Towards a random-based approach for synthetic pandas query generation

Liam Scalzulli
McGill University
liam.scalzulli@mail.mcgill.ca

November 30, 2024

Abstract

This paper presents a random-based procedural algorithm for generating synthetic pandas queries. We show that this approach, although vastly simple, can be used to generate valid and complex queries, suitable for processes in data science and machine learning, such as evaluating database systems, predicting cardinality, and estimating execution times.

1 Introduction

Query generation workflows remain a critical feature of modern data science and machine learning workflows. They are involved in cardinality predictor programs, query execution time estimators, and system performance analyzers.

Historically, SQL has been the target market for these kinds of programs, leveraging relational data schemas and existing query sets.

The widespread adoption of the **pandas** data analysis library for Python necessitates further inquiry into how systems might be built to facilitate synthetic query generators for the pandas query language.

What is *pandas*?

pandas is a data analysis library for Python that provides a fast and efficient DataFrame object for data manipulation with integrated indexing.

A DataFrame is two-dimensional, size-mutable, potentially heterogeneous tabular data. We can easily create them from native Python structures:

```
>>> d = {'col1': [1, 2], 'col2': [3, 4]}
>>> df = pd.DataFrame(data=d)
>>> df
   col1  col2
0     1     3
1     2     4
```

We can write queries against this dataframe (df) using familiar Python syntax.

Selection (σ) filters rows based on boolean conditions:

```
>>> df[df['col1'] > 1]
   col1  col2
1     2     4
```

Projection (π) selects specific columns:

```
>>> df[['col1']]  # Select only col1
   col1
0     1
1     2
```

Merging ($\bowtie$) combines DataFrames based on common columns:

```
>>> df1 = pd.DataFrame({'key': ['A'], 'val1': [1]})
>>> df2 = pd.DataFrame({'key': ['A'], 'val2': [3]})
>>> df1.merge(df2, on='key')
  key  val1  val2
0   A     1     3
```

GroupBy (γ) operations aggregate data by specified columns:

```
>>> df = pd.DataFrame({
...   'group': ['A', 'A', 'B', 'B'],
...   'value': [1, 2, 3, 4]
... })
>>> df.groupby(by=['group']).agg('sum')
       value
group
A          3
B          7
```

These fundamental operations form the basis for more complex queries that can be generated by our algorithm. When combined, they enable sophisticated data manipulation and analysis capabilities.

Prior Art

Work has been done at McGill University in the Distributed Information Systems Group on a query generator for the **pandas** query language.

This generator has taken on a few iterations, and each iteration has tacked on complexity, culminating in an approach that could benefit from a restructuring.

Below I will describe these iterations and how they relate to each other:

- Iteration 1 - *Building a foundation.*

  This first iteration worked on by Dailun Li converted pandas operations into an intermediate representation (IR) in Python, e.g. `df[df[ <conditions>]]` would get translated into `Selection(df, <conditions>)` [3]. This IR would then get converted into a string, the final query result.

  The IR is flat, which is unable to accommodate any sub-queries within merge operations, and the output is single-line only.

  This iteration generated queries without merges first, calling them 'unmerged' queries, and then combining them later on to form 'merged' queries.

  There was no user-input allowed, and the program only worked on the TPC-H dataset.

- Iteration 2 - *Describing your data.*

  This second iteration worked on by Hongxin Huo invented a schema format (JSON) required as input by the user in order to generate queries [2].

  This schema describes meta-information of the underlying data we're interested in generating queries for. For instance, it includes things such as column names, their types, valid data ranges, foreign key relationships, etc.

  They then generate what are called 'base' queries, which are queries that are composed solely of selection and projection operations.

  For each of these 'base' queries, more selections and projections are tacked onto them. They then try to merge these queries together, forming 'merged' queries. This is a slight modification of the approach adopted by the first iteration.

  This iteration also adopts tree-based structures, allowing for sub-queries within merge operations, with support for multi-line output.

- Iteration 3 - *User-input driven generation.*

  The third iteration worked on by Ege Satir extended the program with more custom user input to drive the query generation process [1], in addition to adding support for more data types in the required schema format. It also adds a better selection generation algorithm than was previously devised.

This iteration is focused on consolidating this approach into a single, unified random-based algorithm where we start with two user inputs: a schema describing the underlying data we're generating queries for, and parameters that describe information about a single generated query. It is inspired by ideas from all three approaches to achieve its result.

2 Problem Statement

The generation of synthetic database queries presents several key challenges that this work aims to address:

1. **Query Validity**: Generated queries must be syntactically correct and logically consistent, ensuring they can execute successfully against real datasets.
2. **Human-like Structure**: The queries should reflect patterns commonly used by data analysts, avoiding artificial or unrealistic constructions that might bias downstream applications.
3. **Complexity Control**: The generation process should support varying levels of query complexity, from simple operations to multi-stage trans-

formations combining multiple DataFrame operations.

Formally, given a DataFrame schema S, parameters that constrain the query generation process P_Q, and a set of valid pandas operations $O = \{\sigma, \pi, \bowtie, \gamma\}$, we aim to generate a query Q such that:

- Q is a valid composition of operations from O, satisfying the constraints given by P_Q.
- Q produces a non-empty result set when executed on data conforming to schema S.
- Q exhibits structural similarities to human-written queries in terms of operation frequency and complexity.

Our solution focuses on a randomized approach that balances simplicity with effectiveness, while maintaining control over the statistical properties of the generated queries.

3 Algorithm

The algorithm is concerned with solving the query generation problem in the most simple manner. To achieve this we leverage a wide array of user input to guide our generation steps, using random number generators where applicable.

Input

To start, we accept a schema S. This schema gives us meta-data about the data we're generating queries for. It contains information such as the primary keys for tables, foreign key relationships, column information, etc. This information guides us throughout the various sub-algorithms employed for each operation type.

Moreover, we accept a few parameters that describe the structure of individually generated queries:

- Probability of generating a selection (P_σ).
- Maximum number of selection conditions (C_σ).

  These two inputs are used when generating selections for each query. The probability that a query generates a selection is denoted by P_σ. If we're generating a selection, we then use C_σ to determine how many conditions it has by choosing a random number between 1 and C_σ.

- Probability of generating a projection (P_π).
- Maximum number of projection columns (C_π).

  Similar to selections, we use P_π to determine if we're generating a projection for a query. If we do, we use C_π to determine how many columns it has by choosing a random number between 1 and C_π. The number of columns we project will always be between 1 and C_π.

- Maximum number of merge operations ($N_\bowtie$).

  This input determines the maximum amount of merges a query can have. We choose the number of merges for a query by generating a random number between 0 and $N_\bowtie$.

- Probability of generating a group-by (P_γ).
- Maximum number of group-by columns (C_γ).
- Maximum number of aggregation columns (C_Σ).

  Similar to selections and projections, we use P_γ as the probability that a query will have a group-by operation. For pandas-related constraints, we *always* generate a group-by followed by an aggregation operation. We choose how many columns to group-by on with C_γ, choosing a random number between 1 and C_γ and what columns to aggregate on using C_Σ, choosing a random number between 1 and C_Σ.

All this to say, these components inform us what to do during the core query building step of the algorithm.

State

We maintain a few state variables throughout the algorithm:

- Entity (E). The acting entity of the query being generated.

  When we generate queries, there's a chance we're going to step recursively to generate a sub-query. We need to keep track of the entity we're currently dealing with in order to (1) keep track of which properties we have access to given by schema S, and (2) use this entity when generating the string representation of the query.

- The currently available columns (C).

  We need to keep track of what columns we have access to when generating certain operations. If a projection was done beforehand, we don't want to generate a selection using columns outside of that previous projection set.

- The required columns (R). Used when orchestrating projection operations.

  When we generate a merge, we need to preserve the join key for sub-queries. If we step recursively to generate a sub-query, we need to know what columns we must take into account when generating a projection. If we throw away a join column, the entire merge is no longer valid, thus rendering the entire query invalid.

- Merge entities (M). A set of entities that this query is composed of.

  Let an atomic query be a query that acts on a single entity. A generated query is composed of atomic queries. When merging queries together, we need to consider the entity set given by its atomic query set, because we don't want these sets to intersect. For instance, if we have a query that's composed of entities customers and orders, we don't want to merge it with a query that's composed of a single entity, orders.

By definition, $R \subseteq C$ holds throughout the life-cycle of the algorithm.

Query Building

The query building step in the algorithm decides which operation we should be generating, based on the input parameters described above.

The pseudo-code for this procedure is described as follows:

```
if |C| > 0 and C_σ > 0 and rand() < P_σ
  generate a selection

if |C| > 0 and C_π > 0 and rand() < P_π
  generate a projection

from 0 to rand(0, N_⋈)
  if we can generate a merge, do it
  otherwise give up

if |C| > 0 and C_γ > 0 and rand() < P_γ
  generate a group by operation
```

This procedure constrains the generated query in a few subtle ways:

- Each individually generated query can have **at most** 1 selection and 1 projection operation. Generated queries can have sub-queries, so the total amount will vary.
- A group by operation is always generated at the end, allowing us to tweak query parameters to ensure we only append one at the end of the outer-most query. This will become more apparent once we describe the merge generation process.

These subtle constraints are in place because we want generated queries to exhibit structural similarities to human-written queries. Without these constraints, we could generate queries with all sorts of bizarre operation patterns, such as a multitude of projections one after the other.

Operation Types

As seen above, the query building step dispatches to our individual operation generation algorithms. In this section, we describe the algorithm for each operation type. Our working set of operations is assumed to be $\{\sigma, \pi, \bowtie, \gamma\}$.

Selection (σ)

Selections, alongside merges, are one of the trickier operations to generate.

A purely random approach to generating conditions could yield nonsensical results, culminating in an empty intersection. For instance, if we're generating conditions for an entity with an integer property x, a possible condition set could be $x < 5 \wedge x > 10$.

The goal is to generate a condition set that satisfies the constraints imposed by C_σ and is logically consistent (no empty intersections).

Assume the data types and the data ranges for each column on all tables are given by schema S. We can proceed with a base algorithm:

```
set COND to be an empty list

set X to be rand(1, min(C_σ, |C|))

from 1 to X
  pick a random column C_i ∈ C

  set T_i to the type of column C_i
  set R_i to be the range of column C_i

  generate and append a condition that
      considers T_i, R_i and references C_i,
      to COND

select on the condition set COND
```

Next we describe generating conditions and adding them to $COND$, while maintaining consistency of the entire condition set.

Consistency conflicts can only be generated when operating on the same value. For instance, a possible solution to this problem would be to cap the number of conditions per column to 1. Since we want to avoid doing this, we have to figure out a way to satisfy the desired number of conditions while maintaining consistency.

We approach this problem by keeping track of previously generated conditions for columns, and grouping conditions for each column in parentheses, joined by random operations. For example, a valid condition set would look like $(x > 3 \land x < 10) \lor (y = 5) \land (z \geq 10 \lor z \leq 40)$.

For numerical columns, maintain current minimum and maximum bounds; generate inequalities that further narrow this range without invalidating it. For example, if a previous condition is $x > 3$, you might add $x < 10$, resulting in a combined condition of $3 < x < 10$.

For categorical columns, avoid adding multiple equality conditions that could conflict; if an equality condition already exists for a column, refrain from adding additional conditions for it.

Group the conditions for each column in parentheses and combine them using logical operators such as `AND` ($\land$) and `OR` ($\lor$) to form a coherent condition set.

This structured method ensures logical consistency and prevents the creation of contradictory conditions that would lead to empty result sets. By carefully tracking and updating the state of each column's conditions, you can build complex yet valid selection criteria that enhance the realism of the generated queries.

In practice, however, it may be more efficient to adopt a purely random-based approach to generating selection condition sets and validate the resulting queries against a test dataset. This strategy simplifies the generation process by foregoing intricate consistency checks during condition creation. Instead, it relies on empirical validation to ensure that the queries produce meaningful results, accepting that some generated conditions may be discarded if they yield empty result sets.

Projection (π)

A projection operation simply selects out of our available columns set C a random subset to project. Before we do this however, we subtract the required column set R from the available columns, appending them right before generating the projection:

```
set P to C - R

if P is empty
  project C

set X to rand(1, min(C_π - |R|, len(P)))

select a subset P_S ⊆ P, |P_S| = X

set C to P_S + R

project P_S + R
```

Merge ($\bowtie$)

Merge operations are special in the sense that they can produce sub-queries. For instance, inspect the following query:

```
nation.merge(
  region[(region['R_NAME'] != 'M')],
  left_on='N_REGIONKEY',
  right_on='R_REGIONKEY'
)
```

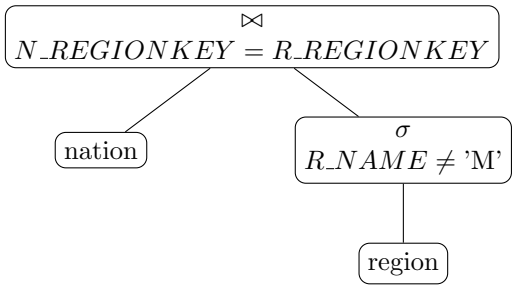
We merge the entity *nation* with rows in *region* whose R_NAME property is not equal to M. The first argument passed to merge is itself a query.

Assume we can get foreign key relationships from S by an entity E ($S_{FK}[E]$). These are given as tuples (c_l, c_f, ε) where c_l is the local column on E, c_f is the foreign column and ε is the foreign entity (or table) containing c_f.

We use all of this information to suggest a recursive approach, which is outlined in the following algorithm:

```
set candidates to an empty list

for c_l, c_f, ε from S_FK[E]
  if c_l ∈ C
    add c_l, c_f, ε to candidates

if candidates is empty
  give up, we can't perform a merge

choose a c_l, c_f, ε ∈ candidates

set X to be a new empty query on ε

set parameters for X
  P_γ,X = 0 // Ensures we have no groupby
  N_⋈,X = N_⋈ - # merges for this query
  ... inherit from the current query

build a query X with R_X + {c_f}, C_X =
    all columns on ε

set C to the union of C with the columns
     returned by X

set M to the union of M with the merge
    entities of query X

subtract N_⋈ by the number of merges in
    query X

merge the current query with X
```

To get a feel for how this algorithm works, lets take a look at the relational algebra tree for the example query shown above:

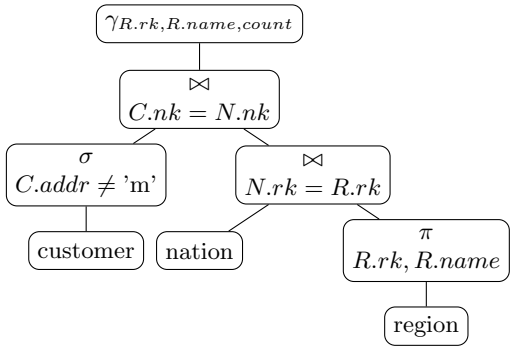


Our algorithm starts off with $E = nation$. We hit a merge operation right away, we remember this and start bottom up from a new entity *region*. Once the *region* query is built we join the two together.

Lets look at another, more complex, query:

```
customer[
  (customer['C_ADDRESS'] != 'm')
].merge(
    nation.merge(
      region[['R_REGIONKEY', 'R_NAME']],
      left_on='N_REGIONKEY',
      right_on='R_REGIONKEY'
    ),
  left_on='C_NATIONKEY',
  right_on='N_NATIONKEY'
).groupby(
  by=['R_REGIONKEY', 'R_NAME']
).agg('count')
```

The relational algebra tree is as follows:



We start with a left entity $E = customer$. We build this query up until we hit the merge operation, where we then start bottom-up from *nation*, stepping once again recursively building up the *region* query. We join each atomic query (a specific set of sub-trees) together with merge operations, appending a groupby

operation to the outer-most query.

Queries are *always* built bottom-up starting from individual entities (or tables) described by the schema S. Each query has a sole acting entity, and we combine these atomic queries with merge operations to form one large and complex query, which may act on all entities described by S.

We're able sustain such complex recursion trees that maintain query validity simply because we preserve join columns all the way down. The projection generation process always takes into account R, a required column set.

Groupby (γ)

For simplicity, we *always* generate the group-by with an aggregation. Since this is always appended at the end of the outer-most query, we need not worry about required columns, so we pick a random subset from our available columns with a mapping of columns to aggregation functions.

```
set X to rand(1, min(C_γ, |C|))

select a subset G_S ⊆ C, |G_S| = X

set A to be C - G_S

if A is empty
  group by on G_S, without aggregation

set Y to be rand(1, min(C_Σ, |A|))

select a subset A_S ⊆ A, |A_S| = Y

set F to be an empty mapping

for each column ∈ A_S
  map column -> a function, in F

group by on G_S, aggregate with F
```

The mapping of columns to a function should be done knowing the columns type. For instance, a column whose type is string should not get mapped with *mean*. This mapping can be defined statically in code, associating column types with appropriate functions.

Output

The algorithm outputs a set of operations, call it the resulting operation set O_R where $\forall o \in O_R, o \in O$.

4 Software Implementation

We provide a full and comprehensive Python package that implements the aforementioned algorithm, called **pqg**, that lets users generate thousands of complex queries for their schemas very quickly.

The package contains both command-line and library interfaces, alongside a companion web application with a friendlier user experience.

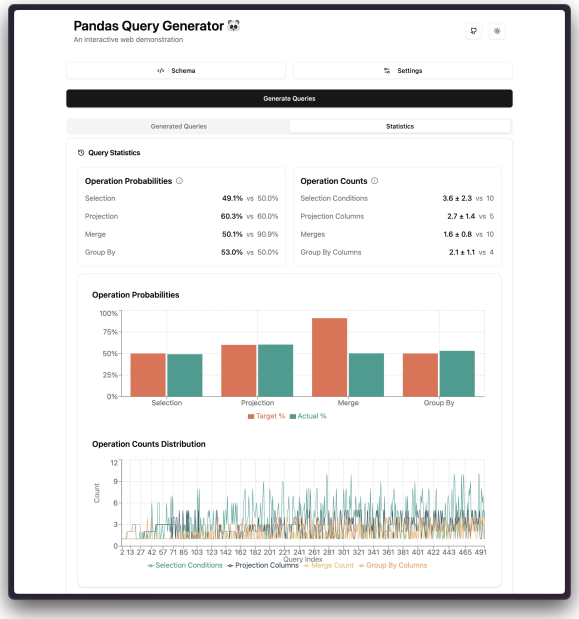


Figure 1: Statistics view in the web application

The core of the tool is written in Python, which consists of the command-line tool and library, whereas the web application is written in React and TypeScript, making use of the library interface distributed over the Python package index (PyPI), using Pyodide.

The code is fully open source, and it lives at https://github.com/DISLMcGill/pandas-query-generator.

Results

The tool is able to generate and execute both single-line and multi-line queries, with options to always ensure they produce some output when ran on sample data. Here are some example queries generated by the tool:

1. Query with a simple selection, merge and groupby.

```
nation[nation['N_NAME'].str.startswith('K')]
  .merge(
    region,
    left_on='N_REGIONKEY',
    right_on='R_REGIONKEY'
  )
  .groupby(by=['R_COMMENT']).agg('count')
```

2. Multiple merge operations.

```
supplier.merge(
  nation.merge(
    region,
      left_on='N_REGIONKEY',
      right_on='R_REGIONKEY'
    ),
    left_on='S_NATIONKEY',
    right_on='N_NATIONKEY'
  )
  .groupby(by=[
    'N_REGIONKEY',
    'N_NATIONKEY'
  ])
  .agg('min', numeric_only=True)
```

3. Multi-line format.

```
df1 = lineitem[[
  'L_PARTKEY',
  'QUANTITY',
  'SHIPDATE',
  'RETURNFLAG',
  'SHIPMODE',
  'LINENUMBER'
]]

df2 = partsupp[
  (partsupp['PS_SUPPKEY'] <= 1720)
]

df3 = df2[[
  'AVAILQTY',
  'PS_PARTKEY',
  'PS_SUPPKEY'
]]

df4 = part[[
  'P_NAME',
  'CONTAINER',
  'PT_COMMENT',
  'RETAILPRICE',
  'MFGR',
  'BRAND',
  'SIZE',
  'P_PARTKEY'
]]

df5 = df3.merge(
  df4,
  left_on='PS_PARTKEY',
  right_on='P_PARTKEY'
)

df6 = df1.merge(
  df5,
  left_on='L_PARTKEY',
  right_on='PS_PARTKEY'
)
```

Each of these queries was generated using the TPC-H reference schema.

Benchmarks

Performance was always at the top of mind during implementation, and it shows in the query generation and execution times.

Below are benchmarks for tasks **pqg** can handle. We use the hyperfine command-line tool to compile an average runtime for these tasks, all ran on an M2 Macbook Pro.

Task	Time (s)
Generating 100 queries	0.8291
Generating 1k queries	1.587
Producing 1k non-empty result queries	3.291
Generating 10k queries	10.028

Table 1: Benchmarks for various tasks

The third iteration provided very few parameters for tweaking the query generation process. It simply attempted to generate and ensure non-empty results for the number of queries specified. Running hyperfine on the program with parameters equivalent to our

Producing 1000 non-empty result queries run gives us **52.747s** on average.

Our implementation is **16.03x**, or **93.76%** faster, on average, than the old one.

Next Steps

There's a ton of cool stuff we can add to this tool, this section describes a few of these ideas:

1. Generating logically consistent selection sets.

   We currently adopt a strictly random-based approach when generating selection condition sets. A purely random-based approach could yield nonsensical results, such as sets that contain empty intersections (e.g. $x < 3 \land x > 5$). One possible solution is to only generation 1 condition per column. This should be avoided, however, and the algorithm should support more than one condition per column.

2. Select aggregation functions based on column types.

   The software currently selects a single aggregation function which gets applied to the rest of the columns which aren't part of the groupby column set. This doesn't make sense in certain cases (e.g. min on a string column). We should take into account column types when generating aggregation functions, and use **pandas** dictionary syntax when generating these types of operations.

3. Handle schemas with multiple entities who have the same property names.

   Our generated queries don't disambiguate column names between tables. If a table shares a column name with another, we have no way to tell which table it came from. We should automatically add a unique prefix or suffix when constructing the query output string, so as to allow for duplicate column names in the schema format.

4. Support multiple output formats.

   We currently only support outputting to a single text file with the command-line interface and web application. We should consider other output formats, such as JSON, Markdown, etc.

5. Allow users to input their own data.

   We currently generate dataframes with at most 1000 rows for each table in the user-provided schema. We execute generated queries on this dataset to test correctness and whether or not they returned an empty result. It would be nice for users to input their own data to execute generated queries on, better reflecting reality with regard to performance.

A comprehensive issue tracker is live on the GitHub repository. The issue tracker documents more than what was just described.

5 Conclusion

This paper presented a simplified, random-based approach to synthetic pandas query generation. By leveraging user-provided schema information and generation parameters, our algorithm produces complex, valid queries that closely resemble real-world data analysis patterns. The implementation, available as the open-source **pqg** package, demonstrates significant performance improvements over previous approaches, achieving a 93.76% reduction in query generation time while maintaining query validity and complexity.

Key contributions of this work include:

- A streamlined algorithm that handles the core pandas operations (σ, π, $\bowtie$, γ) while maintaining query validity through careful state management.
- A flexible parameter system that allows fine-grained control over query complexity and structure.
- An efficient implementation that can generate thousands of queries in seconds, making it suitable for large-scale testing and analysis workflows.

The success of this approach suggests that complex query generation tasks can be effectively addressed through well-designed random-based algorithms, providing a foundation for future work in automated

query generation and testing systems. As pandas continues to evolve as a critical tool in data science workflows, tools like **pqg** will play an increasingly important role in system evaluation, performance testing, and machine learning applications.

6 References

[1] Ege Satir. Customized Query Generation for Pandas. Mcgill University, 2024.

[2] Hongxin Huo. Refined Automated Query Generation for Pandas. Mcgill University, 2023.

[3] Dailun Li. Automated Query Generation for Pandas. Mcgill University, 2022.