**Refined Automated query generation for**

**pandas**

**Ege Satir**

***Mcgill University***

**Relational Schema and Query Generation Parameters**

Before running the query generator, the user must input a relational schema and query generation parameters each in a JSON file. First, create a JSON file storing the relational schema of the datasets with information about entities (dataframes), their attributes, data types and ranges for each attribute, primary keys and foreign keys. In the relational schema, do not include duplicate column names across tables. For example, if you have customer and supplier entities with a PHONE attribute, rename the attributes to C\_PHONE and S\_PHONE. Here is an example relational schema for the TPC-H benchmark:

{

    "entities": {

        "customer": {

            "properties": { "C\_CUSTKEY": { "type": "int", "min": 1, "max": 100 }, "C\_NAME": { "type": "string", "starting character": ["C"] }, "C\_ADDRESS": { "type": "string", "starting character": ["I", "H", "X", "s", "9", "n", "z", "K", "T", "u", "Q", "O", "7", "o", "M", "c", "i", "3", "8", "L", "g", "4", "m", "S", "E", "x", "6", "P", "Y", "J", "j", "q", "a", "e", "b", "0", ",", "B", "F", "R", "r", "p", "D", "l", "U", "h", "w", "d", "v", "f"] }, "C\_NATIONKEY": { "type": "int", "min": 0, "max": 23 }, "C\_PHONE": { "type": "string", "starting character": ["1", "2", "3", "25-", "13-", "27-", "18-", "22-"] }, "C\_ACCTBAL": { "type": "float", "min": -917.25, "max": 9983.38 }, "MKTSEGMENT": { "type": "enum", "values": ["BUILDING", "AUTOMOBILE", "MACHINERY", "HOUSEHOLD", "FURNITURE"] }, "C\_COMMENT": { "type": "string", "starting character": ["i", " ", "s", "l", "r", "c", "t", "e", "o", "n", "a", "p", "h", "u", "k", "g", "y", ".", ",", "d", "f", "q", "w"] } },

            "primary\_key": "C\_CUSTKEY",

            "foreign\_keys": { "C\_NATIONKEY": ["N\_NATIONKEY","nation"] }

        },

*… (lineitem, nation, orders, part, partsupp and region entities)*

        "supplier": {

            "properties": { "S\_SUPPKEY": { "type": "int", "min": 1, "max": 200 }, "S\_NAME": { "type": "string", "starting character": ["S"] }, "S\_ADDRESS": { "type": "string", "starting character": ["N", "e", "f", "J", "o", "c", "b", "u", "p", "8", "q", "S", "Y", "i", "C", "g", "m", "L", "r", "W", "O", "7", "T", " ", "B", "G", "s", "9", "1", "H", "R", "y", "x", "Z", "z", "k", "j", "w", "I", "n", "M", "4", "5", "V", "F", "a", "l", "Q", "0", "U", "D", "h", "v", "2", "X", ",", "t", "E", "P", "6", "3", "d", "K"] }, "S\_NATIONKEY": { "type": "int", "min": 0, "max": 24 }, "S\_PHONE": { "type": "string", "starting character": ["1", "2", "3", "28-", "32-", "26-", "14-", "17-"] }, "S\_ACCTBAL": { "type": "float", "min": -966.20, "max": 9915.24 }, "S\_COMMENT": { "type": "string", "starting character": ["e", " ", "s", "a", "i", "r", "l", "u", "y", "n", "t", "c", "g", "h", "o", "d", "f", "x", "b", "k", ",", ".", "w", "!", "j", "v", "q", "the", "es ", " sl", "bli", "al "]  } },

            "primary\_key": "S\_SUPPKEY",

            "foreign\_keys": { "S\_NATIONKEY": ["N\_NATIONKEY","nation"]}

        }

    }

}

Then create another JSON file to store the following query generation parameters:

* num\_selection: Maximum number of selection conditions per table, an integer from 0 to 3
* projection: Whether to include projections, True or False
* group by: Whether to include group by operations, True or False
* aggregation: Whether to include aggregation operations, True or False
* num\_merges: Maximum number of merges in the generated queries, an integer from 0 to 5
* num\_queries: Number of queries to generate, accepts an integer from 1 to 5000
* multi\_line: Output format for the merged and unmerged queries. If 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.

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.

**Query generation**

To generate queries using the specifications above, 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 is replaced with the actual file names for the relational schema and the query parameters.

The queries are then generated as follows:

1. For each entity in the relational schema, a sample dataframe is created using the create\_dataframe function. This function randomly generates a dataframe with up to 200 rows according to the data ranges for each attribute provided in the relational schema. The dataframe may have less than 200 rows if the enitity has a unique primary key with a range less than 200. The sample dataframe is associated with a variable of the same name using the globals() method:

if not isinstance(entity\_schema['primary\_key'], list):

globals()[entity] = create\_dataframe(entity\_schema, num\_rows=200, primary\_key\_unique=True)

else:

   globals()[entity] = create\_dataframe(entity\_schema, num\_rows=200, primary\_key\_unique=False)

dataframes[entity] = globals()[entity]

1. A table source object is then created for each dataframe. The table source object contains the dataframe, its name and its foreign key pairs.

tbl = TBL\_source(globals()[entity], entity)

tbl\_sources[entity] = tbl

1. The foreign keys pairs are added to the source tables for each entity using the add\_foreignkeys helper function:

def add\_foreignkeys(TBL1:TBL\_source, col1, TBL2:TBL\_source, col2):

    TBL1.add\_edge(col1, col2, TBL2)

    TBL2.add\_edge(col2, col1, TBL1)

1. 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 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 get\_a\_selection() method then returns a list of four pandas query objects on the table source.

1. For each base query of each table source, the gen\_queries() function will generate a list of 100 pandas queries. First, the function iterates through the operations of the base query.

If the operation is a selection, up to 50 selection conditions per column are generated and stored in a dictionary mapping column names to a list of conditions. These conditions are concatenated with AND/OR operators for a selection length of up to the num\_selections parameter in the input file. The is\_logically\_consistent() method checks if the selection condition has no empty intersections. If the newly created selection is logically consistent, it is added to a list of operations which can contain up to 1000 possible selections on the same table source.

If the operation is a projection, up to 50 new projection operations are generated using the generate\_possible\_column\_combinations() method.

Then the selections or projections are returned as a 2D list of operations, where each inner list represents a newly generated query. 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() then randomly selects 100 queries from the output of gen\_queries() and executes them on the sample dataframes created using create\_dataframe() to make sure the queries are syntactically correct. It returns a list of queries which did not raise on exception

1. The resulting queries for all base tables are then pooled into a pandas\_query\_pool object, which contains the list of pandas\_query objects generated in the previous step and parameters to merge the queries.
2. 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 unmerged operations are first sampled from the list of unmerged queries generated in step 5. Then, with a probability of 50%, group by and aggregation operations are added to each unmerged query. The groupby column is randomly selected from the columns in the source dataframe and the aggregation statistic, from either min, max, mean or count.

To generate a query with one merge, two unmerged queries are randomly selected. If the two queries do not have the same source dataframes, 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 query is evaluated on the samples dataframes generated in create\_dataframe(). If it 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 with a 50% probability. The resulting query is added to a dictionary mapping number of merges to a list of queries. It will then be used to generate queries with increasing number of merges. This process is repeated until we generate the desired number of queries with one merge operation.

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 one 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 merged queries is returned.

1. If needed, steps 5-7 can be repeated to generate more queries.
2. 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 whole query or each query operation is written on one line.

**Query execution**

For performance evaluation and other use cases, it might be useful to execute the output queries on test datasets that follow the provided relational schema. The README file 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.

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 is set to 0, there are no empty result sets.