

Homework 4: Text-to-SQL

1. Auto-Generating SQL Schema

Approach:

- Use a python script to convert JSON schema into SQL schema.

Implementation:

The main process of the script is:

1. Parses a sample JSON block response.
2. Extracts keys and determines types.
3. Generates SQL schema dynamically.

I used the below code to auto generate the SQL schema from the JSON object:

```
1  import json
2  from typing import List, Any
3
4  def infer_sql_type(value: Any) -> str:
5      """
6      Infer SQL data type from a Python value with improved type mapping.
7
8      Args:
9          value: Any Python value to analyze
10
11      Returns:
12          str: Corresponding SQL data type
13      """
14      if value is None:
15          return "TEXT"
16      elif isinstance(value, bool):
17          return "BOOLEAN"
18      elif isinstance(value, int):
19          return "INTEGER"
20      elif isinstance(value, float):
21          return "REAL"
22      elif isinstance(value, (list, dict)):
23          return "JSON"
24      elif isinstance(value, str):
25          if len(value) > 1000:
```

```

26         return "TEXT"
27     return "VARCHAR(255)"
28 else:
29     return "TEXT"
30
31 def sanitize_identifier(name: str) -> str:
32     """
33     Sanitize table and column names to be SQL-safe.
34
35     Args:
36         name: Raw identifier name
37
38     Returns:
39         str: Sanitized identifier name
40     """
41     sanitized = ''.join(c if c.isalnum() else '_' for c in name)
42     if sanitized[0].isdigit():
43         sanitized = f"t_{sanitized}"
44     return sanitized.lower()
45
46 def generate_schema(table_name: str, data: Any, parent_table: str = None) -> List[str]:
47     """
48     Recursively generate SQL schema from data.
49
50     Args:
51         table_name: Name of the table to create
52         data: Data to analyze for schema generation
53         parent_table: Name of the parent table if this is a nested structure
54
55     Returns:
56         List[str]: Tables DDL statements with inline foreign key constraints
57     """
58     table_name = sanitize_identifier(table_name)
59     tables = []
60     columns = []
61
62     columns.append("id INTEGER PRIMARY KEY AUTOINCREMENT")
63     if parent_table:
64         parent_fk = f"{sanitize_identifier(parent_table)}_id INTEGER"
65         columns.append(f"{parent_fk} REFERENCES {parent_table}(id)")
66

```

```

67     if isinstance(data, dict):
68         for key, value in data.items():
69             col_name = sanitize_identifier(key)
70
71             if isinstance(value, dict):
72                 if value: # Non-empty dictionary
73                     nested_table_name = f"{table_name}_{col_name}"
74                     nested_tables = generate_schema(nested_table_name,
value, table_name)
75                     tables.extend(nested_tables)
76                     columns.append(f"{col_name}_id INTEGER REFERENCES
{nested_table_name}(id)")
77                 else: # Empty dictionary
78                     sql_type = infer_sql_type(value)
79                     nullable = "NOT NULL" if value is not None else "NU
LL"
80                     columns.append(f"{col_name} {sql_type} {nullable}")
81             else:
82                 sql_type = infer_sql_type(value)
83                 nullable = "NOT NULL" if value is not None else "NULL"
84                 columns.append(f"{col_name} {sql_type} {nullable}")
85
86     columns_sql = ",\n    ".join(columns)
87     create_table = f"""CREATE TABLE IF NOT EXISTS {table_name} (
88         {columns_sql}
89     );"""
90     tables.insert(0, create_table)
91     return tables
92
93 def generate_sql_schema(data: Any, base_table_name: str) -> str:
94     """
95     Generate complete SQL schema including tables with inline constrain
ts.
96
97     Args:
98         data: Data to analyze for schema generation
99         base_table_name: Name of the root table
100
101     Returns:
102         str: Complete SQL schema
103     """
104     schema = ["PRAGMA foreign_keys = ON;", ""]
105     schema.append("-- Table Definitions")
106     tables = generate_schema(base_table_name, data)

```

```

107     schema.extend(tables)
108     return "\n\n".join(schema) + "\n"
109
110 # Example usage
111 if __name__ == "__main__":
112     # Read JSON file
113     with open("/home/tourist/neu/INF07500-cryptocurrency/hw3/block_data/block_0.json", "r") as f:
114         json_obj = json.load(f)
115
116     # Generate schema
117     sql_schema = generate_sql_schema(json_obj["result"], "block")
118
119     # Write to file
120     with open("/home/tourist/neu/INF07500-cryptocurrency/hw4/schema.sql", "w") as f:
121         f.write(sql_schema)

```

SQL Schema:

The generated SQL schema for the given JSON data would look like:

```

1  CREATE TABLE IF NOT EXISTS block (
2      id INTEGER PRIMARY KEY AUTOINCREMENT,
3      hash VARCHAR(255) NOT NULL,
4      confirmations INTEGER NOT NULL,
5      height INTEGER NOT NULL,
6      version INTEGER NOT NULL,
7      versionhex VARCHAR(255) NOT NULL,
8      merkleroot VARCHAR(255) NOT NULL,
9      time INTEGER NOT NULL,
10     mediantime INTEGER NOT NULL,
11     nonce INTEGER NOT NULL,
12     bits VARCHAR(255) NOT NULL,
13     difficulty REAL NOT NULL,
14     chainwork VARCHAR(255) NOT NULL,
15     ntx INTEGER NOT NULL,
16     previousblockhash VARCHAR(255) NOT NULL,
17     nextblockhash VARCHAR(255) NOT NULL,
18     strippedsize INTEGER NOT NULL,
19     size INTEGER NOT NULL,
20     weight INTEGER NOT NULL,
21     tx JSON NOT NULL
22

```

```
);
```

2. Keeping Database Updated

Approach:

- Write a program that calls `getblocks` RPC periodically.
- Extracts block and transaction data.
- Converts JSON into SQL INSERT statements.
- Ensures data consistency.

Implementation:

```
1  import modal
2  from modal import App, Volume, Secret
3  import os
4  import requests
5  import sqlite3
6  import json
7  import time
8  from typing import Dict, Any
9
10 app = App(name="chongchen-bitcoin-explorer") # Use modal.App
11
12 # Define the volume and Docker image
13 volume = Volume.from_name("chongchen-bitcoin-data", create_if_missing=True)
14 bitcoin_image = modal.Image.debian_slim().pip_install("requests")
15
16 class BitcoinRPC:
17     """Handles RPC communication with Bitcoin node via Chainstack"""
18     def __init__(self):
19         self.rpc_username = os.environ["RPC_USERNAME"]
20         self.rpc_password = os.environ["RPC_PASSWORD"]
21         self.rpc_host = os.environ["RPC_HOST"]
22         self.rpc_port = os.environ["RPC_PORT"]
23         self.rpc_path = os.environ["RPC_PATH"]
24         self.rpc_endpoint = f"https://{self.rpc_host}:{self.rpc_port}/{self.rpc_path}"
25         self.auth = (self.rpc_username, self.rpc_password)
26
27     def make_rpc_call(self, method: str, params: list) -> Dict[str, Any]
```

```

28 y]:
29     """Execute JSON-RPC call"""
30     payload = {
31         "jsonrpc": "2.0",
32         "method": method,
33         "params": params,
34         "id": 1
35     }
36     try:
37         response = requests.post(
38             self.rpc_endpoint,
39             auth=self.auth,
40             json=payload,
41             headers={'Content-Type': 'application/json'},
42             timeout=10
43         )
44         response.raise_for_status()
45         return response.json()
46     except Exception as e:
47         print(f"RPC Error: {e}")
48         raise
49
50     def get_block_count(self) -> int:
51         """Fetch current blockchain height"""
52         resp = self.make_rpc_call("getblockcount", [])
53         return resp["result"]
54
55     def get_block_hash(self, height: int) -> str:
56         """Get block hash by height"""
57         resp = self.make_rpc_call("getblockhash", [height])
58         return resp["result"]
59
60     def get_block(self, block_hash: str) -> Dict:
61         """Retrieve block data with transactions"""
62         resp = self.make_rpc_call("getblock", [block_hash, 2])
63         return resp["result"]
64
65     def get_db_connection():
66         """Connect to SQLite database in Modal Volume"""
67         return sqlite3.connect('/data/bitcoin.db')
68
69     def init_db():
70         """Initialize database schema if not exists"""
71         with get_db_connection() as conn:

```

```

72         conn.execute("""
73             CREATE TABLE IF NOT EXISTS block (
74                 id INTEGER PRIMARY KEY AUTOINCREMENT,
75                 hash VARCHAR(255) NOT NULL,
76                 confirmations INTEGER NOT NULL,
77                 height INTEGER NOT NULL,
78                 version INTEGER NOT NULL,
79                 versionHex VARCHAR(255) NOT NULL,
80                 merklroot VARCHAR(255) NOT NULL,
81                 time INTEGER NOT NULL,
82                 mediantime INTEGER NOT NULL,
83                 nonce INTEGER NOT NULL,
84                 bits VARCHAR(255) NOT NULL,
85                 difficulty REAL NOT NULL,
86                 chainwork VARCHAR(255) NOT NULL,
87                 nTx INTEGER NOT NULL,
88                 previousblockhash VARCHAR(255) NOT NULL,
89                 nextblockhash VARCHAR(255) NOT NULL,
90                 strippedsize INTEGER NOT NULL,
91                 size INTEGER NOT NULL,
92                 weight INTEGER NOT NULL,
93                 tx JSON NOT NULL
94             );
95         """)
96         conn.commit()
97
98     def get_max_height() -> int:
99         """Get the highest block height from the database"""
100         with get_db_connection() as conn:
101             cursor = conn.cursor()
102             cursor.execute("SELECT MAX(height) AS max_height FROM block")
103             row = cursor.fetchone()
104             return row[0] if row[0] is not None else -1
105
106     def save_block(block_data: Dict):
107         """Save block to database and Volume"""
108         # Insert into SQLite
109         with get_db_connection() as conn:
110             cursor = conn.cursor()
111             cursor.execute("""
112                 INSERT INTO block (
113                     hash, confirmations, height, version, versionHex, merkl
eroot,
114                     time, mediantime, nonce, bits, difficulty, chainwork, n

```

```

114 Tx,
           previousblockhash, nextblockhash, strippedsize, size, w
115 eight, tx
           ) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,
116 ?, ?, ?)
117     """ , (
118         block_data['hash'],
119         block_data.get('confirmations', 0),
120         block_data['height'],
121         block_data['version'],
122         block_data['versionHex'],
123         block_data['merkleroot'],
124         block_data['time'],
125         block_data.get('mediantime', block_data['time']),
126         block_data['nonce'],
127         block_data['bits'],
128         block_data['difficulty'],
129         block_data['chainwork'],
130         block_data['nTx'],
131         block_data.get('previousblockhash', ''),
132         block_data.get('nextblockhash', ''),
133         block_data['strippedsize'],
134         block_data['size'],
135         block_data['weight'],
136         json.dumps(block_data['tx'])
137     ))
138     conn.commit()
139
140     # Save JSON to Volume
141     # block_dir = "/data/blocks"
142     # os.makedirs(block_dir, exist_ok=True)
143     # with open(f"{block_dir}/block_{block_data['height']}.json", 'w')
144 as f:
145     #     json.dump(block_data, f)
146
147 @app.function(
148     volumes={"/data": volume},
149     image=bitcoin_image,
150     secrets=[Secret.from_name("chongchen-bitcoin-chainstack")],
151     timeout=86400 # Extend timeout for long syncing
152 )
153 def sync_blocks():
154     """Main function to sync blocks continuously"""
155     init_db()

```



```

155     rpc = BitcoinRPC()
156
157     while True:
158         current_height = rpc.get_block_count()
159         max_synced = get_max_height()
160
161         if max_synced >= current_height:
162             print("All blocks synced. Sleeping for 10 minutes.")
163             time.sleep(600)
164             continue
165
166         print(f"Syncing blocks {max_synced + 1} to {current_height}")
167         for height in range(max_synced + 1, current_height + 1):
168             try:
169                 block_hash = rpc.get_block_hash(height)
170                 block_data = rpc.get_block(block_hash)
171                 save_block(block_data)
172                 print(f"Block {height} synced")
173             except Exception as e:
174                 print(f"Failed to sync block {height}: {e}")
175                 break # Retry from current height on next iteration
176
177 if __name__ == "__main__":
178     with app.run():
179         sync_blocks.call()

```

The program synchronizes Bitcoin blockchain data using RPC calls, ensuring the database remains up-to-date and consistent. It fetches the latest block height with:

```

1 | current_height = rpc.get_block_count()

```

Then, it checks the highest stored block:

```

1 | max_synced = get_max_height()

```

If new blocks exist, it retrieves and stores them using:

```

1 | block_hash = rpc.get_block_hash(height)
2 | block_data = rpc.get_block(block_hash)
3 | save_block(block_data)

```

The process runs every few minutes, sleeping if no new blocks are found:

```

1 | time.sleep(600)

```

3. Natural Language to SQL Queries

Steps:

1. Extract SQL schema dynamically from the SQLite database.
2. Construct a prompt for OpenAI API.
3. Generate SQL queries based on natural language input.

Python Implementation:

```
1  import modal
2  import sqlite3
3  import os
4  from openai import OpenAI
5  from datetime import datetime
6
7  app = modal.App("bitcoin-sql-qa")
8  volume = modal.Volume.from_name("chongchen-bitcoin-data", create_if_missing=True)
9  image = modal.Image.debian_slim().pip_install("openai")
10
11  SYSTEM_PROMPT = """You are a SQL developer that is expert in Bitcoin and you answer natural \
12      language questions about the bitcoind database in a sqlite database. \
13      You always only respond with SQL statements that are correct."""
14
15  def get_schema(conn):
16      """Extract schema from SQLite database."""
17      cursor = conn.cursor()
18      cursor.execute("SELECT sql FROM sqlite_master WHERE type IN ('table', 'view') AND sql IS NOT NULL")
19      schemas = cursor.fetchall()
20      return '\n'.join([schema[0] for schema in schemas])
21
22  def execute_sql(conn, sql):
23      """Execute SQL query and return results or error."""
24      try:
25          cursor = conn.cursor()
26          cursor.execute(sql)
```

```

27         result = cursor.fetchall()
28         return result, None
29     except sqlite3.Error as e:
30         return None, str(e)
31
32 def log_qa_history(volume, question, sql, result, error):
33     """Log QA history to a file in the Modal Volume."""
34     log_dir = "/data/qa_history"
35     if not os.path.exists(log_dir):
36         os.makedirs(log_dir, exist_ok=True)
37
38     timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
39     log_file = os.path.join(log_dir, f"{timestamp}.txt")
40
41     log_content = f"Question: {question}\nGenerated SQL: {sql}\n"
42     if error:
43         log_content += f"Error: {error}\n"
44     else:
45         log_content += f"Result: {result}\n"
46
47     with open(log_file, 'w') as f:
48         f.write(log_content)
49
50     volume.commit() # Persist changes to the volume
51
52 @app.function(
53     image=image,
54     volumes={"/data": volume},
55     secrets=[modal.Secret.from_name("chongchen-llm-api-key")]
56 )
57 def answer_question(question: str, db_path: str):
58     """Main function to answer natural language questions using the SQL
59     ite database."""
60     # Connect to the database
61     conn = sqlite3.connect(db_path)
62
63     # Extract schema and prepare user prompt
64     schema = get_schema(conn)
65     user_prompt = f"Database schema:\n{schema}\n\nQuestion: {question}"
66
67     # Generate SQL using OpenAI API
68     llm_api_key = os.environ["DMX_API"]
69     client = OpenAI(
70         base_url="https://www.dmxapi.com/v1",

```

```

70         api_key=llm_api_key
71     )
72
73     response = client.chat.completions.create(
74         model="grok-3",
75         messages=[
76             {"role": "system", "content": SYSTEM_PROMPT},
77             {"role": "user", "content": user_prompt}
78         ],
79         temperature=0.2,
80         max_tokens=500
81     )
82     generated_sql = response.choices[0].message.content.strip()
83
84     # Execute the generated SQL
85     result, error = execute_sql(conn, generated_sql)
86     conn.close()
87
88     # Log the interaction
89     log_qa_history(volume, question, generated_sql, result, error)
90
91     return {"result": result, "error": error}
92
93 # Local testing entry point
94 if __name__ == "__main__":
95     import argparse
96     parser = argparse.ArgumentParser(description="Answer natural language questions about a Bitcoin SQLite database.")
97     parser.add_argument("question", type=str, help="Natural language question")
98     parser.add_argument("db_path", type=str, help="Absolute path to the SQLite database file")
99     args = parser.parse_args()
100
101     with app.run():
102         answer = answer_question.remote(args.question, args.db_path)
103         if answer["error"]:
104             print(f"Error: {answer['error']}")
105         else:
106             print(f"Result: {answer['result']}")

```

Key Points:

1. Inputs

- **Natural language question**
- **SQLite database path**

2. Prompt Generation

- **System prompt** defines the task.
- **Database schema** is extracted dynamically using `get_schema(conn)`.
- **User query and schema** form the input for OpenAI API.

3. LLM API Usage

- Uses LLM model.
- Authenticated via **Modal Secrets** (`chongchen-llm-api-key`).

4. SQL Execution

- Runs the generated SQL query using `execute_sql(conn, sql)`.
- Returns **results or error messages**.

5. Logging

- Stores **queries, SQL, results, and errors** in Modal Volume `/data/qa_history`.

4. 10 Normal Test Cases

The below code use 10 normal test cases to validate the functionality of **Text-to-SQL** implementation.

```
1  import modal
2  import sqlite3
3  import os
4  from openai import OpenAI
5  from datetime import datetime
6
7  app = modal.App("bitcoin-sql-qa")
8  volume = modal.Volume.from_name("chongchen-bitcoin-data", create_if_missing=True)
9  image = modal.Image.debian_slim().pip_install("openai")
10
11 test_cases = [
12     {
13         "question": "What is the hash of the genesis block (block at height 0)?",
14         "expected_sql": "SELECT hash FROM block WHERE height = 0;"
15     },
```

```

16     {
17         "question": "How many confirmations does block #100000 have?",
18         "expected_sql": "SELECT confirmations FROM block WHERE height =
100000;"
19     },
20     {
21         "question": "What is the average block size for blocks between
height 50000 and 55000?",
22         "expected_sql": "SELECT AVG(size) FROM block WHERE height BETWE
EN 50000 AND 55000;"
23     },
24     {
25         "question": "Which block has the highest difficulty between blo
cks 150000 and 160000?",
26         "expected_sql": "SELECT hash, height, difficulty FROM block WHE
RE height BETWEEN 150000 AND 160000 ORDER BY difficulty DESC LIMIT 1;"
27     },
28     {
29         "question": "How many transactions (ntx) were there in total ac
ross all blocks in the range 123000 to 123100?",
30         "expected_sql": "SELECT SUM(ntx) FROM block WHERE height BETWEE
N 123000 AND 123100;"
31     },
32     {
33         "question": "What is the timestamp (time) of the latest block i
n the database?",
34         "expected_sql": "SELECT time, height FROM block ORDER BY height
DESC LIMIT 1;"
35     },
36     {
37         "question": "Find the 5 blocks with the largest size difference
compared to their previous block between heights 75000 and 80000.",
38         "expected_sql": "SELECT b.height, b.hash, b.size, p.size AS pre
v_size, (b.size - p.size) AS size_diff FROM block b JOIN block p ON b.p
reviousblockhash = p.hash WHERE b.height BETWEEN 75000 AND 80000 ORDER
BY ABS(b.size - p.size) DESC LIMIT 5;"
39     },
40     {
41         "question": "What was the average time (in seconds) between blo
cks from height 140000 to 140100?",
42         "expected_sql": "WITH block_times AS (SELECT height, time, LAG
(time) OVER (ORDER BY height) AS prev_time FROM block WHERE height BETW
EEN 140000 AND 140100) SELECT AVG(time - prev_time) FROM block_times WH
ERE prev_time IS NOT NULL;"

```

```

43     },
44     {
45         "question": "How many blocks have a nonce value greater than 30
00000000 between heights 50000 and 60000?",
46         "expected_sql": "SELECT COUNT(*) FROM block WHERE height BETWEE
N 50000 AND 60000 AND nonce > 3000000000;",
47     },
48     {
49         "question": "What is the distribution of block sizes by month i
n 2012? Show the month, average size, min size, and max size.",
50         "expected_sql": "SELECT strftime('%Y-%m', datetime(time, 'unixe
poch')) AS month, AVG(size) AS avg_size, MIN(size) AS min_size, MAX(siz
e) AS max_size FROM block WHERE strftime('%Y', datetime(time, 'unixepoc
h')) = '2012' GROUP BY month ORDER BY month;",
51     },
52     {
53         "question": "Find blocks where the difficulty increased by more
than 10% compared to the previous block in the range 80000 to 90000.",
54         "expected_sql": "SELECT b.height, b.hash, b.difficulty, p.diffi
culty AS prev_difficulty, (b.difficulty - p.difficulty)/p.difficulty*10
0 AS difficulty_increase_pct FROM block b JOIN block p ON b.previousblo
ckhash = p.hash WHERE b.height BETWEEN 80000 AND 90000 AND (b.difficult
y - p.difficulty)/p.difficulty > 0.1 ORDER BY difficulty_increase_pct D
ESC;",
55     },
56     {
57         "question": "What is the correlation between block size and num
ber of transactions (ntx) for blocks 100000 to 110000?",
58         "expected_sql": "SELECT (COUNT(*) * SUM(size * ntx) - SUM(size)
* SUM(ntx)) / (SQRT(COUNT(*) * SUM(size * size) - SUM(size) * SUM(siz
e)) * SQRT(COUNT(*) * SUM(ntx * ntx) - SUM(ntx) * SUM(ntx))) AS correla
tion FROM block WHERE height BETWEEN 100000 AND 110000;",
59     },
60     {
61         "question": "How has the average block size changed each year f
rom 2009 to 2015?",
62         "expected_sql": "SELECT strftime('%Y', datetime(time, 'unixepoc
h')) AS year, AVG(size) AS avg_size FROM block WHERE strftime('%Y', dat
etime(time, 'unixepoch')) BETWEEN '2009' AND '2015' GROUP BY year ORDER
BY year;",
63     },
64     {
65         "question": "Find the top 5 blocks with the most transactions
(ntx) between height 160000 and 170000.",

```

```

66         "expected_sql": "SELECT height, hash, ntx FROM block WHERE height BETWEEN 160000 AND 170000 ORDER BY ntx DESC LIMIT 5;"
67     }
68 ]
69
70 SYSTEM_PROMPT = """
71     You are a SQL developer that is expert in Bitcoin and you answer natural \
72     language questions about the bitcoind database in a sqlite database. \
73     You always only respond with SQL statements that are correct, \
74     you just need to give the SQL statement, nothing extra.
75 """
76
77 def get_schema(conn):
78     """Extract schema from SQLite database."""
79     cursor = conn.cursor()
80     cursor.execute("SELECT sql FROM sqlite_master WHERE type IN ('table', 'view') AND sql IS NOT NULL")
81     schemas = cursor.fetchall()
82     return '\n'.join([schema[0] for schema in schemas])
83
84 def execute_sql(conn, sql):
85     """Execute SQL query and return results or error."""
86     try:
87         cursor = conn.cursor()
88         cursor.execute(sql)
89         result = cursor.fetchall()
90         return result, None
91     except sqlite3.Error as e:
92         return None, str(e)
93
94 def format_result_for_markdown(result):
95     """Format SQL result for markdown display"""
96     if not result or len(result) == 0:
97         return "No results"
98
99     # For single value results
100     if len(result) == 1 and len(result[0]) == 1:
101         return str(result[0][0])
102
103     # For multi-row results, create a simplified representation
104     if len(result) <= 5:
105         return str(result)

```



```

106         else:
107             return f"{str(result[:5])}... (showing 5 of {len(result)} rows)"
108
109 def log_qa_history(volume, question, sql, result, error):
110     """Log QA history to a file in the Modal Volume."""
111     log_dir = "/data/qa_history"
112     if not os.path.exists(log_dir):
113         os.makedirs(log_dir, exist_ok=True)
114
115     timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
116     log_file = os.path.join(log_dir, f"{timestamp}.txt")
117
118     log_content = f"Question: {question}\nGenerated SQL: {sql}\n"
119     if error:
120         log_content += f"Error: {error}\n"
121     else:
122         log_content += f"Result: {result}\n"
123
124     with open(log_file, 'w') as f:
125         f.write(log_content)
126
127     volume.commit() # Persist changes to the volume
128
129 @app.function(
130     image=image,
131     volumes={"/data": volume},
132     secrets=[modal.Secret.from_name("chongchen-llm-api-key")]
133 )
134 def answer_question(question: str, db_path: str):
135     conn = sqlite3.connect(db_path)
136     schema = get_schema(conn)
137     user_prompt = f"Database schema:\n{schema}\n\nQuestion: {question}"
138
139     # Generate SQL
140     llm_api_key = os.environ["DMX_API"]
141     client = OpenAI(base_url="https://www.dmxapi.com/v1", api_key=llm_api_key)
142     response = client.chat.completions.create(
143         model="grok-3",
144         messages=[
145             {"role": "system", "content": SYSTEM_PROMPT},
146             {"role": "user", "content": user_prompt}
147         ],

```

```

148         temperature=0.1,
149         max_tokens=500
150     )
151     generated_sql = response.choices[0].message.content.strip()
152
153     # Execute SQL
154     result, error = execute_sql(conn, generated_sql)
155     conn.close()
156
157     # Log history
158     log_qa_history(volume, question, generated_sql, result, error)
159
160     return {
161         "generated_sql": generated_sql,
162         "result": result,
163         "error": error
164     }
165
166 @app.function(
167     image=image,
168     volumes={"/data": volume},
169     secrets=[modal.Secret.from_name("chongchen-llm-api-key")]
170 )
171 def generate_markdown_report(db_path = "/data/bitcoin.db"):
172     conn = sqlite3.connect(db_path)
173
174     # Initialize markdown content
175     markdown_content = "# Bitcoin Database Natural Language to SQL Test Results\n\n"
176     markdown_content += "This report shows the results of testing natural language queries against a Bitcoin blockchain database.\n\n"
177     markdown_content += "| # | Question | SQL Statement | Result |\n"
178     markdown_content += "|---|-----|-----|-----|\n"
179
180     # Process each test case
181     for idx, case in enumerate(test_cases, 1):
182         question = case["question"]
183         expected_sql = case["expected_sql"]
184
185         # Execute SQL to get the answer
186         result, error = execute_sql(conn, expected_sql)
187
188         # Format result for markdown
189         if error:

```

```

190         result_text = f"ERROR: {error}"
191     else:
192         result_text = format_result_for_markdown(result)
193
194         # Add to markdown table
195         markdown_content += f"| {idx} | {question} | `{expected_sql}` | {result_text} |\n"
196
197     conn.close()
198
199     # Save the markdown file
200     report_dir = "/data/reports"
201     os.makedirs(report_dir, exist_ok=True)
202     timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
203     report_file = os.path.join(report_dir, f"bitcoin_sql_report_{timestamp}.md")
204
205     with open(report_file, 'w') as f:
206         f.write(markdown_content)
207
208     volume.commit() # Persist changes to the volume
209
210     print(f"Markdown report generated: {report_file}")
211     return report_file
212
213 @app.function(
214     image=image,
215     volumes={"/data": volume},
216     secrets=[modal.Secret.from_name("chongchen-llm-api-key")]
217 )
218 def run_tests_and_generate_report(db_path = "/data/bitcoin.db"):
219     """Run all test cases and generate both individual test results and a comprehensive report"""
220     # Run test cases
221     conn = sqlite3.connect(db_path)
222
223     # Initialize markdown content
224     markdown_content = "# Bitcoin Database Natural Language to SQL Test Results\n\n"
225     markdown_content += "This report shows the results of testing natural language queries against a Bitcoin blockchain database.\n\n"
226     markdown_content += "## Test Case Results\n\n"
227
228     for test_id, case in enumerate(test_cases, 1):

```

```

229     question = case["question"]
230     expected_sql = case["expected_sql"]
231
232     # Get expected answer
233     expected_result, expected_error = execute_sql(conn, expected_sq
1)
234
235     # Get system's response (using LLM)
236     response = answer_question.remote(question, db_path)
237     generated_sql = response["generated_sql"]
238     generated_result = response["result"]
239     generated_error = response["error"]
240
241     # Format results for markdown
242     if expected_error:
243         expected_result_text = f"ERROR: {expected_error}"
244     else:
245         expected_result_text = format_result_for_markdown(expected_
result)
246
247     if generated_error:
248         generated_result_text = f"ERROR: {generated_error}"
249     else:
250         generated_result_text = format_result_for_markdown(generate
d_result)
251
252     # Add section for this test case
253     markdown_content += f"### Test Case {test_id}\n\n"
254     markdown_content += f"**Question:** {question}\n\n"
255     markdown_content += f"**Expected SQL:**\n```\n{expected_sql}\n```\n\n".fo
rmat(expected_sql)
256     markdown_content += f"**Expected Result:** {expected_result_text}\n\n"
257     markdown_content += f"**Generated SQL:**\n```\n{generated_sql}\n```\n\n".f
ormat(generated_sql)
258     markdown_content += f"**Actual Result:** {generated_result_text}\n\n"
259
260     # Add comparison
261     if expected_error is None and generated_error is None:
262         if str(expected_result) == str(generated_result):
263             markdown_content += "✅ **Result Match**: The generate
d query produced the correct result.\n\n"
264         else:

```

```

265         markdown_content += "❌ **Result Mismatch**: The gener
ated query produced a different result than expected.\n\n"
266     else:
267         markdown_content += "⚠️ **Error in Execution**: One or bot
h queries produced an error.\n\n"
268
269         markdown_content += "---\n\n"
270
271     conn.close()
272
273     # Save the markdown file
274     report_dir = "/data/reports"
275     os.makedirs(report_dir, exist_ok=True)
276     timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
277     report_file = os.path.join(report_dir, f"bitcoin_sql_report_{timest
amp}.md")
278
279     with open(report_file, 'w') as f:
280         f.write(markdown_content)
281
282     volume.commit() # Persist changes to the volume
283
284     print(f"Comprehensive markdown report generated: {report_file}")
285     return report_file
286
287 @app.function(
288     image=image,
289     volumes={"/data": volume},
290     secrets=[modal.Secret.from_name("chongchen-llm-api-key")]
291 )
292 def generate_summary_table(db_path = "/data/bitcoin.db"):
293     """Generate a single markdown file with just the summary table of a
ll test cases"""
294     conn = sqlite3.connect(db_path)
295
296     # Initialize markdown content
297     markdown_content = "# Bitcoin Database Natural Language to SQL Test
Cases\n\n"
298     markdown_content += "| # | Question | SQL Statement | Result |\n"
299     markdown_content += "|---|-----|-----|-----|\n"
300
301     # Process each test case
302     for idx, case in enumerate(test_cases, 1):
303         question = case["question"]

```

```

304         expected_sql = case["expected_sql"]
305
306         # Execute SQL to get the answer
307         result, error = execute_sql(conn, expected_sql)
308
309         # Format result for markdown
310         if error:
311             result_text = f"ERROR: {error}"
312         else:
313             result_text = format_result_for_markdown(result)
314
315         # Add to markdown table
316         markdown_content += f"| {idx} | {question} | `{expected_sql}` | {result_text} |\n"
317
318     conn.close()
319
320     # Save the markdown file
321     report_dir = "/data/reports"
322     os.makedirs(report_dir, exist_ok=True)
323     report_file = os.path.join(report_dir, "bitcoin_sql_summary.md")
324
325     with open(report_file, 'w') as f:
326         f.write(markdown_content)
327
328     volume.commit() # Persist changes to the volume
329
330     print(f"Summary table report generated: {report_file}")
331     return report_file
332
333 if __name__ == "__main__":
334     with app.run():
335         # Path to your SQLite database
336         db_path = "/data/bitcoin.db"
337
338         # Generate the markdown report with all test cases in a single
table
339         generate_summary_table.call(db_path)
340
341         # Optionally, run tests with LLM and generate detailed report
342         # run_tests_and_generate_report.call(db_path)

```

The full QA historial is logged in the volume: https://modal.com/storage/neu-info5100-oak-spr-2025/main/chongchen-bitcoin-data/sql_tests/normal.

This implementation evaluates the accuracy of a natural language-to-SQL conversion system for querying Bitcoin blockchain data stored in SQLite. It uses OpenAI's LLM API to generate SQL queries from user questions, executes them, and compares the results with predefined correct answers. The system includes 10 test cases covering various SQL functionalities, such as counting, filtering, aggregation, and ordering. Each test logs the expected SQL, generated SQL, expected answer, and actual answer for validation. By automating query execution and result comparison, the implementation ensures accuracy and reliability in translating Bitcoin-related questions into precise SQL queries.

Test Case 1

Question: What is the hash of the genesis block (block at height 0)?

Expected SQL:

```
1 | SELECT hash FROM block WHERE height = 0;
```

Expected Result: 000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f

Generated SQL:

```
1 | SELECT hash FROM block WHERE height = 0;
```

Actual Result: 000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f

Test Case 2

Question: How many confirmations does block #100000 have?

Expected SQL:

```
1 | SELECT confirmations FROM block WHERE height = 100000;
```

Expected Result: 785648

Generated SQL:

```
1 | SELECT confirmations FROM block WHERE height = 100000;
```

Actual Result: 785648

Test Case 3

Question: What is the average block size for blocks between height 50000 and 55000?

Expected SQL:

```
1 | SELECT AVG(size) FROM block WHERE height BETWEEN 50000 AND 55000;
```

Expected Result: 496.68886222755447

Generated SQL:

```
1 | SELECT AVG(size)
2 | FROM block
3 | WHERE height BETWEEN 50000 AND 55000;
```

Actual Result: 496.68886222755447

Test Case 4

Question: Which block has the highest difficulty between blocks 150000 and 160000?

Expected SQL:

```
1 | SELECT hash, height, difficulty FROM block WHERE height BETWEEN 150000
   | AND 160000 ORDER BY difficulty DESC LIMIT 1;
```

Expected Result: [('00000000000000a3290f20e75860d505ce0e948a1d1d846bec7e39015d242884b',
150000, 1468195.427220831)]

Generated SQL:

```
1 | SELECT hash, difficulty
2 | FROM block
3 | WHERE height BETWEEN 150000 AND 160000
4 | ORDER BY difficulty DESC
5 | LIMIT 1
```

Actual Result: [('00000000000000a3290f20e75860d505ce0e948a1d1d846bec7e39015d242884b',
1468195.427220831)]

Test Case 5

Question: How many transactions (ntx) were there in total across all blocks in the range 123000 to 123100?

Expected SQL:

```
1 | SELECT SUM(ntx) FROM block WHERE height BETWEEN 123000 AND 123100;
```

Expected Result: 1596

Generated SQL:

```
1 | SELECT SUM(nTx)
2 | FROM block
3 | WHERE height BETWEEN 123000 AND 123100
```

Actual Result: 1596

Test Case 6

Question: What is the timestamp (time) of the latest block in the database?

Expected SQL:

```
1 | SELECT time, height FROM block ORDER BY height DESC LIMIT 1;
```

Expected Result: [(1366117453, 231644)]

Generated SQL:

```
1 | SELECT time FROM block ORDER BY height DESC LIMIT 1
```

Actual Result: 1366117453

Test Case 7

Question: Find the 5 blocks with the largest size difference compared to their previous block between heights 75000 and 80000.

Expected SQL:

```
1 | SELECT b.height, b.hash, b.size, p.size AS prev_size, (b.size - p.size)
   | AS size_diff FROM block b JOIN block p ON b.previousblockhash = p.hash
   | WHERE b.height BETWEEN 75000 AND 80000 ORDER BY ABS(b.size - p.size) DE
   | SC LIMIT 5;
```

Expected Result: [(76159, '000000000005560bafd779f7a68cac34d8bfd7862bbae3eb25761a97244458f2', 77018, 440, 76578), (76160, '00000000003110b77de74d891a29ecd76298cf9fff812b70466bf57927eb46be', 1159, 77018, -75859), (77289, '000000000014e61addbe1d35dac5be43a9ed4a36018037580c3b7c8f2fd9477a', 215, 41130, -40915), (77288, '0000000000442f240d796f1eaf66ff4c52bd5875dcff334508cecfaacb0fc505', 41130, 2709, 38421), (77469, '0000000000393b038f47d7bcd64b3823f49715f5f980248b922dd7e91bfa37af', 22509, 216, 22293)]

Generated SQL:

```

1 | SELECT b1.hash, b1.height, b1.size, b2.size AS prev_size, (b1.size - b
2 | 2.size) AS size_diff
3 | FROM block b1
4 | JOIN block b2 ON b1.previousblockhash = b2.hash
5 | WHERE b1.height BETWEEN 75000 AND 80000
6 | ORDER BY size_diff DESC
7 | LIMIT 5

```

Actual Result: [('000000000005560bafd779f7a68cac34d8bfd7862bbae3eb25761a97244458f2', 76159, 77018, 440, 76578), ('00000000000442f240d796f1eaf66ff4c52bd5875dcff334508cecfaacb0fc505', 77288, 41130, 2709, 38421), ('00000000000393b038f47d7bcd64b3823f49715f5f980248b922dd7e91bfa37af', 77469, 22509, 216, 22293), ('000000000005cd2bca975d46d05d638942137725fbb691a5a2f7f53e3a4e2b7ac', 76584, 20657, 489, 20168), ('000000000005eb1280e6aaa91dfa5a417315b283e51a4000f1d8527be03903cce', 77554, 14740, 475, 14265)]

Test Case 8

Question: What was the average time (in seconds) between blocks from height 140000 to 140100?

Expected SQL:

```

1 | WITH block_times AS (SELECT height, time, LAG(time) OVER (ORDER BY heig
2 | ht) AS prev_time FROM block WHERE height BETWEEN 140000 AND 140100) SEL
3 | ECT AVG(time - prev_time) FROM block_times WHERE prev_time IS NOT NULL;

```

Expected Result: 683.99

Generated SQL:

```

1 | SELECT AVG(b2.time - b1.time)
2 | FROM block b1
3 | JOIN block b2 ON b1.height + 1 = b2.height
4 | WHERE b1.height BETWEEN 140000 AND 140100 - 1;

```

Actual Result: 683.99

Test Case 9

Question: How many blocks have a nonce value greater than 3000000000 between heights 50000 and 60000?

Expected SQL:

```
1 | SELECT COUNT(*) FROM block WHERE height BETWEEN 50000 AND 60000 AND non  
   | ce > 3000000000;
```

Expected Result: 353

Generated SQL:

```
1 | SELECT COUNT(*)  
2 | FROM block  
3 | WHERE nonce > 3000000000  
4 | AND height BETWEEN 50000 AND 60000
```

Actual Result: 353

Test Case 10

Question: What is the distribution of block sizes by month in 2012? Show the month, average size, min size, and max size.

Expected SQL:

```
1 | SELECT strftime('%Y-%m', datetime(time, 'unixepoch')) AS month, AVG(size)  
   | AS avg_size, MIN(size) AS min_size, MAX(size) AS max_size FROM block  
   | WHERE strftime('%Y', datetime(time, 'unixepoch')) = '2012' GROUP BY mon  
   | th ORDER BY month;
```

Expected Result: [('2012-01', 20554.389544688027, 195, 334262), ('2012-02', 21596.938231917335, 213, 218762), ('2012-03', 20485.796958663526, 213, 211123), ('2012-04', 24840.587885985748, 190, 327826), ('2012-05', 66805.68305391935, 190, 499240)]... (showing 5 of 12 rows)

Generated SQL:

```
1 | SELECT  
2 |     strftime('%m', datetime(time, 'unixepoch')) AS month,  
3 |     AVG(size) AS avg_size,  
4 |     MIN(size) AS min_size,  
5 |     MAX(size) AS max_size  
6 | FROM block  
7 | WHERE strftime('%Y', datetime(time, 'unixepoch')) = '2012'  
8 | GROUP BY strftime('%m', datetime(time, 'unixepoch'))  
9 | ORDER BY month
```

Actual Result: [('01', 20554.389544688027, 195, 334262), ('02', 21596.938231917335, 213, 218762), ('03', 20485.796958663526, 213, 211123), ('04', 24840.587885985748, 190, 327826), ('05', 66805.68305391935, 190, 499240)]... (showing 5 of 12 rows)

Test Case 11

Question: Find blocks where the difficulty increased by more than 10% compared to the previous block in the range 80000 to 90000.

Expected SQL:

```
1  SELECT b.height, b.hash, b.difficulty, p.difficulty AS prev_difficulty,
   (b.difficulty - p.difficulty)/p.difficulty*100 AS difficulty_increase_p
ct FROM block b JOIN block p ON b.previousblockhash = p.hash WHERE b.he
ight BETWEEN 80000 AND 90000 AND (b.difficulty - p.difficulty)/p.diffic
ulty > 0.1 ORDER BY difficulty_increase_pct DESC;
```

Expected Result: [(86688,
'000000000015bfe777e893c4ebd1307541792630c2932278bfe8cf3ae82668ce', 2149.021814946726,
1378.028165037326, 55.94904875463966), (88704,
'000000000012384edfbd167c7778aec3e84bb1795b907cc795912e643c2cff04', 3091.736890411797,
2149.021814946726, 43.86717105002682), (82656,
'000000000024fc69f5415908b1960092a8e81b9d3b9a03c1133f5cb0a2d3c2af', 1318.670050153592,
917.8307413015116, 43.67246495619423), (80640,
'0000000000307c80b87edf9f6a0697e2f01db67e518c8a4d6065d1d859a3a659', 917.8307413015116,
712.8848645520973, 28.74880460229451)]

Generated SQL:

```
1  SELECT b1.hash, b1.height, b1.difficulty, b2.difficulty AS prev_difficu
lty,
2      ((b1.difficulty - b2.difficulty) / b2.difficulty * 100) AS diffi
culty_increase
3  FROM block b1
4  JOIN block b2 ON b1.previousblockhash = b2.hash
5  WHERE b1.height BETWEEN 80000 AND 90000
6  AND ((b1.difficulty - b2.difficulty) / b2.difficulty * 100) > 10
```

Actual Result: [('0000000000307c80b87edf9f6a0697e2f01db67e518c8a4d6065d1d859a3a659',
80640, 917.8307413015116, 712.8848645520973, 28.74880460229451),
(('000000000024fc69f5415908b1960092a8e81b9d3b9a03c1133f5cb0a2d3c2af', 82656,
1318.670050153592, 917.8307413015116, 43.67246495619423),
(('000000000015bfe777e893c4ebd1307541792630c2932278bfe8cf3ae82668ce', 86688,
2149.021814946726, 1378.028165037326, 55.94904875463966),
(('000000000012384edfbd167c7778aec3e84bb1795b907cc795912e643c2cff04', 88704,
3091.736890411797, 2149.021814946726, 43.86717105002682)]

Test Case 12

Question: What is the correlation between block size and number of transactions (ntx) for blocks 100000 to 110000?

Expected SQL:

```
1  SELECT (COUNT(*) * SUM(size * ntx) - SUM(size) * SUM(ntx)) / (SQRT(COUNT(*) * SUM(size * size) - SUM(size) * SUM(size)) * SQRT(COUNT(*) * SUM(ntx * ntx) - SUM(ntx) * SUM(ntx))) AS correlation FROM block WHERE height BETWEEN 100000 AND 110000;
```

Expected Result: 0.7257125719443465

Generated SQL:

```
1  SELECT
2      (COUNT(*) * SUM(size * nTx) - SUM(size) * SUM(nTx)) /
3      SQRT((COUNT(*) * SUM(size * size) - SUM(size) * SUM(size)) *
4          (COUNT(*) * SUM(nTx * nTx) - SUM(nTx) * SUM(nTx)))
5  FROM block
6  WHERE height BETWEEN 100000 AND 110000
```

Actual Result: 0.7257125719443465

Test Case 13

Question: How has the average block size changed each year from 2009 to 2015?

Expected SQL:

```
1  SELECT strftime('%Y', datetime(time, 'unixepoch')) AS year, AVG(size) AS avg_size FROM block WHERE strftime('%Y', datetime(time, 'unixepoch')) BETWEEN '2009' AND '2015' GROUP BY year ORDER BY year;
```

Expected Result: [('2009', 226.58307171437366), ('2010', 777.7929034157833), ('2011', 13445.744461401713), ('2012', 69627.1657741261), ('2013', 156263.41400304413)]

Generated SQL:

```
1  SELECT
2      strftime('%Y', datetime(time, 'unixepoch')) as year,
3      AVG(size) as avg_block_size
4  FROM block
5  WHERE year BETWEEN '2009' AND '2015'
6  GROUP BY year
7  ORDER BY year;
```

Actual Result: [('2009', 226.58307171437366), ('2010', 777.7929034157833), ('2011', 13445.744461401713), ('2012', 69627.1657741261), ('2013', 156263.41400304413)]

Test Case 14

Question: Find the top 5 blocks with the most transactions (ntx) between height 160000 and 170000.

Expected SQL:

```
1 | SELECT height, hash, ntx FROM block WHERE height BETWEEN 160000 AND 170000 ORDER BY ntx DESC LIMIT 5;
```

Expected Result: [(166966, '000000000000007eaeaefaf88bc9c055011e3f71df490556df289545f891421e0', 233), (166723, '00000000000000a56b4dff35485f5e42ebe518bcc60ea2b1f6f7bc3916e33ce9a', 226), (166221, '0000000000000097b7c075ccc95aa2f99e326aa5e48192ac00888f989a501d111', 218), (162928, '00000000000000c861e7811237029484dab6ec704b76b328af1560a7cda640d5d', 216), (166105, '000000000000008e772f92b56b79031b7000848a112912cdd7ede464202ff5c19', 202)]

Generated SQL:

```
1 | SELECT hash, height, nTx
2 | FROM block
3 | WHERE height BETWEEN 160000 AND 170000
4 | ORDER BY nTx DESC
5 | LIMIT 5
```

Actual Result: [('000000000000007eaeaefaf88bc9c055011e3f71df490556df289545f891421e0', 166966, 233), ('00000000000000a56b4dff35485f5e42ebe518bcc60ea2b1f6f7bc3916e33ce9a', 166723, 226), ('0000000000000097b7c075ccc95aa2f99e326aa5e48192ac00888f989a501d111', 166221, 218), ('00000000000000c861e7811237029484dab6ec704b76b328af1560a7cda640d5d', 162928, 216), ('000000000000008e772f92b56b79031b7000848a112912cdd7ede464202ff5c19', 166105, 202)]

5. 3 Hard Test Cases

Below code write three very hard test cases that are so hard that the system is not able to answer correctly. The purpose of this is to find the limit of what's possible for this task.

```
1 | import modal
2 | import sqlite3
3 | import os
4 | from openai import OpenAI
5 | from datetime import datetime
6 |
```

```

7 app = modal.App("bitcoin-sql-qa")
8 volume = modal.Volume.from_name("chongchen-bitcoin-data", create_if_mis
sing=True)
9 image = modal.Image.debian_slim().pip_install("openai")
10
11 hard_test_cases = [
12     {
13         "question": "What is the median nonce value for blocks mined in
February 2013?",
14         "expected_sql": """
15             SELECT AVG(nonce)
16             FROM (
17                 SELECT nonce
18                 FROM block
19                 WHERE strftime('%Y-%m', datetime(time, 'unixepoch')) =
'2013-02'
20
21                 ORDER BY nonce
22                 LIMIT 2 - (
23                     SELECT COUNT(*)
24                     FROM block
25                     WHERE strftime('%Y-%m', datetime(time, 'unixepoc
h')) = '2013-02'
26
27                     ) % 2
28                 OFFSET (
29                     SELECT COUNT(*)
30                     FROM block
31                     WHERE strftime('%Y-%m', datetime(time, 'unixepoc
h')) = '2013-02'
32
33                     ) / 2
34                 );
35         """,
36         "question": "Analyze the 'fee market' development by calculatin
g the implicit fee per transaction in satoshis for each block from 1500
00 to 160000. For this, estimate the mining reward by using the formul
a: (block_reward_bitcoins * 10^8 + (block_size - 80) * 10). Then calcul
ate fee = (reward - expected_subsidy) / ntx where expected_subsidy is 5
0 BTC per block multiplied by 10^8 to convert to satoshis. Show the top
10 blocks with highest average fee per transaction, including block hei
ght, time (formatted as date), number of transactions, and average fee
per transaction.",
36         "expected_sql": """
37             WITH block_rewards AS (

```

```

38         SELECT
39             height,
40             hash,
41             ntx,
42             size,
43             datetime(time, 'unixepoch') AS block_date,
44             (size - 80) * 10 AS size_reward_satoshis,
45             CASE
46                 WHEN height < 210000 THEN 5000000000 -- 50 BTC
in satoshis
47                 WHEN height < 420000 THEN 2500000000 -- 25 BTC
in satoshis
48                 WHEN height < 630000 THEN 1250000000 -- 12.5 BT
C in satoshis
49                 ELSE 625000000 -- 6.25 BTC in satoshis
50             END AS block_subsidy_satoshis
51         FROM block
52         WHERE height BETWEEN 150000 AND 160000 AND ntx > 1
53     )
54     SELECT
55         height,
56         hash,
57         block_date,
58         ntx,
59         size,
60         block_subsidy_satoshis,
61         size_reward_satoshis,
62         CASE
63             WHEN ntx > 1 THEN ROUND((size_reward_satoshis - blo
ck_subsidy_satoshis) / (ntx - 1), 2)
64             ELSE 0
65         END AS avg_fee_per_tx_satoshis
66     FROM block_rewards
67     ORDER BY avg_fee_per_tx_satoshis DESC
68     LIMIT 10;
69     """"
70 },
71 {
72     "question": "Calculate the mining difficulty adjustment pattern
by finding the percentage change in difficulty between each difficulty
adjustment period (every 2016 blocks) from block 50000 to 100000. Show
the starting block of each period, the average block time in minutes fo
r that period, and the percentage difficulty change.",
73     "expected_sql": """"

```



```

74         WITH adjustment_periods AS (
75             SELECT
76                 height,
77                 difficulty,
78                 time,
79                 height / 2016 AS period_number
80             FROM block
81             WHERE height BETWEEN 50000 AND 100000
82         ),
83         period_stats AS (
84             SELECT
85                 period_number,
86                 MIN(height) AS start_block,
87                 MAX(difficulty) AS difficulty,
88                 (MAX(time) - MIN(time)) / (COUNT(*) - 1) / 60.0 AS
avg_block_time_minutes,
89                 LAG(MAX(difficulty)) OVER (ORDER BY period_number)
AS prev_difficulty
90             FROM adjustment_periods
91             GROUP BY period_number
92         )
93         SELECT
94             start_block,
95             avg_block_time_minutes,
96             difficulty,
97             prev_difficulty,
98             CASE
99                 WHEN prev_difficulty IS NULL THEN NULL
100                 ELSE ROUND((difficulty - prev_difficulty) / prev_difficulty * 100, 2)
101             END AS difficulty_change_percent
102         FROM period_stats
103         ORDER BY start_block;
104     """
105 }
106 ]
107
108 SYSTEM_PROMPT = """
109     You are a SQL developer that is expert in Bitcoin and you answer natural \
110     language questions about the bitcoind database in a sqlite database. \
111     You always only respond with SQL statements that are correct, \
112     you just need to give the SQL statement, nothing extra.

```

```

113     """
114
115     def get_schema(conn):
116         """Extract schema from SQLite database."""
117         cursor = conn.cursor()
118         cursor.execute("SELECT sql FROM sqlite_master WHERE type IN ('tabl
119 e', 'view') AND sql IS NOT NULL")
120         schemas = cursor.fetchall()
121         return '\n'.join([schema[0] for schema in schemas])
122
123     def execute_sql(conn, sql):
124         """Execute SQL query and return results or error."""
125         try:
126             cursor = conn.cursor()
127             cursor.execute(sql)
128             result = cursor.fetchall()
129             return result, None
130         except sqlite3.Error as e:
131             return None, str(e)
132
133     def log_qa_history(volume, question, sql, result, error):
134         """Log QA history to a file in the Modal Volume."""
135         log_dir = "/data/qa_history"
136         if not os.path.exists(log_dir):
137             os.makedirs(log_dir, exist_ok=True)
138
139         timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
140         log_file = os.path.join(log_dir, f"{timestamp}.txt")
141
142         log_content = f"Question: {question}\nGenerated SQL: {sql}\n"
143         if error:
144             log_content += f"Error: {error}\n"
145         else:
146             log_content += f"Result: {result}\n"
147
148         with open(log_file, 'w') as f:
149             f.write(log_content)
150
151         volume.commit() # Persist changes to the volume
152
153     @app.function(
154         image=image,
155         volumes={"/data": volume},
156         secrets=[modal.Secret.from_name("chongchen-llm-api-key")]

```

```

156 )
157 def answer_question(question: str, db_path: str):
158     conn = sqlite3.connect(db_path)
159     schema = get_schema(conn)
160     user_prompt = f"Database schema:\n{schema}\n\nQuestion: {question}"
161
162     # Generate SQL
163     llm_api_key = os.environ["DMX_API"]
164     client = OpenAI(base_url="https://www.dmxapi.com/v1", api_key=llm_a
pi_key)
165     response = client.chat.completions.create(
166         model="grok-3",
167         messages=[
168             {"role": "system", "content": SYSTEM_PROMPT},
169             {"role": "user", "content": user_prompt}
170         ],
171         temperature=0.2,
172         max_tokens=500
173     )
174     generated_sql = response.choices[0].message.content.strip()
175
176     # Execute SQL
177     result, error = execute_sql(conn, generated_sql)
178     conn.close()
179
180     # Log history
181     log_qa_history(volume, question, generated_sql, result, error)
182
183     return {
184         "generated_sql": generated_sql,
185         "result": result,
186         "error": error
187     }
188
189 def log_test_result(test_type: str, content: str, filename: str):
190     log_dir = f"/data/sql_tests/{test_type}"
191     os.makedirs(log_dir, exist_ok=True)
192     # timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
193     log_file = os.path.join(log_dir, f"{filename}.txt")
194
195     with open(log_file, 'w') as f:
196         f.write(content)
197     volume.commit()
198

```

```

199 @app.function(
200     image=image,
201     volumes={"/data": volume},
202     secrets=[modal.Secret.from_name("chongchen-llm-api-key")]
203 )
204 def test_hard_cases(db_path = "/data/bitcoin.db"):
205     conn = sqlite3.connect(db_path)
206     for test_id, case in enumerate(hard_test_cases, start=1):
207         question = case["question"]
208         expected_sql = case["expected_sql"]
209
210         # Get expected answer
211         expected_answer, _ = execute_sql(conn, expected_sql)
212
213         # Get system's response
214         response = answer_question.remote(question, db_path)
215         generated_sql = response["generated_sql"]
216         generated_answer = response["result"]
217         error = response["error"]
218
219         # Build log content
220         log_content = (
221             f"Question: {question}\n"
222             f"Expected SQL: {expected_sql}\n"
223             f"Expected Answer: {expected_answer}\n"
224             f"Generated SQL: {generated_sql}\n"
225             f"Generated Answer: {generated_answer}\n"
226         )
227         if error:
228             log_content += f"Error: {error}\n"
229
230         log_test_result("hard", log_content, f"test_{test_id}")
231     conn.close()

```

The full QA historial is logged in the volume: https://modal.com/storage/neu-info5100-oak-spr-2025/main/chongchen-bitcoin-data/sql_tests/hard.

Test Case 1

Question: What is the median nonce value for blocks mined in February 2013?

Expected SQL:

```

1 SELECT AVG(nonce)
2 FROM (

```

```

3      SELECT nonce
4      FROM block
5      WHERE strftime('%Y-%m', datetime(time, 'unixepoch')) = '2013-02'
6      ORDER BY nonce
7      LIMIT 2 - (
8          SELECT COUNT(*)
9          FROM block
10         WHERE strftime('%Y-%m', datetime(time, 'unixepoch')) = '2013-0
11         2'
12     ) % 2
13     OFFSET (
14         SELECT COUNT(*)
15         FROM block
16         WHERE strftime('%Y-%m', datetime(time, 'unixepoch')) = '2013-0
17         2'
18     ) / 2
19 );

```

Expected Answer: [(2114258196.5,)]

Generated SQL:

```

1      SELECT median(nonce)
2      FROM block
3      WHERE time >= 1359676800 AND time < 1362096000

```

Generated Answer:

None

Error: no such function: median

Test Case 2

Question: Analyze the 'fee market' development by calculating the implicit fee per transaction in satoshis for each block from 150000 to 160000. For this, estimate the mining reward by using the formula: $(\text{block_reward_bitcoins} * 10^8 + (\text{block_size} - 80) * 10)$. Then calculate $\text{fee} = (\text{reward} - \text{expected_subsidy}) / \text{ntx}$ where expected_subsidy is 50 BTC per block multiplied by 10^8 to convert to satoshis. Show the top 10 blocks with highest average fee per transaction, including block height, time (formatted as date), number of transactions, and average fee per transaction.

Expected SQL:

```

1      WITH block_rewards AS (
2          SELECT
3              height,
4              hash,

```

```

5      ntx,
6      size,
7      datetime(time, 'unixepoch') AS block_date,
8      (size - 80) * 10 AS size_reward_satoshis,
9      CASE
10         WHEN height < 210000 THEN 5000000000 -- 50 BTC in satoshis
11         WHEN height < 420000 THEN 2500000000 -- 25 BTC in satoshis
12         WHEN height < 630000 THEN 1250000000 -- 12.5 BTC in satoshi
13         ELSE 625000000 -- 6.25 BTC in satoshis
14     END AS block_subsidy_satoshis
15 FROM block
16 WHERE height BETWEEN 150000 AND 160000 AND ntx > 1
17 )
18 SELECT
19     height,
20     hash,
21     block_date,
22     ntx,
23     size,
24     block_subsidy_satoshis,
25     size_reward_satoshis,
26     CASE
27         WHEN ntx > 1 THEN ROUND((size_reward_satoshis - block_subsidy_s
28         ELSE 0
29     END AS avg_fee_per_tx_satoshis
30 FROM block_rewards
31 ORDER BY avg_fee_per_tx_satoshis DESC
32 LIMIT 10;

```

Expected Answer: [(154723,

'00000000000003bfc715be0afb06486c325c12dea913766564fd7e9bc453889d', '2011-11-25 12:21:16',
398, 107002, 5000000000, 1069220, -12591765.0), (156869,
'00000000000002bd9cc33fbf8cfef874a79bc51295c3c14a583178c258c1784f', '2011-12-10 04:35:39',
379, 123336, 5000000000, 1232560, -13224252.0), (154725,
'00000000000008c67bc9ac7eba82450a38a94051a0f3751490ecd0d837545b24', '2011-11-25 13:04:15',
239, 69926, 5000000000, 698460, -21005468.0), (155303,
'0000000000000054cafde5ade84adf4cc7b7f7cdf63bf4ba983a80f6fa1ce5c8', '2011-11-30 02:16:01',
209, 49038, 5000000000, 489580, -24036107.0), (155544,
'0000000000000c86066cdc2dc8f6961da85b390e9bae9dd4e5b700830dc20f01', '2011-12-01 12:09:22',
199, 48770, 5000000000, 486900, -25250066.0), (154432,
'00000000000004611ff254ba52111a14a98de7167f79fae9fdb87399e18cca0', '2011-11-23 04:06:37',
198, 49174, 5000000000, 490940, -25378218.0), (158212,

'000000000000084f15663b602830d00964347643f212f44deb882b301d4d9663', '2011-12-19 20:24:27',
197, 72987, 5000000000, 729070, -25506484.0), (154385,
'00000000000008f209cf9ecbfa522b0b4f0ce4a9e3bf840328b695813d89f4c9', '2011-11-22 20:02:44',
196, 76011, 5000000000, 759310, -25637131.0), (157270,
'0000000000000940c5c3d1e0820806370fae3254313f2c45cab7ede2a28f880e', '2011-12-13 00:04:17',
193, 49073, 5000000000, 489930, -26039114.0), (151992,
'0000000000000a544d3ba541dddae74e46469e38b1f8ba4681e0b5f722bb9670', '2011-11-05
15:11:30', 190, 56993, 5000000000, 569130, -26452015.0)]

Generated SQL:

```
1 SELECT
2     height,
3     DATETIME(time, 'unixepoch') AS block_time,
4     nTx AS num_transactions,
5     (((50 * POW(10,8) + (size - 80) * 10) - (50 * POW(10,8)))) / nTx) AS
    avg_fee_per_tx
6 FROM block
7 WHERE height BETWEEN 150000 AND 160000
8 ORDER BY avg_fee_per_tx DESC
9 LIMIT 10
```

Generated Answer: [(157101, '2011-12-11 17:27:34', 2, 353710.0), (154034, '2011-11-20 05:44:54', 2,
226750.0), (152680, '2011-11-10 09:23:34', 5, 153044.0), (154642, '2011-11-24 21:34:38', 3, 135970.0),
(154764, '2011-11-25 20:14:16', 2, 120665.0), (157155, '2011-12-12 01:51:46', 3, 91793.333333333333),
(152752, '2011-11-10 21:12:23', 5, 70412.0), (154761, '2011-11-25 20:00:19', 7, 70192.85714285714),
(153176, '2011-11-14 00:54:35', 9, 68177.777777777778), (152461, '2011-11-08 23:50:28', 2, 68120.0)]

Test Case 3

Question: Calculate the mining difficulty adjustment pattern by finding the percentage change in difficulty between each difficulty adjustment period (every 2016 blocks) from block 50000 to 100000. Show the starting block of each period, the average block time in minutes for that period, and the percentage difficulty change.

Expected SQL:

```
1 WITH adjustment_periods AS (
2     SELECT
3         height,
4         difficulty,
5         time,
6         height / 2016 AS period_number
7     FROM block
8     WHERE height BETWEEN 50000 AND 100000
```

```

9      ),
10     period_stats AS (
11         SELECT
12             period_number,
13             MIN(height) AS start_block,
14             MAX(difficulty) AS difficulty,
15             (MAX(time) - MIN(time)) / (COUNT(*) - 1) / 60.0 AS avg_block_time_minutes,
16             LAG(MAX(difficulty)) OVER (ORDER BY period_number) AS prev_difficulty
17         FROM adjustment_periods
18         GROUP BY period_number
19     )
20     SELECT
21         start_block,
22         avg_block_time_minutes,
23         difficulty,
24         prev_difficulty,
25         CASE
26             WHEN prev_difficulty IS NULL THEN NULL
27             ELSE ROUND((difficulty - prev_difficulty) / prev_difficulty * 100, 2)
28         END AS difficulty_change_percent
29     FROM period_stats
30     ORDER BY start_block;

```

Expected Answer: [(50000, 6.033333333333333, 6.085476906000794, None, None), (50400, 6.816666666666666, 7.819796993353832, 6.085476906000794, 28.5), (52416, 8.916666666666666, 11.46431580551412, 7.819796993353832, 46.61), (54432, 10.85, 12.84918314782378, 11.46431580551412, 12.08), (56448, 7.116666666666666, 11.84622814970471, 12.84918314782378, -7.81), (58464, 9.55, 16.61907875185736, 11.84622814970471, 40.29), (60480, 8.95, 17.38148815814011, 16.61907875185736, 4.59), (62496, 8.25, 19.40479632838217, 17.38148815814011, 11.64), (64512, 5.166666666666667, 23.50125722290317, 19.40479632838217, 21.11), (66528, 2.383333333333333, 45.38582234101263, 23.50125722290317, 93.12), (68544, 7.433333333333334, 181.5432893640505, 45.38582234101263, 300.0), (70560, 6.933333333333334, 244.2132230923753, 181.5432893640505, 34.52), (72576, 6.883333333333334, 352.1612090680101, 244.2132230923753, 44.2), (74592, 8.233333333333333, 511.7735342566043, 352.1612090680101, 45.32), (76608, 8.733333333333333, 623.3869598689275, 511.7735342566043, 21.81), (78624, 7.766666666666667, 712.8848645520973, 623.3869598689275, 14.36), (80640, 6.95, 917.8307413015116, 712.8848645520973, 28.75), (82656, 9.566666666666666, 1318.670050153592, 917.8307413015116, 43.67), (84672, 6.4, 1378.028165037326, 1318.670050153592, 4.5), (86688, 6.95, 2149.021814946726, 1378.028165037326, 55.95), (88704, 6.816666666666666, 3091.736890411797, 2149.021814946726, 43.87), (90720, 6.6, 4536.353723275037, 3091.736890411797, 46.73), (92736, 8.5, 6866.898648972739, 4536.353723275037, 51.37), (94752, 6.583333333333333, 8078.195257925088,

6866.898648972739, 17.64), (96768, 8.45, 12251.9997603745, 8078.195257925088, 51.67), (98784, 9.133333333333333, 14484.1623612254, 12251.9997603745, 18.22)]

Generated SQL:

```
1  SELECT
2      b1.height AS period_start,
3      (CAST(AVG(b2.time - b1.time) AS REAL) / 2016 * 60) AS avg_block_time_minutes,
4      ((b2.difficulty - b1.difficulty) / b1.difficulty * 100) AS difficulty_change_percent
5  FROM block b1
6  JOIN block b2 ON b2.height = b1.height + 2015
7  WHERE b1.height % 2016 = 0
8  AND b1.height BETWEEN 50000 AND 100000 - 2015
9  GROUP BY b1.height, b1.difficulty, b2.difficulty
10 ORDER BY b1.height;
```

Generated Answer: [(50400, 24555.565476190473, 0.0), (52416, 32119.97023809524, 0.0), (54432, 39047.94642857143, 0.0), (56448, 25661.13095238095, 0.0), (58464, 34420.95238095238, 0.0), (60480, 32246.339285714283, 0.0), (62496, 29724.940476190473, 0.0), (64512, 18641.190476190477, 0.0), (66528, 8614.107142857143, 0.0), (68544, 26761.75595238095, 0.0), (70560, 24965.32738095238, 0.0), (72576, 24772.916666666668, 0.0), (74592, 29554.434523809523, 0.0), (76608, 31480.446428571428, 0.0), (78624, 27961.428571428572, 0.0), (80640, 25056.99404761905, 0.0), (82656, 34449.31547619047, 0.0), (84672, 23084.464285714286, 0.0), (86688, 25023.095238095237, 0.0), (88704, 24535.684523809527, 0.0), (90720, 23782.053571428572, 0.0), (92736, 30601.964285714283, 0.0), (94752, 23736.160714285714, 0.0), (96768, 30452.113095238095, 0.0)]

6. Sources and Links

All codes: <https://github.com/chongchen1999/INFO7500-cryptocurrency/tree/main/hw4>

Bitcoin database on Modal Volume: <https://modal.com/api/volumes/neu-info5100-oak-spr-2025/main/chongchen-bitcoin-data/files/content?path=bitcoin.db>

Test cases: https://modal.com/storage/neu-info5100-oak-spr-2025/main/chongchen-bitcoin-data/sql_tests