pgreaper Documentation

Release 1.0.0

Vincent La

Oct 04, 2017

Contents:

CHAPTER 1

PostgreSQL Default Settings

Before uploading to Postgres, you may want to configure the default connection settings. If default settings are provided, PGReaper can use these to create new databases so you won't have to create them manually.

CHAPTER 2

Uploading TXTs and CSVs to Postgres

Loading TXTs and CSVs to PostgreSQL

Details

Loading Large Files

To conserve memory, all CSVs are read in chunks.

Schema Inference

PGReaper uses a custom CSV parser (written in C++) which simultaneously determines sanitizes and analyzes CSV files. It is capable of differentiating between strings, integers, and floats. The data type of an entire column is determined by this rule:

- If all values are integers, then the column type is **bigint**
- If all values are either floats or integers, then the column type is **double precision**
- Otherwise, the column type is **text**

Minor Caveats:

- Trailing and leading whitespace is ignored when determining data types, so * ” 3.14 ” is considered a floating point number
- Quoted numeric fields are automatically unquoted (Postgres does not tolerate quoted numeric fields)

Corrections PGReaper Can Make

Not all CSV files are perfect, but PGReaper is capable of making some corrections:

- Sanitizing column names

- Dropping rows that are too short or too long
- Unquoting numeric fields before copying

API

CHAPTER 3

Reading JSON

Loading JSON

From Files

PGReaper can load arbitrarily large JSON files assuming they are structured as collections of JSON objects, e.g.

```
[
    {
      "Name": "Julia",
      "Age": 29,
      "Occupation": "Database Administrator"
    }, {
      "Name": "Mark",
      "Age": 30,
      "Occupation": "Barista",
      "Phone": 999-999-9999
    }
]
```

Because it uses a custom JSON reader (which cuts up potentially large files), it is also capable of reading other variants of JSON that Python's *json* module can't handle, such as newline delimited JSON.

Flattening

Currently, PGReaper can optionally flatten out JSON data by its outermost keys. In the example above, the resultant table would have two rows with the columns "name", "age", "occupation", and "phone". The majority of this work is done via Postgres' JSON functions, which is much faster than anything done in pure Python.

API

CHAPTER 4

Reading From Compressed (ZIP) Files

PGReaper is capable of copying specific files located in ZIP archives without decompressing them. (If you are interested in reading in GZIP, BZIP, or LZMA compressed files, use the *compression* parameter on the *copy_csv()* function.)

Reading ZIP Archives

PGReaper provides an intutive way to access files stored in ZIP archives. You can also pass the references to these files to PGReaper's normal reader functions like *copy_csv()*.

Step 1: Read the ZIP Archive

Step 2: Get the Specific File

Notes: File Opening Safety

Opening a file within a ZIP archive using the methods above creates a *ZipReader* object. These objects are like any other file-like objects in Python– supporting *read()* and *readline()* methods, but can only be used as context managers.

CHAPTER 5

The *Table* Data Structure

PGReaper contains a two-dimensional data structure creatively named *Table*. These are lightweight structures which are built on Python's *list* containers but contain a lot of features for easily mapping them into SQL tables.

Creating and Loading *Table* Objects

Motivation

The *Table* is similar in concept to the *DataFrame* (either in R or in pandas) but is optimized for fast appends, iteration, and copying into databases. Because they are structured as lists of lists, all of the Python methods for operating with lists apply to Tables. Furthermore, *Table* objects provide their own set of specialized methods, with an API inspired by R's *dplyr* package.

The key difference between an R or pandas DataFrame and a pgreaper Table is that the Table is designed to be copied into a SQL database, rather than attempt to replace its functionality. As such, most of the methods are geared towards collecting, cleaning and restructuring data, rather than analyzing it.

Structure

Each list in a Table represents a row, while an item in each row represents a cell. If you had a table stored as–say–*world_gdp_data*, then *world_gdp_data[0]* would return a list representing the first row and *world_gdp_data[0][1]* would be the second column of the first row.

Type-Inference

Table objects keep track of the data types inserted into every column, and uses this information to determine the final column type. By default, PGReaper will attempt to treat unrecognized data types as *text* columns. Currently, *Table* is able to recognize text, integer, float, jsonb (list or dict), and datetime types as well as most *numpy* types.

Brief Example: Creating and Loading *Table* Objects

```
from pgreaper import Table, table_to_pg

planes = Table(
 name='planes',
 col_names=['weight', 'length', 'wingspan', 'cost', 'manufacturer']
)

planes.append(...)

table_to_pg(planes, dbname='vehicles')
```

Full Reference: Creating and Modifying Tables

Slicing

Exotic Methods

These are methods which be handy for some edge cases

Operations Which Create New Tables

Groupby

Adding Rows to a Table

Since a *Table* is really just a nested list, you can use the *append()* method. However, Table objects will refuse to add rows which are shorter or longer than the existing rows. This is motivated by the fact that a lot of input sources, especially HTML tables, are not very clean and do not guarantee consistent record lengths even if they should.

If you want to intentionally extend a table, you can use the *add_col()* method to create a new column, or *add_dict()* method which implicitly creates new columns.

Full Reference: Loading Tables to Postgres

Dumping to Files

It is also possible to dump the contents of Table objects into files instead of SQL databases.

Jupyter Notebooks

Table objects take advantage of Jupyter's pretty HTML display.

CHAPTER 6

pandas Integration

Loading pandas DataFrames

By supporting both INSERTs and UPSERTs for DataFrames, when combined with pandas' *read_sql()* functionality, PGReaper turns any Postgres database into a robust store for your pandas-based projects. Compared with other methods for loading DataFrames such as *to_sql()* or a combination of *to_csv()* and *copy_from()*, PGReaper provides:

- Automatic type inference for basic Python types and most *numpy* types
- Properly encoding *jsonb* (dict or list) and *timestamp* (datetime) objects
- Automatic correction of problematic column names, e.g. those that are Postgres keywords
- Support for composite primary keys
- **Support for both INSERT OR REPLACE and UPSERT operations**
  - Faster UPSERT performance using *SELECT unnest()* rather than slower batched INSERTs

Benchmark/Example: pandas DataFrame COPY and UPSERT

COPY

Here, we're going to use the excellent *mimesis* package to generate 500,000 rows of fake data to populate a pandas DataFrame.

Note: For robustness, PGReaper uses both fake and real data in its test suite.

Structure

The resulting SQL table should have 4 text, 1 bigint, and 1 jsonb column.

Results

And for the moment of truth...

```
1 loop, best of 1: 10.7 s per loop
```

UPSERT

Suppose now that we live in such an amazing economy that everybody past 50 has enough money to retire. This means we'll need to update our data to reflect this. As you can see for yourself, this operation will affect about 160,000 rows.

Results

```
1 loop, best of 1: 7.8 s per loop
```

Checking Our Work

```
50 rows affected.
```

Where's the Bottleneck

Apparently it only takes Python about 2.5 seconds to create the 160,000 row UPSERT statement (which includes properly encoding dicts, escaping quotes, and so on). Since *psycopg2* (which PGReaper sends the UPSERT statement to) is basically a C library with Python bindings, and we're only sending one statement, the 5 remaining seconds is most likely taken up primarily by Postgres itself.

```
1 loop, best of 3: 2.42 s per loop
```

CHAPTER 7

HTML Parsing

pgreaper contains a rich HTML parsing module featuring automated <*table*> parsing and Jupyter notebook integration. Because it is a large module on its own, it has its own documentation page.

HTML Table Parsing

Introduction

A lot of useful data is stored in HTML tables, but parsing HTML is an arduous task. Using Python's standard library *html.pars*

- Creating a list of separate HTML tables
- Trying to automatically find column headers
- **Handling different table designs, i.e. handling**
  - <tbody> and <thead> tags
  - rowspan and colspan attributes

Note: Using SQLify's HTML parser in conjunction with Jupyter notebooks is recommended.

Step 1: Reading in HTML

There are two avenues for reading in HTML.

a) Locally Saved HTML Files

b) From the Web

Step 2: Reviewing the Output

The functions above return *TableBrowser* objects, which are basically lists of HTML tables that were found. If viewing in Jupyter Notebook, the code above will display every table with an index next to the name of the table, e.g. *[5] Players of the week*.

Step 3: Cleaning the Tables

If the tables you wanted were parsed 100% correctly and don't require any further processing steps, proceed to step 4. Otherwise, read on.

As seen above, when using the indexing operator on a *TableBrowser* object, you will get a *Table* object back. *Table* objects support a small set of data cleaning methods and contain attributes you may want to use or modify.

Step 4: Saving the Results

After you've cleaned the Table to your satisfaction, you can save the results as either a:

- CSV file
- JSON file
- PostgreSQL Table

PostgreSQL

When saving to a new PostgreSQL database, you can either manually create it, or tell SQLify your preferred default database which should be used to create new databases.

CHAPTER 8

Internals

Information for maintainers and forkers of *pgreaper*.

PGReaper Internals: Schema Inference

This page documents the interals of PGReaper. Unless you are developing, maintaining, forking, or just really curious in PGReaper, this is not really going to be of interest to you.

ColumnList

Originally, the column names and types for a *Table* were stored as the *col_names* and *col_types* attributes respectively. Simply being lists of strings, this approach–while simple–had many limitations. As PGReaper expanded its capabilities, more and more were being demanded of these lists, such as:

- Making sure *col_names* and *col_types* were of the same length
- Returning SQL safe column names while somehow being to keep track of the original column names
- Validating primary keys, i.e. making sure they referred to columns that actually existed
- **Comparing column lists to other column lists to see if one was a subset of another or not**
  - This comes up when performing UPSERTs against existing SQL tables
- Comparing column lists to other column lists to see if they were really the same columns in different orders (a common problem with JSON parsing)
- Having a method to return the integer index corresponding to a column name
- Taking a list of column names and mapping to to their respective integer indices (again, comes up in JSON parsing and also used to implement the *add_dict()* method

Adding all this code to the *Table* structure made it messy, confusing, and harder to test. Therefore, a standalone class that managed column information was created.

SQLType

The *SQLType* object is used to map Python types to SQL types.

PGReaper Internals: Mappings

CHAPTER 9

Index

- genindex
- modindex
- search