Code Generation with StarCoder2-Instruct

DS677 Deep Learning

Professor: Hai Phan

Project by:

Uyen Nguyen

tn268@njit.edu

Aravind Kalyan Sivakumar

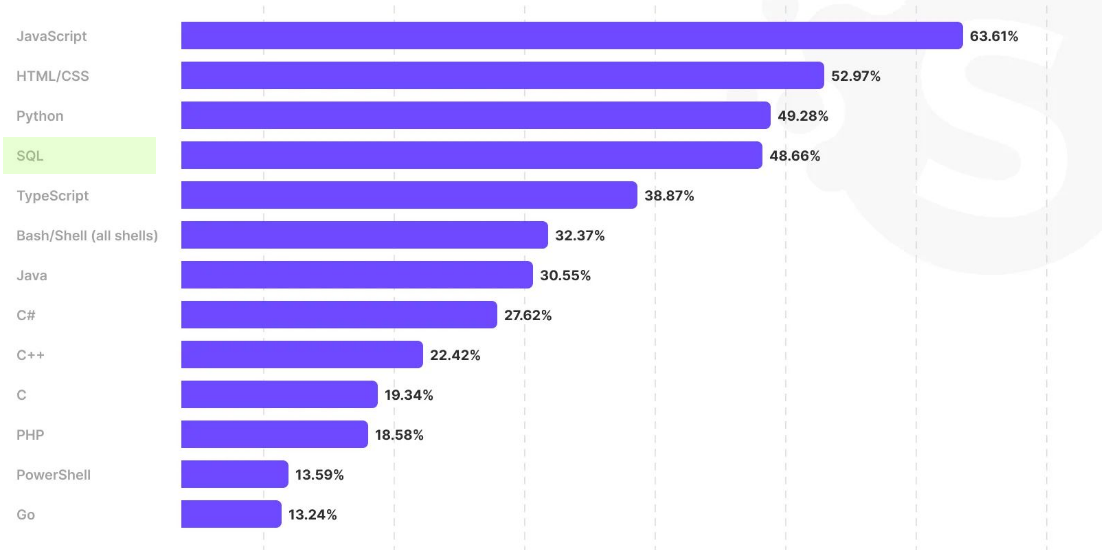
as4588@njit.edu

Kartik Pandit

ksp82@njit.edu

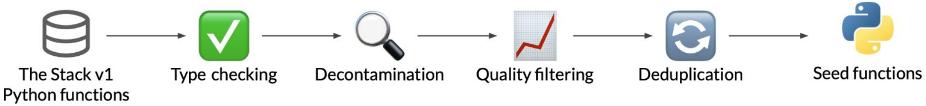
About the project

The project involves leveraging StarCoder2-15B, a large language model and its automated pipeline for generating high-quality code instructions and responses.

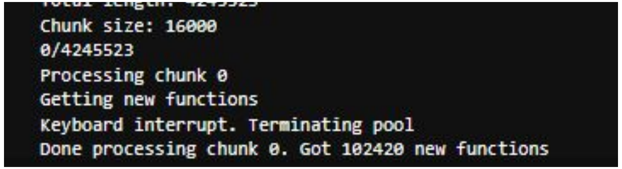


https://www.softermii.com/blog/top-programming-languages-and-frameworks-for-software-development

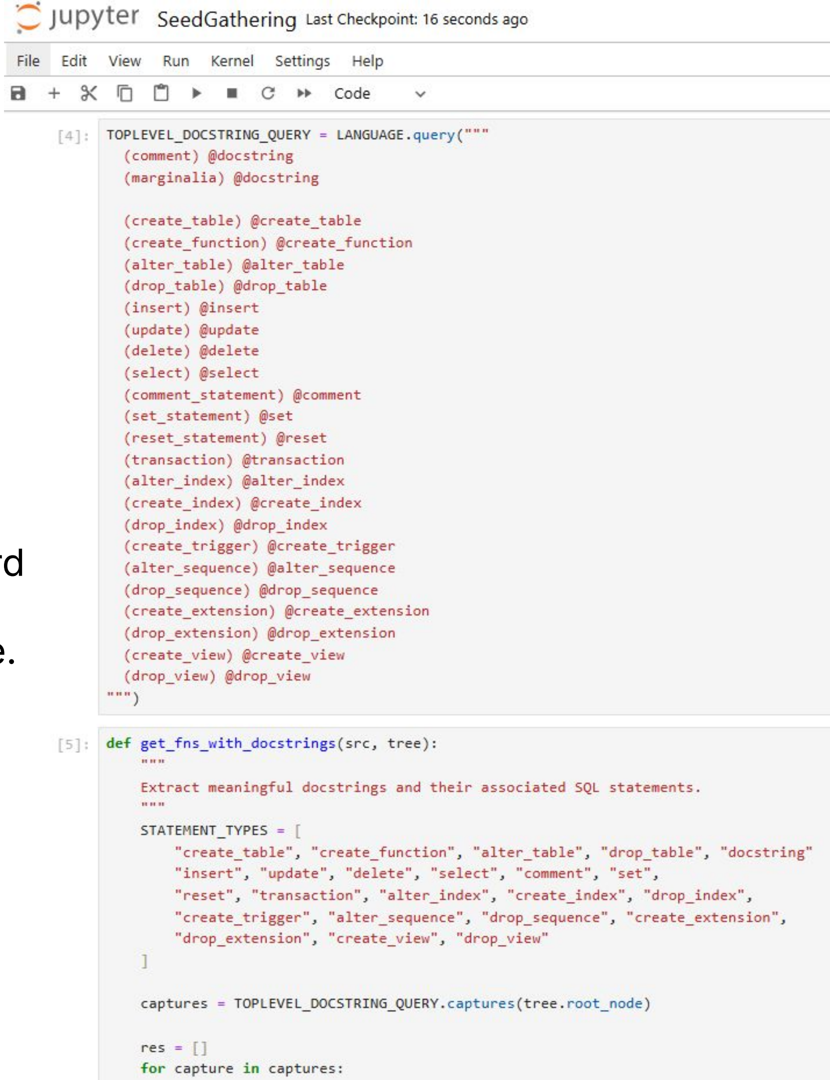
Step 1: Seed Dataset Curation



Actions:
- Set up HF Token and Login
- **`DerekStride/tree-sitter-sql`**
SQL grammar parser for tree-sitter
- Check `grammar.js` to understand syntax and the keyword for the lib parse and tokenize SQL code.
- Cleaned and formatted the seed JSON for the `S->C` stage.
- Extract 100K rows for quick experimentation



```
Total length: 4245523
Chunk size: 16000
0/4245523
Processing chunk 0
Getting new functions
Keyboard interrupt. Terminating pool
Done processing chunk 0. Got 102420 new functions
```


jupyter SeedGathering Last Checkpoint: 16 seconds ago
File Edit View Run Kernel Settings Help
Code

```
[4]: TOPLEVEL_DOCSTRING_QUERY = LANGUAGE.query("""
       (comment) @docstring
       (marginalia) @docstring

       (create_table) @create_table
       (create_function) @create_function
       (alter_table) @alter_table
       (drop_table) @drop_table
       (insert) @insert
       (update) @update
       (delete) @delete
       (select) @select
       (comment_statement) @comment
       (set_statement) @set
       (reset_statement) @reset
       (transaction) @transaction
       (alter_index) @alter_index
       (create_index) @create_index
       (drop_index) @drop_index
       (create_trigger) @create_trigger
       (alter_sequence) @alter_sequence
       (drop_sequence) @drop_sequence
       (create_extension) @create_extension
       (drop_extension) @drop_extension
       (create_view) @create_view
       (drop_view) @drop_view
     """)
```

```
[5]: def get_fns_with_docstrings(src, tree):
         """
         Extract meaningful docstrings and their associated SQL statements.
         """
         STATEMENT_TYPES = [
             "create_table", "create_function", "alter_table", "drop_table", "docstring"
             "insert", "update", "delete", "select", "comment", "set",
             "reset", "transaction", "alter_index", "create_index", "drop_index",
             "create_trigger", "alter_sequence", "drop_sequence", "create_extension",
             "drop_extension", "create_view", "drop_view"
         ]

         captures = TOPLEVEL_DOCSTRING_QUERY.captures(tree.root_node)

         res = []
         for capture in captures:
```

How Tree-Sitter-SQL parser Works:

- **Grammar Definition**:

The SQL grammar is defined in a `.js` file (as part of the Tree-sitter-SQL package). This grammar specifies the syntax rules for SQL, including keywords, clauses, operators, and constructs like `SELECT`, `WHERE`, and `JOIN`.

- **Vocab/Lexical Analysis**:

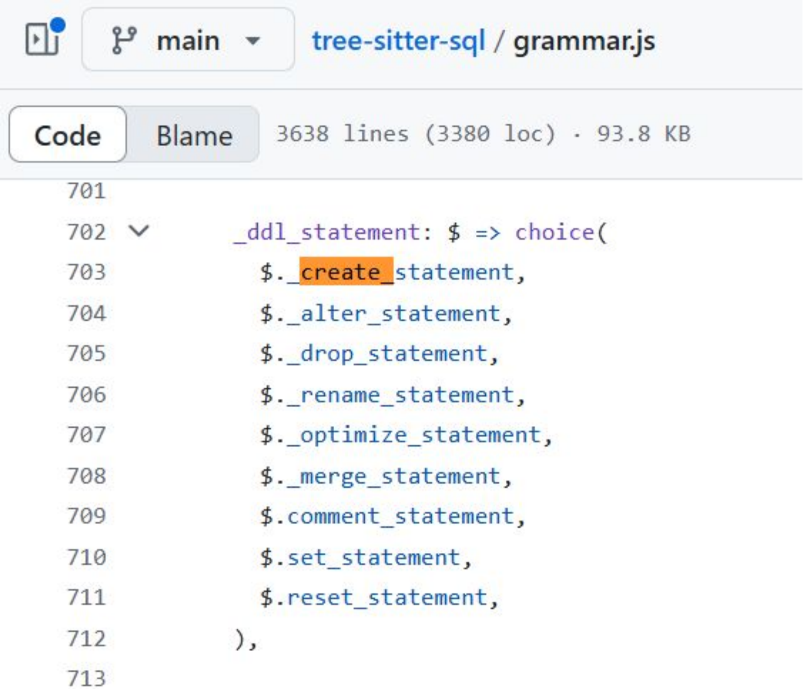
The parser tokenizes the SQL input into smaller syntactic units such as keywords (`SELECT`, `FROM`), identifiers (table/column names), operators (`=`, `AND`), and literals (e.g., `'2023-01-01'`).

- **Tree Construction**:

Using the predefined grammar, Tree-sitter constructs a **Concrete Syntax Tree (CST)**, representing the hierarchical structure of the SQL code. Each node in the tree corresponds to a grammatical element (e.g., a `select_statement`, `where_clause`, or `expression`).

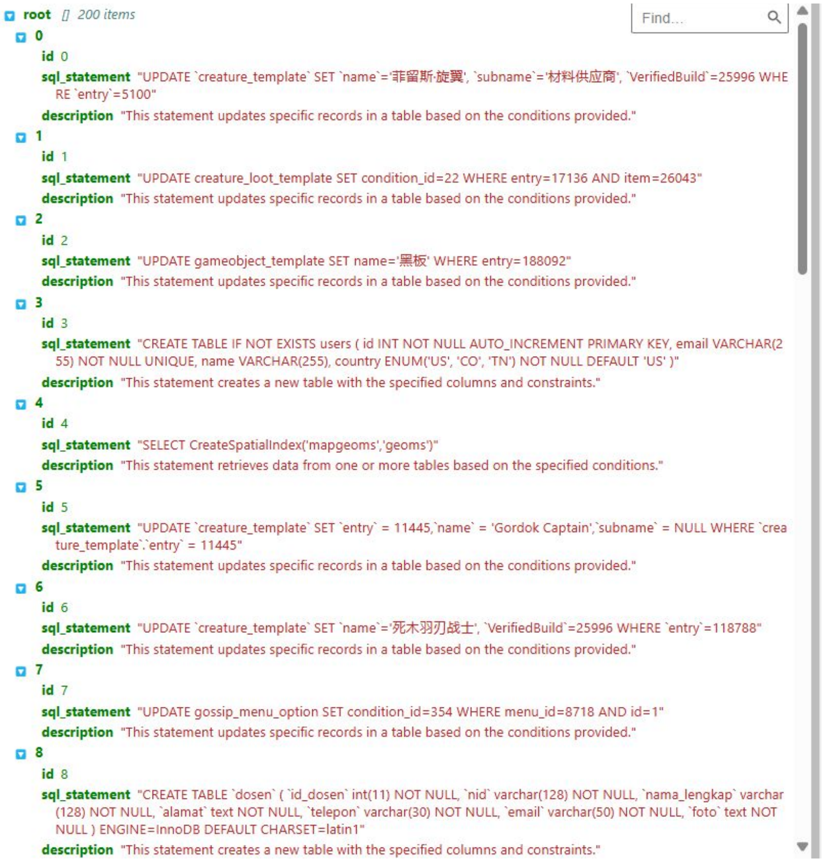
- **Tree Querying**:

The tree can be queried programmatically using APIs provided by Tree-sitter to extract specific parts of the SQL code, such as table names, columns, or conditions in the `WHERE` clause.

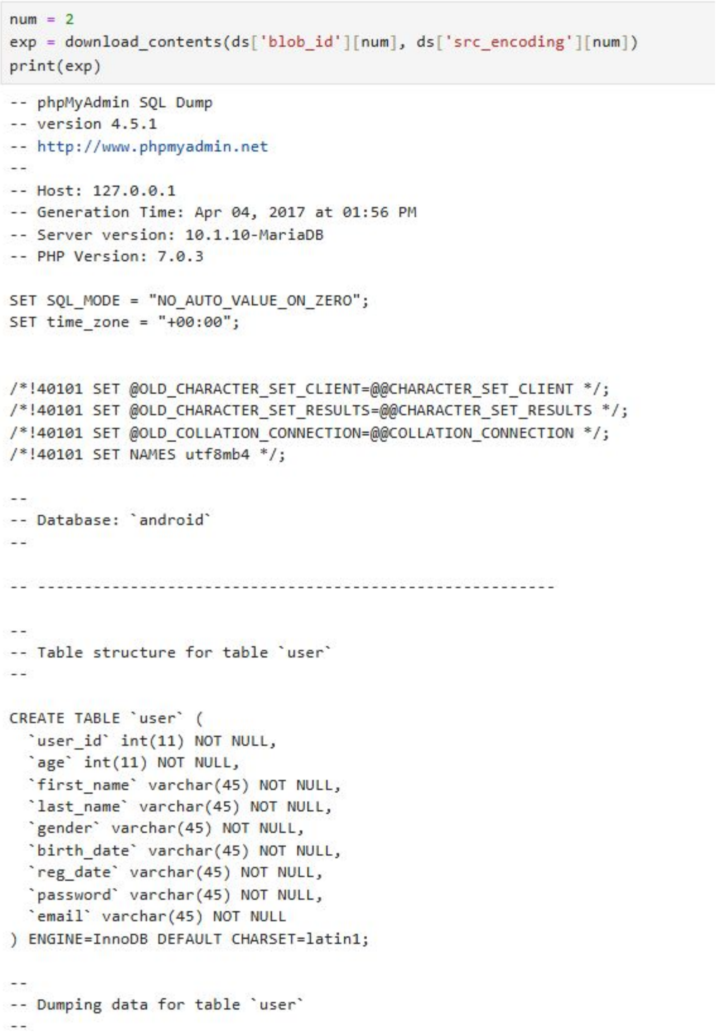


```

    _ddl_statement: $ => choice(
      $._create_statement,
      $._alter_statement,
      $._drop_statement,
      $._rename_statement,
      $._optimize_statement,
      $._merge_statement,
      $.comment_statement,
      $.set_statement,
      $.reset_statement,
    ),

```

SEED GATHERING OUTPUT (input for S→C)

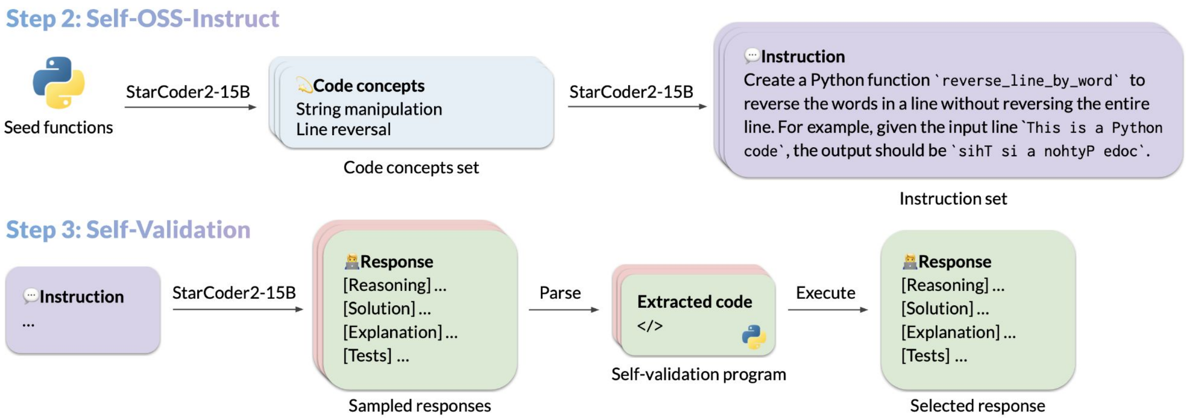


```
root  [] 200 items
  0
    id 0
    sql_statement "UPDATE `creature_template` SET `name`='菲留斯·旋翼', `subname`='材料供应商', `VerifiedBuild`=25996 WHE
      RE `entry`=5100"
    description "This statement updates specific records in a table based on the conditions provided."
  1
    id 1
    sql_statement "UPDATE creature_loot_template SET condition_id=22 WHERE entry=17136 AND item=26043"
    description "This statement updates specific records in a table based on the conditions provided."
  2
    id 2
    sql_statement "UPDATE gameobject_template SET name='黑板' WHERE entry=188092"
    description "This statement updates specific records in a table based on the conditions provided."
  3
    id 3
    sql_statement "CREATE TABLE IF NOT EXISTS users ( id INT NOT NULL AUTO_INCREMENT PRIMARY KEY, email VARCHAR(2
      55) NOT NULL UNIQUE, name VARCHAR(255), country ENUM('US', 'CO', 'TN') NOT NULL DEFAULT 'US' )"
    description "This statement creates a new table with the specified columns and constraints."
  4
    id 4
    sql_statement "SELECT CreateSpatialIndex('mapgeoms','geoms')"
    description "This statement retrieves data from one or more tables based on the specified conditions."
  5
    id 5
    sql_statement "UPDATE `creature_template` SET `entry` = 11445,`name` = 'Gordok Captain',`subname` = NULL WHERE `crea
      ture_template`.`entry` = 11445"
    description "This statement updates specific records in a table based on the conditions provided."
  6
    id 6
    sql_statement "UPDATE `creature_template` SET `name`='死木羽刃战士', `VerifiedBuild`=25996 WHERE `entry`=118788"
    description "This statement updates specific records in a table based on the conditions provided."
  7
    id 7
    sql_statement "UPDATE gossip_menu_option SET condition_id=354 WHERE menu_id=8718 AND id=1"
    description "This statement updates specific records in a table based on the conditions provided."
  8
    id 8
    sql_statement "CREATE TABLE `dosen` ( `id_dosen` int(11) NOT NULL, `nid` varchar(128) NOT NULL, `nama_lengkap` varchar
      (128) NOT NULL, `alamat` text NOT NULL, `telepon` varchar(30) NOT NULL, `email` varchar(50) NOT NULL, `foto` text NOT
      NULL ) ENGINE=InnoDB DEFAULT CHARSET=latin1"
    description "This statement creates a new table with the specified columns and constraints."
```



```python
num = 2
exp = download_contents(ds['blob_id'][num], ds['src_encoding'][num])
print(exp)
```

```
-- phpMyAdmin SQL Dump
-- version 4.5.1
-- http://www.phpmyadmin.net
--
-- Host: 127.0.0.1
-- Generation Time: Apr 04, 2017 at 01:56 PM
-- Server version: 10.1.10-MariaDB
-- PHP Version: 7.0.3

SET SQL_MODE = "NO_AUTO_VALUE_ON_ZERO";
SET time_zone = "+00:00";


/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
/*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS */;
/*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;
/*!40101 SET NAMES utf8mb4 */;

--
-- Database: `android`
--

-- --------------------------------------------------------

--
-- Table structure for table `user`
--

CREATE TABLE `user` (
  `user_id` int(11) NOT NULL,
  `age` int(11) NOT NULL,
  `first_name` varchar(45) NOT NULL,
  `last_name` varchar(45) NOT NULL,
  `gender` varchar(45) NOT NULL,
  `birth_date` varchar(45) NOT NULL,
  `reg_date` varchar(45) NOT NULL,
  `password` varchar(45) NOT NULL,
  `email` varchar(45) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

--
-- Dumping data for table `user`
--
```



Create a Python function `reverse_line_by_word` to reverse the words in a line without reversing the entire line. For example, given the input line `This is a Python code`, the output should be `sihT si a nohtyP edoc`.

The script's steps (S→C, C→I, I→R) focuses on inference and evaluation (rather than training). It looks at evaluating the model’s ability to follow instructions, generate relevant outputs, and validate the results.

Actions:
Self-OSS-Instruct
- Wulver: ssh to account and configure environment using `run.sh` script
- Set up script for each substep
- VLLM serve to start the vllm
- Modify the examples in `self-ossinstruct-fewshot.txt`
- Run the script that activates the InstructMode

Self-Validation
- Execute the extracted Response generated by the model and the Test to run programmatically

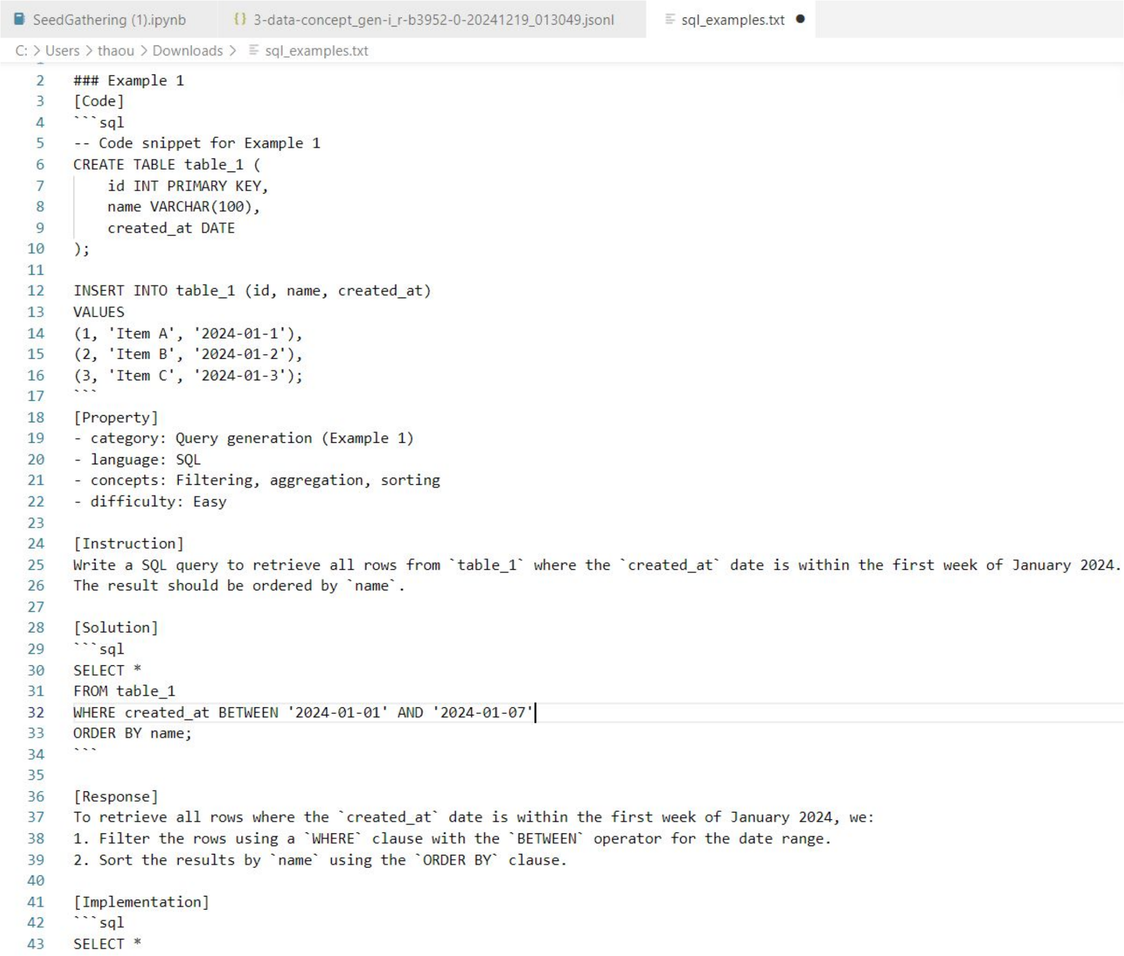
Few-shot Prompting for SQL

`self-ossinstruct-fewshot.txt` includes SQL examples structured similarly to the Python examples. Each example must include:

- `[Code]` containing SQL code snippets.
- `[Property]` describing the SQL task, language, and concepts.
- `[Instruction]` defining what the user needs to implement or answer.
- `[Solution]` providing the SQL solution (optional).
- `[Response]` explaining the reasoning behind the solution.

Purpose:

- Guiding the Model's Behavior
- Improving Response Quality


SeedGathering (1).ipynb
3-data-concept_gen-i_r-b3952-0-20241219_013049.jsonl
sql_examples.txt
C: > Users > thaou > Downloads > sql_examples.txt

~~~
### Example 1
[Code]
```sql
-- Code snippet for Example 1
CREATE TABLE table_1 (
    id INT PRIMARY KEY,
    name VARCHAR(100),
    created_at DATE
);

INSERT INTO table_1 (id, name, created_at)
VALUES
(1, 'Item A', '2024-01-1'),
(2, 'Item B', '2024-01-2'),
(3, 'Item C', '2024-01-3');
~~~
[Property]
- category: Query generation (Example 1)
- language: SQL
- concepts: Filtering, aggregation, sorting
- difficulty: Easy

[Instruction]
Write a SQL query to retrieve all rows from `table_1` where the `created_at` date is within the first week of January 2024.
The result should be ordered by `name`.

[Solution]
```sql
SELECT *
FROM table_1
WHERE created_at BETWEEN '2024-01-01' AND '2024-01-07'
ORDER BY name;
```

[Response]
To retrieve all rows where the `created_at` date is within the first week of January 2024, we:
1. Filter the rows using a `WHERE` clause with the `BETWEEN` operator for the date range.
2. Sort the results by `name` using the `ORDER BY` clause.

[Implementation]
```sql
SELECT *
```

S→C
(Seeds to Concepts)

Goal: Generate **concepts** from the seed functions.

What Happens:

- The script uses the provided dataset (e.g., a JSONL file with seed code snippets).
- For each seed, the model is prompted with a few-shot example (loaded from self-ossinstruct-fewshot.txt)
- Extract relevant **concepts** or keywords that summarize the snippet.

C→I
(Concepts to Instructions)

Goal: Generate **instructions** from a set of concepts or properties.

What Happens:

- The script takes the concepts generated from the S→C step.
- Few-shot examples in self-ossinstruct-fewshot.txt demonstrate how to map concepts into full, human- readable instructions.
- For each concept, the model generates instructions that are clear and specific.

I→R
(Instructions to Response)

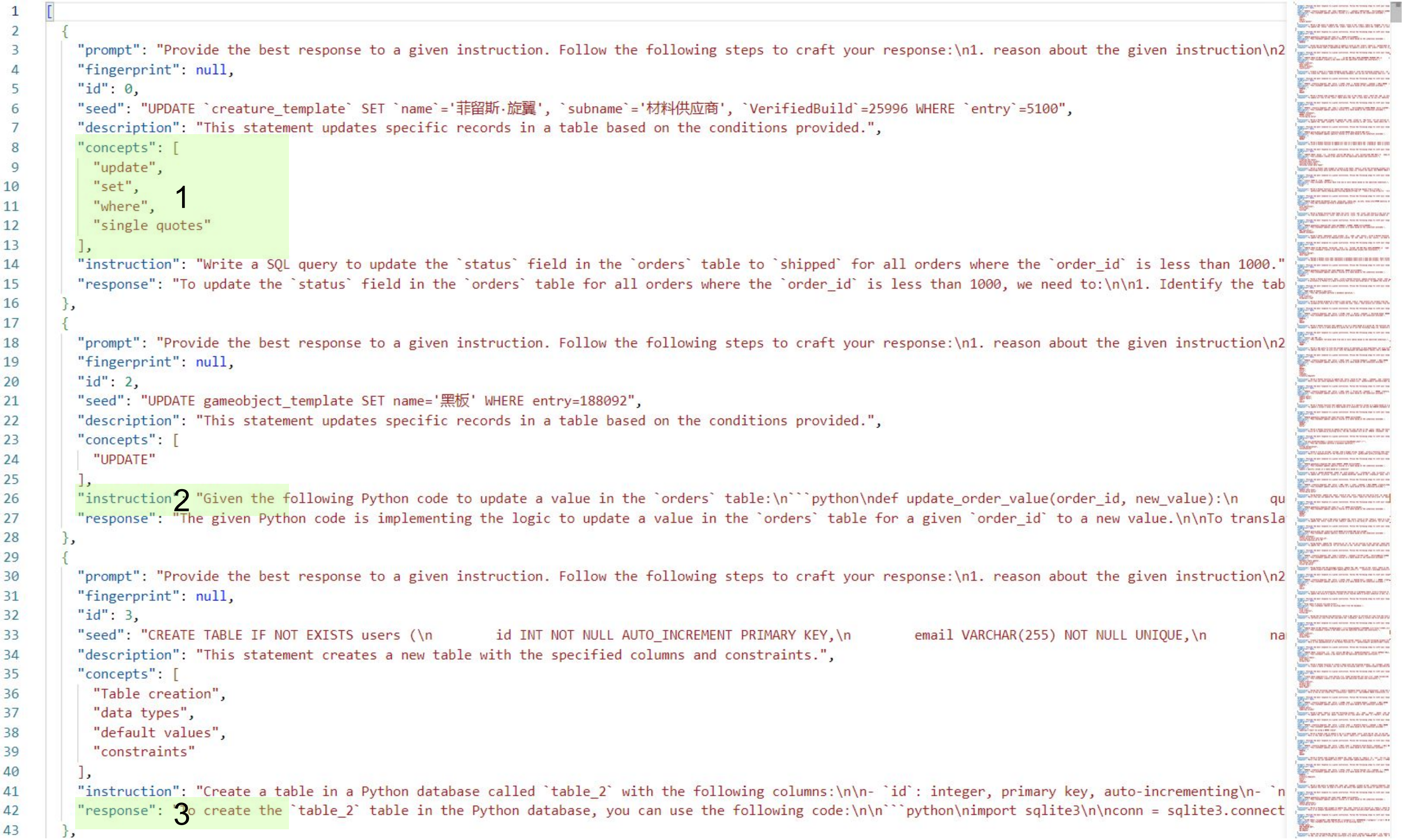
Goal: Generate detailed **responses** or solutions for a given instruction.

What Happens:

- The script takes the instructions created in the `C->I` step.
- Few-shot examples show how to map an instruction into a meaningful and actionable response (e.g., function implementation, detailed explanations, or other structured outputs).
- The model generates responses or solutions for each instruction.

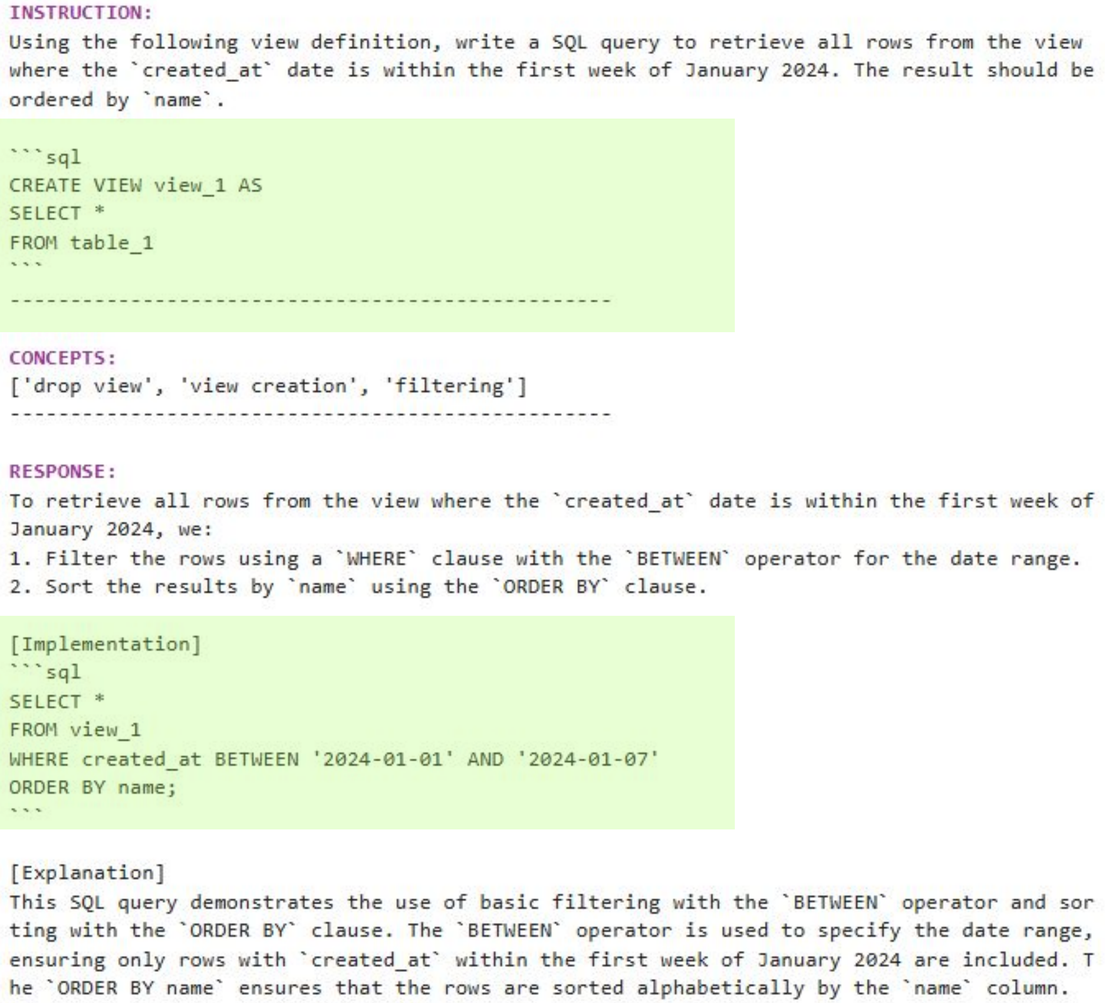
Extra Step for future reference S→I

- Expanding the Instruction dataset
- Reverse mapping for training
- Produce high-quality Instruction dataset



```
[
  {
    "prompt": "Provide the best response to a given instruction. Follow the following steps to craft your response:\n1. reason about the given instruction\n2
    "fingerprint": null,
    "id": 0,
    "seed": "UPDATE `creature_template` SET `name`='菲留斯·旋翼', `subname`='材料供应商', `VerifiedBuild`=25996 WHERE `entry`=5100",
    "description": "This statement updates specific records in a table based on the conditions provided.",
    "concepts": [
      "update",
      "set",
      "where",
      "single quotes"
    ],
    "instruction": "Write a SQL query to update the `status` field in the `orders` table to `shipped` for all orders where the `order_id` is less than 1000."
    "response": "To update the `status` field in the `orders` table for all orders where the `order_id` is less than 1000, we need to:\n\n1. Identify the tab
  },
  {
    "prompt": "Provide the best response to a given instruction. Follow the following steps to craft your response:\n1. reason about the given instruction\n2
    "fingerprint": null,
    "id": 2,
    "seed": "UPDATE gameobject_template SET name='黑板' WHERE entry=188092",
    "description": "This statement updates specific records in a table based on the conditions provided.",
    "concepts": [
      "UPDATE"
    ],
    "instruction": "Given the following Python code to update a value in the `orders` table:\n```python\ndef update_order_value(order_id, new_value):\n    qu
    "response": "The given Python code is implementing the logic to update a value in the `orders` table for a given `order_id` to a new value.\n\nTo transla
  },
  {
    "prompt": "Provide the best response to a given instruction. Follow the following steps to craft your response:\n1. reason about the given instruction\n2
    "fingerprint": null,
    "id": 3,
    "seed": "CREATE TABLE IF NOT EXISTS users (\n        id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,\n        email VARCHAR(255) NOT NULL UNIQUE,\n        na
    "description": "This statement creates a new table with the specified columns and constraints.",
    "concepts": [
      "Table creation",
      "data types",
      "default values",
      "constraints"
    ],
    "instruction": "Create a table in a Python database called `table_2` with the following columns:\n\n- `id`: integer, primary key, auto-incrementing\n- `n
    "response": "To create the `table_2` table in the Python database, you can use the following code:\n\n```python\nimport sqlite3\nconn = sqlite3.connect
  },
```

✅ GOOD RESPONSE EXAMPLES



INSTRUCTION:

Using the following view definition, write a SQL query to retrieve all rows from the view where the `created_at` date is within the first week of January 2024. The result should be ordered by `name`.

~~~~
```sql
CREATE VIEW view_1 AS
SELECT *
FROM table_1
```

--------------------------------------------------
~~~~

CONCEPTS:

['drop view', 'view creation', 'filtering']

RESPONSE:

To retrieve all rows from the view where the `created_at` date is within the first week of January 2024, we:

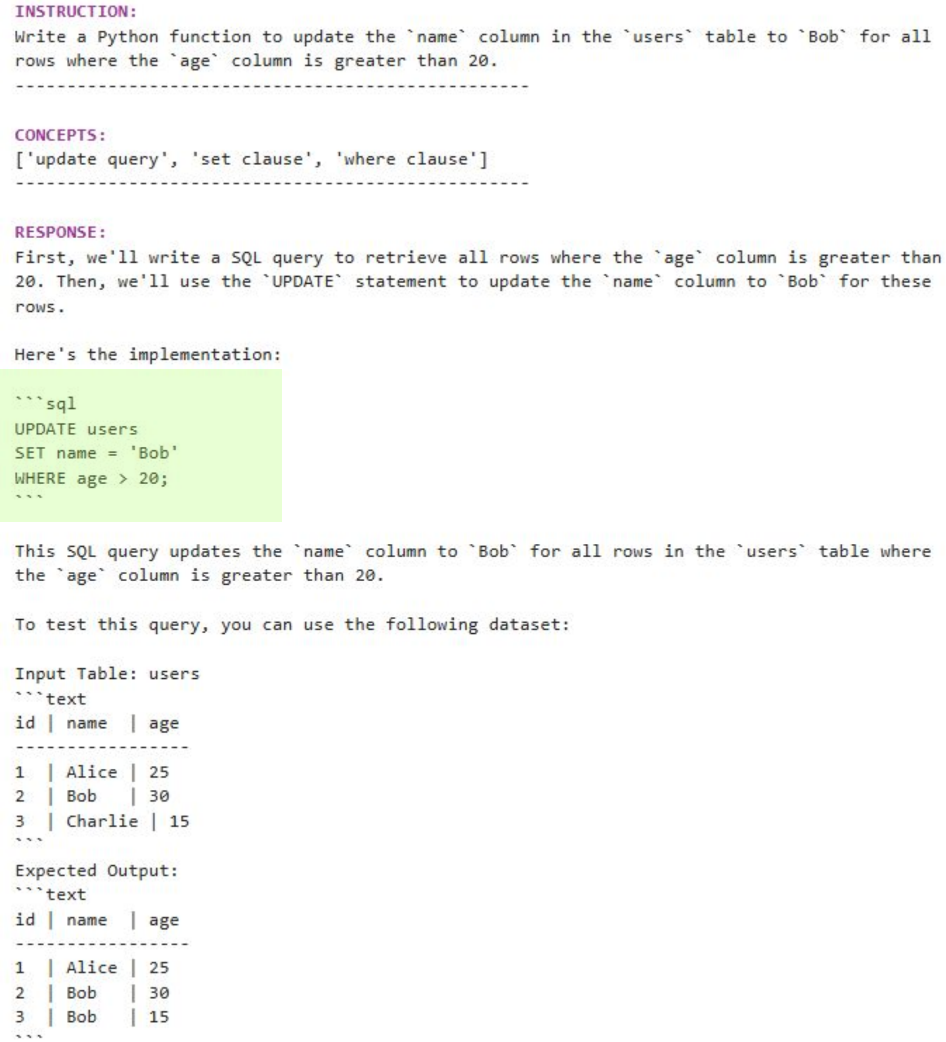
1. Filter the rows using a `WHERE` clause with the `BETWEEN` operator for the date range.
2. Sort the results by `name` using the `ORDER BY` clause.

~~~~
[Implementation]
```sql
SELECT *
FROM view_1
WHERE created_at BETWEEN '2024-01-01' AND '2024-01-07'
ORDER BY name;
```
~~~~

[Explanation]

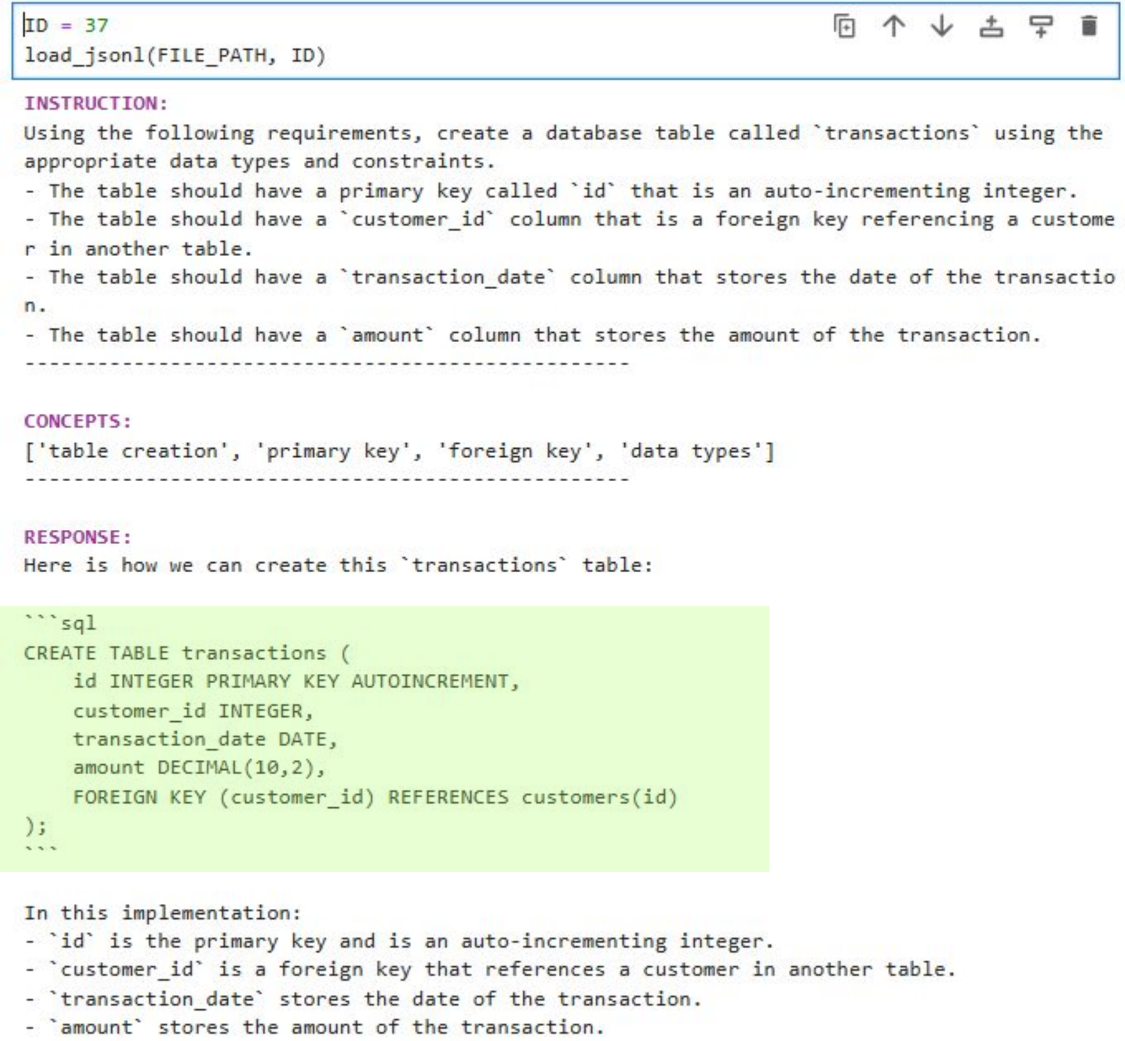
This SQL query demonstrates the use of basic filtering with the `BETWEEN` operator and sorting with the `ORDER BY` clause. The `BETWEEN` operator is used to specify the date range, ensuring only rows with `created_at` within the first week of January 2024 are included. The `ORDER BY name` ensures that the rows are sorted alphabetically by the `name` column.

✅ GOOD RESPONSE EXAMPLES



~~~
INSTRUCTION:
Write a Python function to update the `name` column in the `users` table to `Bob` for all
rows where the `age` column is greater than 20.
--------------------------------------------------

CONCEPTS:
['update query', 'set clause', 'where clause']
--------------------------------------------------

RESPONSE:
First, we'll write a SQL query to retrieve all rows where the `age` column is greater than
20. Then, we'll use the `UPDATE` statement to update the `name` column to `Bob` for these
rows.

Here's the implementation:

```sql
UPDATE users
SET name = 'Bob'
WHERE age > 20;
```

This SQL query updates the `name` column to `Bob` for all rows in the `users` table where
the `age` column is greater than 20.

To test this query, you can use the following dataset:

Input Table: users
```text
id | name   | age
-----------------
1  | Alice | 25
2  | Bob    | 30
3  | Charlie | 15
```
Expected Output:
```text
id | name   | age
-----------------
1  | Alice | 25
2  | Bob    | 30
3  | Bob    | 15
```
~~~

✅ GOOD RESPONSE EXAMPLES



```python
ID = 37
load_jsonl(FILE_PATH, ID)
```

~~~
INSTRUCTION:
Using the following requirements, create a database table called `transactions` using the
appropriate data types and constraints.
- The table should have a primary key called `id` that is an auto-incrementing integer.
- The table should have a `customer_id` column that is a foreign key referencing a custome
r in another table.
- The table should have a `transaction_date` column that stores the date of the transactio
n.
- The table should have a `amount` column that stores the amount of the transaction.
--------------------------------------------------

CONCEPTS:
['table creation', 'primary key', 'foreign key', 'data types']
--------------------------------------------------

RESPONSE:
Here is how we can create this `transactions` table:

```sql
CREATE TABLE transactions (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    customer_id INTEGER,
    transaction_date DATE,
    amount DECIMAL(10,2),
    FOREIGN KEY (customer_id) REFERENCES customers(id)
);
```

In this implementation:
- `id` is the primary key and is an auto-incrementing integer.
- `customer_id` is a foreign key that references a customer in another table.
- `transaction_date` stores the date of the transaction.
- `amount` stores the amount of the transaction.
~~~

INSTRUCTION:

Write a Python code snippet to create a new table `table_1` with the following columns:

- `id`: integer, primary key
- `name`: string, not null
- `created_at`: datetime, not null
- `updated_at`: datetime, not null

CONCEPTS:

['creating new table', 'defining table columns', 'setting primary key', 'defining column data types']

RESPONSE:

[Reasoning]
This query performs the following steps:
1. Create the table: Use CREATE TABLE to define the structure of the table.
2. Define the columns: Specify the column names and their data types.
3. Define primary key: Use PRIMARY KEY to define the primary key constraint on the `id` column.
4. Define not null: Use NOT NULL to define the not null constraint on the `name`, `created_at`, and `updated_at` columns.

[Implementation]
Here is the implementation:

```sql
CREATE TABLE table_1 (
    id INTEGER PRIMARY KEY,
    name TEXT NOT NULL,
    created_at DATETIME NOT NULL,
    updated_at DATETIME NOT NULL
);

    return result
```

[Explanation]
This query performs the following steps:
1. Create the table: Use CREATE TABLE to define the structure of the table.
2. Define the columns: Specify the column names and their data types.
3. Define primary key: Use PRIMARY KEY to define the primary key constraint on the `id` column.
4. Define not null: Use NOT NULL to define the not null constraint on the `name`, `created_at`, and `updated_at` columns.

[Tests]
To test this query, use the following dataset:

Input Table: table_1

```text
id | name   | created_at            | updated_at
-----------------------------------------------
1  | 'John' | '2024-01-01 12:00:00' | '2024-01-01 12:00:00'
2  | 'Jane' | '2024-01-01 12:00:00' | '2024-01-01 12:00:00'
3  | 'Bob'  | '2024-01-01 12:00:00' | '2024-01-01 12:00:00'
```

Expected Output:
```text
id | name   | created_at            | updated_at
-----------------------------------------------
1  | 'John' | '2024-01-01 12:00:00' | '2024-01-01 12:00:00'
2  | 'Jane' | '2024-01-01 12:00:00' | '2024-01-01 12:00:00'
3  | 'Bob'  | '2024-01-01 12:00:00' | '2024-01-01 12:00:00'
```

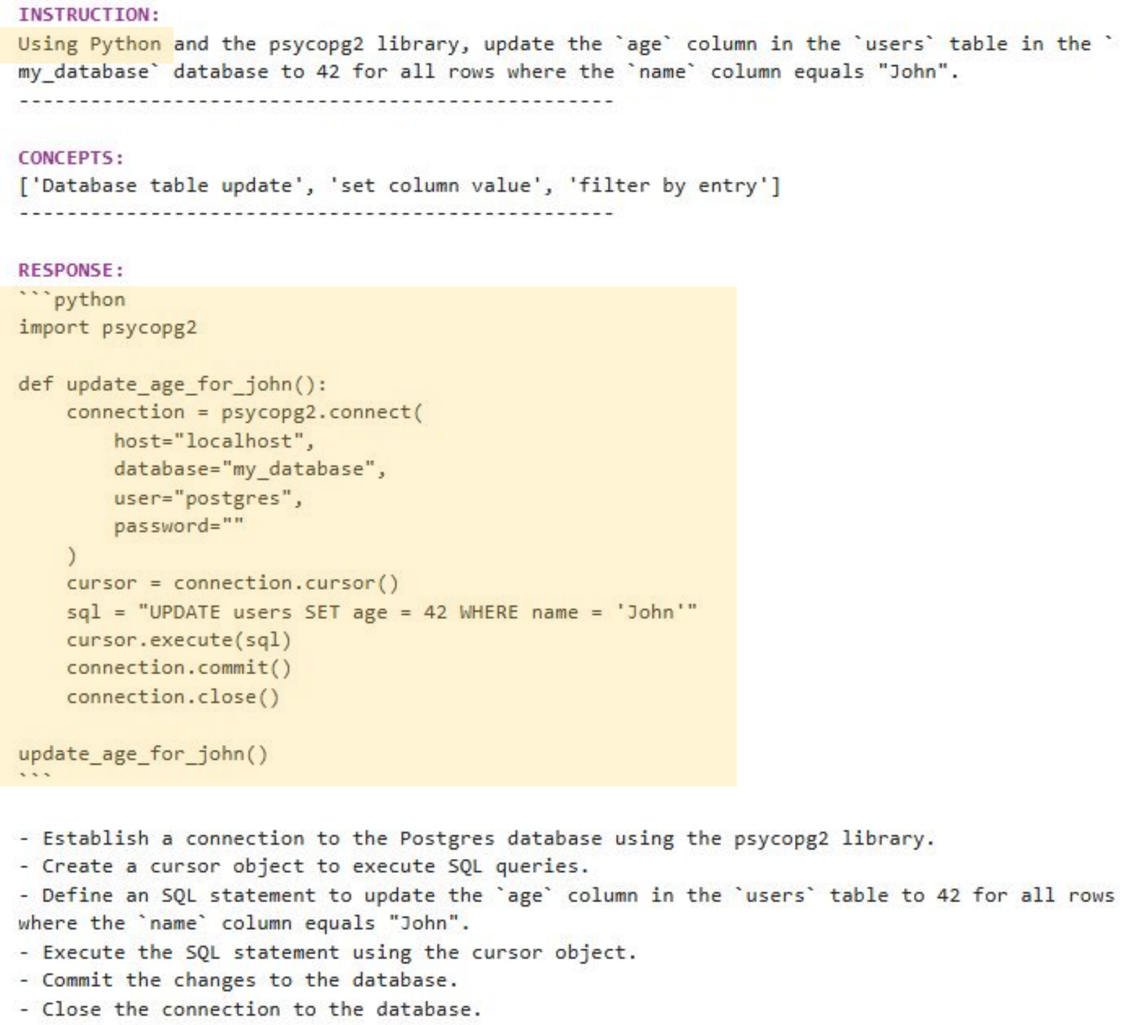
❌ BAD RESPONSE EXAMPLES

1) Concept Extraction (S→C):
- The extracted concepts are relevant
- correctly represent the core task

2) Instruction Generation (C→I):
- clear, concise, and accurate
- did not instruct to implement in SQL

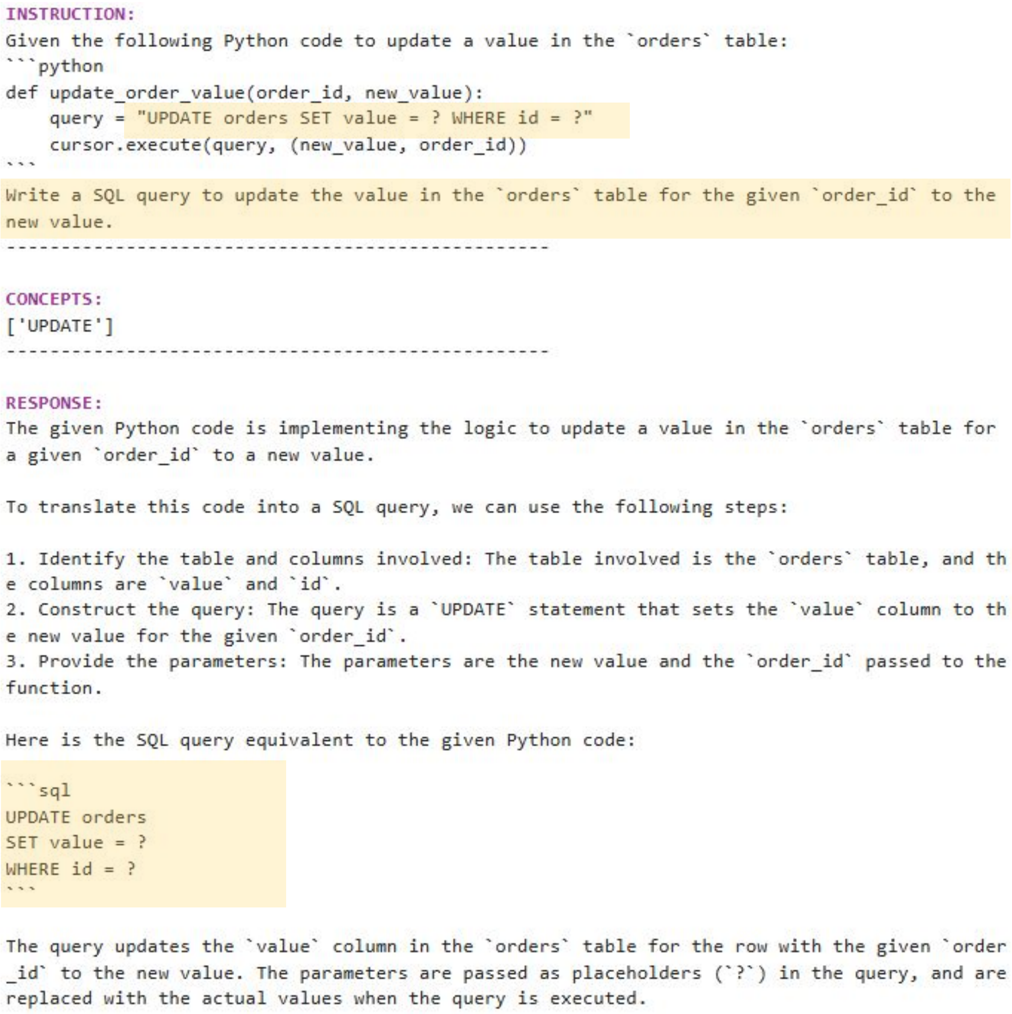
3) Response (I→R):
- did not just return SQL statements, generated additional Python code.

The focus on Python implementation suggests the model is not tuned for generating SQL queries.



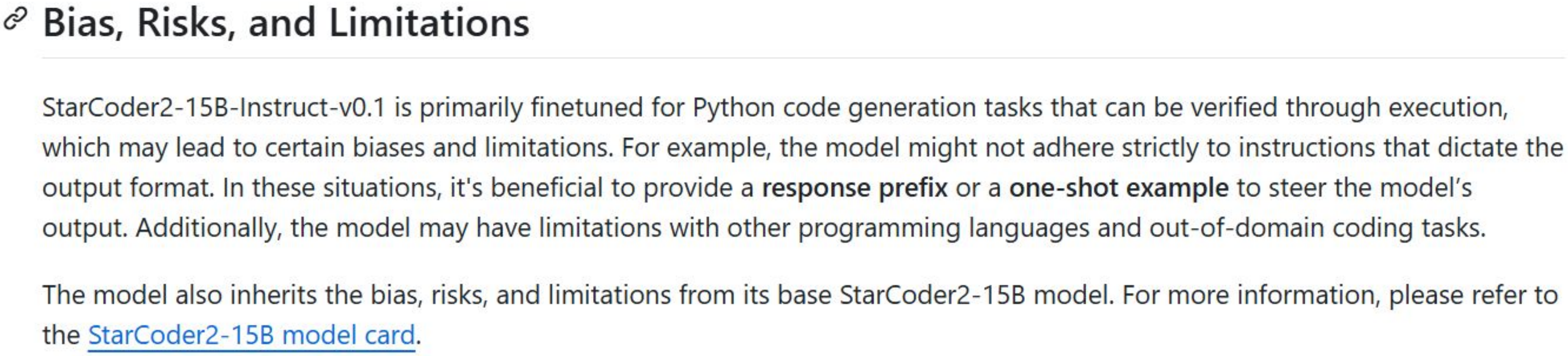
~~~
INSTRUCTION:
Using Python and the psycopg2 library, update the `age` column in the `users` table in the `
my_database` database to 42 for all rows where the `name` column equals "John".
--------------------------------------------------

CONCEPTS:
['Database table update', 'set column value', 'filter by entry']
--------------------------------------------------

RESPONSE:
```python
import psycopg2

def update_age_for_john():
    connection = psycopg2.connect(
        host="localhost",
        database="my_database",
        user="postgres",
        password=""
    )
    cursor = connection.cursor()
    sql = "UPDATE users SET age = 42 WHERE name = 'John'"
    cursor.execute(sql)
    connection.commit()
    connection.close()

update_age_for_john()
```

- Establish a connection to the Postgres database using the psycopg2 library.
- Create a cursor object to execute SQL queries.
- Define an SQL statement to update the `age` column in the `users` table to 42 for all rows
where the `name` column equals "John".
- Execute the SQL statement using the cursor object.
- Commit the changes to the database.
- Close the connection to the database.
~~~

❌ BAD RESPONSE EXAMPLES



~~~
INSTRUCTION:
Given the following Python code to update a value in the `orders` table:
```python
def update_order_value(order_id, new_value):
    query = "UPDATE orders SET value = ? WHERE id = ?"
    cursor.execute(query, (new_value, order_id))
```
Write a SQL query to update the value in the `orders` table for the given `order_id` to the
new value.
--------------------------------------------------

CONCEPTS:
['UPDATE']
--------------------------------------------------

RESPONSE:
The given Python code is implementing the logic to update a value in the `orders` table for
a given `order_id` to a new value.

To translate this code into a SQL query, we can use the following steps:

1. Identify the table and columns involved: The table involved is the `orders` table, and th
e columns are `value` and `id`.
2. Construct the query: The query is a `UPDATE` statement that sets the `value` column to th
e new value for the given `order_id`.
3. Provide the parameters: The parameters are the new value and the `order_id` passed to the
function.

Here is the SQL query equivalent to the given Python code:

```sql
UPDATE orders
SET value = ?
WHERE id = ?
```

The query updates the `value` column in the `orders` table for the row with the given `order
_id` to the new value. The parameters are passed as placeholders (`?`) in the query, and are
replaced with the actual values when the query is executed.
~~~

LIMITATIONS



Bias, Risks, and Limitations

StarCoder2-15B-Instruct-v0.1 is primarily finetuned for Python code generation tasks that can be verified through execution, which may lead to certain biases and limitations. For example, the model might not adhere strictly to instructions that dictate the output format. In these situations, it's beneficial to provide a **response prefix** or a **one-shot example** to steer the model's output. Additionally, the model may have limitations with other programming languages and out-of-domain coding tasks.

The model also inherits the bias, risks, and limitations from its base StarCoder2-15B model. For more information, please refer to the StarCoder2-15B model card.