parsesql

Release 0.1

Wade Lieurance

Sep 15, 2021

CONTENTS:

CHAPTER

ONE

PARSESQL

Parses SQL into vector statements, formats with parameters, identifies character states/key words, and formats the result for printing.

1.1 Description

This project provides classes in both python and R which can be used to do simple Structured Query Language (SQL) parsing in both languages. The *sql_parser* class functions by first doing a parameter replacement in the SQL if any are passed by the user and found within the string. Parameters in the SQL code are surrounded by {} brackets and named parameters are passed to the class by the user at initialization. The class then assigns a 'state' to each character in the SQL code, where the state can be one of the following:

- default state (none of the other states)
- line comment
- block comment
- dollar quote
- dollar tag
- single quote
- quoted identifier (double quote)

Characters sharing the same state are then concatenated between each state change and stored within dictionary vectors in python or tibbles in R. This allows users to do string replacement depending on state (e.g. only replace a word if it is quoted or replace a function name only if it is neither quoted nor commented). State-identified strings are split into a vector of statements by splitting the text by semi-colons within the default state. Key words for PostgreSQL, SQL-92, SQL:2011, and SQL:2016 are then identified within the default state per user choice of language/standard at class initialization. Finally, the strings are recombined for each statement to create two vector class variables, one with SQL suitable for single statement execution by python or R, and another for printing out a style/color formatted version of each statement.

1.2 Getting Started

Users can initialize an instance of the class most simply in python using the following commands:

```
from parsesql.classes import SqlParser
```

```
my_instance = SqlParser(args)
```

or in R using the R6 class instancing syntax:

```
library(parsesql)
```

```
my_instance = sql_parser$new(args)
```

Doing this will parse the SQL provided and make the output available in *my_instance.sql* (python) or *my_instance$sql* (R). Users can also print a formatted version of the SQL statements by calling print on the instance.

For those users who may want to use regex or string replacement functions, the *string_states* class variable will contains the individual states for each statement.

1.2.1 Dependencies

python:

- colorama

R:

- dplyr
- crayon
- glue
- R6
- readr
- stringr
- tibble

1.2.2 Installing

python (from the terminal):

```
pip install git+https://github.com/wlieurance/parsesql
```

R (from within the R console):

```
devtools::install_github("wlieurance/parsesql/R")
```

1.3 Help

The parser currently relies on the fact that every parameter in the SQL text (i.e. text within brackets {}) needs to be provided by the user at class initialization. If users do not provide a parameter dictionary (python) or a named list(R) the class with still initialize, but providing only partial parameters will cause errors. Support for partial parameter formatting is planned for future releases.

1.4 Authors

Wade Lieurance

1.5 Version History

- 0.1
  - Initial Release

1.6 License

This project is licensed under the GPL3 License - see the LICENSE.md file for details.

CHAPTER

TWO

CLASSES

class classes.**SqlParser**(*text=None*, *file=None*, *standard='SQL:2016'*, *params=None*)

A class to represent SQL statements and their attributes.

Parameters

- **text** (*str*) – Text containing one or more SQL statements separated by a semicolon.
- **file** (*str*) – A path to a file containing SQL to parse.
- **standard** (*str*) – An SQL standard from which to identify key words from. One of: ('SQL-92', 'SQL:2011', 'SQL:2016', 'PostgreSQL')
- **params** (*dict*) – bracketed text in the SQL to replace with a parameter.

char_states

List[List[(str, dict)]]. Each list element is an SQL statement and each sublist is a list of two values in a tuple, *char* and *state*, a single character and a dictionary containing that character's state respectively.

combine_states()

Combines characters that have the same state into strings and stores them along with state and order in a list.

combine_stmts()

Combines SQL with different states into a single statements, both unformatted for execution and formatted for printing/display.

static **format_char**(*my_chars*, *my_state*, *reserved=None*)

Takes a character and a state and assigns a color and style to the character using the colorama package.

Parameters

- **my_chars** (*str*) – A character string containing the text to format.
- **my_state** – A variable containing state information for my_chars.

Returns A color/style formatted version of my_chars.

Return type str

formatted

List[str]. The final SQL statements, split into list format and formatted with color and style via the colorama module.

get_state()

Parses character string multi-statement SQL one character at a time and assigns a state to that character pertaining to whether it is single-quoted, dollar quoted, a quoted identifier, or a comment.

`param_text`
: str. Sourced from *text* or *file* where parameters which have been enclosed in brackets in the string have been replaced by their values in *params*.

`params`
: dict. Contains parameters to replace in the SQL code. Parameters need to be identically named and contained by brackets {} in the SQL file and dictionary names must match the text enclosed in brackets.

`read_file`(*file*)
: Assigns the *text* instance variable with the contents of *file*.

  Parameters `file` (*str*) – The path to a file containing SQL to parse.

`reserved`
: List[str]. Populated from the global 'STANDARDS' variable and filtered during class initialization by choice of the *standard* field.

`sql`
: List[str]. The final SQL statements, split into list format.

`standard`
: str. An SQL standard from which to identify key words from. One of: ('SQL-92', 'SQL:2011', 'SQL:2016', 'PostgreSQL')

`string_states`
: List[List[(str, dict)]]. Contains the same data as *stripped_states*, but the characters sharing the same sequential state have been concatenated and assigned their shared state.

`strip_ws`()
: Finds white space characters at the beginning and end of a list produced by the *get_state* method and removes them.

`stripped_states`
: List[List[(str, dict)]]. Contains the same data as *char_states*, but with whitespace trimmed from the beginning and end of each statement sublist.

`text`
: str. Text containing one or more SQL statements separated by a semicolon.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search