Customized Query Generation for Pandas

Ege Satir
Mcgill University

Abstract

Query generation is crucial for Data Science and Machine Learning model training. While various SQL query generation tools exist, automated Pandas query generation remains underdeveloped, limiting cross-system scalability and data security. Existing prototypes of the automated Pandas query generator have laid the groundwork but face significant limitations, including the generation of numerous invalid queries and lack of flexibility in the types of queries generated. This project aims to address these shortcomings by enhancing the query generator with user-defined parameters, support for additional data types, and improved algorithms to reduce the number of queries with empty result sets. The resulting system can generate varied, semantically and syntactically correct Pandas queries based on a provided data schema and user-defined parameters, facilitating efficient query execution and robust model training.

1. Introduction

1.1 Motivation

The query generator was developed to facilitate zero-shot learning for estimating execution time and cardinality of queries. “The main promise of zero-shot cost estimation is that a trained model can predict the query runtime on a new database” [1], which is critical for efficient database management and analytics. Research methodologies like zero-shot learning require training on thousands of diverse queries involving multiple table joins. Consequently, achieving scalability in query generation across diverse schemas is increasingly critical.

Another application of the query generator is testing scheduler performance of Database Management Systems (DBMS), which requires generating test databases and a set of queries that will be executed against these databases. One approach to generating test databases involves using a database schema with a set of primary and foreign key constraints [2]. Another technique is reverse query processing (RQP), which takes a query and its result as input to populate datasets with test data. [3] While these are effective approaches to generate test data on SQL DBMS, little research has been conducted on query generation for Python Pandas. The Pandas library is widely used in the data science community due to its user-friendly syntax and optimized performance for handling relational and labeled datasets. It also introduces powerful data structures such as Series (one-dimensional) and DataFrames (two-dimensional).

2. Background Information

2.1 First Prototype

The first prototype for the automated query generator [4], developed by Dailun Li, generates an intermediate representation for row operations on Pandas DataFrames as indicated in Table 1:

Operation	Pandas Representation	Intermediate Representation
Selection	df[df[<conditions>]]	Selection (df, <conditions>)
Projection	df[[<columns>]]	Projection (df, <columns>)
Merge	df1.merge(df2,<l_col>,<r_col>)	Merge(df2, <l_col>, <r_col>)
Groupby	df.groupby(<columns>)	Groupby(df, <columns>)
Aggregation	df.agg(<f>)	Agg(df, <f>)

Table 1: Intermediate Representation for Pandas operations

Each selection condition in the intermediate representation (IR) is defined as a column, an operator and a value. For example, the Pandas query:

df[df["col1"]] == 3

is translated into IR format as:

selection(df, condition = ["col1", operator.eq, val = 3])

The aggregation function <f> is one of "min", "max", "mean" or "count".

These operations in IR format are then combined to form a set of merged and unmerged queries where each query is output on a single line. Dailun's prototype only works on datasets from the TPC-H benchmark [5], which need to be passed as input files to the query generator.

2.2 Second prototype

In addition to Dailun's work, Hongxin Huo's prototype [6] adopts a tree-based data structure to divide the one-liner output queries into several subqueries, each building on the resulting dataframes from the previous lines, as shown in Figure 1. While Dailun's one-liner queries are easier to execute on test dataframes, Hongxin's format is more readable.

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')
```

Figure 1: One-liner vs multi-line queries [6]

Moreover, Hongxin's version generates syntactically correct queries based only on a relational schema from a JSON file, which removes the need for using actual data files like in the first prototype. The relational schema contains information about data types, data ranges, primary keys and foreign keys for each entity and its attributes, as shown in Figure 2.

```
{
   "entities": {
      "orders": {
           "properties": { "O_ORDERKEY": { "type": "int", "min": 1, "max": 800 }, "O_CUSTKEY": {
"type": "int", "min": 302, "max": 149641 }, "ORDERSTATUS": { "type": "enum", "values": ["O", "F",
"P"] }, "TOTALPRICE": { "type": "float", "min": 1156.67, "max": 355180.76 }, "ORDERDATE": {
"type": "date", "min": "1992-01-13", "max": "1998-07-21" }, "ORDERPRIORITY": { "type": "enum",
"values": ["1-URGENT", "2-HIGH", "3-MEDIUM", "4-NOT SPECIFIED", "5-LOW"] }, "CLERK": {
"type": "string", "starting character": ["C"] }, "SHIPPRIORITY": { "type": "int", "min": 0, "max": 0 }} },
         "primary_key": "O_ORDERKEY",
         "foreign_keys": { "O_CUSTKEY": ["C_CUSTKEY","customer"] }
      },
... (customer. lineitem, nation, part, partsupp and supplier entities)
      "region": {
           "properties": { "R_REGIONKEY": { "type": "int", "min": 0, "max": 4 }, "R_NAME": { "type":
"string", "starting character": ["A", "E", "M", "AFR", "AME", "ASI"] }, "R_COMMENT": { "type":
"string", "starting character": ["l", "h", "g", "u"] } },
         "primary_key": "R_REGIONKEY"
      },
   }
}
```


Figure 2: Example Relational Schema for the TPC-H benchmark

2.3 Current prototype

The current prototype has the objective of bringing the automated query generator closer to being publicly released as an open-source project.

While the previous approaches generate syntactically correct queries, little parametrization is possible to generate queries with customized characteristics. The same number of queries are always generated with the same operation types, query complexity and output format. Moreover, some of the generated queries are not adequate for testing due to selection conditions with empty intersections or group by operations before merging. These types of queries produce empty result sets when executed on test dataframes.

The current prototype allows users to customize the query generation process using the following parameters, as indicated in Table 2:

Parameter	Description	Possible values
num_selection	Maximum number of selection conditions per table	An integer from 0 to 3
projection	Whether to include projections	True or False
num_merges	Maximum number of merges in the generated queries	An integer from 0 to 4
group by	Whether to include group by operations	True or False
aggregation	Whether to include aggregation operations	True or False
num_queries	Number of queries to generate	An integer from 1 to 5000
multi_line	Output format for the merged and unmerged queries	True or False

Table 2: Query Generation Parameters

If the multi line parameter is set to "True", each output query is divided into multiple subqueries with one subquery on each line and the main queries are separated by a "Next" delimeter. If set to "False", the queries are output each on one line like in Dailun's prototype.

Users can input these parameters to the query generator in a JSON file passed to the program as a command-line argument upon runtime. At least one of projection or selections must be included to be able to generate queries. If group by is set to True, then aggregation must also be set to True, since a groupby operation without an aggregation does not return a dataframe.

In addition to the query generation parameters, the current version supports more data types in the relational schema and generates queries that are less likely to yield empty result sets during testing.

One limitation of this prototype is that it does not support duplicate column names across tables. In the relational schema for the TPC-H benchmark shown in Figure 1, the PHONE attributes in the Customers and Suppliers entities were renamed to C_PHONE and S_PHONE.

3. Methods

In this section, we will first discuss the general workflow of Hongxin's query generator. Then, we will specify the contributions of the current prototype. A class diagram of the query generator is included in Figure 3 to indicate the data structures and methods used during the query generation process:

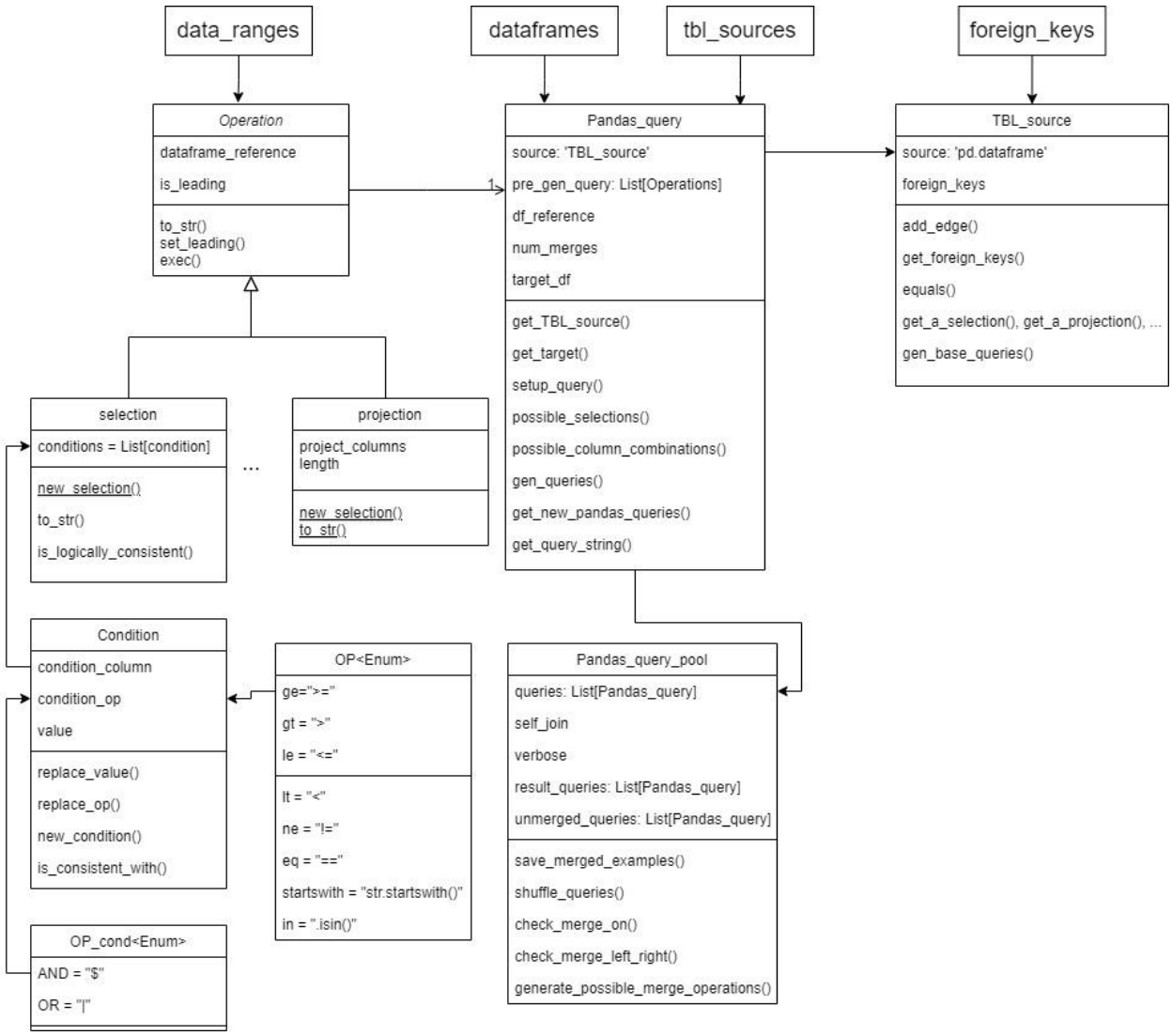

Figure 3: Class Diagram of the Current Prototype

3.1 Algorithmic Description

Query generation

To generate queries using the specifications above, users run the query generator with the following command:

python query_generator.py --schema data_structure.json --params query_parameters.json

where data_structure.json and query_parameters.json are replaced with the actual file names for the relational schema and the query parameters.

The queries are then generated as follows:

1) The JSON files containing the relational schema and the query parameters are parsed into Python dictionaries to make them accessible to the program.

2) Using the information from the parsed input files, the data structures required for the query generation are initialized and stored in Python dictionaries. These include data ranges for each attribute (data_ranges), foreign key relationships between entities (foreign_keys), and a wrapper class for each entity (TBL_sources) containing foreign keys of a specific dataframe. The generator will then extract the information from the relational schema and store them into the dictionaries. These dictionaries will be used to generate the query operations in intermediate representation format.

3) Pandas DataFrames are created for each entity according to the specifications outlined within the parsed JSON schema. In the **create_dataframe()** method of Hongxin's prototype, each dataframe is initially populated with two rows to ensure robustness in scenarios where the data type of the first element in an unpopulated data column is subject to validation. In the latest prototype, each dataframe is populated with 200 rows of data, randomly generated within the data ranges of each attribute and with unique values for primary keys. The purpose of increasing the number of rows to 200 is to execute the generated queries on these dataframes using **pd.eval (query_string, local_dict= local_dict)** to ensure that the queries do not return empty result sets. The number of rows in each dataframe is designed to minimize the number of empty result sets when users test the generated queries on their own dataframes while not being too selective to generate the desired number of queries. The dataframe may have less than 200 rows if the entity has a unique primary key with a range less than 200. The created dataframes are stored into a dictionary (dataframes), and will be used along with the data_ranges, foreign_keys and TBL_sources dictionaries when calling **pd.eval()** on the generated queries.

   Since the generator later uses the **pd.eval()** function to evaluate the string expression defined in **query_string,** the variables referencing the dataframes in **local_dict** must share the same name as the dataframes themselves. This is achieved using the **globals()** function to dynamically create and assign a pandas DataFrame to a global variable named after the value contained in the variable entity, which is the name of the corresponding dataframe.

   ```
   globals()[entity] = create_dataframe(entity_schema, num_rows=200)
   dataframes[entity] = globals()[entity]
   ```

4) For each table source, four base queries are generated using the **gen_base_queries()** function. Each query is a list of selection and/or projection operations. More specifically,

the four base queries consist of four selections, four projections or a mix of selections and projections (2 queries with a selection, 1 with a projection and 1 with a selection and a projection) depending on the query parameters in the input file.

The selection operations are generated with the **get_a_selection()** method, which randomly generates a selection condition on a single attribute by choosing a column from the table source, an operator and a value within the data range of the selected column.

The projection operations are generated in the **get_a_projection()** method, which randomly selects a projection length and a sample of columns from the table source.

The **gen_base_queries()** method then returns a list of four pandas query objects for each table source object.

5) For each base query of each TBL_source, the **gen_queries()** function will generate a nested list of up to 1000 operation lists, where each operation list can be directly transferred into a pandas query object with the TBL_source as the base query.

   First, the function iterates through the operations of each base query. If the operation is a selection, up to 50 new selection conditions per source table attribute are generated using the **possible_selections()** method. These conditions are concatenated with AND/OR operators to form a selection with a length of up to the num_selections parameter in the input file. The resulting selections are on the same TBL_source object as the base query but can be on different attributes. The **is_logically_consistent()** method is added in the current prototype to check if the selection condition has no empty intersections (e.g. x > 5 AND x < 3). If the newly created selection is logically consistent, it is added to a list of operations which can contain up to 1000 unique selections on the same TBL_source object.

   If the operation in the base query is a projection, up to 50 distinct projection operations are generated on the same TBL_source as the base query using the **generate_possible_column_combinations()** method.

   Then the selections or projections are returned as a nested list of operations, where each inner list represents a newly generated query. If the base query has only one selection or projection, each operation list will have a single operation. If the base query has both a selection and a projection, the inner lists contain all possible combinations of one selection and one projection.

   The **get_new_pandas_queries()** method then randomly selects 100 operation lists from the output of **gen_queries()** and executes them on the dataframes generated using **create_dataframe()** to make sure the queries are semantically correct. For example, it excludes queries with selections on columns that are no longer available in the dataframe because of a previous projection. It returns a list of valid pandas_query objects generated from the valid operation lists.

6) The resulting queries are then pooled into a pandas_query_pool object, which contains a list of pandas_query objects generated in the previous step for all the TBL_source objects. The pandas_query_pool class also contains a self-join attribute to specify whether to generate merge operations on a source table with itself.

7) The merged queries are generated by calling the **generate_possible_merge_operations()** method on the pandas_query_pool object. The merged queries are uniformly distributed by number of merges. For example, if we have num_queries = 1000 and num_merges = 3, then it will generate 250 queries each with 0 to 3 merges.

   The resulting unmerged queries are first sampled from the list of pandas_query objects queries generated in step 5. Then, group by and aggregation operations are added to half of the unmerged pandas queries to ensure a balanced distribution of operation types. The group by column is randomly selected from the columns in the source dataframe and the aggregation function, from either min, max, mean or count.

   To generate a query with one merge, two unmerged queries are randomly selected from the list of pandas queries in the pandas_query_pool object. If the two queries do not have the same source tables, the **check_merge_left_right()** function returns the two columns which the dataframes will be joined on. The two dataframes are then merged on the join attributes and the resulting merged query is evaluated on the dataframes generated in **create_dataframe()**. If the query returns an empty result set, then two different unmerged queries are selected. If the result set is non empty, then we add a projection to merged query half the time. Similarly, groupby and aggregation operations are added to approximately half of the merged queries. This process is repeated until the desired number of queries with a single merge operation are generated.

   To generate queries with two merges, we use the same method as above, but instead of two unmerged queries we select one unmerged query and one query with a single merge. The group by operation, if any, is excluded from the singly merged query to avoid generating queries that have a group by before a merge. For queries with three merges, we select one unmerged query and one query with two merges, and so on until we reach the maximum number of merges. Then a list of all the resulting pandas_query objects is returned.

8) The resulting queries are saved into an output file with the name "merged_queries_auto_sf0000". Depending on the value of the multi_line parameter in the input file, either each entire query or each query operation is written on one line.

Figure 4 provides a summary of the query generation process described above:

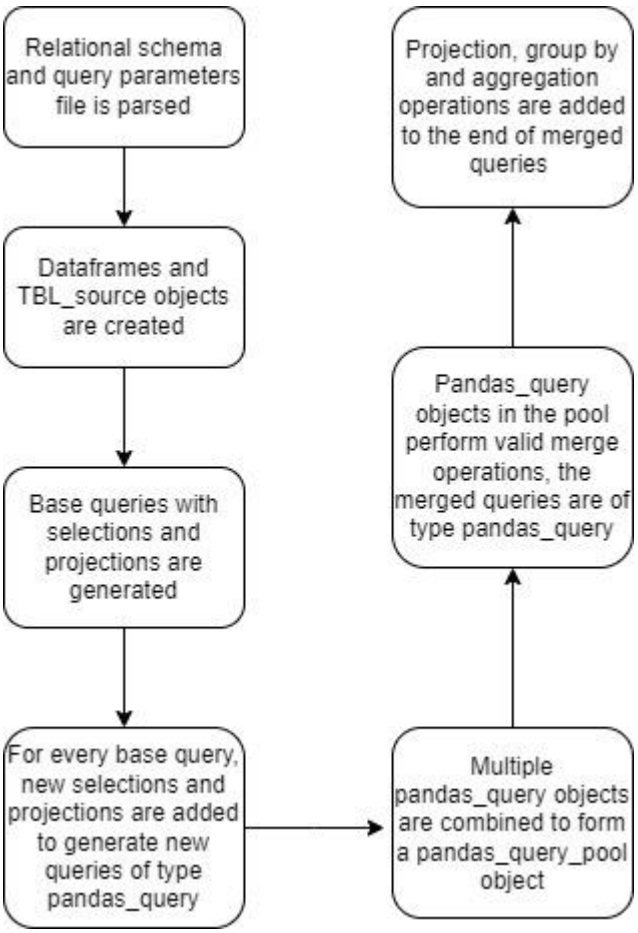

Figure 4: Query Generation Process

The numbers used during the query generation process such as the 50 selections per column and up to 1000 selections per table in **gen_queries()**, selecting 100 queries in **get_new_pandas_queries()** and generating 4 base queries in **gen_base_queries()** were arbitrarily selected in Dailun Li's prototype as a threshold to prevent extra time consumption. This was acceptable in the previous version since it did not have a user-defined parameter to determine the desired number of queries. Using these fixed numbers is a limitation of the current prototype. For example, if the user requests 10 000 queries using only 3 tables, the query generator would not be able to produce 10 000 queries due to the limited number of base queries. On the other hand, in the case where we have 10 tables and we only need 500 queries, the query generator would not be efficient since most of the queries that it generates would not be included in the 500 output queries. An improved query generation algorithm is discussed in the *Further Improvements* section of this report.

3.2 Query parameters configuration

The previous prototype always generates 1300 unmerged queries and 1000 merged queries with a maximum of 3 merges. Users do not have the option to exclude certain operations from the query generation process, and the operations in the resulting queries are unevenly distributed, with all unmerged queries having selections and only 30% with a group by operation. Having an excessive number of selections in the unmerged queries tends to produce more queries with empty result sets when they are merged. Moreover, the complexity of the generated queries is unevenly distributed, with most queries having up to one merge and very few queries with three merges.

The current prototype is more flexible with the types of operations allowed, the number of queries it generates and the number of merges per query. It can exclude any type of query operation, and generate up to 5000 queries with up to 4 merges. The output queries have a balanced distribution of each type of operation and a varying number of merges.

This section details the methodology I employed to implement the query generation parameters described in section 2.4 to generate customized queries with a balanced distribution of operations.

3.2.1 Operation Types and Number of Queries

The num_selections, projection, num_merges, group by and aggregation parameters are used at each step of the query generation process as follows:

The num_selections and projection parameters are first used in the **gen_base_queries()** method before generating the base pandas queries. The four base queries are selection-only if projection is False and projection-only if num_selections is set to 0. If both projection is True and num_selections is non-zero, then two of the base queries are selection-only, one is projection-only and the last one is a combination of a selection and a projection. This means approximately 50% of output queries will have a projection and 75% will have a selection.

The projection parameter is also used in **generate_possible_merge_operations()** to check if a projection operation should be added after merging two queries. Similarly, group by and aggregation operations are added to the end of a query only if the corresponding parameters are set to True.

The num_merges and num_queries parameters determine when to stop generating queries with a given number of merge operations using the following condition:

```
if q_generated >= 2*(k+1)*(max_q // (max_merge+1)):
    break
```

where k is a counter for the current number of merges, q_generated is a counter for the number of valid queries produced so far, max_q is the total number of queries to be generated and max_merge is the maximum number of merges allowed per query

If we set max_q=1000 and max_merge=3, then for k=0 to k=3, we increase the number of merges every time q_generated is equal to a multiple of 500 up to a maximum of 2000 queries. Notice that we only need 1000/4 = 250 queries for each number of merges but we generate double that number. This is because as each query becomes more complex (i.e. has more merge operations), an increasing number of queries produce empty result sets when executed on the sample dataframes from the **create_dataframes()** method. Therefore, more queries must be generated at each complexity level to ensure that we have enough valid queries to generate up to 5000 queries in total and up to 4 merge operations per query.

3.2.2 Multi Line Queries

Dailun's prototype output the generated queries on a single line each. As queries get more complex with increasing number of merges, this output format is less readable. Hongxin's version divides each output query into several subqueries with one query operation on each line, and each query is separated by a "Next" delimiter. One disadvantage of this approach is the difficulty executing these divided queries on a test dataset, since the intermediate dataframe names (e.g. df1, df2) often do not match the name of the source dataframes. The current prototype, through the multi_line parameter in the JSON input file, combines these two approaches by giving the user the option to output each query on a single line or divide it onto multiple lines.

3.3 Extended relational schema

Hongxin's prototype accepts integer, float and string datatypes for attributes in the relational schema. While minimum and maximum values are appropriate data ranges for integers and floats, the relational schema only has one possible startswith character for strings. This makes it difficult to generate meaningful selections on strings, since most datasets with string attributes start with more than one character. Moreover, date attributes are classified as strings, which makes it impossible to generate selections on them using the startswith operator.

The current prototype extends the relational schema in the previous version with date and enum type attributes. The data ranges for dates are minimum and maximum values, and selection operations are generated on dates with the same operators as numerical data types. The range for enum values is simply a list of possible values it can take. Selections are generated on enum type attributes with the equals, not equal to and .isin() membership operator which takes a subset from the list of possible values from the attribute's data range. This approach of generating selections is more precise than using the startswith operator, since an enum might have several distinct values which all start with the same letter.

3.4 Reduction in Queries that Produce Empty Result Sets

This section details the methodology I employed to generate meaningful queries which do not return empty result sets.

3.4.1 Improved Selection on Numerical Data Types

The previous prototype generates a selection condition on integer and float attributes by randomly choosing a column from the source table, an operator and a numerical value within the data range of that attribute. When combining multiple selection conditions on the same attribute, this approach might produce logically inconsistent selections if we have two or more conditions on the same attribute combined with an AND. For example, selections like Customer[C_CUSTKEY > 5 & C_CUSTKEY < 3] or Customer[C_CUSTKEY == 5 & C_CUSTKEY == 3] would always return empty results since the intersection of the two conditions is empty.

The current prototype includes a **is_consistent_with()** method which compares the operators and the values of two conditions on the same attribute and checks if the intersection is empty. However, if two conditions are separated by the OR operator, the selection is still logically consistent even if the intersection is empty. After a selection operation is generated, the **is_logically_consistent()** method gets the conditions which are separated by an AND, then uses **is_consistent_with()** to check if these conditions have an empty intersection.

Another issue with the previous version is that it generates is selection conditions which select values outside of the attributes data range. For example, if the C_CUSTKEY attribute has a range from 1 to 100, the conditions C_CUSTKEY < 1 or C_CUSTKEY > 100 would always return empty result sets. This issue is resolved in the current prototype by changing the **possible_selections()** and **get_a_selection()** methods, which are used to generate selection operations. If the chosen value for the selection condition is the minimum value for that attribute, the < and <= operators are excluded, and for maximum values, the > and >= operators are excluded from the condition.

The current prototype also avoids using the equals (==) operator on conditions where the column is a float. Although these types of selection conditions would not always produce an empty result set in theory, this is often the case since floats have an infinite number of possible values. Therefore, it does not make sense to generate queries like CUSTOMER[C_ACCTBAL==976.50] where C_ACCTBAL is of type float.

Finally, the current prototype includes the **ranges_overlap()** method to check if the data ranges for join attributes are mutually exclusive. If this is the case, then the **check_merge_left_right()** method avoids using this join attribute for a merge operation since it would always return empty result sets. For example, if PS_PARTKEY in the partsupp table has a range of [0,50] and is a foreign key to L_PARTKEY in the lineitem table which has a range of [450, 200 000], these two attributes would not be used to join the partsupp and lineitem tables.

3.4.2 Improved Selection on Strings

In the previous prototype, the “starting character” attribute in the relational schema generates a selection condition on strings using only one possible starting character. This approach leads to the generation of queries with empty result sets, since most string attributes start with a variety of different characters. The current prototype extends the “starting character” attribute in the

relational schema to a list of possible characters or substrings. The **possible_selections()** and **get_a_selection()** methods then select one of the values from that list.

Similarly to integers and floats, the current prototype also checks for logical consistency while generating selection conditions on strings. For example, two conditions on the same attribute with different values for the starting character (e.g. Customer[C_ADDRESS.startswith(‘a’) & C_ADDRESS.startswith(‘c’)] have an empty intersection. The **is_logically_consistent()** method ensures that conditions like these are not combined with an AND operator while generating selections.

3.4.3 Improved Group by and Aggregation

The previous prototype generates group by and aggregation operations on dataframes before merging them, which leads to producing queries with empty result sets since the group by drastically reduces the number of rows and the aggregation function changes the values on the join attributes. For example, here is one of the output queries in Hongxin’s prototype:

```
df54 = coach[(coach['Role'] >= (12)) | (coach['Role'] >= (12)) | (coach['Role'] == (20))]
df55 = df54[['Role','National_name']]
df56 = team[['National_name','Group','Number_player']]
df57 = df56.groupby(by=['Group'])
df58 = df57.agg('count')
df59 = df55.merge(df58, left_on=National_name, right_on=National_name)
```

The above query performs a selection and a projection operation on the *coach* dataframe, then does a projection, group by and aggregation on the *team* dataframe before merging the resulting two dataframes. The *team* dataframe after the group by contains the number of distinct values of ‘National_name’ for each ‘Group’, while the coach dataframe contains the original values of the ‘National_name’ attribute. Merging these two dataframes would not generate a meaningful resulting table, which is almost always empty.

To avoid this issue, the current prototype uses an updated version of the **generate_possible_merge_operations()** method, which only adds group by and aggregation operations to a query after all merges have been executed. Although these two operations are added at the end of the resulting queries for each query complexity level (i.e. number of merges), they are excluded from each pair of queries before merging them. For example, suppose we want to join the following two unmerged queries to generate a merge operation.

```
df704 = customer[(customer['C_NATIONKEY'] != 17)]
df705 = df704[['C_ADDRESS','C_NATIONKEY','C_PHONE','C_ACCTBAL’]]
df706 = df705.groupby(by=['C_ACCTBAL'])
df707 = df706.agg('min', numeric_only=True)

df710 = supplier[['S_ADDRESS','S_ACCTBAL', ‘S_NATIONKEY’]]
df711 = df710.groupby(by=['S_ACCTBAL'])
df712 = df711.agg('count')
```

In the above example, **generate_possible_merge_operations()** would merge df705 with df710 on the join attribute before adding a group by and aggregation on the result.

4 Analysis

4.1 Query Generation Results

Utilizing the data schema illustrated in Figure 1, which consists of eight entities from the TPC-H benchmark, and the parameters from Table 2 with the following values (num_selections=2, projection=True, num_merges=3, group by=True, aggregation=True, num_queries=1000, multi_line=True), query generation tests were conducted. The system produced 1000 queries in 215 seconds. The generation of merged queries took 184 seconds, and the rest of the execution time was taken by the generation of selection, projection, group by and aggregation operations on the base queries. There was a balanced distribution of operation types, with 510 queries having group by and aggregation operations, 550 queries with a projection, 750 queries with a selection, and 250 queries each with 0 to 3 merges.

The performance of the current query generator was much slower than the previous prototype, which generated 1300 unmerged queries in 8 seconds and 1000 merged queries in 50 seconds.

The main reason for this is that we are generating more operations than we need at each step of the query generation process. While generating selection operations on the base queries in the **gen_queries()** method, checking logical consistency for each operation means that some of the generated selections are excluded from the output queries. In **generate_possible_merge_ operations()**, we exclude merged queries which produce empty result sets when executed on the tables from **create_dataframe()**. Since these tables only have a maximum of 200 rows, a significant proportion of the queries are excluded for this reason, especially queries with high number of merges. Moreover, the current prototype generates more queries than its predecessor with two or more merges, and these complex queries take a long time to be executed on the dataframes. Again, more queries than needed are saved for each complexity level (i.e. number of merges) to ensure that we are able to generate enough queries with two or more merges.

4.2 Sample Queries

Some sample generated queries are listed below:

```
df712 = lineitem[(lineitem['RECEIPTDATE'] <= '1995-08-01')]
df713 = df712[['L_PARTKEY','L_SUPPKEY','LINENUMBER', 'DISCOUNT','TAX',
'LINESTATUS','SHIPDATE','COMMITDATE']]
df714 = df713.groupby(by=['L_SUPPKEY'])
df715 = df714.agg('count')
```

The above query performs a projection operation on the *lineitem* dataframe, and outputs the count of each projected attribute for each distinct value of L_SUPPKEY.

Here is more complex query involving multiple table joins:

```
df2503= customer[(customer['MKTSEGMENT'].isin(['AUTOMOBILE','HOUSEHOLD',
'MACHINERY']))]
df2504 = df2503[['C_NAME','C_NATIONKEY','C_PHONE','MKTSEGMENT']]
df2505 = nation[(nation['N_NAME'].str.startswith('C'))]
df2506 = supplier[(supplier['S_NAME'].str.startswith('S')) &
(supplier['S_PHONE'].str.startswith('3'))]
df2507 = df2505.merge(df2506, left_on=N_NATIONKEY, right_on=S_NATIONKEY)
df2508 = df2504.merge(df2507, left_on=C_NATIONKEY, right_on=N_NATIONKEY)
df2509 = df2508[['S_NAME','N_NATIONKEY','N_COMMENT','S_COMMENT',
'S_NATIONKEY','S_ACCTBAL','MKTSEGMENT','S_ADDRESS','N_REGIONKEY','S_PHO
NE','S_SUPPKEY','C_NATIONKEY','C_NAME','C_PHONE','N_NAME']]
```

The above query joins the *customer*, *nation* and *supplier* dataframes. Before merging, it performs a selection and a projection on *customer*, a selection on *nation*, and a selection on *supplier*. Then it merges *nation* with *supplier*, and the resulting dataframe with *customer* on the NATIONKEY foreign key pair. A projection is performed on the final dataframe, which contains attributes from each of the three source dataframes.

4.3 Example program to test output queries

To provide an example of how to use the output of the query generator, the Github repository includes an example program which executes the queries on datasets from the TPC-H benchmark. The program outputs a separate file called “merged_query_execution_results”, which stores the execution time, cardinality of the result set and number of each type of operations for each query. The first few lines of the output file from the example program are shown in Figure 5. The execution metrics are collected to evaluate the performance of the current query generator in terms of generating valid queries (non-empty result) with a mix of different types of operations. Users may collect these execution metrics across different relational schemas to obtain training data for zero-shot learning models designed to predict query execution cost and cardinality.

The query execution in the example program can be replicated for any relational schema using the following steps:

1) The csv files for each dataframe are loaded and the local dictionary defining the context for pd.eval is updated to include these dataframes.

2) The file "merged_queries_auto_sf0000" containing the output queries is opened and read line by line. If the merged queries are output each on a single line, then we only need to extract the relevant part after the "=" sign (as in the execute_merged_queries() function). If each query is output on multiple lines, further processing is needed to combine the operations on each line into a single query (as in the execute_merged_queries_multiline() function).

3) Execute the queries using pd.eval() and get the desired execution metrics.

```
Query, Valid, Execution Time, Cardinality, Selections, Projections, Group by, Aggregations
df0 = orders[['O_CUSTKEY','CLERK']]
df1 = df0.groupby(by=['CLERK'])
df2 = df1.agg('min', numeric_only=True), True, 0.002000093460083008, 182,0,1,0,1,1
df3 = part[(part['P_PARTKEY'] != 2)]
df4 = df3[['P_NAME','MFGR','SIZE','CONTAINER']], True, 0.0010066032409667969, 199,1,1,0,0,0
df5 = customer[(customer['C_CUSTKEY'] == 48)]
df6 = df5[['C_ADDRESS','C_ACCTBAL','MKTSEGMENT']], True, 0.002063274383544922, 1,1,1,0,0,0
df7 = orders[(orders['O_CUSTKEY'] >= 44402)]
df8 = df7[['O_CUSTKEY','ORDERDATE','ORDERPRIORITY','O_COMMENT']]
df9 = df8.groupby(by=['O_CUSTKEY'])
df10 = df9.agg('min', numeric_only=True), True, 0.0019202232360839844, 140,1,1,0,1,1
df11 = customer[(customer['C_CUSTKEY'] < 78)]
df12 = df11[['C_PHONE','MKTSEGMENT','C_COMMENT']], True, 0.0009930133819580078, 77,1,1,0,0,0
df13 = partsupp[(partsupp['P_COMMENT'].str.startswith(' '))]
df14 = df13.groupby(by=['PS_PARTKEY'])
df15 = df14.agg('max', numeric_only=True), True, 0.0019969940185546875, 20,1,0,0,1,1
df16 = orders[(orders['ORDERDATE'] >= '1995-02-11')], True, 0.0, 104,1,0,0,0,0
df17 = orders[(orders['O_COMMENT'].str.startswith('-'))], True, 0.0, 1,1,0,0,0,0
df18 = supplier[(supplier['S_ACCTBAL'] < 2629.86)]
df19 = df18.groupby(by=['S_PHONE'])
df20 = df19.agg('min', numeric_only=True), True, 0.0010058879852294922, 71,1,0,0,1,1
```

Figure 5: Query Execution Metrics for TPC-H Benchmark

4.4 Query Execution Results

In the execution metrics for the example program, most of the output queries return a non-empty result set when executed on the test dataframes. With the following query parameters (num_selections=2, projection=True, num_merges=3, group by=True, aggregation=True, num_queries=1000, multi_line=True), we get approximately 100 out of 1000 queries with empty result sets, which are mostly due to startswith selection conditions on string attributes across merged tables. When num_selections or num_merges is changed to 0, there are almost no empty result sets, meaning that selections and merges must both be present for the output queries to produce empty results.

5 Conclusion and Further Improvements

5.1 Conclusion

The objective of the latest prototype is to make the Pandas query generator ready to be publicly released as an open-source project which would facilitate the testing of scheduler performance and support zero-shot learning in pandas query execution analytics. The current prototype generates meaningful queries, which are less likely to produce empty result sets, with varying complexity and operation types defined by the user. These queries can then be executed on test dataframes which follow the data schema using our example program.

5.2 Further Improvements

One significant improvement to the current prototype involves developing a more efficient algorithm for query generation. Instead of generating a fixed number of base queries and expanding them iteratively, we can adopt a randomized approach to generate each query from scratch according to the desired query characteristics. The new algorithm would work as follows: for each query, select the operations, number of merges, and source tables randomly. Begin by randomly choosing the number of operations and their types (e.g., selection, projection, group by, aggregation). For selection operations, randomly pick columns and values within their data ranges, ensuring logical consistency. For projection operations, select a random subset of columns. Next, determine the number of merges for the query and randomly select pairs of source tables and join attributes, ensuring that the join attributes have overlapping data ranges. Combine the operations into a query, execute it on the sample dataframes to verify it does not produce an empty result set, and store the valid query. Repeat this process until the required number of queries is generated. This approach would improve the efficiency of the query generation process by dynamically adapting to the desired query characteristics and generating exactly the required number of queries.

In the **generate_merge_operations()** method of the current prototype, if the merged query does not return an empty result set on the dataframes with 200 rows from **create_dataframe()**, it might still return empty results on the user's test dataframes, since we do not know the number of rows and the distribution of values in the user's dataframe. Moreover, executing each query on the dataframes initialized by the query generator greatly increases execution time. A more efficient approach could be to accept additional parameters for selection operations, which would determine how selective they are. Since selections with low reduction factors are more likely produce empty result sets, this parameter would allow the user control over the validity of the output queries.

References

[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] Dailun Li. Automated Query Generation for Pandas. Mcgill University.

[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

[6] Hongxin Huo. Refined Automated Query Generation for Pandas. Mcgill University.