到期日期升序排序。

```
SELECT Name, LicenseExpirationDate
FROM AgentInfo
ORDER BY LicenseExpirationDate;
```

6、获取所有保险产品产品名称和保费，按照保费降序排序。

```
SELECT ProductName, Premium
FROM ProductInfo
ORDER BY Premium DESC;
```

7、查询所有在特定销售区域工作的员工的姓名和职位。

```
SELECT Name, Position
FROM EmployeeInfo
WHERE EmployeeID IN (
    SELECT EmployeeID
    FROM EmployeeInfo
    WHERE Location = 'Sales Region 1'
);
```

# 保险场景 SQL查询

8、找出所有年龄在30岁以下的客户，并列出其客户ID、姓名和出生日期。

```
SELECT CustomerInfo.CustomerID, CustomerInfo.Name,  
CustomerInfo.DateOfBirth  
FROM CustomerInfo  
WHERE CustomerInfo.DateOfBirth < '1990-01-01'
```

9、查找所有已审核但尚未支付的理赔记录，包括理赔号、审核人和审核日期。

```
SELECT  
    ClaimInfo.ClaimNumber,  
    ClaimInfo.ClaimDate,  
    ClaimInfo.ClaimType,  
    ClaimInfo.ClaimAmount,
```

```
    ClaimInfo.ClaimStatus,  
    ClaimInfo.ClaimDescription,  
    ClaimInfo.BeneficiaryID,  
    ClaimInfo.MedicalRecords,  
    ClaimInfo.AccidentReport,  
    ClaimInfo.ClaimHandler,  
    ClaimInfo.ReviewDate,  
    ClaimInfo.PaymentMethod,  
    ClaimInfo.PaymentDate,  
    ClaimInfo.DenialReason  
FROM  
    ClaimInfo  
WHERE  
    ClaimInfo.ClaimStatus = 'Pending'
```

# 保险场景 SQL查询

---

10、获取每个产品类型下的平均保费，以及该产品类型下的产品数量。

```
SELECT ProductType, AVG(Premium) AS AveragePremium,  
COUNT(PolicyNumber) AS PolicyCount  
FROM PolicyInfo AS pi  
JOIN ProductInfo AS pi2  
ON pi.ProductID = pi2.ProductID  
GROUP BY ProductType;
```

# Prompt works!

Thinking: 以下哪种prompt可以写出更好的SQL?

```
prompt = f"""# language: SQL
/*
{query}你需要先判断需要哪个数据表和字段,
然后基于它们编写SQL。数据库中有以下数
据表:
=====
{table_description}
*/
# {query}"""
```

写法1

```
prompt = f"""
-- language: SQL
/*{query}
以下是数据表
=====
{table_description}
=====
编写一条SQL: {query}
*/"""
```

写法2

```
prompt = f"""-- language: SQL
### Question: {query}
### Input: {create_sql}
### Response:
Here is the SQL query I have generated to
answer the question `{query}`:
```sql
"""
```

写法3

table\_description是数据表的中文描述, create\_sql 是数据表的建表语句, query 是用户查询的问题

# Prompt works!

---

Prompt写法很重要

- 1) 说明语言类型, -- language: SQL
- 2) 将SQL建表语句放到SQL prompt中, 因为大语言是通过SQL建表语句进行识别的
- 3) SQL编写用 ``sql, 放到prompt最后

Prompt中的首尾很重要

```
prompt = f"""-- language: SQL
### Question: {query}
### Input: {create_sql}
### Response:

Here is the SQL query I have generated to
answer the question `{query}`:




``sql
"""
```

# 学员讨论和呈现



在你的工作中，都有哪些SQL查询的场景？（对应的数据表、SQL查询语句，LLM能否完成，是否有临时SQL的需求）

No	业务场景	价值性	可行性



Thank You  
Using data to solve problems