Code Generation Using LLMs and Evaluation

George Mason University
AIT726-DL1| Prof. Lindi Liao

Mukesh Rajmohan
George Mason University
Fairfax, Virginia.
mrajmoha@gmu.edu

Praneeth Ravirala
George Mason University
Fairfax, Virginia.
praviral@gmu.edu

Utkarsh Jayesh Desai
George Mason University
Fairfax, Virginia.
udesai4@gmu.edu

Abstract:

The rise of Large Language Models (LLMs) such as OpenAI's Codex and GPT has revolutionized the field of automated code generation, offering solutions to a wide range of programming challenges. However, despite their remarkable advancements, these models face significant limitations when generating code for complex tasks that require multi-step reasoning, contextual awareness, and logical coherence. Challenges such as syntactic errors, incomplete implementations, and logical inconsistencies often arise in single-pass generation approaches, particularly for intricate programming problems. This leads to four main problems: code modularity, code maintainability and code security.

This paper introduces a novel framework leveraging hierarchical decomposition to overcome these challenges especially modularity and maintainability. Inspired by human problem-solving strategies, hierarchical decomposition involves breaking a programming task into smaller, logically distinct subtasks, solving each subtask independently, and integrating the solutions into a cohesive final output. By incorporating this framework into the workflow of LLMs, we aim to enhance their code generation capabilities in terms of accuracy, modularity, and logical consistency. This structured method significantly reduced common errors, such as variable mismanagement and incomplete implementations, while allowing for targeted debugging and traceability of generated code.

The findings of this study highlight the potential of hierarchical decomposition to address the inherent limitations of LLMs in code generation, offering a scalable and robust solution for complex programming tasks. Future work will focus on automating the decomposition process and extending the framework to domain-specific applications such as medical and financial coding challenges.

Keywords: Code Generation, Large Language Models, Hierarchical Decomposition, AI Evaluation, Programming Tasks

Introduction:

This is the era of Large Language Models, and we see a wide range of specialities where they are employed. Once such field is Code generation, where a Large Language Model provides automated solutions for various programming challenges. It opens a wide world of opportunities for aiding the programmers increase efficiency and non-programmers to do certain operations without the need for hiring specialised personal. We have seen models like OpenAI's Codex, GPT, Claude's Sonnet, etc perform exceptionally in the field.

But we still face many situations where the generated code leads to error as it is not able to perfectly adapt to the requirements of the user. We see similar drawbacks in security and maintainability of the code. We often encounter difficulties like logical inconsistencies in multi-step tasks as well as modularity and scalability issues for larger code requirements. Such examples are quite frequent when dealing with scenarios of generating domain specific software or debugging complex algorithms.

In this paper we are trying to utilize the approach of Hierarchical Decomposition as a strategy for mitigating these issues and improving modularity and maintainability of the code. We decided to start with SQL code generation as there has been a lot of work done in the field which gives us space for comparison and the simplicity of the SQL language makes it easier to implement. This method mirrors the learning method of humans in that a complex question is divided into simpler subparts making it easier to understand as well as remember. Decomposing the query into parts

makes it easier for the Language Model to solve complex questions at the same time improving the efficiency as each segment of the code is targeted separately leading to improved aggregated code.

The primary focus of this study is to understand the effectiveness of Hierarchical Decomposition to improve Modularity, Maintainability, accuracy and Logical Coherence of the generated code. In this paper we have decided to take simple T5 model as our baseline model to compare with our proposed solution. We decided to select T5 as it is a bidirectional encoder and a unidirectional encoder having great results for text generation. We will be using SQL-create-context dataset which is a mixture of popular WikiSQL dataset and Spider dataset.

In this paper we will be introducing prior studies done on the subject, our baseline model, our proposed model, how it performs, its advantages and its inadequacies, the experimental results, their analysis, and finally a conclusion and possibilities for future work.

Related Work:

Several approaches have been suggested for code generation by LLMs. The state-of-the-art models, proposed in the works discussed below, show great promise but also a few notable limitations.

1.**Content-Enhanced BERT-based Text-to-SQL Generation [1]**

It focuses on enhancing the process of text-to-SQL by making use of table content and structure for more accurate query generation. Indeed, the authors incorporate table-specific knowledge and achieve radical improvements of model performance in SQL-related tasks. This approach involves encoding table structures for SQL generation using BERT models. Furthermore, it doesn't take into consideration the broader challenge of code generation for multiple languages or modularity and security.

2.**CodexDB: Generating Code for SQL Processing Using GPT-3 Codex [2]**

CodexDB provides an SQL processing engine powered by the GPT-3 Codex model, which can be customized using natural language instructions. This system then effectively breaks down comprehensive SQL queries into simple steps. However, this system basically focuses on the customization of query processing and the generation of SQL, without focusing on code security, maintainability, or modular programming principles.

3.**A Review on Code Generation with LLMs: Application and Evaluation [3]**

It had summarized the potentials that LLMs have gained so far in various software engineering tasks like code generation, code completion, and automatic program repair. This review paper further made the following important remarks: while the LLMs such as Codex generate syntactically correct code that often runs and produces the intended functionality, the security, maintainability, and modularity of generated code were not considered in the reviews. The major portion of research in LLM-based code generation focuses on functional correctness rather than aspects related to security and maintainability.

4.**GitHub Copilot AI Pair Programmer: Asset or Liability? [4]**

GitHub Copilot, built on OpenAI Codex, aids in automatic code generation and completion. While it performs well in solving fundamental algorithmic problems and offers competitive suggestions compared to human programmers, many of its solutions are buggy or non-reproducible. The research primarily highlights Copilot's strengths in code functionality but notes significant gaps in ensuring secure, maintainable, and modular code, especially for novice developers who may struggle to filter out flawed suggestions.

The Dataset:

For this project we have decided to focus on SQL to implement the working of our proposed solution of Hierarchical Decomposition for code generation. Here we use the following dataset:

1. **SQL-Create-Context-Database [5]**

   This dataset focuses on SQL query generation based on natural language context, which will help the model

understand and generate secure SQL queries.

This dataset provides a comprehensive range of scenarios for question and answer regarding SQL queries.

The Approaches/Methods:

Framework:

The Hierarchical Decomposition of SQL queries include the following steps:

- **Stage 1 - Component Identification**
  Identifies high-level SQL components
- **Stage 2 - SQL Detail Filling**
  Predicts details for each component
- **Stage 3 - Query Construction**
  Combines details into a valid query

Hierarchical decomposition is a modular approach for complex NLP tasks, such as translating natural language questions into SQL queries. This approach breaks down the task into multiple stages, each focused on a specific aspect, improving robustness, interpretability, and maintainability.

Algorithms/Approaches:

- **Existing Algorithms/Baseline Solution:**

1. **Dataset Loading and Preprocessing**
   A subset of 10,000 samples was imported from the SQL-Create-Context dataset using Pandas. This was done due to computational resource limitations.
2. **Data Splitting**
   The dataset was divided into 8,000 training samples and 2,000 testing samples using Sklearn's train_test_split function.
3. **Feature Encoding**
   The "question" and "context" were combined as input features. The "answer" field was designated as the output label.
4. **Data Preparation for Training**
   The data was converted into a tensor format suitable for PyTorch, with a batch size of 2. This batch size was chosen to optimize training performance and memory utilization.
5. **Training the T5 Model**
   The T5 model was trained using the Adam optimizer to minimize the loss function. Step-by-step explanations were added as comments within the training code to improve clarity and maintainability.
6. **Model Testing**
   The model's performance was evaluated using a subset of the test data to manage memory constraints. Predicted SQL queries were decoded using the T5 tokenizer to generate the final SQL output.

- **Hierarchical Decomposition:**

Our proposed approach improves upon baseline methods by introducing decomposition at the core of the workflow.

I. **Subtask Creation**
We have decided to divide the SQL query into three main parts: Firstly, Data Retrieval which includes SELECT and FROM statements, Data Filtering which includes WHERE statement and finally Data Processing which includes various commands like ORDER BY, GROUP BY, etc. Each subtask was designed to represent a distinct logical unit. This made the generation process explainable and traceable.

II. **Improved Modularity**
By solving each subtask independently, we reduced the complexity of prompt formulation and allowed the LLM to focus on solving simpler problems sequentially.

III. **Error Mitigation**
Hierarchical decomposition inherently minimizes the risk of logical inconsistencies. If one subtask's solution is incorrect, it can be addressed independently without requiring regeneration of the entire solution. This is also the case for overfitting and underfitting scenarios while training. Each segment can be dealt with independently which leads to lower computational resources

required while retraining and dealing with errors. This helps make it easier for the model to stay up to date with newer scenarios and increases its ability to learn new things.

IV. **SW/HW Development Platforms**

For this project we used:

Software: Python, T5 model, and Jupyter Notebooks for experimentation.

Hardware: Experiments were conducted on a GPU-enabled server for faster model inference.

Experimental Results and Analysis:

Baseline Solution:

Step 1: Importing Required Libraries such as pandas, pytorch and transformers

```
#Importing Required Libraries
import pandas as pd
from sklearn.model_selection import train_test_split
import torch
from torch.utils.data import DataLoader, TensorDataset, RandomSampl
from transformers import T5Tokenizer, T5ForConditionalGeneration, A
```

Step2: Importing the dataset using pandas, due to insufficient computational resources in training the model we have taken subset of the dataset containing 10000 samples.

```
[4] #Importing the dataset
data=pd.read_csv('/content/drive/MyDrive/Colab Notebooks/sql_code_dataset.csv')

[5] #Retrieving the structure of the dataset
data.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 78577 entries, 0 to 78576
Data columns (total 3 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   question  78577 non-null  object
 1   context   78577 non-null  object
 2   answer    78577 non-null  object
dtypes: object(3)
memory usage: 1.8+ MB

#Due to limited computing resources, we are taking subset containing 10000 samples
data1=data[:10000]
```

Step3: Using Sklearn library we have split the dataset into training and testing where training data has 8000 samples and testing set has 2000 samples.

```
#Splitting Dataset into Training and Testing with 80% training data and 20% Testing Data
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(data1[['question','context']],
                                                    data1['answer'], test_size=0.2, random_sta

[8] #Structure of Train Data
x_train.info()

<class 'pandas.core.frame.DataFrame'>
Index: 8000 entries, 9254 to 7270
Data columns (total 2 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   question  8000 non-null   object
 1   context   8000 non-null   object
dtypes: object(2)
memory usage: 187.5+ KB
```

Step4: Splitting the "question" and "context" features, they have been encoded combinedly for training and testing to better understand the structure of sql query.

```
#Splitting the Input "Question" and "Context" to give for Tokenizer as a pair
context_train=x_train['context']
context_test=x_test['context']
x_train=x_train['question']
x_test=x_test['question']

[10] device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

[16] #Defining T5 Transformer Model and its tokenizer for encoding and decoding operations.
T5Tokenizer = T5Tokenizer.from_pretrained('t5-base')
T5Model = T5ForConditionalGeneration.from_pretrained('t5-base').to(device)

/usr/local/lib/python3.10/dist-packages/transformers/tokenization_utils_base.py:1601: FutureWarning: `clean_up_tokenization_s
  warnings.warn(

#Tokenising input "question" and "context" pair into tokens and encoding tokens for both training and testing datasets.
#The returned encoded data is in tensor 2D list format and each each list contains max 512 tokens.
x_train_encoded = T5Tokenizer(x_train.tolist(), context_train.tolist(), padding=True, truncation=True,
                              return_tensors='pt', max_length=512).to(device)
x_test_encoded = T5Tokenizer(x_test.tolist(), context_test.tolist(), padding=True, truncation=True,
                             return_tensors='pt', max_length=512).to(device)
```

Step5: Similarly encoding the label('answer') data respectively for training.

```
[19] #Similarly tokenising and encoding the train label
y_train_encoded = T5Tokenizer(y_train.tolist(),padding=True, truncation=True, return_tensors='pt', max_length=512).to(device)

#The Encoded data contains the ids of tokens and attention mask which describes the importance of tokens.
y_train_encoded

{'input_ids': tensor([[    3, 23143, 14196,  ...,     0,     0,     0],
        [    3, 23143, 14196,  ...,     0,     0,     0],
        [    3, 23143, 14196,  ...,     0,     0,     0],
        ...,
        [    3, 23143, 14196,  ...,     0,     0,     0],
        [    3, 23143, 14196,  ...,     0,     0,     0],
        [    3, 23143, 14196,  ...,     0,     0,     0]], device='cuda:0'), 'attention_mask': tensor([[1, 1, 1,  ..., 0, 0, 0],
        [1, 1, 1,  ..., 0, 0, 0],
        [1, 1, 1,  ..., 0, 0, 0],
        ...,
        [1, 1, 1,  ..., 0, 0, 0],
        [1, 1, 1,  ..., 0, 0, 0],
        [1, 1, 1,  ..., 0, 0, 0]], device='cuda:0')}
```

Step6: Transforming the encoded train data to form a tensor dataset suitable for pytorch training function. Followed by Randomly sampling data samples and forming batches based on given batch size which is 2 for our case. We have used Adam Optimizer to updated weights and minimize loss function.

```
#Transform the input and label pair to a tensor dataset supported by pytorch
Transform_Train_Tensor = TensorDataset(x_train_encoded['input_ids'], x_train_encoded['attention_mask'],
                                       y_train_encoded['input_ids'])
#Randomly Sample the dataset to batches for parallel processing based on given batch size
Sample_Train_Tensor = RandomSampler(Transform_Train_Tensor)
Load_Train_Batches = DataLoader(Transform_Train_Tensor, sampler=Sample_Train_Tensor, batch_size=8)
#Defining Optimizer to update the weights and minimising the loss function
T5ModelOptimizer = torch.optim.AdamW(T5Model.parameters(), lr=2e-5)
```

Step7: Training the T5 Model. Explanation of Steps are given as Comments.

```
# Defining the training function which does the following:
# Set Epochs to 2
#Iterating through batches
#For each batch computing weights based on input_id, attention_mask and label of that corresponding batch
#computing loss function for the corresponding batch
#computing gradient and updating weights by backpropogating to minimize overall loss function.
T5Model.train()
for i in range(2):
    epoch_loss = 0
    for b in Load_Train_Batches:
        input_id_batch=b[0]
        attention_mask_batch=b[1]
        label_batch = b[2]
        outputs_weights_batch=T5Model(input_ids=input_id_batch,attention_mask=attention_mask_batch,labels=label_batch)
        batch_loss = outputs_weights_batch.loss
        epoch_loss = epoch_loss+batch_loss.item()
        batch_loss.backward()
        T5ModelOptimizer.step()
        T5Model.zero_grad()
        torch.cuda.empty_cache()

    Average_Epoch_Loss = epoch_loss / len(Load_Train_Batches)
    print('Epoch '+str(i)+' Loss = '+str(Average_Epoch_Loss))

Passing a tuple of `past_key_values` is deprecated and will be removed in Transformers v4.48.0. You should pass an instance of
Epoch 0 Loss = 0.4777791198454797
Epoch 1 Loss = 0.07594330419041216
```

Step8: Using Subset of encoded test data and use it for predicting the test code because predicting 2000 codes is making system go out of memory. We are predicting the code using pytorch's eval function and sending encoded test data as input parameter.

```
#Subset of test data with 500 samples for predicting the code.
x_test_encoded1=x_test_encoded[:500]
```

```
[ ] x_test_encoded1
```

```
{'input_ids': tensor([[ 366,  968, 5865,  ...,    0,    0,    0],
        [ 363,   19,    8,  ...,    0,    0,    0],
        [8298,   66,    8,  ...,    0,    0,    0],
        ...,
        [2588,    8, 3056,  ...,    0,    0,    0],
        [ 570, 3056,   13,  ...,    0,    0,    0],
        [ 549,  107,    3,  ...,    0,    0,    0]], device='cuda
 'attention_mask': tensor([[1, 1, 1,  ..., 0, 0, 0],
        [1, 1, 1,  ..., 0, 0, 0],
        [1, 1, 1,  ..., 0, 0, 0],
        ...,
        [1, 1, 1,  ..., 0, 0, 0],
        [1, 1, 1,  ..., 0, 0, 0],
        [1, 1, 1,  ..., 0, 0, 0]], device='cuda:0')}
```

Step9: After receiving the predicted code in encoded format, we decode the output to generate sql query using T5 tokenizer.

```
[ ] T5Model.eval()
    with torch.no_grad():
        all_outputs = []
        for i in range(500):
            generated_output = T5Model.generate(
                input_ids=x_test_encoded1['input_ids'][i:i+1].to(device),
                attention_mask=x_test_encoded1['attention_mask'][i:i+1].to(device),
                max_length=100,
                num_beams=5,
                early_stopping=True
            )
            all_outputs.append(generated_output)
            torch.cuda.empty_cache()
```

Results and Analysis of Baseline Model:

From the below actual test code and generated test code outputs we can depict that T5 model can be able to efficiently handle text-text sequences and accurately generate sql queries.

```
predicted_code=[]
for i in all_outputs:
  pred=T5Tokenizer.decode(i[0], skip_special_tokens=True, clean_up_tokenization_spaces=True)
  predicted_code.append(pred)
for input_text, code in zip(y_test, predicted_code):
    print(f"Input: {input_text}\nGenerated Code:\n{code}\n")
```

```
Input: SELECT assembled FROM table_1827690_4 WHERE elected = "1472"
Generated Code:
SELECT assembled FROM table_1827690_4 WHERE elected = "1472"

Input: SELECT willow_canyon FROM table_11340432_1 WHERE shadow_ridge = "Stallion"
Generated Code:
SELECT willow_canyon FROM table_11340432_1 WHERE shadow_ridge = "Stallion"

Input: SELECT fname, lname FROM authors ORDER BY lname
Generated Code:
SELECT fname, lname FROM authors WHERE alphabetical order of last names

Input: SELECT gvm__kg__technical_capacity FROM table_11497980_1 WHERE model = "15.180E"
Generated Code:
SELECT gvm__kg__technical_capacity FROM table_11497980_1 WHERE model = "15.180e"

Input: SELECT school_club_team FROM table_10015132_16 WHERE years_in_toronto = "1995-96"
Generated Code:
SELECT school_club_team FROM table_10015132_16 WHERE years_in_toronto = 1995-96

Input: SELECT minor_sponsors FROM table_187239_1 WHERE kit_manufacturer = "Olympikus" AND main_sponsor = "Bat
Generated Code:
SELECT minor_sponsors FROM table_187239_1 WHERE kit_manufacturer = "batavo" AND main_sponsor = "olympikus"
```

Proposed Model:

Stage 1: Query Component Identification

Objective: Identify the high-level components (SELECT, FROM, WHERE, ORDER BY, etc.) required for the SQL query.

- **Input**: Natural language question and schema context.
- **Output**: A structured list of SQL components to be included in the query.

- **Advantages**:

  - Understanding the structure of the SQL query is foundational to query generation.
  - This stage reduces ambiguity by focusing on identifying the query's overall structure before adding specific details.

- **Implementation**:

  - Extraction of SQL components, select_columns, from_tables and where_conditions based on the target query.
  - A classification model (e.g., BERT or T5) is trained to predict which SQL components are relevant based on the question and schema.
  - Example:
    - **Input**: "How many departments have a budget greater than $1 million?"
    - **Output**: ["SELECT", "FROM", "WHERE"].

- **Execution:**

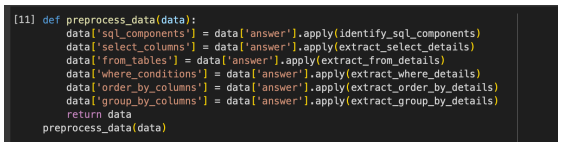
```
[ ] def identify_sql_components(answer):
        components = []
        if 'SELECT' in answer:
          components.append('SELECT')
        if 'FROM' in answer:
          components.append('FROM')
        if 'WHERE' in answer:
          components.append('WHERE')
        if 'ORDER BY' in answer:
          components.append('ORDER BY')
        if 'GROUP BY' in answer:
          components.append('GROUP BY')
        return components
```

Here we are extracting the SQL components from the queries form the answer column in the dataset to decompose it into various elements.

```
[ ] def extract_select_details(answer):
        if 'SELECT' in answer and 'FROM' in answer:
            return answer.split('FROM')[0].replace('SELECT', '').strip()
        return ''

[ ] def extract_from_details(answer):
        if 'FROM' in answer:
            parts = answer.split('FROM')[1].split('WHERE')[0] if 'WHERE' in answer else answer.split('FROM')[1]
            return parts.split()[0].strip()
        return ''

[ ] def extract_where_details(answer):
        if 'WHERE' in answer:
            return answer.split('WHERE')[1].split('ORDER BY')[0] if 'ORDER BY' in answer else answer.split('WHERE')[1]
        return ''
```

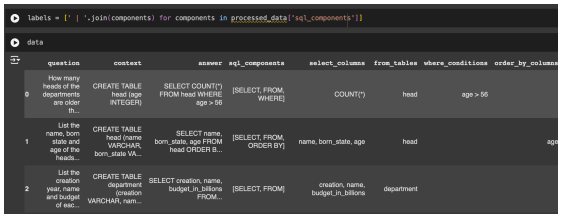
```
[10] def extract_group_by_details(answer):
        if 'GROUP BY' in answer:
            return answer.split('GROUP BY')[1].strip()
        return ''
```

Using the above 5 functions we are mapping each component of the query with the dataset.



```
[11] def preprocess_data(data):
        data['sql_components'] = data['answer'].apply(identify_sql_components)
        data['select_columns'] = data['answer'].apply(extract_select_details)
        data['from_tables'] = data['answer'].apply(extract_from_details)
        data['where_conditions'] = data['answer'].apply(extract_where_details)
        data['order_by_columns'] = data['answer'].apply(extract_order_by_details)
        data['group_by_columns'] = data['answer'].apply(extract_group_by_details)
        return data
    preprocess_data(data)
```

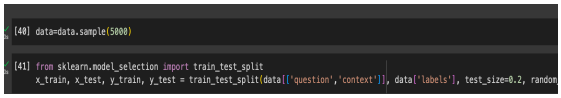
Applying the above functions to our dataset to get the required values.



```
labels = [' | '.join(components) for components in processed_data['sql_components']]
data
```

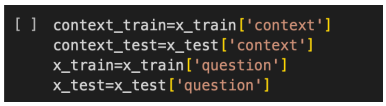
	question	context	answer	sql_components	select_columns	from_tables	where_conditions	order_by_columns
0	How many heads of the departments are older th...	CREATE TABLE head (age INTEGER)	SELECT COUNT(*) FROM head WHERE age > 56	[SELECT, FROM, WHERE]	COUNT(*)	head	age > 56	
1	List the name, born state and age of the heads...	CREATE TABLE head (name VARCHAR, born_state VA...	SELECT name, born_state, age FROM head ORDER B...	[SELECT, FROM, ORDER BY]	name, born_state, age	head		age
2	List the creation year, name and budget of eac...	CREATE TABLE department (creation VARCHAR, nam...	SELECT creation, name, budget_in_billions FROM...	[SELECT, FROM]	creation, name, budget_in_billions	department		

Concatenating the extracted components for classification.

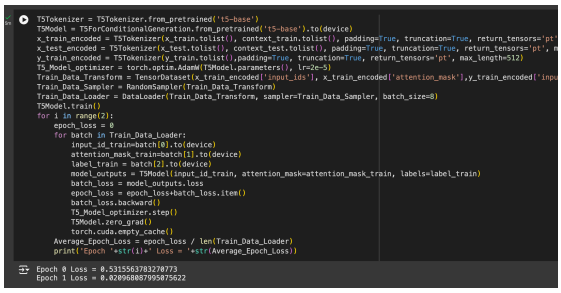


```
[40] data=data.sample(5000)

[41] from sklearn.model_selection import train_test_split
     x_train, x_test, y_train, y_test = train_test_split(data[['question','context']], data['labels'], test_size=0.2, random
```

Splitting the data into train and test subsets.



```
[ ]  context_train=x_train['context']
     context_test=x_test['context']
     x_train=x_train['question']
     x_test=x_test['question']
```



```
T5Tokenizer = T5Tokenizer.from_pretrained('t5-base')
T5Model = T5ForConditionalGeneration.from_pretrained('t5-base').to(device)
x_train_encoded = T5Tokenizer(x_train.tolist(), context_train.tolist(), padding=True, truncation=True, return_tensors='pt'
x_test_encoded = T5Tokenizer(x_test.tolist(), context_test.tolist(), padding=True, truncation=True, return_tensors='pt', m
y_train_encoded = T5Tokenizer(y_train.tolist(),padding=True, truncation=True, return_tensors='pt', max_length=512)
T5_Model_optimizer = torch.optim.AdamW(T5Model.parameters(), lr=2e-5)
Train_Data_Transform = TensorDataset(x_train_encoded['input_ids'], x_train_encoded['attention_mask'],y_train_encoded['inpu
Train_Data_Sampler = RandomSampler(Train_Data_Transform)
Train_Data_Loader = DataLoader(Train_Data_Transform, sampler=Train_Data_Sampler, batch_size=8)
T5Model.train()
for i in range(2):
    epoch_loss = 0
    for batch in Train_Data_Loader:
        input_id_train=batch[0].to(device)
        attention_mask_train=batch[1].to(device)
        label_train = batch[2].to(device)
        model_outputs = T5Model(input_id_train, attention_mask=attention_mask_train, labels=label_train)
        batch_loss = model_outputs.loss
        epoch_loss = epoch_loss+batch_loss.item()
        batch_loss.backward()
        T5_Model_optimizer.step()
        T5Model.zero_grad()
        torch.cuda.empty_cache()
    Average_Epoch_Loss = epoch_loss / len(Train_Data_Loader)
    print('Epoch '+str(i)+' Loss = '+str(Average_Epoch_Loss))

Epoch 0 Loss = 0.5315563793270773
Epoch 1 Loss = 0.02090808799507562
```

Training T5 model, with baseline parameters but the features are context and question and the target is label column (decomposed SQL components) instead of answer column (single SQL query).

```
x_test_encoded1=x_test_encoded[:500]

T5Model.eval()
with torch.no_grad():
    all_outputs = []
    for i in range(500):
        generated_output = T5Model.generate(
            input_ids=x_test_encoded1['input_ids'][i:i+1].to(device),
            attention_mask=x_test_encoded1['attention_mask'][i:i+1].to(device),
            max_length=100,
            num_beams=5,
            early_stopping=True
        )
        all_outputs.append(generated_output)
        torch.cuda.empty_cache()

predicted_code0=[]
for i in all_outputs:
  pred=T5Tokenizer.decode(i[0], skip_special_tokens=True, clean_up_tokenization_spaces=True)
  predicted_code0.append(pred)
for input_text, code in zip(y_test, predicted_code0):
    print(f"Input: {input_text}\nGenerated Code:\n{code}\n")

Input: SELECT | FROM | WHERE
Generated Code:
SELECT | FROM | WHERE

Input: SELECT | FROM | WHERE
Generated Code:
SELECT | FROM | WHERE

Input: SELECT | FROM | WHERE
Generated Code:
SELECT | FROM | WHERE
```

Taking subset of test set with 500 instances, predicting SQL components with question and context as input. These predicted components will be used for SQL query aggregation and reconstruction at the end.

Stage 2: SQL Detail Filling

Objective: Predict the specific details for each identified SQL component.

- **Input**: Natural language question, schema context, and the components identified in Stage 1.
- **Output**: Filled details for each SQL component, such as columns for SELECT, table names for FROM, and conditions for WHERE.

- **Advantages**:

  - This stage ensures modularity by focusing on individual components rather than the entire query at once.
  - Allows for specialized models to handle different SQL components, improving accuracy and reducing complexity.

- **Implementation**:

  - Each SQL component is treated as a separate task:
    - SELECT: Predict the columns to be retrieved.

- FROM: Predict the relevant table(s).
- WHERE: Predict the conditions or filters.

- Models like T5 are fine-tuned for each task to predict the details based on the input context.
- Example:
  - **Input for SELECT**: "How many departments have a budget greater than $1 million? | Schema: department (budget)"
  - **Output**: COUNT(*).
  - **Input for FROM**: Same input but predicting the table name
  - **Output**: dept_budget
  - **Input for WHERE**: Same input but predicting the conditions.
  - **Output**: budget > 1000000.

- **Execution:**

```
subtasks = ["select_columns", "from_tables", "where_conditions","order_by_columns","group_by_

select_data = []
from_data = []
where_data = []
order_by_data = []
group_by_data = []

for subtask in subtasks:
    for _, row in data.iterrows():
        input_text = f"Fill {subtask}: {row['question']} | {row['context']} | {row['sql_compo
        target_text = row[subtask] if not pd.isna(row[subtask]) else "NULL"
        entry = {"input": input_text, "label": target_text}

        if subtask == "select_columns":
            select_data.append(entry)
        elif subtask == "from_tables":
            from_data.append(entry)
        elif subtask == "where_conditions":
            where_data.append(entry)
        elif subtask == "order_by_columns":
            order_by_data.append(entry)
        elif subtask == "group_by_columns":
            group_by_data.append(entry)

select_df = pd.DataFrame(select_data)
from_df = pd.DataFrame(from_data)
where_df = pd.DataFrame(where_data)
order_by_df = pd.DataFrame(order_by_data)
group_by_df = pd.DataFrame(group_by_data)
```

```
data1=select_df
data2=from_df
data3=where_df
data4=order_by_df
data5=group_by_df

print(data1.head(1))
print(data2.head(1))
print(data3.head(1))
print(data4.head(1))
print(data5.head(1))
```

```
                                                input label
0  Fill select_columns: Name the city for enrollm...  city
                                                input             l
0  Fill from_tables: Name the city for enrollment...  table_273785
                                                input
0  Fill where_conditions: Name the city for enrol...   enrollment
                                                input label
0  Fill order_by_columns: Name the city for enrol...
                                                input label
0  Fill group_by_columns: Name the city for enrol...
```

So, we divide the whole structure of SQL query into subtasks, such as SELECT, FROM, WHERE, etc. We will divide the details of each SQL query into different datasets.

```
[60] from sklearn.model_selection import train_test_split
     x1_train, x1_test, y1_train, y1_test = train_test_split(data1['input'],
                                                             data1['label'], test_size=0.2, random_state=42)

[61] from sklearn.model_selection import train_test_split
     x2_train, x2_test, y2_train, y2_test = train_test_split(data2['input'],
                                                             data2['label'], test_size=0.2, random_state=42)

[62] from sklearn.model_selection import train_test_split
     x3_train, x3_test, y3_train, y3_test = train_test_split(data3['input'],
                                                             data3['label'], test_size=0.2, random_state=42)

[63] from sklearn.model_selection import train_test_split
     x4_train, x4_test, y4_train, y4_test = train_test_split(data4['input'],
                                                             data4['label'], test_size=0.2, random_state=42)

[64] from sklearn.model_selection import train_test_split
     x5_train, x5_test, y5_train, y5_test = train_test_split(data5['input'],
                                                             data5['label'], test_size=0.2, random_state=42)
```

Now, we split the datasets into its respective train and test subparts.

```
[65] x1_train_encoded = T5Tokenizer(x1_train.tolist(), padding=True, truncation=True,
                                    return_tensors='pt', max_length=512)
     x1_test_encoded = T5Tokenizer(x1_test.tolist(), padding=True, truncation=True,
                                   return_tensors='pt', max_length=512)
     y1_train_encoded = T5Tokenizer(y1_train.tolist(),padding=True, truncation=True, return_tensors='pt', max_length=512)

[66] x2_train_encoded = T5Tokenizer(x2_train.tolist(), padding=True, truncation=True,
                                    return_tensors='pt', max_length=512)
     x2_test_encoded = T5Tokenizer(x2_test.tolist(), padding=True, truncation=True,
                                   return_tensors='pt', max_length=512)
     y2_train_encoded = T5Tokenizer(y2_train.tolist(),padding=True, truncation=True, return_tensors='pt', max_length=512)

[67] x3_train_encoded = T5Tokenizer(x3_train.tolist(), padding=True, truncation=True,
                                    return_tensors='pt', max_length=512)
     x3_test_encoded = T5Tokenizer(x3_test.tolist(), padding=True, truncation=True,
                                   return_tensors='pt', max_length=512)
     y3_train_encoded = T5Tokenizer(y3_train.tolist(),padding=True, truncation=True, return_tensors='pt', max_length=512)

     x4_train_encoded = T5Tokenizer(x4_train.tolist(), padding=True, truncation=True,
                                    return_tensors='pt', max_length=512)
     x4_test_encoded = T5Tokenizer(x4_test.tolist(), padding=True, truncation=True,
                                   return_tensors='pt', max_length=512)
     y4_train_encoded = T5Tokenizer(y4_train.tolist(),padding=True, truncation=True, return_tensors='pt', max_length=512)

[69] x5_train_encoded = T5Tokenizer(x5_train.tolist(), padding=True, truncation=True,
                                    return_tensors='pt', max_length=512)
     x5_test_encoded = T5Tokenizer(x5_test.tolist(), padding=True, truncation=True,
                                   return_tensors='pt', max_length=512)
     y5_train_encoded = T5Tokenizer(y5_train.tolist(),padding=True, truncation=True, return_tensors='pt', max_length=512)
```

Similarly, we will encode features and target using tokenizer.

```
[70] Transform_Train_Tensor = TensorDataset(x1_train_encoded['input_ids'], x1_train_encoded['attention_mask'],
                                            y1_train_encoded['input_ids'])
     Sample_Train_Tensor = RandomSampler(Transform_Train_Tensor)
     Load_Train_Batches = DataLoader(Transform_Train_Tensor, sampler=Sample_Train_Tensor, batch_size=8)
     T5ModelOptimizer = torch.optim.AdamW(T5Model.parameters(), lr=2e-5)

[71] T5Model.train()
     for i in range(2):
         epoch_loss = 0
         for batch in Load_Train_Batches:
             input_id_train=batch[0].to(device)
             attention_mask_train=batch[1].to(device)
             label_train = batch[2].to(device)
             model_outputs = T5Model(input_id_train, attention_mask=attention_mask_train, labels=label_train)
             batch_loss = model_outputs.loss
             epoch_loss = epoch_loss+batch_loss.item()
             batch_loss.backward()
             T5_Model_optimizer.step()
             T5Model.zero_grad()
             torch.cuda.empty_cache()
         Average_Epoch_Loss = epoch_loss / len(Load_Train_Batches)
         print('Epoch '+str(i)+' Loss = '+str(Average_Epoch_Loss))

     Epoch 0 Loss = 1.185669342212379
     Epoch 1 Loss = 0.14307400258630515
```

Here we train the T5 model for SELECT clause, to predict SELECT component for a given input.

```
[72] x1_test_encoded1=x1_test_encoded[:500]
     x1_test_encoded1

{'input_ids': tensor([[12607,  1738,   834,  ...,     0,     0,     0],
                      [12607,  1738,   834,  ...,     0,     0,     0],
                      [12607,  1738,   834,  ...,     0,     0,     0],
                      ...,
                      [12607,  1738,   834,  ...,     0,     0,     0],
                      [12607,  1738,   834,  ...,     0,     0,     0],
                      [12607,  1738,   834,  ...,     0,     0,     0]]),
 'attention_mask': tensor([[1, 1, 1,  ..., 0, 0, 0],
                           [1, 1, 1,  ..., 0, 0, 0],
                           [1, 1, 1,  ..., 0, 0, 0],
                           ...,
                           [1, 1, 1,  ..., 0, 0, 0],
                           [1, 1, 1,  ..., 0, 0, 0],
                           [1, 1, 1,  ..., 0, 0, 0]])}
```

```
[73] T5Model.eval()
     with torch.no_grad():
         all_outputs = []
         for i in range(500):
             generated_output = T5Model.generate(
                 input_ids=x1_test_encoded1['input_ids'][i:i+1].to(device),
                 attention_mask=x1_test_encoded1['attention_mask'][i:i+1].to(device),
                 max_length=100,
                 num_beams=5,
                 early_stopping=True
             )
             all_outputs.append(generated_output)
             torch.cuda.empty_cache()
```

```
predicted_code1=[]
for i in all_outputs:
  pred=T5Tokenizer.decode(i[0], skip_special_tokens=True, clean_up_tokenization_spaces=True)
  predicted_code1.append(pred)
for input_text, code in zip(y1_test, predicted_code1):
    print(f"Input: {input_text}\nGenerated Code:\n{code}\n")
```

```
Input: league
Generated Code:
league

Input: manufacturer
Generated Code:
manufacturer

Input: max_power_at_rpm
Generated Code:
max_power_at_rpm

Input: COUNT(party)
Generated Code:
party
```

```
[75] Transform_Train_Tensor = TensorDataset(x2_train_encoded['input_ids'], x2_train_encoded['attention_mask'
                                            y2_train_encoded['input_ids'])
     Sample_Train_Tensor = RandomSampler(Transform_Train_Tensor)
     Load_Train_Batches = DataLoader(Transform_Train_Tensor, sampler=Sample_Train_Tensor, batch_size=8)
     T5ModelOptimizer = torch.optim.AdamW(T5Model.parameters(), lr=2e-5)
```

```
T5Model.train()
for i in range(2):
    epoch_loss = 0
    for batch in Load_Train_Batches:
        input_id_train=batch[0].to(device)
        attention_mask_train=batch[1].to(device)
        label_train = batch[2].to(device)
        model_outputs = T5Model(input_id_train, attention_mask=attention_mask_train, labels=label_train
        batch_loss = model_outputs.loss
        epoch_loss = epoch_loss+batch_loss.item()
        batch_loss.backward()
        T5_Model_optimizer.step()
        T5Model.zero_grad()
        torch.cuda.empty_cache()
    Average_Epoch_Loss = epoch_loss / len(Load_Train_Batches)
    print('Epoch '+str(i)+' Loss = '+str(Average_Epoch_Loss))
```

```
Epoch 0 Loss = 0.0330268563607909
Epoch 1 Loss = 0.011325780261890031
```

```
x2_test_encoded1=x2_test_encoded[:500]
x2_test_encoded1

{'input_ids': tensor([[12607,    45,   834,  ...,     0,     0,     0],
                      [12607,    45,   834,  ...,     0,     0,     0],
                      [12607,    45,   834,  ...,     0,     0,     0],
                      ...,
                      [12607,    45,   834,  ...,     0,     0,     0],
                      [12607,    45,   834,  ...,     0,     0,     0],
                      [12607,    45,   834,  ...,     0,     0,     0]]),
 'attention_mask': tensor([[1, 1, 1,  ..., 0, 0, 0],
                           [1, 1, 1,  ..., 0, 0, 0],
                           [1, 1, 1,  ..., 0, 0, 0],
                           ...,
                           [1, 1, 1,  ..., 0, 0, 0],
                           [1, 1, 1,  ..., 0, 0, 0],
                           [1, 1, 1,  ..., 0, 0, 0]])}
```

Similar training function for FROM component (content) of the SQL query.

```
[79] T5Model.eval()
     with torch.no_grad():
         all_outputs = []
         for i in range(500):
             generated_output = T5Model.generate(
                 input_ids=x2_test_encoded1['input_ids'][i:i+1].to(device),
                 attention_mask=x2_test_encoded1['attention_mask'][i:i+1].to(device),
                 max_length=100,
                 num_beams=5,
                 early_stopping=True
             )
             all_outputs.append(generated_output)
             torch.cuda.empty_cache()
```

```
predicted_code2=[]
for i in all_outputs:
  pred=T5Tokenizer.decode(i[0], skip_special_tokens=True, clean_up_tokenization_spaces=True)
  predicted_code2.append(pred)
```

```
[81] for input_text, code in zip(y2_test, predicted_code2):
         print(f"Input: {input_text}\nGenerated Code:\n{code}\n")
```

```
Input: table_12043148_2
Generated Code:
table_12043148_2

Input: table_name_24
Generated Code:
table_name_24

Input: table_name_90
Generated Code:
table_name_90

Input: table_name_33
Generated Code:
table_name_33

Input: table_27734286_7
Generated Code:
table_27734286_7
```

```
[82] Transform_Train_Tensor = TensorDataset(x3_train_encoded['input_ids'], x3_train_encoded['attention_mask'],
                                            y3_train_encoded['input_ids'])
     Sample_Train_Tensor = RandomSampler(Transform_Train_Tensor)
     Load_Train_Batches = DataLoader(Transform_Train_Tensor, sampler=Sample_Train_Tensor, batch_size=8)
     T5ModelOptimizer = torch.optim.AdamW(T5Model.parameters(), lr=2e-5)
```

```
[83] T5Model.train()
     for i in range(2):
         epoch_loss = 0
         for batch in Load_Train_Batches:
             input_id_train=batch[0].to(device)
             attention_mask_train=batch[1].to(device)
             label_train = batch[2].to(device)
             model_outputs = T5Model(input_id_train, attention_mask=attention_mask_train, labels=label_train)
             batch_loss = model_outputs.loss
             epoch_loss = epoch_loss+batch_loss.item()
             batch_loss.backward()
             T5_Model_optimizer.step()
             T5Model.zero_grad()
             torch.cuda.empty_cache()
         Average_Epoch_Loss = epoch_loss / len(Load_Train_Batches)
         print('Epoch '+str(i)+' Loss = '+str(Average_Epoch_Loss))
```

```
Epoch 0 Loss = 0.1773597779497504
Epoch 1 Loss = 0.08164384141936898
```

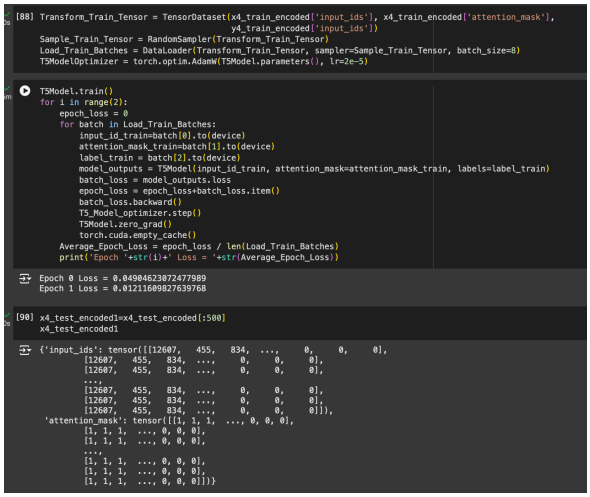
```
x3_test_encoded1=x3_test_encoded[:500]
x3_test_encoded1

{'input_ids': tensor([[12607,   213,   834,  ...,     0,     0,     0],
                      [12607,   213,   834,  ...,     0,     0,     0],
                      ...,
                      [12607,   213,   834,  ...,     0,     0,     0],
                      [12607,   213,   834,  ...,     0,     0,     0],
                      [12607,   213,   834,  ...,     0,     0,     0]]),
 'attention_mask': tensor([[1, 1, 1,  ..., 0, 0, 0],
                           [1, 1, 1,  ..., 0, 0, 0],
                           ...,
                           [1, 1, 1,  ..., 0, 0, 0],
                           [1, 1, 1,  ..., 0, 0, 0],
                           [1, 1, 1,  ..., 0, 0, 0]])}
```

Training for WHERE component (content) of SQL query

```
T5Model.eval()
with torch.no_grad():
    all_outputs = []
    for i in range(500):
        generated_output = T5Model.generate(
            input_ids=x3_test_encoded1['input_ids'][i:i+1].to(device),
            attention_mask=x3_test_encoded1['attention_mask'][i:i+1].to(device),
            max_length=100,
            num_beams=5,
            early_stopping=True
        )
        all_outputs.append(generated_output)
        torch.cuda.empty_cache()
```

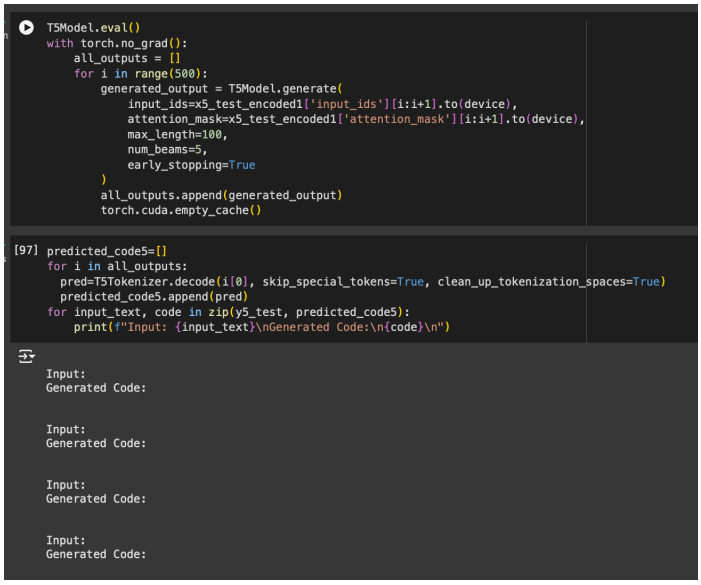
```
predicted_code3=[]
for i in all_outputs:
  pred=T5Tokenizer.decode(i[0], skip_special_tokens=True, clean_up_tokenization_spaces=True)
  predicted_code3.append(pred)
for input_text, code in zip(y3_test, predicted_code3):
    print(f"Input: {input_text}\nGenerated Code:\n{code}\n")
```

```
week  12 AND date = "november 25, 1965"

Input:  number_of_decks > 6
Generated Code:
number_of_decks > 6

Input:  average_attendance = 17148
Generated Code:
average_attendance = "17148"

Input:  draft > 1980 AND player = "shane doan category:articles with hcards"
Generated Code:
draft > 1980 AND player = "Shane Doan Category:Articles with hCards"

Input:  original_artist = "Anna Nalick"
Generated Code:
original_artist = "Anna Nalick"
```

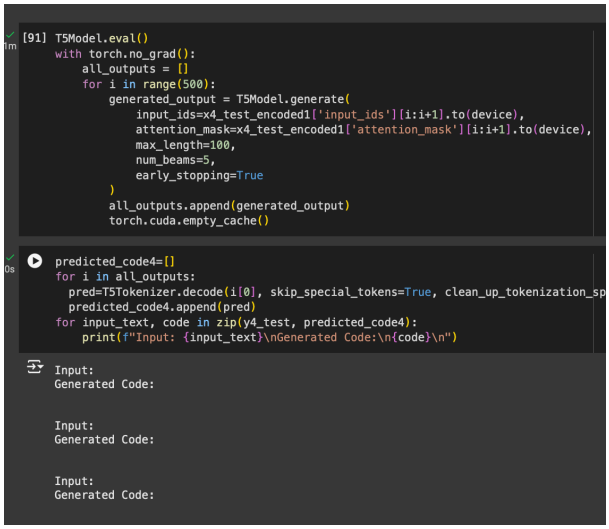


```
[88] Transform_Train_Tensor = TensorDataset(x4_train_encoded['input_ids'], x4_train_encoded['attention_mask'],
                                            y4_train_encoded['input_ids'])
     Sample_Train_Tensor = RandomSampler(Transform_Train_Tensor)
     Load_Train_Batches = DataLoader(Transform_Train_Tensor, sampler=Sample_Train_Tensor, batch_size=8)
     T5ModelOptimizer = torch.optim.AdamW(T5Model.parameters(), lr=2e-5)

     T5Model.train()
     for i in range(2):
         epoch_loss = 0
         for batch in Load_Train_Batches:
             input_id_train=batch[0].to(device)
             attention_mask_train=batch[1].to(device)
             label_train = batch[2].to(device)
             model_outputs = T5Model(input_id_train, attention_mask=attention_mask_train, labels=label_train)
             batch_loss = model_outputs.loss
             epoch_loss = epoch_loss+batch_loss.item()
             batch_loss.backward()
             T5_Model_optimizer.step()
             T5Model.zero_grad()
             torch.cuda.empty_cache()
         Average_Epoch_Loss = epoch_loss / len(Load_Train_Batches)
         print('Epoch '+str(i)+' Loss = '+str(Average_Epoch_Loss))

Epoch 0 Loss = 0.04904623072477989
Epoch 1 Loss = 0.01211609827639768

[90] x4_test_encoded1=x4_test_encoded[:500]
     x4_test_encoded1

{'input_ids': tensor([[12607,   455,   834,  ...,     0,     0,     0],
                      [12607,   455,   834,  ...,     0,     0,     0],
                      [12607,   455,   834,  ...,     0,     0,     0],
                      ...,
                      [12607,   455,   834,  ...,     0,     0,     0],
                      [12607,   455,   834,  ...,     0,     0,     0],
                      [12607,   455,   834,  ...,     0,     0,     0]]),
 'attention_mask': tensor([[1, 1, 1,  ..., 0, 0, 0],
                           [1, 1, 1,  ..., 0, 0, 0],
                           [1, 1, 1,  ..., 0, 0, 0],
                           ...,
                           [1, 1, 1,  ..., 0, 0, 0],
                           [1, 1, 1,  ..., 0, 0, 0],
                           [1, 1, 1,  ..., 0, 0, 0]])}
```

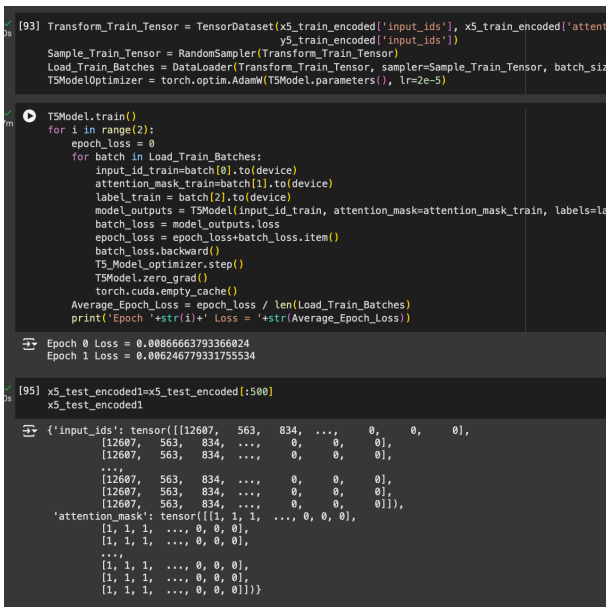
Training for ORDER BY component (content) for SQL query.



```
T5Model.eval()
with torch.no_grad():
    all_outputs = []
    for i in range(500):
        generated_output = T5Model.generate(
            input_ids=x5_test_encoded1['input_ids'][i:i+1].to(device),
            attention_mask=x5_test_encoded1['attention_mask'][i:i+1].to(device),
            max_length=100,
            num_beams=5,
            early_stopping=True
        )
        all_outputs.append(generated_output)
        torch.cuda.empty_cache()

[97] predicted_code5=[]
     for i in all_outputs:
       pred=T5Tokenizer.decode(i[0], skip_special_tokens=True, clean_up_tokenization_spaces=True)
       predicted_code5.append(pred)
     for input_text, code in zip(y5_test, predicted_code5):
         print(f"Input: {input_text}\nGenerated Code:\n{code}\n")

Input:
Generated Code:


Input:
Generated Code:


Input:
Generated Code:


Input:
Generated Code:
```



```
[91] T5Model.eval()
     with torch.no_grad():
         all_outputs = []
         for i in range(500):
             generated_output = T5Model.generate(
                 input_ids=x4_test_encoded1['input_ids'][i:i+1].to(device),
                 attention_mask=x4_test_encoded1['attention_mask'][i:i+1].to(device),
                 max_length=100,
                 num_beams=5,
                 early_stopping=True
             )
             all_outputs.append(generated_output)
             torch.cuda.empty_cache()

     predicted_code4=[]
     for i in all_outputs:
       pred=T5Tokenizer.decode(i[0], skip_special_tokens=True, clean_up_tokenization_sp
       predicted_code4.append(pred)
     for input_text, code in zip(y4_test, predicted_code4):
         print(f"Input: {input_text}\nGenerated Code:\n{code}\n")

Input:
Generated Code:


Input:
Generated Code:


Input:
Generated Code:
```



```
[93] Transform_Train_Tensor = TensorDataset(x5_train_encoded['input_ids'], x5_train_encoded['attent
                                            y5_train_encoded['input_ids'])
     Sample_Train_Tensor = RandomSampler(Transform_Train_Tensor)
     Load_Train_Batches = DataLoader(Transform_Train_Tensor, sampler=Sample_Train_Tensor, batch_siz
     T5ModelOptimizer = torch.optim.AdamW(T5Model.parameters(), lr=2e-5)

     T5Model.train()
     for i in range(2):
         epoch_loss = 0
         for batch in Load_Train_Batches:
             input_id_train=batch[0].to(device)
             attention_mask_train=batch[1].to(device)
             label_train = batch[2].to(device)
             model_outputs = T5Model(input_id_train, attention_mask=attention_mask_train, labels=la
             batch_loss = model_outputs.loss
             epoch_loss = epoch_loss+batch_loss.item()
             batch_loss.backward()
             T5_Model_optimizer.step()
             T5Model.zero_grad()
             torch.cuda.empty_cache()
         Average_Epoch_Loss = epoch_loss / len(Load_Train_Batches)
         print('Epoch '+str(i)+' Loss = '+str(Average_Epoch_Loss))

Epoch 0 Loss = 0.008666663793366024
Epoch 1 Loss = 0.006246779331755534

[95] x5_test_encoded1=x5_test_encoded[:500]
     x5_test_encoded1

{'input_ids': tensor([[12607,   563,   834,  ...,     0,     0,     0],
                      [12607,   563,   834,  ...,     0,     0,     0],
                      [12607,   563,   834,  ...,     0,     0,     0],
                      ...,
                      [12607,   563,   834,  ...,     0,     0,     0],
                      [12607,   563,   834,  ...,     0,     0,     0],
                      [12607,   563,   834,  ...,     0,     0,     0]]),
 'attention_mask': tensor([[1, 1, 1,  ..., 0, 0, 0],
                           [1, 1, 1,  ..., 0, 0, 0],
                           ...,
                           [1, 1, 1,  ..., 0, 0, 0],
                           [1, 1, 1,  ..., 0, 0, 0],
                           [1, 1, 1,  ..., 0, 0, 0]])}
```

Training for GROUP BY component (content) for SQL query.

Stage 3: SQL Query Construction

Objective: Combine the outputs from Stage 2 to form a complete and valid SQL query.

- **Input**: The predicted details for each SQL component.
- **Output**: A syntactically and semantically correct SQL query.

- **Advantages**:

  - This stage ensures that the final query is coherent and follows SQL syntax.
  - Modular combination allows flexibility in handling different query types.

- **Implementation**:

  - A query-building function constructs the SQL query by inserting the predicted components into the appropriate SQL structure.
  - Example:
    - **Input Components**:
      - SELECT: COUNT(*)
      - FROM: department
      - WHERE: budget > 1000000
    - **Output**:
    - select count(*) from department where budget > 1000000;

- **Execution:**
-

```
[98] y1=y1_test[:500].reset_index(drop=True)
     y2=y2_test[:500].reset_index(drop=True)
     y3=y3_test[:500].reset_index(drop=True)
     y4=y4_test[:500].reset_index(drop=True)
     y5=y5_test[:500].reset_index(drop=True)

     y0=y_test[:500].reset_index(drop=True)

[115] x_test[:500].tolist()

[121] test_list=[]
      predicted_list=[]
      for i in range(500):
        print('Test Code: ')
        y00=y0[i].split('|')
        y00 = [k.strip() for k in y00]
        t=''
        if 'SELECT' in y00:
          t=t+'SELECT '+y1[i]
        if 'FROM' in y00:
          t=t+' FROM '+y2[i]
        if 'WHERE' in y00:
          t=t+' WHERE '+y3[i]
        if 'ORDER BY' in y00:
          t=t+' ORDER BY '+y4[i]
        if 'GROUP BY' in y00:
          t=t+' GROUP BY '+y5[i]
        print(t)
        test_list.append(t)
        print('Predicted Code: ')
        predicted00=predicted_code0[i].split('|')
        predicted00 = [k.strip() for k in predicted00]
        p=''
        if 'SELECT' in predicted00:
          p=p+predicted00[0]+' '+predicted_code1[i]
        if 'FROM' in predicted00:
          p=p+' '+predicted00[1]+' '+predicted_code2[i]
        if 'WHERE' in predicted00:
          p=p+' '+predicted00[2]+' '+predicted_code3[i]
        if 'ORDER BY' in predicted00:
          p=p+' '+predicted00[3]+' '+predicted_code4[i]
        if 'GROUP BY' in predicted00:
          p=p+' '+predicted00[4]+' '+predicted_code5[i]
        print(p)
        predicted_list.append(p)
```

Aggregating all the predicted components and its respective contents to form a meaningful SQL query for the given prompt.

```
Test Code:
SELECT club FROM table_name_96 WHERE  played = "22" AND drawn = "0" AND lost = "14" AND losing_bp = "
Predicted Code:
SELECT club FROM table_name_96 WHERE played = "22 rounds" AND drawn = "0" AND losing_bp = 5
Test Code:
SELECT illustrator FROM table_name_79 WHERE  year = 1987
Predicted Code:
SELECT illustrator FROM table_name_79 WHERE year = 1987
Test Code:
SELECT departure_pune FROM table_29301050_1 WHERE  arrival_lonavla = "22:22"
Predicted Code:
SELECT departure_pune FROM table_29301050_1 WHERE arrival_lonavla = "22:22"
Test Code:
SELECT nfl_team FROM table_2508633_6 WHERE  pick__number = 151
Predicted Code:
SELECT nfl_team FROM table_2508633_6 WHERE pick__number = "151"
Test Code:
SELECT player FROM table_name_24 WHERE  country = "united states" AND money___$__ = 356
Predicted Code:
SELECT player FROM table_name_24 WHERE country = "usa" AND money___$__ = "356"
Test Code:
SELECT MIN(stage) FROM table_15088557_1 WHERE  mountains_classification = "Aitor Osa" AND winner = "A
Predicted Code:
SELECT stage FROM table_15088557_1 WHERE mountains_classification = "aitor osa" AND winner = "aitor g
Test Code:
SELECT weight FROM table_name_41 WHERE  denomination = "three paise"
Predicted Code:
SELECT weight FROM table_name_41 WHERE denomination = "three paise"
Test Code:
```

Generated Output to compare given SQL query in test dataset and the predicted SQL query by our Proposed Model.

Performance Analysis:

Fuzz Ratio:

Baseline Model:

```
[ ] Test_list=y_test[:500].tolist()
    Outputlist=predicted_code

[ ] testlist=pd.DataFrame(Test_list)
    outputlist=pd.DataFrame(Outputlist)

[ ] pip install rapidfuzz

Collecting rapidfuzz
  Using cached rapidfuzz-3.10.1-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (11 kB)
Using cached rapidfuzz-3.10.1-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (3.1 MB)
Installing collected packages: rapidfuzz
Successfully installed rapidfuzz-3.10.1

from rapidfuzz import fuzz
total=0
for i in range(0,500):
  result=fuzz.ratio(testlist[0][i],outputlist[0][i])
  total=total+result

print(total/500)

86.84779392126414
```

Proposed Model:

```
from rapidfuzz import fuzz
total=0
for i in range(0,500):
  result=fuzz.ratio(test_list[i],predicted_list[i])
  total=total+result

print(total/500)

91.4104434809133
```

Rouge Score:

Baseline Model:

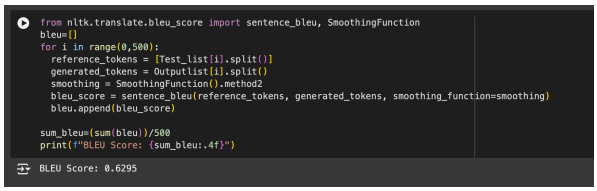
```
from rouge_score import rouge_scorer

scorer = rouge_scorer.RougeScorer(['rouge1', 'rouge2', 'rougeL'], use_stemmer=False)
R1=[]
R2=[]
RL=[]
for i in range(0,500):
  scores = scorer.score(Test_list[i], Outputlist[i])
  R1.append(scores['rouge1'].fmeasure)
  R2.append(scores['rouge2'].fmeasure)
  RL.append(scores['rougeL'].fmeasure)

for i in range(0,10):
  sum_R1=sum(R1)
  sum_R2=sum(R2)
  sum_RL=sum(RL)

print('Rouge1 Score: '+str(sum_R1/500))
print('Rouge2 Score: '+str(sum_R2/500))
print('RougeL Score: '+str(sum_RL/500))

Rouge1 Score: 0.8732782169120589
Rouge2 Score: 0.7864498523807367
RougeL Score: 0.8523645264523053
```

Proposed Model:

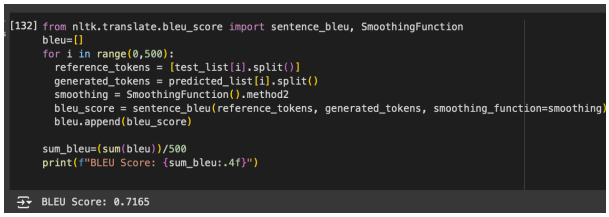
```
[127] from rouge_score import rouge_scorer

scorer = rouge_scorer.RougeScorer(['rouge1', 'rouge2', 'rougeL'], use_stemmer=False)
R1=[]
R2=[]
RL=[]
for i in range(0,500):
  scores = scorer.score(test_list[i], predicted_list[i])
  R1.append(scores['rouge1'].fmeasure)
  R2.append(scores['rouge2'].fmeasure)
  RL.append(scores['rougeL'].fmeasure)

for i in range(0,500):
  sum_R1=sum(R1)
  sum_R2=sum(R2)
  sum_RL=sum(RL)

print('Rouge1 Score: '+str(sum_R1/500))
print('Rouge2 Score: '+str(sum_R2/500))
print('RougeL Score: '+str(sum_RL/500))

Rouge1 Score: 0.9254991466021099
Rouge2 Score: 0.8604519015826005
RougeL Score: 0.9152792638922922
```

Bleu Score:

Baseline Model:



Proposed Model:



Conclusion:

The hierarchical decomposition framework proved to be a significant enhancement over traditional single-pass LLM approaches for code generation. By leveraging modular problem-solving, the method not only improved accuracy and consistency but also introduced greater transparency into the generation process. We saw that for Fuzz Ratio, Rouge Score, and Blue Score our proposed model of Hierarchical Decomposition proved to be a better model as compared to the baseline T5 single-pass model.

Baseline Model

T5 showed promise but was limited by memory usage and accuracy on complex queries. Non-modular approach led to difficulties in maintaining and debugging generated SQL queries.

Enhanced Model

Hierarchical decomposition significantly improved modularity and accuracy. Errors localized to specific stages, enhancing debugging and overall robustness.

Lessons Learned:

- LLMs benefit immensely from structured problem-solving frameworks.
- Breaking down tasks into logical subtasks is crucial for complex programming challenges.

Future Work:

- Dynamic Subtask Identification: Automating the decomposition process using advanced NLP techniques.
- Broader Dataset Coverage: Expanding experiments to include domain-specific codebases, such as medical or financial software.
- Multi-Agent Collaboration: Investigating how multiple LLMs can work together hierarchically to solve tasks collaboratively.

Acknowledgements:

We extend our gratitude to Prof. Lindi Liao and Teaching Assistant Mostafa for guidance and continued support.

References:

[1] “Content-Enhanced BERT-based Text-to-SQL Generation”, Tong Guo, Huilin Gao,

https://arxiv.org/pdf/1910.07179v5

[2] “CodexDB: Generating Code for SQL Processing Using GPT-3 Codex”, Immanuel Trummer, https://arxiv.org/pdf/2204.08941

[3] “A Review on Code Generation with LLMs: Application and Evaluation”, Jianxun Wang, Yixiang Chen, https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=10403378

[4] “GitHub Copilot AI Pair Programmer: Asset or Liability?”, Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais,

https://arxiv.org/pdf/2206.15331

[5] b-mc2/sql-create-context · Datasets at Hugging Face. (2001, May) https://huggingface.co/datasets/b-mc2/sql-create-context

[6] GeeksforGeeks. (2024, September 23). 10 Exciting project ideas using Large Language Models (LLMS).

https://www.geeksforgeeks.org/project-ideas-using-large-language-models/

[7].T5. (n.d.). https://huggingface.co/docs/transformers/en/model_doc/t5

[8].Papers with Code - Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. (2019, October 23). https://paperswithcode.com/paper/exploring-the-limits-of-transfer-learning#code

[9].Isaacndirangumuturi. (2023, October 5). Code Generation with T5 Transformer. Kaggle. https://www.kaggle.com/code/isaacndirangumuturi/code-generation-with-t5-transformer