Improved Automated Query Generation for Pandas

Hongxin Huo

McGill University

Abstract

This project is an improved and ameliorated version of automated query generation for Pandas. The essential goal is to simplify the overcomplicated formality of output and reduce the computational timing consumption. With a simple and clear format of output queries, it assists data scientists to understand and analyze the data, and makes it easier to detect the undesired queries. It also eliminates some useless edge cases for efficiency. For example, cases such as having a selection operation which eliminates the desired columns for other operations like projection or aggregation.

1. Introduction

1.1 Motivation

Some data scientists aren’ t familiar with the concept and syntax of Pandas query. In order to provide them with an intuitive and clear understanding of the generated queries, the outputs are broken into smaller steps with a different dataframe index to indicate. My work turns the initially overcomplicated output from one single line to several intermediate steps, with clear indications of which operation has been processed to the queries.

```
df0 = customer[(customer['C_ACCTBAL'] >= 6320) & (customer['C_CUSTKEY'] >= 56)]
[['C_CUSTKEY','C_NAME','C_NATIONKEY','C_PHONE','C_ACCTBAL','C_MKTSEGMENT']]
```

```
df0 = customer[(customer['c_custkey'] >= 3) | (customer['c_acctbal'] < 815.61) |
(customer['c_nationkey'] > 15) | (customer['c_acctbal'] >= 574.61)]
df1 = df0[['c_custkey','c_address','c_nationkey','c_phone','c_acctbal','c_comment']]
df2 = df1[(df1['c_nationkey'] >= 14) | (df1['c_acctbal'] >= 497.65) | (df1['c_custkey'] > 3)]
```

Table 1.1: The above is the previous one line output, while the one below is an example of simplified formality

As for the queries with multiple merge operations, my work uses a binary tree order traversal to keep track of the desired dataframe index. When detecting a merge operation, it first keeps track of the operations before the merge, and then goes into the dataframe query inside the merge operation, and recursively counts the index until merge finishes.

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

```
Next
df0 = lineitem[(lineitem['l_partkey'] <= 207) & (lineitem['l_quantity'] >= 10)]
df1=df0[['l_orderkey','l_partkey','l_suppkey','l_linenumber','l_quantity','l_extendedprice','l_discou
nt','l_returnflag','l_linestatus','l_shipdate','l_receiptdate','l_shipinstruct','l_comment']]
df2 = df1[(df1['l_suppkey'] > 6) | (df1['l_suppkey'] < 6) | (df1['l_orderkey'] != 1) | (df1['l_tax']
>= 0.02) & (df1['l_linenumber'] == 2)]
Next
df3 = orders[(orders['o_orderkey'] == 4) | (orders['o_shippriority'] > 0) | (orders['o_totalprice']
>= 118980.58)]
df4=df3[['o_orderkey','o_custkey','o_orderstatus','o_totalprice','o_orderdate','o_clerk','o_comment']
]
df5 = df4[(df4['o_custkey'] >= 83) | (df4['o_orderkey'] < 4)]
df6 = df2.merge(df5, left_on=l_orderkey, right_on=o_orderkey)
df7=df6[['l_orderkey','l_discount','l_returnflag','l_shipinstruct','o_totalprice','l_receiptdate','l_
quantity','l_linestatus','l_linenumber','l_partkey','o_orderdate','l_extendedprice','o_orderstatus','
o_orderkey','l_suppkey','o_clerk','o_custkey','o_comment','l_comment']]
```

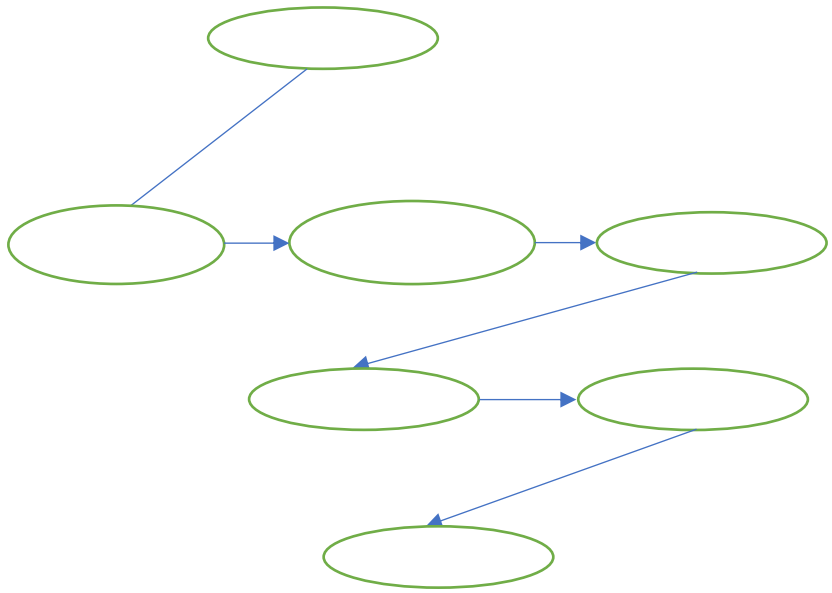
Table1.2: The above is the original output formality, while the one below is an example of simplified version.

2. Methods

In this section, we will discuss the general algorithm and workflow of the updates.

2.1 Recursive binary tree order traversal

As stated above, we need to keep track of the correct dataframe index outside and inside merge operations. For example, if we have a pandas dataframe with multiple layers of merge operations, such as pd1(selection1, projection1, merge1(pd2(aggregation2, merge2(pd3(selection3))))). The binary tree we use to keep track of the dataframe index is as follows:



The desired index output is

df0=pd1(selection1)

df1=df0(projection1)

df2=pd2(aggregation2)

df3=pd3(selection3)

df4=df2(merge, df3)

df5=df1(merge, df4)

2.2 Edge cases

When we have multiple operations performed on the same columns, there might be conflicts between those operations depending on the order. For instance, if there is a projection on column 1 and column 3, then we can’ t have another selection on column 2 since it no longer exists after the first operation. Thus, we need to check for availability before performing each operation. Similar cases are like merge, group by operation after projection etc. For each of these cases, we create a temperate available columns attribute under the pandas query object after each selection or group_by operation. After each projection operation, this available columns attribute will be modified to record the available columns so far.

3. Analysis

We use the TPC-H database as source tables to generate queries.The TPC-H Benchmark is a transaction processing and database benchmark specific to decision support. We set the scale factor to be 0.1 for 100MB amount of data. These include 8 tables with maximum 600000 data entries for a table. Our query generator can successfully generate 3000 unmerged queries in less than 15 seconds, mainly because one has to execute the

operations to generate the target query for merge operations. When it comes to generating merged queries, it takes significantly longer, especially when

considering multiple merges, since the number of columns will be a union of two source tables; if it is not followed by a projection, the average time it takes to loop through the columns of the new dataframe will grow up exponentially. It takes the original version of the query generator less than 100 seconds to generate about 5000 queries, However, since we have binary tree order traversal and eliminating edge cases in the update, it takes less than 140 seconds to generate 5000 queries.

4. Conclusion

The improved query generator has a simplified output formality and reduced unnecessary edge cases. The additional computational time due to recursion is within an accepted range. Such queries can be used in testing scheduler performance or constructing zero-shot learning in pandas query execution analytics. In addition, our intermediate transformation can also be used in encoding the queries to form feature vectors for model training processes.