Refined Automated query generation for pandas

Hongxin Huo
Mcgill University

Abstract

Query generation plays an essential role in Data Science and Machine Learning model training. Nowadays, various approaches for SQL query generation are available online. However, more effort needs to be made for pandas query generation to achieve across-system scalability and enhanced data security. As the importance of data privacy and expedited development cycles intensifies in the development of today's machine learning models, there is a growing demand for innovative methods to generate pandas queries that doesn't rely on actual data files. In this paper, we introduced a novel approach for pandas query generation, incorporating specific generation techniques tailored to pandas functions. Given a general data schema and meta information such as foreign keys, primary key, data types and ranges linked to each table (represented as pandas dataframes), the above approach generates a set of varied, semantically and syntactically correct pandas queries based on the provided conditions.

1. Introduction

1.1 Motivation

Research methodologies like zero-shot learning, crucial for estimating query execution costs and predicting cardinality, require training on thousands of diverse queries across varying table schemas. For example, [1] introduced a transferable database and query representation. Developing a robust zero-shot encoding necessitates a large dataset of queries across different table schemas involving multiple table joins. Consequently, achieving scalability in query generation across diverse schemas is increasingly critical.

Former approaches like [2] and [3] either uses information from queries and the database table schema resulting in an alarming amount of invalid queries, (e.g. result in an empty table after execution) or requires users to manually input the queries to guide the generation process. Although these techniques have been validated on SQL DBMS, little research has been conducted on automated query generation for pandas. While studies like [4] have explored converting pandas operations into SQL through transpilers, potentially limiting the full use of pandas' capabilities and idiomatic features like chained operations.

1.2 Foundation

This gap motivated the development of Dailun Li's automated query generator prototype. It utilized an intermediate representation to transform a sequence of operations into operation-based objects. With these objects that obtained information from table source data, one can keep track of the source and result table and also add, remove and modify operations to create a set of distinct pandas queries. This generator specifically worked for TPC-H database [5] as source tables to generate queries. The resulting queries are one-liner queries with a sequence of operations. In addition to his work, I adopted a tree-based data structure and deployed divide and conquer algorithm to transform the one-liner queries

into subsequent lines of subqueries built on the previous resulting dataframes, as shown in Figure 1. Both of these work serve as the foundation of our improved automated query generator. The goal is to accept general data schema and meta information from a JSON file, and dynamically generates a set of varied, semantically and syntactically correct pandas queries based on the specific requirements from users.

One example of the resulting queires from one-liner query

df0 = partsupp[partsupp['ps_supplycost'] >= 149].merge(supplier[(supplier['s_nationkey'] >= 20) | (supplier['s_acctbal'] >= 4757)],left_on='ps_suppkey', right_on='s_suppkey')

can be translated into

df0 = partsupp[partsupp['ps_supplycost'] >= 149]

df1 = supplier[(supplier['s_nationkey'] >= 20) | (supplier['s_acctbal'] >= 4757)]

df2 = df0.merge(df1, left_on='ps_suppkey', right_on='s_suppkey')

If a Merge Operation is present in the list, the preceding operations will be indexed as usual. Subsequently, the dataframe intended for merging will execute its operations. Finally, the Merge Operation will integrate the two dataframes, effectively linking them together.

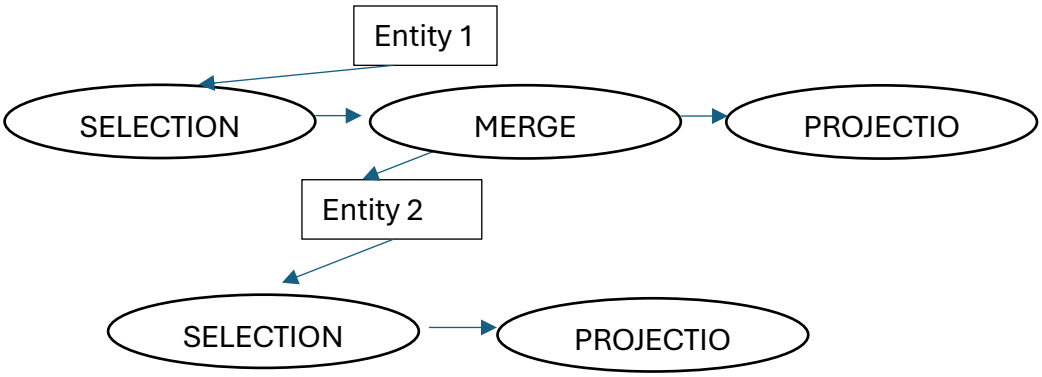


Figure 1: Tree-Based Entity & Operation Data Structure

2. Background Information

2.1 Pandas

The Pandas library is extensively utilized within the data science community due to its robust capabilities in managing and manipulating relational or labeled datasets. Its versatility extends beyond handling .csv and .xlsx file formats, encompassing SQL tabular data structures and numerical matrices as well. The primary data structures in Pandas are typically represented in two prevalent tabular formats.

2.1.1 Pandas Series

The Pandas Series is a type-specific 1D array that is size-immutable

Operation	SQL Representation	Pandas Representation
Selection	WHERE <conditions>	df[df[<conditions>]]
Projection	SELECT <columns> FROM	df[[<columns>]]
Aggregation	SELECT f<column>	df.agg(<f>)
Groupby	GROUP BY <columns>	df.groupby(<columns>)
Merge	A JOIN B ON <l_col>, <r_col>	df1.merge(df2,<l_col>,<r_col>)

Table 1: The operations for SQL representation and Pandas representation

2.1.2 Pandas Dataframe

The pandas DataFrame is more frequently used and can be seen as a container of multiple Pandas Series. A DataFrame is size mutable on both columns and rows. In this project, we will only focus on the row operations, as shown in Table 1.

2.2 Pandas and Intermediate Representation Form Operation

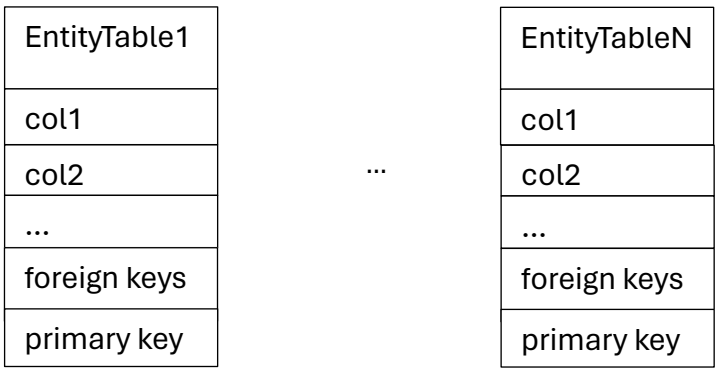
In IR format, each pandas query can be viewed as a list of operation objects. Table 2 explains the translation from pandas to IR format syntax.

Operation	Our Intermediate Representation
Selection	Selection (df, <conditions>)
Projection	Projection (df, <columns>)
Aggregation	Agg(df, <f>)
Groupby	Groupby(df, <columns>)
Merge	Merge(df2, <l_col>, <r_col>)

Table 2: Intermediate Representation for Pandas from the Prototype

2.3 Prototype Direct File Access and General Schema Query Generator

The enhanced query generator represents a substantial advancement over its predecessor in terms of versatility and efficiency in resource utilization. This novel iteration necessitates merely the schema details, including the names of the pertinent data tables and their quantity, to initiate the query generation process. Unlike the initial prototype, which was confined to the processing of CSV files specific to the TPC-H benchmark and depended on the availability of actual data files, the updated model boasts the capability to operate utilizing data schemas and metadata. This eliminates the necessity for direct file manipulation. Figure 2 delineates the general data schema requisite for the refined query generator.



```
{
    "entities": {
        "team": {
            "properties": { "National_name": { "type": "int", "min": 1, "max": 20 }, "Group": { "type": "string",
"starting character": "a" }, "Number": { "type": "int", "min": 1, "max": 20 } },
            "primary_key": "National_name"
        },
        "association": {
            "properties": { "Association_name": { "type": "int", "min": 1, "max": 20 }, "National_name": { "type":
"string", "min": 1, "max": 20 }, "URL": { "type": "string", "min": 1, "max": 100 } },
            "primary_key": "Association_name",
            "foreign_keys": { "National_name": "team" }
        },
        "player": {
            "properties": { "Shirt_number": { "type": "int", "min": 1, "max": 20 }, "National_name": { "type": "string",
"min": 1, "max": 20 }, "DOB": { "type": "date" }, "Name": { "type": "string", "min": 1, "max": 20 }, "General_position":
{ "type": "string", "min": 1, "max": 20 } },
            "primary_key": ["Shirt_number", "National_name"],
            "foreign_keys": { "National_name": "team" }
        },
        "coach": {
            "properties": { "Role": { "type": "int", "min": 1, "max": 20 }, "National_name": { "type": "string", "min":
1, "max": 20 }, "DOB": { "type": "date" }, "Name": { "type": "string", "min": 1, "max": 20 } },
            "primary_key": ["Role", "National_name"],
            "foreign_keys": { "National_name": "team" }
        },
        "stadium": {
            "properties": { "name": { "type": "int", "min": 1, "max": 20 }, "city": { "type": "string", "min": 1, "max":
20 }, "location": { "type": "string", "min": 1, "max": 20 } },
            "primary_key": "name"
        },
        "match": {
            "properties": { "ID": { "type": "int", "min": 1, "max": 64 }, "teamName1": { "type": "string", "min": 1,
"max": 20 }, "teamName2": { "type": "string", "min": 1, "max": 20 }, "stadiumName": { "type": "string", "min": 1, "max":
20 }, "date": { "type": "date" }, "time": { "type": "time" }, "length": { "type": "integer", "min": 1, "max": 20 },
"round": { "type": "string", "min": 1, "max": 1 } },
            "primary_key": "ID",
            "foreign_keys": { "teamName1": "team", "teamName2": "team", "stadiumName": "stadium" }
        }
    }
}
```


Figure 2: Data schema and Entity Table Information

3. Methods

In this section, we will discuss the general workflow and extended functionalities of the improved generator

3.1 General Schema and Enhanced versatility

This section delineates the methodology I employed for schema parsing and the utilization of meta-information, such as data types, data ranges, foreign keys, and primary keys, for the dynamic creation of intermediate representation dataframes.

3.1.1 JSON Schema Parsing

The process of JSON Schema Parsing is foundational to the dynamic data generator, serving as the initial step where the schema definitions are loaded and interpreted. This process begins with the ingestion of a JSON file, which contains structured metadata about various entities that mimic database table definitions. Each entity in the JSON file describes properties such as column names, data types, constraints (like minimum and maximum values for numerical data or specific formats for string data), and relationships with other entities (including primary and foreign keys).

The parsing is implemented using Python's json library. Upon loading the JSON schema, the system reads and decodes the JSON file into a Python dictionary, making the schema programmatically accessible.

```
with open('/Users/data_structure.json') as f:
        schema_info = json.load(f)
for entity, entity_info in schema_info["entities"].items():
        entity_schema = {
```

"properties": entity_info['properties'],

"primary_key": entity_info.get('primary_key', None),

"foreign_keys": entity_info.get('foreign_keys', {})

}

3.1.2 Data Structure Initialization

Following the schema parsing, the Data Structure Initialization phase sets up the necessary infrastructure to manage the data according to the schema. This stage involves initializing several Python dictionaries to store the DataFrames (dataframes), their associated data ranges for validation purposes (data_ranges), relationships as defined by foreign keys (foreign_keys), and a hypothetical structure (tbl_sources) defined from the previous prototype to wrap information like foreign keys of a specific dataframe. The generator will then extract the meta information from schema and store them into the dictionaries. This meta information is crucial for speeding up the operation generation in IR format, as shown in Figure 3.

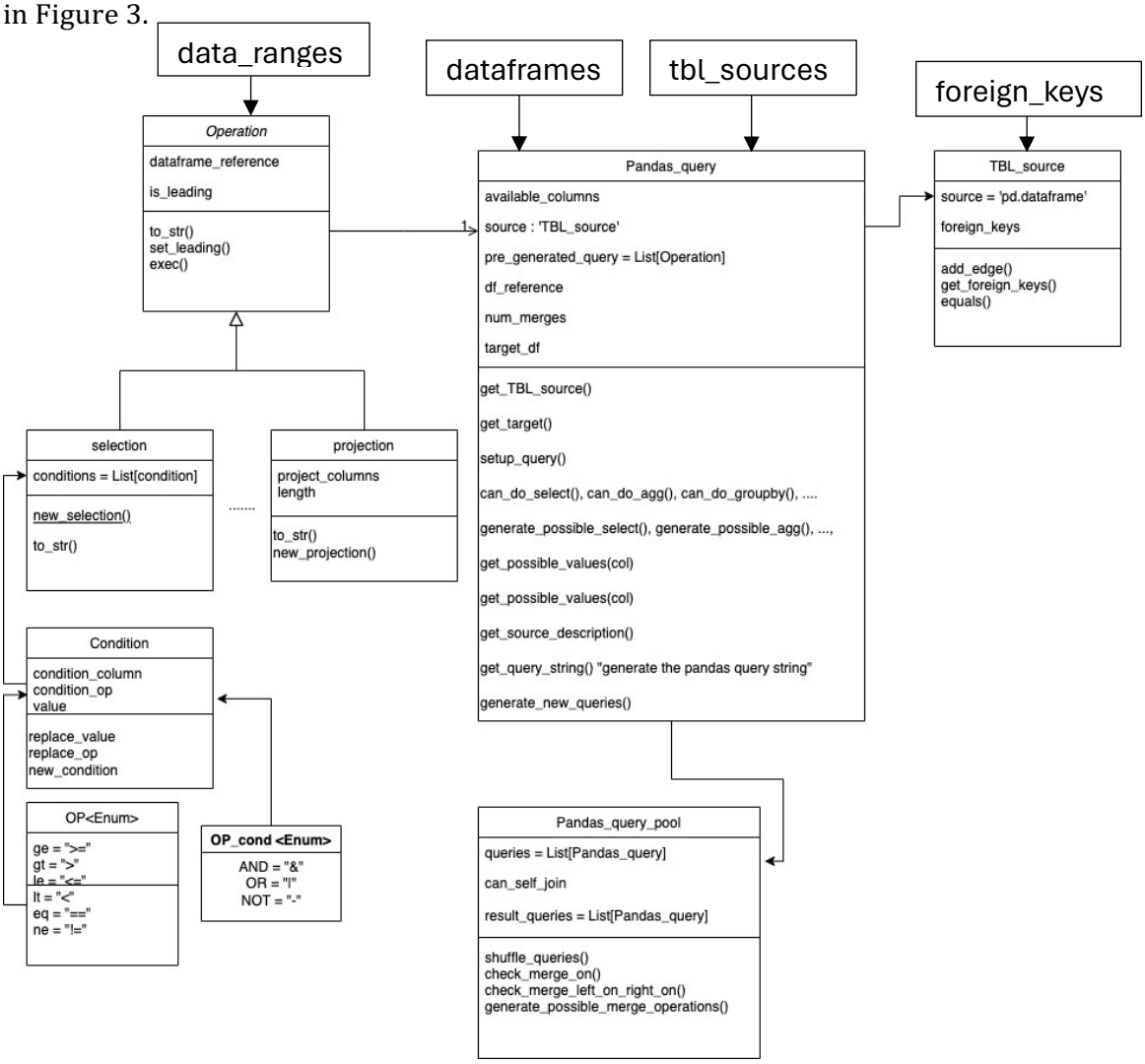


Figure 3: Utilizing Meta information for IR format operation generation

3.1.3 Dynamic Intermediate Representation DataFrame Creation

This stage encompasses the dynamic assembly of IR pandas DataFrames, which are structured according to the specifications outlined within the parsed JSON schema. By calling **create_dataframe()** function, the methodology systematically progresses through the properties attributed to each entity as delineated in the schema. Concurrently, this function allocates a suitable pandas data type for each property. To circumvent any potential `IndexError`, the DataFrame is initially populated with two rows of data, randomly generated, ensuring robustness in scenarios where the data type of the first element in an unpopulated data column may be subject to validation.

However, there is one issue worth noting. Since the generator later uses **pd.eval()** function to evaluate a string expression in the context of pandas data structures,

```
pd.eval(query_string, local_dict=local_dict)
```

The query_string paramenter contains a pandas-compatible expression, and local_dict refers to the dataframes dictionary created before. When **pd.eval()** is called with these parameters, pandas parses the query_string and looks up any referenced variables in the local_dict. If the reference variable to the dataframe doesn't share the same name as the dataframe, **pd.eval()** won't find the corresponding dataframe to perform pandas queries on, and will lead to a KeyError.

False example:

```
team = create_dataframe(team_schema, num_rows = 2)
dataframes["team"] = team
```

To dynamically create variables with the same names as the associated dataframes, I adopted **globals()** function which returns a dictionary representing the current global symbol table.

```
globals()[entity] = create_dataframe(entity_schema, num_rows=2)
dataframes[entity] = globals()[entity]
```

It dynamically creates and assigns a pandas DataFrame to a global variable named after the value contained in the variable entity, which is the name of the corresponding dataframe. Then the generator will create the wrapper class instance tbl_source for each dataframe, and utilizes **foreign_keys** dictionary and **add_foreignkeys()** function from helpers.py to build links between tbl_source objects.

```
tbl = TBL_source(globals()[entity], entity)
tbl_sources[entity] = tbl
h.add_foreignkeys(tbl_sources[entity1], col, tbl_sources[entity2], col)
```

3.2 Selection Opertions on More Data Types

3.2.1 Improved Selection Operation on Numerical Data Type

The initial selection procedure was restricted to columns of numerical data types, assessing the type based on the first element within each column to discern if it is an integer or a floating-point number. Subsequently, the process employed the **df.describe()** function inherent to the DataFrame to procure a comprehensive statistical summary, encapsulating measures of central tendency and variability. Following this, a value was randomly generated within the established range to facilitate the selection operation. This method necessitated an exhaustive iteration over the complete dataset, resulting in considerable consumption of resources and delays in execution.

To enhance efficiency, I implemented a strategy that leverages the metadata attributes, specifically the 'min' and 'max' values delineated in the data schema. Utilizing this metadata allows for the generation of a random value that aligns with the specified requirements. Consequently, this value is incorporated into a condition object, which is then applied to the selection operation, streamlining the process by circumventing the need for full dataset traversal.

```
min, max = data_ranges[self.df_name][key]

cur_val = random.randint(min, max)

OPs = [OP.gt, OP.ge, OP.le, OP.eq, OP.lt, OP.ne

cur_condition = condition(key, random.choice(OPs), cur_val)
```

Given that the meta information is already available, this methodology significantly enhances execution efficiency and minimizes latency.

3.2.2 Expanded Selection Operation on String Data Type

As for columns of string data type, I expand the selection operation to include conditions which filters the data starting with a given character. The specific requirement is also reflected in the data schema and stored inside **data_ranges** dictionary.

```
{"entities": {"team": { "properties": {"Group": { "type": "string", "starting
character": "a" } } } } }

cur_val = data_ranges[self.df_name][column]

cur_condition = condition(key, OP.startswith, cur_val)
```

Overall, Using the meta information from the schema facilitates expanding selection operations following the users' requirements and greatly reduces execution time especially when we have a large amount of data.

3.3 Refined Projection operation preserve primary key information

Given that the primary key uniquely identifies each table and reference keys frequently

target these primary key columns, it is essential to retain the primary key within projection operations. I extract the primary key metadata for each entity table, ensuring its inclusion in the projection alongside a random selection of other possible columns, which may consist solely of the primary key.

This practice minimizes instances where projecting a dataframe excludes the primary key, which also functions as a foreign key, potentially causing errors during subsequent merges with other dataframes.

4 Analysis

Utilizing the data schema illustrated in Figure 2, which consists of six unpopulated tables, I conducted query generation tests. The system swiftly generated 1,300 individual queries within eight seconds. However, creating merged queries required significantly more time, especially with multiple merges due to the column union from two source tables. This process, without subsequent projection, led to an exponential increase in the time needed to iterate through the columns of the resulting dataframe. Overall, the system managed to produce approximately 2,300 queries in less than 50 seconds.

The performance of the refined query generator was not markedly different from the initial prototype. A possible explanation is that while the meta-information expedited the operation generation phase as shown in Figure 3, it did not significantly reduce the total execution time. The direct execution needed for generating target queries in merge operations, particularly with multiple merges, also contributed to an exponential increase in processing time. Additionally, the previously employed divide and conquer strategy may have prolonged execution times compared to more straightforward methods of query generation. Some sample generated queries are listed below:

4.1 Sample Queries

```
df1 = team[['Group']]

df2 = df1.groupby(by=['Group'])

df3 = df2.agg('max', numeric_only=True)
```

The above query is generated by the team table. A projection was first done on ‘Group’ column, and a group_by operation was performed on the same column of the resulting dataframe, followed by an an aggregation operatin using “max” function.

```
df9 = player[['National_name','Name','General_position']]

Next

df10 = association[['National_name','URL']]

Next

df11 = team[['National_name','Group']]
```

```
df12 = df10.merge(df11, left_on=National_name, right_on=National_name)

df13 = df9.merge(df12, left_on=National_name, right_on=National_name)
```

can be translated into pandas query as:

```
df9 =
player[['National_name','Name','General_position']].merge(association[['Nat
ional_name','URL']].merge(team[['National_name','Group']],
left_on=National_name, right_on=National_name ), left_on=National_name,
right_on=National_name)
```

The above query is a complex example. This is a chain of merge operations between player table and the resulting table from another nested merge operation between team table and association table. Each table also has a projection operation before they're being merged.

5 Conclusion and Further Improvements

5.1 Conclusion

Our query generator efficiently produces queries using only the general data schema, facilitating the testing of scheduler performance and supporting zero-shot learning in pandas query execution analytics. Additionally, this tool proves invaluable in developing environments where direct access to data files is unavailable, or where data privacy concerns prevent the use of actual datasets.

5.2 Further Improvements:

While the refined query generator effectively handles user-defined schemas, there is potential to further enhance its capabilities through API integration. This improvement could significantly extend its functionality in areas such as remote data source integration and cross-platform compatibility.

Reference

[1] Benjamin Hilprecht, Carsten Binnig One Model to Rule them All: Towards Zero-Shot Learning for Databases; arXiv:2105.00642

[2] Claudio de la Riva, María José Suárez-Cabal, and Javier Tuya. 2010. Constraint-based test database generation for SQL queries. In Proceedings of the 5th Workshop on Automation of Software Test (AST '10). Association for Computing Machinery, New York, NY, USA, 67–74. https://doi.org/10.1145/1808266.1808276

[3] Carsten Binnig, Donald Kossmann, and Eric Lo. Reverse query processing. In ICDE, pages 506–515. IEEE, 2007.

[4] Hagedorn, Stefan, Steffen Kläbe, and Kai-Uwe Sattler. "Putting Pandas in a Box." CIDR. 2021.

[5]TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark Performance Characterization and Benchmarking, 2014, Volume 8391 ISBN : 978-3-319-04935-9, Peter Boncz, Thomas Neumann, Orri Erling