Creating an LLM to Generate SQL Queries for GridDB

November 13, 2024
Version 1.0

Table Of Contents

Introduction

GridDB is a time series database available both on-premise and on the cloud optimized for IoT and Big Data that was developed by Toshiba Digital Solutions Corporation. It features a unique key-container model designed specifically to handle both metadata and time series data. Its in-memory architecture allows incredible ingestion and query performance. GridDB is also horizontally scalable to improve both performance and reliability.

With the rise in popularity of using Large Language Models (LLMs), general purpose models like GPT-4 have attracted the most interest and media coverage, but many other specialized models exist for the purpose of code generation. While many of these models are extremely large and require immense compute power to train and execute, some LLMs can be trained and executed on platforms that are practical and cost effective.

Every database uses a slightly different form of SQL and GridDB is no different. In particular, GridDB has different time functions compared to other SQL databases and also uses a unique Key-Container data model where it is encouraged to store homogeneous data in multiple tables. With these differences in mind, the LLM must be fine-tuned for GridDB and experience tells us that an off the shelf LLM would not produce suitable queries.

There are both consumer and business use cases for using an LLM to query GridDB. For consumers, the LLM would enable the end user to ask simple questions about their own data such as “When do I consume the most electricity?”. For business analysts and managers, it extends their business intelligence tools allowing for more ad-hoc queries to dive deeper into their organization’s data.

In this technical report, we demonstrate how software developers can utilize an LLM to generate queries for use with GridDB’s SQL interface to enhance their application. The process used to create and use an LLM for GridDB is as follows:

1. Determine SQL Generation performance of other models and the feasibility of fine tuning these other models.
2. Find the datasets that were used to train the models selected in Step 1.
3. Filter out any queries that are not supported by GridDB and fine-tune the model to ensure accuracy is still reasonable.
4. Create a data set that uses GridDB specific features: time series range selection and Key-Container.
5. Fine-tune the model with our new GridDB specific dataset and evaluate the accuracy as measured by the percentage of the number of responses that matched the human answer in the evaluation data split.
6. Demonstrate inference within a Python Flask application.

Existing Model Evaluation

Closed source, general purpose models such as GPT-4 were immediately dismissed as the bulk of SQL related material they consumed for training would have been for mainstream

databases such as Oracle, Postgres, and SQLite. Not only that, but their closed source nature would make it difficult to train or fine-tune the models for GridDB.

DeFog.AI's SQLCoder model based on LLAMA was tested and performed well, but the original models did not support GridDB's time series SQL semantics. Furthermore, the hardware and time requirements to fine-tune or run inference with SQLCoder was not feasible. Likewise, StarCoder was examined and its reported accuracy was deemed to be significantly poorer than SQLCoder while being just as difficult to fine-tune.

The last two models left for consideration were OpenAI's GPT-2 and Google's T5-Small models. After fine tuning both with our selected datasets, T5-Small was generally more accurate and trouble-free while performing fine tuning. Other parties had already used GPT-2 and T5-Small to create SQL generating LLMs and between the two, accuracy was better with T5-Small.

Dataset Selection

Three individual datasets with similar characteristics that had been used to train text to SQL models were found:

- https://huggingface.co/datasets/b-mc2/sql-create-context
- https://huggingface.co/datasets/Clinton/Text-to-sql-v1
- https://huggingface.co/datasets/knowrohit07/know_sql

Between the three datasets, they have nearly 400,000 rows of data. Each row contains:

- A context or the SQL schema
- The question to be converted into a SQL query.
- The answer, the SQL query based on the question and context.

For example:

```
{
  "answer" : "SELECT COUNT(*) FROM head WHERE age > 56",
  "question" : "How many heads of the departments are older than 56 ?"
  "context" : "CREATE TABLE head (age INTEGER)"
}
```

Dataset Filtering

The first problem seen in these third party datasets is that some of the queries do not work with GridDB. A simple script was created to execute the context statement and then the query statement for each row in the dataset, if the query was executed successfully, then it would be saved to the filtered dataset.

```
for line in fd.readlines():
    data = json.loads(line)
    data[answer_name] = re.sub('"', '\'', data[answer_name])
    try:
        for stmt in data[context_name].split(";"):
            stmt = stmt.strip()
            table = stmt.split(" ")[2]
            curs.execute("DROP TABLE IF EXISTS "+table)
            curs.execute(stmt)
    except:
        pass

    try:
        curs.execute(data[answer_name])
        good=good+1
        print(json.dumps({"question": data[question_name], "context":
data[context_name], "answer": data[answer_name]}))

    except:
        bad=bad+1
```

Of the nearly 400,000 queries in the original datasets, 170,000 of them functioned in GridDB and were used to perform initial model fine tuning. The dataset was split 80/10/10 for training, validation, and testing and fine tuning ran on top of the base T5-small model.

Dataset Creation

None of the data in the filtered dataset supports GridDB specific functionality. While GridDB has many unique SQL features not shared with any other database, we wanted to focus on the two most basic.

The first is GridDB's Key-Container data model. Most relational databases store all data in one table, while GridDB recommends splitting data into multiple tables. For example, data for device #1 would be stored in tsdata_1 and data for device #20 in tsdata_20 so the human question of "What is the maximum temperature recorded on device 12?" would be the SQL query of `SELECT max(temp) FROM tsdata_12` instead of `SELECT max(temp) FROM tsdata WHERE device = '12'"` as would be normal in another database.

The second feature we wanted to support was fetching data, in particular aggregations for a given period. GridDB uses the TIMESTAMP() SQL function to compare time series data in a query. For example, to select average temperature in 2024, you would use the query `SELECT avg(temp) from tsdata_12 where ts >= TIMESTAMP('2024-01-01') and ts < TIMESTAMP('2025-01-01')`.

To do this, a new tool was developed that could generate a human question with corresponding SQL query answer based on a given template. Iterating on the template with

random values for the time period, container identifier, aggregation, etc would build a reasonable sized data set to fine tune the model with.

An example input template:

```
{
    "context" : "CREATE TABLE IF NOT EXISTS devices (device_id INTEGER, ts
TIMESTAMP, co DOUBLE, humidity DOUBLE,light BOOL,lpg DOUBLE,motion
BOOL,smoke DOUBLE,temp DOUBLE);",
    "queries" : [
        {
            "columns" : ["co", "humidity", "lpg", "smoke", "temp"],
            "human" : "What is the {HUMAN_AGGREGATE} {COLUMN} ?",
            "sql" : "SELECT {AGGREGATE}({COLUMN}) FROM devices;"
        }
     ]
}
```

Would produce :

```
{"context": "CREATE TABLE IF NOT EXISTS devices (device_id INTEGER, ts
TIMESTAMP, co DOUBLE, humidity DOUBLE,light BOOL,lpg DOUBLE,motion
BOOL,smoke DOUBLE,temp DOUBLE);",
"question": "What is the lowest smoke in 2009 for all devices?",
"answer": "SELECT MIN(smoke) FROM devices WHERE ts >
TIMESTAMP('2009-01-01T00:00:00Z') and ts <
TIMESTAMP('2010-01-01T00:00:00Z');"}
{"context": "CREATE TABLE IF NOT EXISTS devices (device_id INTEGER, ts
TIMESTAMP, co DOUBLE, humidity DOUBLE,light BOOL,lpg DOUBLE,motion
BOOL,smoke DOUBLE,temp DOUBLE);",
"question": "What is the highest humidity in June 2011 for all devices?",
"answer": "SELECT MAX(humidity) FROM devices WHERE ts >
TIMESTAMP('2011-06-01T00:00:00Z') and ts <
TIMESTAMP('2011-07-01T00:00:00Z');"}
```

We created five to six templated queries that both did and did not use the TIMESTAMP() function for five different contexts which both did and did not use multiple tables per the Key-Container model and then with the dataset creation tool, generated 100 different question/answer pairs per query for a total of 3600 queries. As automatic dataset splitting resulted in a disproportionate amount of one context over another in the test dataset, a second test dataset was generated with only a single question/answer pair for each templated query.

Fine Tuning

For both the filtered data set and the generated GridDB data set, each training data item combined into a single string of the format:

```
Tables:
{context}

Question:
{question}
Answer:
```

The above string is then tokenized using the HuggingFace AutoTokenizer and used as the input identifier while the answer is tokenized as the labels. After the dataset has been tokenized, it is trained using HuggingFace's Trainer library.

Additional tokens for < and <= need to be added to the tokenizer otherwise those symbols with the SQL statements would be ignored by the model during training.

```
def tokenize_function(example):

    start_prompt = "Tables:\n"
    middle_prompt = "\n\nQuestion:\n"
    end_prompt = "\n\nAnswer:\n"

    data_zip = zip(example['context'], example['question'])
    prompt = [start_prompt + context + middle_prompt + question +
end_prompt for context, question in data_zip]
    example['input_ids'] = tokenizer(prompt, padding='max_length',
truncation=True, return_tensors="pt").input_ids
    example['labels'] = tokenizer(example['answer'], padding='max_length',
truncation=True, return_tensors="pt").input_ids

    return example


finetuned_model = AutoModelForSeq2SeqLM.from_pretrained(model_name,
torch_dtype=torch.bfloat16)
tokenizer = AutoTokenizer.from_pretrained(tok_model_name)
tokenizer.add_tokens(['<=', '<= ', ' <=', ' <', '<', '< ', '>= ', ' >=',
'>='])
finetuned_model.resize_token_embeddings(len(tokenizer))

tokenized_datasets = dataset.map(tokenize_function, batched=True)
```

```
output_dir = f'./sql-training-{str(int(time.time()))}'

training_args = TrainingArguments(
    output_dir=output_dir,
    learning_rate=5e-3,
    num_train_epochs=2,
    per_device_train_batch_size=8,     # batch size per device during
training
    per_device_eval_batch_size=8,      # batch size for evaluation
    weight_decay=0.01,
    logging_steps=50,
    evaluation_strategy='steps',        # evaluation strategy to adopt
during training
    eval_steps=500,                     # number of steps between
evaluation
)

trainer = Trainer(
    model=finetuned_model,
    args=training_args,
    train_dataset=tokenized_datasets['train'],
    eval_dataset=tokenized_datasets['validation'],
)

trainer.train()
```

Using an AMD Ryzen Threadripper 2990WX with an NVIDIA 4070GTX, training took approximately 3-4 hours to complete for the filtered dataset and under an hour to complete for the generated dataset.

Evaluation

Using either the 10% test split of the training dataset or the generated test dataset, the same tokenization method was used to build input for the model. The output answer was generated for every input and compared using HuggingFace's ROUGE evaluation library.

```
try:
    for stmt in data[context_name].split(";"):
        stmt = stmt.strip()
        table = stmt.split(" ")[2]
        curs.execute("DROP TABLE IF EXISTS "+table)
        curs.execute(stmt)
except:
```

```
    pass

try:
    curs.execute(data[answer_name])
    good=good+1
    print(json.dumps({"question": data[question_name], "context":
data[context_name], "answer": data[answer_name]}))

except:
    bad=bad+1
```

This evaluation was performed for both the original filtered data set and also the generated GridDB specific data set and ROUGE metrics were gathered. ROUGE or Recall-Oriented Understudy for Gisting Evaluation is a set of metrics used to evaluate text transformation or summarization models by comparing human generated baseline answer versus the model generated response. Each ROUGE metric varies from 0 to 1, with 1 being a perfect match.

Metric	Filtered Queries	GridDB Specific Queries
ROUGE-1	0.9220341258369449	0.893189189189189
ROUGE-2	0.8328271928176021	0.8556992481203007
ROUGE-L	0.9039756047111251	0.8807387387387388

- ROUGE-1 measures the overlap of the words between the original and infered answer.
- ROUGE-2 refers to the overlap of pairs of words between the reference and infered answer.
- ROUGE-L measures the longest sequence of words between the reference and infered answer that match.

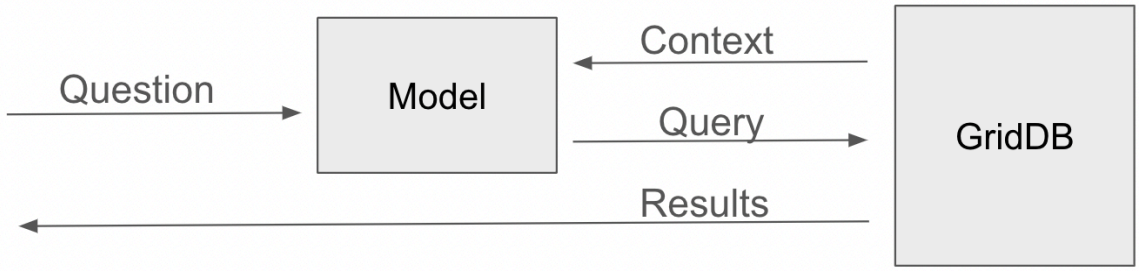
Application Implementation

There are many ways to integrate the LLM into an application. LLM inference could be performed on the edge on the user's device which would allow for greater scalability but also much higher end user system requirements.

If the inference is performed on the server side, it can be bundled into the current application or as a separate service that communicates with the current application. This

would allow inference to run on dedicated high performance instances and thus inference would have minimal impact on the existing application's performance.

We will directly bundle the LLM into our application into the demo for simplicity's sake.



Now adding the code to use the model in your application is straight forward. The context can be fetched using GridDB's NoSQL API:

```
containers = []
x = 0
while x < gridstore.partition_info.partition_count:
    containers.extend(gridstore.partition_info.get_container_names(x, 0))
    x=x+1

conts_and_schemas = {}
for cont in containers:
    col_list = gridstore.get_container_info(cont).column_info_list
    schema = {}
    for row in col_list:
        schema[row[0]] = type_mapping(row[1])
    conts_and_schemas[cont] = schema

create_stmts = []
for key in conts_and_schemas:
    create_stmts.append(create_table_statement(key,
conts_and_schemas[key]))
return create_stmts
```

Inference for a single question is performed in a similiar fashion to how evalution was performed.

```
model = AutoModelForSeq2SeqLM.from_pretrained("griddb_model_2_epoch")
tokenizer = AutoTokenizer.from_pretrained("t5-small")
def translate_to_sql_select(context, question):
    prompt = f"""Tables:
```

```
{context}
Question:
{question}
Answer:
"""
    input_ids = tokenizer.encode(prompt, return_tensors="pt")
    outputs = model.generate(input_ids)
    sql_query = tokenizer.decode(outputs[0], skip_special_tokens=True)
    return sql_query
```

Finally a Flask route gets the local context, calls the model, executes the query, and returns the response.

```
@app.route('/nlquery')
def nlquery():
    question = request.args.get('question')
    context = get_local_context()
    query = translate_to_sql_select(context,question)
    curs = conn.cursor()
    try:
        curs.execute(query)
        rows = curs.fetchall()
        return json.dumps(rows)
    else:
        abort(400, 'Generated query was not successful')
```

While the model is easily incorporated into any Flask or other Python application as shown, scalability may be difficult as each LLM invocation takes approximately 500 milliseconds using an AMD Ryzen Threadripper and NVIDIA 4070GTX. There are other projects such as https://github.com/Ki6an/fastT5 that will greatly improve the scalability of the GridDB LLM model.

Conclusion

We hope the process of creating a training dataset, performing the training and using the resulting LLM within an application to query your data was insightful and educational.

Using LLM, end users including IoT device owners, corporate analysts, managers, customer service, and others are able to query data stored in GridDB without having to know SQL. While the queries used to demonstrate the LLM in this project are relatively simple, the model appears to be extensible to other query types and methods. Furthermore, the T5-small model is efficient to train, not requiring large investments in hardware to to train or run inference on.

In the future, with a larger, more diverse training dataset and advancements even in the base model performing natural language queries will become even more commonplace and accurate. The source code used in the project is available at

https://github.com/griddbnet/sql_llm_model. The finished model can be downloaded from HuggingFace https://huggingface.co/griddbnet.