sqlfluff

Release 0.3.6

Alan Cruickshank

Nov 26, 2020

DOCUMENTATION FOR SQLFLUFF:

Bored of not having a good SQL linter that works with whichever dialiect you're working with? Fluff is an extensible and modular linter designed to help you write good SQL and catch errors and bad SQL before it hits your database.

Note: **Sqlfluff** is still in an open alpha phase - expect the tool to change significantly over the coming months, and expect potentially non-backward compatable api changes to happen at any point. In particular:

- **0.1.x** involved a major re-write of the parser, completely changing the behaviour of the tool with respect to complex parsing.
- **0.2.x** added templating support and a big restructure of rules and changed how users might interact with sqlfluff on templated code.
- **0.3.x** drops support for python 2.7 and 3.4, and also reworks the handling of indentation linting in a potentially not backward compatable way.

Want to see where and how people are using Sqlfluff in their projects? Head over to *SQLfluff in the Wild* for inspiration.

CHAPTER

ONE

GETTING STARTED

To get started just install the package, make a sql file and then run sqlfluff and point it at the file. For more details or if you don't have python or pip already installed see *Getting Started*.

```
$ pip install sqlfluff
$ echo "  SELECT a  +  b FROM tbl;  " > test.sql
$ sqlfluff lint test.sql
== [test.sql] FAIL
L:   1 | P:   1 | L003 | Single indentation uses a number of spaces not a multiple of␣
 ↪4
L:   1 | P:  14 | L006 | Operators should be surrounded by a single space unless at␣
 ↪the start/end of a line
L:   1 | P:  27 | L001 | Unnecessary trailing whitespace
```

CHAPTER

TWO

CONTENTS

2.1 Getting Started

To get started with *sqlfluff* you'll need python and pip installed on your machine, if you're already set up, you can skip straight to *Installing sqlfluff*.

2.1.1 Installing Python

How to install *python* and *pip* depends on what operating system you're using. In any case, the python wiki provides up to date instructions for all platforms here.

There's a chance that you'll be offered the choice between python versions. Support for python 2 was dropped in early 2020, so you should always opt for a version number starting with a 3. As for more specific options beyond that, *sqlfluff* aims to be compatible with all current python versions, and so it's best to pick the most recent.

You can confirm that python is working as expected by heading to your terminal or console of choice and typing `python --version` which should give you a sensible read out and not an error.

```
$ python --version
Python 3.6.7
```

For most people, their installation of python will come with `pip` (the python pacakge manager) preinstalled. To confirm this you can type `pip --version` similar to python above.

```
$ pip --version
pip 10.0.1 from ...
```

If however, you do have python installed but not `pip`, then the best instructions for what to do next are on the python website.

2.1.2 Installing sqlfluff

Assuming that python and pip are already installed, then installing *sqlfluff* is straight forward.

```
$ pip install sqlfluff
```

You can confirm it's installation by getting *sqlfluff* to show it's version number.

```
$ sqlfluff version
0.3.1
```

2.1.3 Basic Usage

To get a feel for how to use *sqlfluff* it helps to have a small `.sql` file which has a simple structure and some known issues for testing. Create a file called `test.sql` in the same folder that you're currently in with the following content:

```
SELECT a+b  AS foo,
c AS bar from my_table
```

You can then run `sqlfluff lint test.sql` to lint this file.

```
$ sqlfluff lint test.sql
== [test.sql] FAIL
L:   1 | P:   9 | L006 | Operators should be preceded by a space.
L:   1 | P:  10 | L006 | Operators should be followed by a space.
L:   2 | P:   1 | L003 | Indent expected and not found compared to line #1
L:   2 | P:  10 | L010 | Inconsistent capitalisation of keywords.
L:   2 | P:  15 | L009 | Files must end with a trailing newline.
```

You'll see that *sqlfluff* has failed the linting check for this file. On each of the following lines you can see each of the problems it has found, with some information about the location and what kind of problem there is. The first one has been found on *line 1*, *position 9* (as shown by `L: 1 | P: 9`) and it's a problem with rule *L006* (for a full list of rules, see *Rules Reference*). From this (and the following error) we can see that the problem is that there is no space either side of the + symbol in `a+b`. Head into the file, and correct this issue so that the file now looks like this:

```
SELECT a + b  AS foo,
c AS bar from my_table
```

Rerun the same command as before, and you'll see that the original problems now no longer show up.

```
$ sqlfluff lint test.sql
== [test.sql] FAIL
L:   2 | P:   1 | L003 | Indent expected and not found compared to line #1
L:   2 | P:  10 | L010 | Inconsistent capitalisation of keywords.
L:   2 | P:  15 | L009 | Files must end with a trailing newline.
```

To fix the remaining issues, we're going to use one of the more advanced features of *sqlfluff*, which is the *fix* command. This allows more automated fixing of some errors, to save you time in sorting out your sql files. Not all rules can be fixed in this way and there may be some situations where a fix may not be able to be applied because of the context of the query, but in many simple cases it's a good place to start. Another thing to note is that when fixing, you must always be specific about which rules you wish to fix. This is to minimise any unintended consequences from making large scale changes to your code. In this case we want to try and fix rules *L003*, *L009* and *L010*.

```
$ sqlfluff fix test.sql --rules L003,L009,L010
==== finding violations ====
== [test.sql] FAIL
L:   2 | P:   1 | L003 | Indent expected and not found compared to line #1
L:   2 | P:  10 | L010 | Inconsistent capitalisation of keywords.
L:   2 | P:  15 | L009 | Files must end with a trailing newline.
==== fixing violations ====
3 linting violations found
Are you sure you wish to attempt to fix these? [Y/n]
```

... at this point you'll have to confirm that you want to make the changes by pressing `y` on your keyboard...

```
Are you sure you wish to attempt to fix these? [Y/n] ...
Attempting fixes...
Persisting Changes...
```

(continues on next page)

(continued from previous page)

```
== [test.sql] PASS
Done. Please check your files to confirm.
```

If we now open up `test.sql`, we'll see the content is now different.

```
SELECT a + b  AS foo,
    c AS bar FROM my_table
```

In particular:

- The `FROM` keyword has been capitalised to match the other keywords.
- The second line has been indented to reflect being inside the `SELECT` statement.
- A final newline character has been added at the end of the file (which may not be obvious in the snippet above).

2.1.4 Custom Usage

So far we've covered the stock settings of *sqlfluff*, but there are many different ways that people style their sql, and if you or your organisation have different conventions, then many of these behaviours can be configured. For example, given the example above, what if we actually think that indents should only be two spaces, and rather than upeercase keywords, they should all be lowercase?

To achieve this we create a configuration file named `.sqlfluff` and place it in the same directory as the current file. In that file put the following content:

```
[sqlfluff:rules]
tab_space_size = 2

[sqlfluff:rules:L010]
capitalisation_policy = lower
```

Then rerun the same command as before.

```
$ sqlfluff fix test.sql --rules L003,L009,L010
```

Then examine the file again, and you'll notice that the file has been fixed accordingly.

```
select a + b  as foo,
  c as bar from my_table
```

For a full list of configuration options check out *Default Configuration*. To see how these options apply to specific rules check out the "Configuration" section within each rule's documentation in *Rules Reference*.

2.1.5 Going further

From here, there are several more things to explore.

- To understand how *sqlfluff* is interpreting your file explore the `parse` command. You can learn more about that command and more by running `sqlfluff --help` or `sqlfluff parse --help`.
- To start linting more than just one file at a time, experiment with passing sqlfluff directories rather than just single files. Try running `sqlfluff lint .` (to lint every sql file in the current folder) or `sqlfluff lint path/to/my/sqlfiles`.
- To find out more about which rules are available, see *Rules Reference*.

- To find out more about configuring *sqlfluff* and what other options are available, see *Configuration*.

One last thing to note is that *sqlfluff* is a relatively new project and you may find bugs or strange things while using it. If you do find anything, the most useful thing you can do is to post the issue on github where the maintainers of the project can work out what to do with it. The project is in active development and so updates and fixes may come out regularly.

2.2 SQL in the Wild

SQL has been around for a long time, as a language for communicating with databases, like a communication protocol. More recently with the rise of *data* as a business function, or a domain in it's own right SQL has also become an invaluable tool for defining the *structure* of data and analysis - not just as a one off but as a form of infrastructure as code.

As *analytics* transitions from a profession of people doing one-offs, and moves to building stable and reusable pieces of analytics, more and more principles from software engineering are moving in the analytics space. One of the best articulations of this is written in the viewpoint section of the docs for the open-source tool dbt. Two of the principles mentioned in that article are quality assurance and modularity.

2.2.1 Quality assurance

The primary aim of *sqlfluff* as a project is in service of that first aim of quality assurance. With larger and larger teams maintaining large bodies of SQL code, it becomes more and more important that the code is not just *valid* but also easily *comprehensible* by other users of the same codebase. One way to ensure readability is to enforce a consistent style, and the tools used to do this are called linters.

Some famous linters which are well known in the software community are flake8 and jslint (the former is used to lint the *sqlfluff* project itself).

Sqlfluff aims to fill this space for SQL.

2.2.2 Modularity

SQL itself doesn't lend itself well to modularity, so to introduce some flexibility and reusability it is often templated. Typically this is done in the wild in one of the following ways:

1. Using the limited inbuilt templating abilities of a programming language directly. For example in python this would be using the format string syntax:

   ```
   "SELECT {foo} FROM {tbl}".format(foo="bar", tbl="mytable")
   ```

   Which would evaluate to:

   ```
   SELECT bar FROM mytable
   ```

2. Using a dedicated templating library such as jinja2. This allows a lot more flexibility and more powerful expressions and macros. See the *Jinja Templating Configuration* section for more detail on how this works.

   - Often there are tools like dbt or apache airflow which allow templated sql to be used directly, and they will implement a library like jinja2 under the hood themselves.

All of these templating tools are great for modularity but they also mean that the SQL files themselves are no longer valid SQL code, because they now contain these configured *placeholder* values, intended to improve modularity.

Sqlfluff supports both of the templating methods outlined above, as well as dbt projects, to allow you to still lint these "dynamic" SQL files as part of your CI/CD pipeline (which is great), rather than waiting until you're in production (which is bad , and maybe too late).

During the CI/CD pipeline (or any time that we need to handle templated code), Sqlfluff needs additional info in order to interpret your templates as valid SQL code. You do so by providing dummy parameters in Sqlfluff configuration files. When substituted into the template, these values should evaluate to valid SQL (so Sqlfluff can check its style, formatting, and correctness), but the values don't need to match actual values used in production. This means that you can use *much simpler* dummy values than what you would really use. The recommendation is to use *the simplest* possible dummy value that still allows your code to evaluate to valid SQL so that the configuration values can be a streamlined as possible.

2.3 Vision for SQLfluff

SQLfluff has a few components:

1. A generic parser for SQL which aims to be able to unify SQL written in different dialects into a comparable format. The *parser*.
2. A mechanism for measuring written SQL against a set of rules, with the added ability to fix any violations found. The *linter*.
3. An opinionated set of guidelines for how SQL should be structured and formatted. The *rules*.

The core vision[1] for sqlfluff is to be really good at being the *linter*. The reasoning for this is outlined in *SQL in the Wild*.

Most of the codebase for SQlfluff is the *parser*, mostly because at the point of developing SQLfluff, there didn't appear to be a good option for a whitespace-aware parser that could be used instead.

With regards to the *rules*, SQLfluff aims to be opinionated but it also accepts that many organisations and groups have pre-existing strong conventions around how to write SQL and so ultimately SQLfluff should be flexible enough to support whichever rule set a user wishes to.

Notes

2.4 Let's talk about indentation

If there is one part of building a linter that is going to be controversial it's going to be **indentation** (closely followed by **cApiTaLiSaTiOn**).

Sqlfluff aims to be *opinionated* here, but also *configurable* (see *Configuring Indentation*). The tool will have a default viewpoint and will aim to have views on all of the important aspects of SQL layout, but if you (or your organisation) don't like those views then we aim to allow enough configuration that you can lint in line with your views, and still use *sqlfluff*. For more information on how to configure rules to your own viewpoint see *Configuration*.

So, without further ado, here are the principles we think apply to indentation:

1. **For Keywords within a statement, the first root keyword of each line should be aligned.** For `SELECT` statements, this means that `SELECT`, `FROM`, `WHERE`, `GROUP`, `ORDER`, `HAVING` and `LIMIT`, should all have the same indent. Occasionally, it's actually more legible to have one-line or more compressed statements, and so additionally, if two (or more) of these keywords are on *the same* line, then the second (and any further) keywords won't raise a violation, provided that the *first* was correctly aligned.

[1] Credit to this article for highlighting the importance of a good vision.

- This same logic applies to keywords within subsections, but the likelihood of them being on the same line to start is higher. one example of where this might occur regularly is within aggregate functions.

```
SELECT
   col_a,
   col_b,
   COUNT(*) as num,
   SUM(num) OVER (
      PARTITION BY col_a
      ORDER BY col_b
   ) as an_aggregate_function
FROM tbl_a
GROUP BY 1, 2
```

Note that `PARTITION` and `ORDER` are both aligned on the same line. This also follows the rules around brackets described below.

2. **Line Length**. Long lines are hard to read and many SQL guidelines include a line length restriction. This is (of course) configurable, but the default is 80 characters (in line with the fishtown SQL style guide.)

3. **Bracket behaviour**. For brackets there are three accepted ways:

   a. *Inline brackets*. Bracket expressions that start and end on the same line are fine (providing we don't fall foul of the line length rules above).

```
SELECT GREATEST(1, 6) AS col1 FROM my_table
```

   b. *Brackets with immediate linebreak*. If brackets are followed by an immediate line break (or at least with no other non-code elements after them on that line), then the following line should be indented +1 relative to the indent of the previous line. All elements of the bracketed block should be at this level of indent or deeper. The *closing* bracket of this block should have the same indent as the first element of the line containing the opening bracket.

```
SELECT GREATEST(
    1, 3, 7,
    6, 8, 9
) AS col1
FROM my_table
```

   c. *Brackets with delayed linebreak*. If brackets are followed by content, and then a linebreak *before* the closing bracket, then we assume a *hanging* indent, where the following items of content should have the same indent as the first item of content. In this case, the *closing* bracket should come after the final element *on the same line*.

```
SELECT GREATEST(1, 6, 8,
                6, 7) AS col1
FROM my_table
```

4. **Comments** are dealt with differently, depending on whether they're *block* comments (`/* like this */`), which might optinally inlude newlines, or *inline* comments (`-- like this`) which are necessarily only on one line.

   a. *Block comments* cannot share a line with any code elements (so in effect they must start on their own new line), they cannot be followed by any code elements on the same line (and so in effect must be followed by a newline, if we are to avoid trailing whitespace). None of the lines within the block comment may have an indent less than the first line of the block comment (although additional indentation within a comment is allowed), and that first line should be aligned with the first code element *following* the block comment.

```
SELECT
    /* This is a block comment starting on a new line
    which contains a newline (continuing with at least
    the same indent.
        - potentially containing greater indents
        - having no other code following it in the same line
        - and aligned with the line of code following it */
    this_column as what_we_align_the_column_to
FROM my_table
```

b. *Inline comments* can be on the same line as other code, but are subject to the same line-length restrictions. If they don't fit on the same line (or if it just looks nicer) they can also be the only element on a line. In this latter case they should be aligned with the first code element *following* the comment.

```
SELECT
    -- This is fine
    this_column as what_we_align_to,
    another_column as something_short,  -- Is ok
    case
        -- This is aligned correctly with below
        when indented then take_care
        else try_harder
    end as the_general_guidance
-- Even here we align with the line below
FROM my_table
```

Note: When fixing issues with comment indentation, sqlfluff will attempt to keep comments in their original position but if line length concerns make this difficult, it will either abort the fix, or move *same line* comments up and *before* the line they are currently on. This is in line with the assumption that comments on their own line refer to the elements of code which they come *before*, not *after*.

2.4.1 Configuring Indentation

How indentation is linted is controlled in the rules, but what indentation is expected to be present is controlled by the parser, and therefore configured seperately. One of the key areas for this is the indentation of the `JOIN` expression.

Semantically, a `JOIN` expression is part of the `FROM` expression and therefore would be expected to be indented. However according to many of the most common SQL style guides (including the fishtown SQL style guide and the Mozilla SQL style guide) the `JOIN` keyword is expected to at the same indent as the `FROM` keyword. By default, *sqlfluff* sides with the current consensus, which is to *not* indent the `JOIN` keyword, however this is one element which is configurable.

By setting values in the `sqlfluff:indentation` section of your config file you can control how this is parsed, for example you may work with an indentation similar to that of Baron Schwartz.

By setting your config file to:

```
[sqlfluff:indentation]
indented_joins = True
```

Then the expected indentation will be:

```
SELECT
    a, b, c
```

(continues on next page)

(continued from previous page)

```
FROM my_table
   JOIN another_table
      USING(a)
```

However if no value for `indented_joins` is set, or if it is set to `false` then then following indentation will be expected:

```
SELECT
   a, b, c
FROM my_table
JOIN another_table
   USING(a)
```

By default, *sqlfluff* aims to follow the indentation most common approach to indentation. However, if you have other versions of indentation which are supported by published style guides, then please submit an issue on github to have that variation supported by *sqlfluff*.

2.5 Rules Reference

Rules in *sqlfluff* are implemented as *crawlers*. These are entities which work their way through the parsed structure of a query to evaluate a particular rule or set of rules. The intent is that the definition of each specific rule should be really streamlined and only contain the logic for the rule itself, with all the other mechanics abstracted away.

2.5.1 Specific Rules

Standard SQL Linting Rules.

class Rule_L001(*code*, *description*, ***kwargs*)

`sqlfluff fix` compatible.

Unnecessary trailing whitespace.

Anti-pattern

The • character represents a space.

```
SELECT
    a
FROM foo••
```

Best practice

Remove trailing spaces.

```
SELECT
    a
FROM foo
```

class Rule_L002(*code*, *description*, ***kwargs*)

Mixed Tabs and Spaces in single whitespace.

Configuration

tab_space_size: The number of spaces to consider equal to one tab. Used in the fixing step of this rule. Must be one of range(0, 100).

This rule will fail if a single section of whitespace contains both tabs and spaces.

Anti-pattern

The • character represents a space and the → character represents a tab.
In this example, the second line contains two spaces and one tab.

```
SELECT
••→a
FROM foo
```

Best practice

Change the line to use spaces only.

```
SELECT
••••a
FROM foo
```

class Rule_L003(*code*, *description*, ***kwargs*)

`sqlfluff fix` compatible.

Indentation not consistent with previous lines.

Configuration

tab_space_size: The number of spaces to consider equal to one tab. Used in the fixing step of this rule. Must be one of range(0, 100).

indent_unit: Whether to use tabs or spaces to add new indents. Must be one of ['space', 'tab'].

lint_templated_tokens: Should lines starting with a templating placeholder such as *{{blah}}* have their indentation linted.. Must be one of [True, False].

Note: This rule used to be _"Indentation length is not a multiple of *tab_space_size*"_, but was changed to be much smarter.

Anti-pattern

The • character represents a space.
In this example, the third line contains five spaces instead of four.

```
SELECT
••••a,
•••••b
FROM foo
```

Best practice
Change the indentation to use a multiple of four spaces.

```
SELECT
••••a,
••••b
FROM foo
```

class Rule_L004(*code*, *description*, ***kwargs*)
Mixed Tab and Space indentation found in file.

Anti-pattern
The • character represents a space and the → character represents a tab.
In this example, the second line is indented with spaces and the third one with tab.

```
SELECT
••••a,
→   b
FROM foo
```

Best practice
Change the line to use spaces only.

```
SELECT
••••a,
••••b
FROM foo
```

class Rule_L005(*code*, *description*, ***kwargs*)
`sqlfluff fix` compatible.

Commas should not have whitespace directly before them.

Unless it's an indent. Trailing/leading commas are dealt with in a different rule.

Anti-pattern
The • character represents a space.
There is an extra space in line two before the comma.

```
SELECT
    a•,
    b
FROM foo
```

Best practice
Remove the space before the comma.

```
SELECT
    a,
    b
FROM foo
```

class Rule_L006(*code*, *description*, ***kwargs*)
`sqlfluff fix` compatible.

Operators should be surrounded by a single whitespace.

Anti-pattern
The • character represents a space.
In this example, there is a space missing space between the operator and 'b'.

```
SELECT
    a +b
FROM foo
```

Best practice
Keep a single space.

```
SELECT
    a + b
FROM foo
```

class Rule_L007(*code*, *description*, ***kwargs*)
Operators near newlines should be after, not before the newline.

Anti-pattern
The • character represents a space.
In this example, the operator '+' should not be at the end of the second line.

```
SELECT
    a +
    b
FROM foo
```

Best practice

Place the operator after the newline.

```
SELECT
    a
    + b
FROM foo
```

class Rule_L008(*code*, *description*, ***kwargs*)

`sqlfluff fix` compatible.

Commas should be followed by a single whitespace unless followed by a comment.

Anti-pattern

The • character represents a space.

In this example, there is no space between the comma and 'zoo'.

```
SELECT
    *
FROM foo
WHERE a IN ('plop','zoo')
```

Best practice

Keep a single space after the comma.

```
SELECT
    *
FROM foo
WHERE a IN ('plop',•'zoo')
```

class Rule_L009(*code*, *description*, ***kwargs*)

`sqlfluff fix` compatible.

Files must end with a trailing newline.

class Rule_L010(*code*, *description*, ***kwargs*)

`sqlfluff fix` compatible.

Inconsistent capitalisation of keywords.

Configuration

capitalisation_policy: The capitalisation policy to enforce. Must be one of ['consistent', 'upper', 'lower', 'capitalise'].

Anti-pattern

In this example, 'select 'is in lower-case whereas 'FROM' is in upper-case.

```
select
    a
FROM foo
```

Best practice
Make all keywords either in upper-case or in lower-case

```
SELECT
    a
FROM foo

-- Also good

select
    a
from foo
```

class Rule_L011(*code*, *description*, ***kwargs*)
`sqlfluff fix` compatible.

Implicit aliasing of table not allowed. Use explicit *AS* clause.

Anti-pattern
In this example, the alias 'voo' is implicit.

```
SELECT
    voo.a
FROM foo voo
```

Best practice
Add *AS* to make it explicit.

```
SELECT
    voo.a
FROM foo AS voo
```

class Rule_L012(*code*, *description*, ***kwargs*)
Implicit aliasing of column not allowed. Use explicit *AS* clause.

NB: This rule inherits its functionality from obj:*Rule_L011* but is seperate so that they can be enabled and disabled seperately.

class Rule_L013(*code*, *description*, ***kwargs*)
Column expression without alias. Use explicit *AS* clause.

Configuration
allow_scalar: Whether or not to allow a single element in the select clause to be without an alias. Must be one of [True, False].

Anti-pattern
In this example, there is no alias for both sums.

```
SELECT
    sum(a),
    sum(b)
FROM foo
```

Best practice

Add aliases.

```
SELECT
    sum(a) AS a_sum,
    sum(b) AS b_sum
FROM foo
```

class Rule_L014(*code*, *description*, ***kwargs*)

Inconsistent capitalisation of unquoted identifiers.

Configuration

capitalisation_policy: The capitalisation policy to enforce. Must be one of [‘consistent’, ‘upper’, ‘lower’, ‘capitalise’].

The functionality for this rule is inherited from *Rule_L010*.

class Rule_L015(*code*, *description*, ***kwargs*)

DISTINCT used with parentheses.

Anti-pattern

In this example, parenthesis are not needed and confuse DISTINCT with a function. The parethesis can also be misleading in which columns they apply to.

```
SELECT DISTINCT(a), b FROM foo
```

Best practice

Remove parenthesis to be clear that the DISTINCT applies to both columns.

```
SELECT DISTINCT a, b FROM foo
```

class Rule_L016(*code*, *description*, ***kwargs*)

`sqlfluff fix` compatible.

Line is too long

Configuration

max_line_length: The maximum length of a line to allow without raising a violation. Must be one of range(0, 1000).

tab_space_size: The number of spaces to consider equal to one tab. Used in the fixing step of this rule. Must be one of range(0, 100).

indent_unit: Whether to use tabs or spaces to add new indents. Must be one of ['space', 'tab'].

class Rule_L017(*code*, *description*, ***kwargs*)

`sqlfluff fix` compatible.

Function name not immediately followed by bracket.

Anti-pattern

In this example, there is a space between the function and the parenthesis.

```
SELECT
    sum (a)
FROM foo
```

Best practice

Remove the space between the function and the parenthesis.

```
SELECT
    sum(a)
FROM foo
```

class Rule_L018(*code*, *description*, ***kwargs*)

`sqlfluff fix` compatible.

WITH clause closing bracket should be aligned with WITH keyword.

Anti-pattern

The • character represents a space.

In this example, the closing bracket is not aligned with WITH keyword.

```
WITH zoo AS (
    SELECT a FROM foo
••••)

SELECT * FROM zoo
```

Best practice

Remove the spaces to align the WITH keyword with the closing bracket.

```
WITH zoo AS (
    SELECT a FROM foo
)

SELECT * FROM zoo
```

class Rule_L019(*code*, *description*, ***kwargs*)

`sqlfluff fix` compatible.

Leading/Trailing comma enforcement.

Configuration

comma_style: The comma style to to enforce. Must be one of [‘leading’, ‘trailing’].

Anti-pattern

There is a mixture of leading and trailing commas.

```
SELECT
    a
    , b,
    c
FROM foo
```

Best practice

By default sqlfluff prefers trailing commas, however it
is configurable for leading commas. Whichever option you chose
it does expect you to be consistent.

```
SELECT
    a,
    b,
    c
FROM foo

-- Alternatively, set the configuration file to 'leading'
-- and then the following would be acceptable:

SELECT
    a
    , b
    , c
FROM foo
```

class Rule_L020(*code*, *description*, ***kwargs*)

Table aliases should be unique within each clause.

class Rule_L021(*code*, *description*, ***kwargs*)

Ambiguous use of DISTINCT in select statement with GROUP BY.

Anti-pattern

DISTINCT and GROUP BY are conflicting.

```
SELECT DISTINCT
    a
FROM foo
GROUP BY a
```

Best practice
Remove DISTINCT or GROUP BY. In our case, removing GROUP BY is better.

```
SELECT DISTINCT
    a
FROM foo
```

class Rule_L022(*code*, *description*, ***kwargs*)
`sqlfluff fix` compatible.

Blank line expected but not found after CTE definition.

Anti-pattern
There is no blank line after the CTE. In queries with many
CTEs this hinders readability.

```
WITH plop AS (
    SELECT * FROM foo
)
SELECT a FROM plop
```

Best practice
Add a blank line.

```
WITH plop AS (
    SELECT * FROM foo
)

SELECT a FROM plop
```

class Rule_L023(*code*, *description*, ***kwargs*)
`sqlfluff fix` compatible.

Single whitespace expected after AS in WITH clause.

Anti-pattern

```
WITH plop AS(
    SELECT * FROM foo
)

SELECT a FROM plop
```

Best practice
The • character represents a space.

Add a space after AS, to avoid confusing
it for a function.

```
WITH plop AS•(
    SELECT * FROM foo
)

SELECT a FROM plop
```

class Rule_L024(*code*, *description*, ***kwargs*)

Single whitespace expected after USING in JOIN clause.

Anti-pattern

```
SELECT b
FROM foo
LEFT JOIN zoo USING(a)
```

Best practice

The • character represents a space.
Add a space after USING, to avoid confusing it
for a function.

```
SELECT b
FROM foo
LEFT JOIN zoo USING•(a)
```

class Rule_L025(*code*, *description*, ***kwargs*)

Tables should not be aliased if that alias is not used.

Anti-pattern

```
SELECT
    a
FROM foo AS zoo
```

Best practice

Use the alias or remove it. An usused alias makes code
harder to read without changing any functionality.

```
SELECT
    zoo.a
```

(continues on next page)

(continued from previous page)

```
FROM foo AS zoo

-- Alternatively...

SELECT
    a
FROM foo
```

class Rule_L026(*code*, *description*, ***kwargs*)

References cannot reference objects not present in FROM clause.

Anti-pattern

In this example, the reference 'vee' has not been declared.

```
SELECT
    vee.a
FROM foo
```

Best practice

Remove the reference.

```
SELECT
    a
FROM foo
```

class Rule_L027(*code*, *description*, ***kwargs*)

References should be qualified if select has more than one referenced table/view.

NB: Except if they're present in a USING clause.

Anti-pattern

In this example, the reference 'vee' has not been declared
and the variables 'a' and 'b' are potentially ambiguous.

```
SELECT a, b
FROM foo
LEFT JOIN vee ON vee.a = foo.a
```

Best practice

Add the references.

```
SELECT foo.a, vee.b
FROM foo
LEFT JOIN vee ON vee.a = foo.a
```

class Rule_L028(*code*, *description*, ***kwargs*)

References should be consistent in statements with a single table.

Configuration

single_table_references: The expectation for references in single-table select. Must be one of [‘consistent’, ‘qualified’, ‘unqualified’].

Anti-pattern

In this example, only the field *b* is referenced.

```
SELECT
    a,
    foo.b
FROM foo
```

Best practice

Remove all the reference or reference all the fields.

```
SELECT
    a,
    b
FROM foo

-- Also good

SELECT
    foo.a,
    foo.b
FROM foo
```

class Rule_L029(*code*, *description*, ***kwargs*)

Keywords should not be used as identifiers.

Configuration

only_aliases: Whether or not to flags violations for only alias expressions or all unquoted identifiers. Must be one of [True, False].

Anti-pattern

In this example, SUM function is used as an alias.

```
SELECT
    sum.a
FROM foo AS sum
```

Best practice

Avoid keywords as the name of an alias.

```
SELECT
    vee.a
FROM foo AS vee
```

class Rule_L030(*code*, *description*, ***kwargs*)

Inconsistent capitalisation of function names.

Configuration

capitalisation_policy: The capitalisation policy to enforce. Must be one of [‘consistent’, ‘upper’, ‘lower’, ‘capitalise’].

The functionality for this rule is inherited from *Rule_L010*.

Anti-pattern

In this example, the two SUM functions don’t have the same capitalisation.

```
SELECT
    sum(a) AS aa,
    SUM(b) AS bb
FROM foo
```

Best practice

Make the case consistent.

```
SELECT
    sum(a) AS aa,
    sum(b) AS bb
FROM foo
```

class Rule_L031(*code*, *description*, ***kwargs*)

`sqlfluff fix` compatible.

Avoid table aliases in from clauses and join conditions.

Anti-pattern

In this example, alias ‘o’ is used for the orders table, and ‘c’ is used for ‘customers’ table.

```
SELECT
    COUNT(o.customer_id) as order_amount,
    c.name
FROM orders as o
JOIN customers as c on o.id = c.user_id
```

Best practice

Avoid aliases.

```
SELECT
    COUNT(orders.customer_id) as order_amount,
    customers.name
FROM orders
JOIN customers on orders.id = customers.user_id

-- Self-join will not raise issue

SELECT
    table.a,
    table_alias.b,
FROM
    table
    LEFT JOIN table AS table_alias ON table.foreign_key = table_alias.
↪foreign_key
```

class Rule_L032(*code*, *description*, ***kwargs*)

Prefer specifying join keys instead of using "USING".

Anti-pattern

```
SELECT
    table_a.field_1,
    table_b.field_2
FROM
    table_a
INNER JOIN table_b USING (id)
```

Best practice

Specify the keys directly

```
SELECT
    table_a.field_1,
    table_b.field_2
FROM
    table_a
INNER JOIN table_b
    ON table_a.id = table_b.id
```

class Rule_L033(*code*, *description*, ***kwargs*)
UNION ALL is preferred over UNION.

Anti-pattern
In this example, UNION ALL should be preferred over UNION

```
SELECT a, b FROM table_1 UNION SELECT a, b FROM table_2
```

Best practice
Replace UNION with UNION ALL

```
SELECT a, b FROM table_1 UNION ALL SELECT a, b FROM table_2
```

class Rule_L034(*code*, *description*, ***kwargs*)
`sqlfluff fix` compatible.

Use wildcards then simple select targets before calculations and aggregates.

Anti-pattern

```
select
    a,
    *,
    row_number() over (partition by id order by date) as y,
    b
from x
```

Best practice
Order "select" targets in ascending complexity

```
select
    *,
    a,
    b,
    row_number() over (partition by id order by date) as y
from x
```

2.5.2 Implementation

class RuleSet(*name*, *config_info*)
Class to define a ruleset.

A rule set is instantiated on module load, but the references to each of it's classes are instantiated at runtime. This means that configuration values can be passed to those rules live and be responsive to any changes in configuration from the path that the file is in.

Rules should be fetched using the *get_rulelist()* command which also handles any filtering (i.e. whitelisting and blacklisting).

New rules should be added to the instance of this class using the *register()* decorator. That decorator registers the class, but also performs basic type and name-convention checks.

The code for the rule will be parsed from the name, the description from the docstring. The eval function is assumed that it will be overriden by the subclass, and the parent class raises an error on this function if not overriden.

copy()
Return a copy of self with a seperate register.

document_configuration(*cls*)
Add a 'Configuration' section to a Rule docstring.

Utilize the the metadata in config_info to dynamically document the configuration options for a given rule.

This is a little hacky, but it allows us to propogate configuration options in the docs, from a single source of truth.

document_fix_compatible(*cls*)
Mark the rule as fixable in the documentation.

get_rulelist(*config*)
Use the config to return the appropriate rules.

We use the config both for whitelisting and blacklisting, but also for configuring the rules given the given config.

Returns `list` of instantiated *BaseCrawler*.

register(*cls*)
Decorate a class with this to add it to the ruleset.

```
@myruleset.register
class Rule_L001(BaseCrawler):
    "Description of rule."

    def eval(self, **kwargs):
        return LintResult()
```

We expect that rules are defined as classes with the name *Rule_XXXX* where *XXXX* is of the form *LNNN*, where L is a letter (literally L for *linting* by default) and N is a three digit number.

If this receives classes by any other name, then it will raise a `ValueError`.

class BaseCrawler(*code*, *description*, ***kwargs*)
The base class for a crawler, of which all rules are derived from.

Parameters

- **code** (`str`) – The identifier for this rule, used in inclusion or exclusion.
- **description** (`str`) – A human readable description of what this rule does. It will be displayed when any violations are found.

_eval(***kwargs*)
Evaluate this rule against the current context.

This should indicate whether a linting violation has occured and/or whether there is something to remember from this evaluation.

Note that an evaluate function shoul always accept ***kwargs*, but if it relies on any available kwargs, it should explicitly call them out at definition.

Returns *LintResult* or `None`.

The reason that this method is called *_eval()* and not *eval* is a bit of a hack with sphinx autodoc, to make it so that the rule documentation auto-generates nicely.

crawl(*segment*, *dialect*, *parent_stack=None*, *siblings_pre=None*, *siblings_post=None*, *raw_stack=None*, *fix=False*, *memory=None*)
Recursively perform the crawl operation on a given segment.

Returns A tuple of (vs, raw_stack, fixes, memory)

static filter_meta(*segments*, *keep_meta=False*)
Filter the segments to non-meta.

Or optionally the opposite if keep_meta is True.

classmethod get_parent_of(*segment*, *root_segment*)
Return the segment immediately containing segment.

NB: This is recursive.

Parameters

- **segment** – The segment to look for.
- **root_segment** – Some known parent of the segment we're looking for (although likely not the direct parent in question).

classmethod make_keyword(*raw*, *pos_marker*)
Make a keyword segment.

classmethod make_newline(*pos_marker*, *raw=None*)
Make a newline segment.

classmethod make_whitespace(*raw*, *pos_marker*)
Make a whitespace segment.

class LintResult(*anchor=None*, *fixes=None*, *memory=None*, *description=None*)
A class to hold the results of a crawl operation.

Parameters

- **anchor** (BaseSegment, optional) – A segment which represents the *position* of the a problem. NB: Each fix will also hold it's own reference to position, so this position is mostly for alterting the user to where the *problem* is.
- **fixes** (list of *LintFix*, optional) – An array of any fixes which would correct this issue. If not present then it's assumed that this issue will have to manually fixed.
- **memory** (dict, optional) – An object which stores any working memory for the crawler. The *memory* returned in any *LintResult* will be passed as an input to the next segment to be crawled.
- **description** (str, optional) – A description of the problem identified as part of this result. This will override the description of the rule as what gets reported to the user with the problem if provided.

to_linting_error(*rule*)
Convert a linting result to a SQLLintError if appropriate.

class LintFix(*edit_type*, *anchor*, *edit=None*)
A class to hold a potential fix to a linting violation.

Parameters

- **edit_type** (str) – One of *create*, *edit*, *delete* to indicate the kind of fix this represents.

- **anchor** (BaseSegment) – A segment which represents the *position* that this fix should be applied at. For deletions it represents the segment to delete, for creations it implies the position to create at (with the existing element at this position to be moved *after* the edit), for an *edit* it implies the segment to be replaced.
- **edit** (BaseSegment, optional) – For *edit* and *create* fixes, this hold the segment, or iterable of segments to create to replace at the given *anchor* point.

is_trivial()

Return true if the fix is trivial.

Trivial edits are: - Anything of zero length. - Any edits which result in themselves.

Removing these makes the routines which process fixes much faster.

The *_eval* function of each rule should take enough arguments that it can evaluate the position of the given segment in relation to it's neighbors, and that the segment which finally "triggers" the error, should be the one that would be corrected OR if the rule relates to something that is missing, then it should flag on the segment FOLLOWING, the place that the desired element is missing.

2.5.3 Inline Ignoring Errors

sqlfluff features inline error ignoring. For example, the following will ignore the lack of whitespace surrounding the `*` operator.

```
a.a*a.b AS bad_1  -- noqa: L006
```

Multiple rules can be ignored by placing them in a comma-delimited list.

```
a.a *  a.b AS bad_2,  -- noqa: L007, L006
```

It is also possible to ignore non-rule based errors, and instead opt to ignore templating (TMP) & parsing (PRS) errors.

```
WHERE dt >= DATE_ADD(CURRENT_DATE(), INTERVAL -2 DAY) -- noqa: PRS
```

Should the need arise, not specifying specific rules to ignore will ignore all rules on the given line.

```
a.a*a.b AS bad_3  -- noqa
```

2.6 Dialects Reference

Sqlfluff is designed to be flexible in supporting a variety of dialects. Not all potential dialects are supported so far, but several have been implemented by the community. Below are a list of the currently available dialects. Each inherits from another, up to the root *ansi* dialect.

For a canonical list of supported dialects, run the **sqlfluff dialects** command, which will output a list of the current dialects available on your installation of sqlfluff.

Note: For technical users looking to add new dialects or add new features to existing ones, the dependent nature of how dialects have been implemented is to try and reduce the amount of repetition in how different elements are defined. As an example, when we say that the *Snowflake* dialect *inherits* from the *PostgreSQL* dialect this is not because there is an agreement between those projects which means that features in one must end up in the other, but that the design of the *Snowflake* dialect was heavily *inspired* by the postgres dialect and therefore when defining the dialect within sqlfuff it makes sense to use *PostgreSQL* as a starting point rather than starting from scratch.

Consider when adding new features to a dialect:

- Should I be adding it just to this dialect, or adding it to a *parent* dialect?
- If I'm creating a new dialect, which dialect would be best to inherit from?
- Will the feature I'm adding break any *downstream* dependencies within dialects which inherit from this one?

2.6.1 ANSI

This is the base dialect which holds most of the definitions of common SQL commands and structures. If the dialect which you're actually using isn't specifically implemented by sqlfluff, using this dialect is a good place to start.

This dialect doesn't intend to be brutal in adhering to (and only to) the ANSI SQL spec *(mostly because ANSI charges for access to that spec)*. It aims to be a representation of vanilla SQL before any other project adds their spin to it, and so may contain a slightly wider set of functions than actually available in true ANSI SQL.

2.6.2 PostgreSQL

This is based around the PostgreSQL spec, and is also part of the inheritance chain for several other dialects which were heavily inspired by this one. The *Snowflake* dialect depends directly on this one, and if you're running AWS Redshift or Greenplum this is the dialect to use (until someone makes a specific dialect).

2.6.3 MySql

The dialect for MySQL.

2.6.4 Teradata

The dialect for Teradata.

2.6.5 BigQuery

The dialect for Google BigQuery.

2.6.6 Snowflake

The dialect for Snowflake, which has much of its syntax inherited from *PostgreSQL*.

2.7 Production Usage

Sqlfluff is designed to be used both as a utility for developers but also to be part of CI/CD pipelines.

2.7.1 Using sqlfluff on a whole sql codebase

The exit code provided by sqlfluff when run as a command line utility is designed to assist usefulness in deployment pipelines. If no violations are found then the exit code will be 0. If violations are found then a non-zero code will be returned which can be interrogated to find out more.

- At the moment all error states related to linting return *65*.
- An error as a result of a sqlfluff internal error will return *1*.

2.7.2 Using sqlfluff on changes using *diff-quality*

For projects with large amounts of (potentially imperfect) SQL code, the full SQLFluff output could be very large, which can be distracting – perhaps the CI build for a one-line SQL change shouldn't encourage the developer to fix lots of unrelated quality issues.

To support this use case, SQLFluff integrates with a quality checking tool called *diff-quality*. By running SQLFluff using *diff-quality* (rather than running it directly), you can limit the the output to the new or modified SQL in the branch (aka pull request or PR) containing the proposed changes.

Currently, `diff-quality` requires that you are using `git` for version control.

NOTE: Installing SQLFluff automatically installs the *diff_cover* package that provides the *diff-quality* tool.

Adding *diff-quality* to your builds

In your CI build script:

1. Set the current working directory to the `git` repository containing the SQL code to be checked.
2. Run *diff-quality*, specifying SQLFluff as the underlying tool:

```
diff-quality --violations sqlfluff
```

The output will look something like:

```
-------------
Diff Quality
Quality Report: sqlfluff
Diff: origin/master...HEAD, staged and unstaged changes
-------------
sql/audience_size_queries/constraints/_postcondition_check_gdpr_compliance.sql (0.0%):
sql/audience_size_queries/constraints/_postcondition_check_gdpr_compliance.sql:5:␣
↪Inconsistent capitalisation of unquoted identifiers.
-------------
Total:   1 line
Violations: 1 line
% Quality: 0%
-------------
```

These messages are basically the same as those provided directly by SQLFluff, although the format is a little different. Note that *diff-quality* only lists the line _numbers_, not the character position. If you need the character position, you will need to run SQLFluff directly.

For more information on *diff-quality*, see the [documentation](https://diff-cover.readthedocs.io/en/latest/). It covers topics such as:

- Generating HTML reports

- Controlling which branch to compare against (i.e. to determine new/changed lines). The default is *origin/master*.
- Configuring *diff-quality* to return an error code if the quality is too low
- Troubleshooting

2.8 Configuration

sqlfluff accepts configuration either through the command line or through configuration files. There is *rough* parity between the two approaches with the exception that *templating* configuration must be done via a file, because it otherwise gets slightly complicated.

For details of what's available on the command line check out the *CLI Reference*.

For file based configuration *sqlfluff* will look for the following files in order. Later files will (if found) will be used to overwrite any vales read from earlier files.

- `setup.cfg`
- `tox.ini`
- `pep8.ini`
- `.sqlfluff`

Within these files, they will be read like an cfg file, and *sqlfluff* will look for sections which start with *sqlfluff*, and where subsections are delimited by a semicolon. For example the *jinjacontext* section will be indicated in the section started with *[sqlfluff:jinjacontext]*.

2.8.1 Nesting

Sqlfluff uses **nesting** in it's configuration files, with files closer *overriding* (or *patching*, if you will) values from other files. That means you'll end up with a final config which will be a patchwork of all the values from the config files loaded up to that path. You don't **need** any config files to be present to make *sqlfluff* work. If you do want to override any values though sqlfluff will use files in the following locations in order, with values from later steps overriding those from earlier:

0. *[. . . and this one doesn't really count]* There's a default config as part of the sqlfluff package. You can find this below, in the *Default Configuration* section.
1. It will look in the user's os-specific app config directory. On OSX this is *~/Library/Preferences/sqlfluff*, Unix is *~/.config/sqlfluff*, Windows is *<home>AppDataLocalsqlfluffsqlfluff*, for any of the filenames above in the main *Configuration* section. If multiple are present, they will *patch/override* eachother in the order above.
2. It will look for the same files in the user's home directory (~).
3. It will look for the same files in the current working directory.
4. *[if parsing a file in a subdirectory of the current working directory]* It will look for the same files in every subdirectory between the current working dir and the file directory.
5. It will look for the same files in the directory containing the file being linted.

This whole structure leads to efficient configuration, in particular in projects which utilise a lot of complicated templating.

2.8.2 Jinja Templating Configuration

When thinking about Jinja templating there are two different kinds of things that a user might want to fill into a templated file, *variables* and *functions/macros*. Currently *functions* aren't implemented in any of the templaters.

Variable Templating

Variables are available in the *jinja* and *python* templaters. By default the templating engine will expect variables for templating to be available in the config, and the templater will be look in the section corresponding to the context for that templater. By convention, the config for the *jinja* templater is found in the *sqlfluff:templater:jinja:context* section and the config for the *python* templater is found in the *sqlfluff:templater:python:context* section.

For example, if passed the following *.sql* file:

```
SELECT {{ num_things }} FROM {{ tbl_name }} WHERE id > 10 LIMIT 5
```

. . . and the following configuration in *.sqlfluff* in the same directory:

```
[sqlfluff:templater:jinja:context]
num_things=456
tbl_name=my_table
```

. . . then before parsing, the sql will be transformed to:

```
SELECT 456 FROM my_table WHERE id > 10 LIMIT 5
```

Note: If there are variables in the template which cannot be found in the current configuration context, then this will raise a *SQLTemplatingError* and this will appear as a violation without a line number, quoting the name of the variable that couldn't be found.

Complex Variable Templating

Two more advanced features of variable templating are *case sensitivity* and *native python types*. Both are illustrated in the following example:

```
[sqlfluff:templater:jinja:context]
my_list=['a', 'b', 'c']
MY_LIST=("d", "e", "f")
my_where_dict={"field_1": 1, "field_2": 2}
```

```
SELECT
    {% for elem in MY_LIST %}
        '{{elem}}' {% if not loop.last %}||{% endif %}
    {% endfor %} as concatenated_list
FROM tbl
WHERE
    {% for field, value in my_where_dict.items() %}
        {{field}} = {{value}} {% if not loop.last %}and{% endif %}
    {% endfor %}
```

. . . will render as. . .

```
SELECT
    'd' || 'e' || 'f' as concatenated_list
FROM tbl
WHERE
    field_1 = 1 and field_2 = 2
```

Note that the variable was replaced in a case sensitive way and that the settings in the config file were interpreted as native python types.

Macro Templating

Macros (which also look and feel like *functions* are available only in the *jinja* templater. Similar to *Variable Templating*, these are specified in config files, what's different in this case is how they are named. Similar to the *context* section above, macros are configured seperately in the *macros* section of the config. Consider the following example.

If passed the following *.sql* file:

```
SELECT {{ my_macro(6) }} FROM some_table
```

. . . and the following configuration in *.sqlfluff* in the same directory (note the tight control of whitespace):

```
[sqlfluff:templater:jinja:macros]
a_macro_def = {% macro my_macro(something) %}{{something}} + {{something * 2}}{%␣
↪endmacro %}
```

. . . then before parsing, the sql will be transformed to:

```
SELECT 6 + 12 FROM some_table
```

Note that in the code block above, the variable name in the config is *a_macro_def*, and this isn't apparently otherwise used anywhere else. Broadly this is accurate, however within the configuration loader this will still be used to overwrite previous *values* in other config files. As such this introduces the idea of config *blocks* which could be selectively overwritten by other configuration files downsteam as required.

In addition to macros specified in the config file, macros can also be loaded from a file or folder. The path to this macros folder must be specified in the config file to function as below:

```
[sqlfluff:templater:jinja]
load_macros_from_path=my_macros
```

In this case, sqlfluff will load macros from any `.sql` file found at the path specified on this variable. The path is interpreted *relative to the config file*, and therefore if the config file above was found at `/home/my_project/.sqlfluff` then sqlfluff will look for macros in the folder `/home/my_project/my_macros/`. Alternatively the path can also be a `.sql` itself.

Note: Throughout the templating process **whitespace** will still be treated rigourously, and this includes **newlines**. In particular you may choose to provide your *dummy* macros in your configuration with different to the actual macros you may be using in production.

REMEMBER: The purpose of providing the option of macros is to *enable* the parsing of templated sql without it being a blocker. It shouldn't be a requirement that the *templating* is accurate - only so far as that is required to enable the *parsing* and *linting* to be helpful.

Builtin Macro Blocks

One of the main use cases which inspired *sqlfluff* as a project was dbt. It uses jinja templating extensively and leads to some users maintaining large repositories of sql files which could potentially benefit from some linting.

Note: *sqlfluff* has now a tighter integration with dbt through the "dbt" templater. It is the recommended templater for dbt projects and removes the need for the overwrites described in this section.

To use the dbt templater, go to *Dbt Project Configuration*.

Sqlfluff anticipates this use case and provides some built in macro blocks in the *Default Configuration* which assist in getting started with dbt projects. In particular it provides mock objects for:

- *ref*: The mock version of this provided simply returns the model reference as the name of the table. In most cases this is sufficient.
- *config*: A regularly used macro in dbt to set configuration values. For linting purposes, this makes no difference and so the provided macro simply returns nothing.

Note: If there are other builtin macros which would make your life easier, consider submitting the idea (or even better a pull request) on github.

2.8.3 Dbt Project Configuration

dbt is not the default templater for *sqlfluff* (it is Jinja). For using *sqlfluff* with a dbt project, users can either use the *jinja* templater (which may be slightly faster, but may not support the full spectrum of macros) or the *dbt* templater, which uses the dbt itself to render the sql (meaning that there is a much more reliable representation of macros, but a potential performance hit accordingly). At this stage we recommend that users try both approaches and choose according to the method that they indent to use *sqlfluff*.

A simple rule of thumb might be:

- If you are using *sqlfluff* in a CI/CD context, where speed is not critical but accuracy in rendering sql is, then the *dbt* templater may be more appropriate.
- If you are using *sqlfluff* in an IDE or on a git hook, where speed of response may be more important, then the *jinja* templater may be more appropriate.

In order to get started using *sqlfluff* with a dbt project you will need the following configuration:

In *.sqlfluff*:

```
[sqlfluff]
templater = dbt
# dbt templating does not keep trailing new lines (L009)
exclude_rules = L009
```

In *.sqlfluffignore*:

```
target/
dbt_modules/
macros/
```

2.8.4 CLI Arguments

You already know you can pass arguments (`--verbose`, `--exclude_rules`, etc.) through the CLI commands (`lint`, `fix`, etc.):

```
$ sqfluff lint my_code.sql -v -exclude_rules L022,L027
```

You might have arguments that you pass through every time, e.g rules you *always* want to ignore. These can also be configured:

```
[sqlfluff]
verbose = 1
exclude_rules = L022,L027
```

Note that while the `exclude_rules` config looks similiar to the above example, the `verbose` config has an integer value. This is because `verbose` is *stackable* meaning there are multiple levels of verbosity that are available for configuration. See *CLI Reference* for more details about the available CLI arguments.

2.8.5 .sqlfluffignore

Similar to Git's `.gitignore` and Docker's `.dockerignore`, sqlfluff supports a `.sqfluffignore` file to control which files are and aren't linted. Under the hood we use the python pathspec library which also has a brief tutorial in their documentation.

An example of a potential `.sqfluffignore` placed in the root of your project would be:

```
# Comments start with a hash.

# Ignore anything in the "temp" path
/path/

# Ignore anything called "testing.sql"
tesing.sql

# Ignore any ".tsql" files
*.tsql
```

Ignore files can also be placed in subdirectories of a path which is being linted and the sub files will also be applied within that subdirectory.

2.8.6 Default Configuration

The default configuration is as follows, note the *Builtin Macro Blocks* in section *[sqlfluff:templater:jinja:macros]* as referred to above.

```
[sqlfluff]
verbose = 0
nocolor = False
dialect = ansi
templater = jinja
rules = None
exclude_rules = None
recurse = 0
output_line_length = 80
runaway_limit = 10
```

(continues on next page)

(continued from previous page)

```
11
12  [sqlfluff:indentation]
13  indented_joins = False
14  template_blocks_indent = True
15
16  [sqlfluff:templater:jinja]
17  apply_dbt_builtins = True
18
19  [sqlfluff:templater:jinja:macros]
20  # Macros provided as builtins for dbt projects
21  dbt_ref = {% macro ref(model_ref) %}{{model_ref}}{% endmacro %}
22  dbt_source = {% macro source(source_name, table) %}{{source_name}}_{{table}}{%␣
    ↪endmacro %}
23  dbt_config = {% macro config() %}{% for k in kwargs %}{% endfor %}{% endmacro %}
24  dbt_var = {% macro var(variable) %}item{% endmacro %}
25
26  # Some rules can be configured directly from the config common to other rules.
27  [sqlfluff:rules]
28  tab_space_size = 4
29  max_line_length = 80
30  indent_unit = space
31  comma_style = trailing
32  allow_scalar = True
33  single_table_references = consistent
34  only_aliases = True
35
36  # Some rules have their own specific config.
37  [sqlfluff:rules:L003]
38  lint_templated_tokens = True
39
40  [sqlfluff:rules:L010]
41  capitalisation_policy = consistent
42
43  [sqlfluff:rules:L014]
44  capitalisation_policy = consistent
45
46  [sqlfluff:rules:L030]
47  capitalisation_policy = consistent
```

2.9 Architecture

2.9.1 Stage 1, the templater

The templater takes raw files and optionally fills in any configurable sections before passing onto the linter. In particular most templating systems use curly brackets *{}* to indicate templatable sections and these are not currently set up in any of the lexers, and so will fail at the next step if not dealt with here.

sqlfluff supports 2 templating engines:

Jinja is the default templater used by *sqlfluff*. Under the covers, dbt also uses Jinja but in *sqlfluff* it is a separate templater which relies directly on the dbt compiler.

For more details on how to configure the templater see *Jinja Templating Configuration*.

2.9.2 Stage 2, the lexer

The lexer takes raw text and splits it into tokens. Nothing is *removed* but whitepace and code are seperated. In principle all identifiers should be seperated at this stage, but they should not be imparted any meaning at this stage. Any files which cannot be lexed, should raise a *SQLLexError*.

While the Lexer is passed a series of raw segments, it will return a single segment, usually a `FileSegment`, which will be used to initite parsing.

2.9.3 Stage 3, the parser

We recursively parse each of the elements, using their in built grammars.

1. Initially we start with a segment, which contains only raw segments. This is normally a `FileSegment`, but it could in theory be any kind of segment, and as we recurse it will be lower and lower sublevels.
2. We then *parse* this element by calling `.parse()`.
    1. This first calls uses the `parse_grammar` if that is present, on which we call `.match()`. The *match* method of this grammar will return a potentially refined structure of the segments within this segment in greater detail than what was initially there. In the example of a `FileSegment`, it first divides up the query into statements and then finishes.
    2. The `.match()` method of any grammar is naturally recursive, and is comprised of *segments* and *grammars*. The match step is still not inert however and child elements of a match grammar may return more specific versions of segments during this phase even though we are not calling `.parse()`
        - *Segments* must implement a `match_grammar` to be used in this way. When `.match()` is called on a segment, this is the grammar which is used to decide whether there is a match.
        - *Grammars* are objects which combine *segments* or other *grammars* together in a pre-defined way. For example the `OneOf` grammar will match if any one of it's child elements match.
    3. Regardless of whether the `parse_grammar` was used, the next step is to recursively call the `.parse()` method of each of the child segments of the grammar. This operation is wrapped in a method called `.expand()`. In the `FileSegment`, the first step will have transformed a seried of raw tokens into `StatmentSegment` segments, and the *expand* step will let each of those segments refine the content within them.
    4. Eventually in that recursive operation we reach segments which have no children (raw elements containing a single token), and so the recursion naturally finishes.
3. If no match is found for the current segment, the contents will be wrapped in an `UnparsableSegment` which can be picked up as a *parsing* error later.
4. In analysing the eventual tree, any *UnparsableSegment* objects should raise a *SQLParseError* which can then be formatted and raised to the user.

Principles within the parser

The parser is arguably the most complicated element of sqlfluff, and is relied on by all the other elements of the tool to do most of the heavy lifting. When working on the parser there are a couple of design principles to keep in mind.

- The core of the grammar is stored in a *dialect*, the root dialect being the *ansi* dialect. The intent here is that for other SQL dialects that they will inherit most of the logic from a parent dialect and then just reimplement the sections that they need to. One reason for the *Ref* grammar is that it allows name resolution of grammar elements at runtime and so a *patched* grammar with some elements overriden can still rely on lower level elements which haven't been overwritten.

- *Segments* and *Grammars* operate roughly interchangably from a dialect. A grammar is something which can be matched against and return a result. These will be used as instantiated classes and usually contain a set of elements (stored in the *_elements* attribute) which are the sub elements which the grammar will use for matching. These may in turn be a mixture of segments and grammars. A segment can be used as either a class or an instance of a class. While matching, the class itself is used, but if it does match, rather than just returning the original segments it was passed unchanged, it will optionally return mutated versions of those segments which may have been given a more specific meaning by the matcher. For example a *RawSegment* containing *123.4* may actually be mutated to an instance of the *NumericLiteralSegment* class when matched against the *NumericLiteralSegment* class.
- All grammars and segments attempt to match as much as they can and will return partial matches where possible. It is up to the calling grammar or segment to decide whether a partial or complete match is required based on the context it is matching in.

2.9.4 Stage 4, the linter

Given the complete parse tree, we now walk that tree to assess the tree for linting errors. A linter is an object which is able to traverse the tree itself, allowing it to choose which objects it examines and which it alerts as a problem.

Some linters, may optionally be able to *fix* the problems they find. If this is the case, they will optionally return a mutated tree as one of their return values, which can be passed to the next linter. In normal operation this will not be what is returned, because it becomes confusing with line references for a user fixing issues manually.

2.10 CLI Reference

2.10.1 sqlfluff

Sqlfluff is a modular sql linter for humans.

```
sqlfluff [OPTIONS] COMMAND [ARGS]...
```

Options

--version
Show the version and exit.

dialects

Show the current dialects available.

```
sqlfluff dialects [OPTIONS]
```

Options

-n, --nocolor
No color - if this is set then the output will be without ANSI color codes.

-v, --verbose
Verbosity, how detailed should the output be. This is *stackable*, so *-vv* is more verbose than *-v*. For the most verbose option try *-vvvv* or *-vvvvv*.

--version
Show the version and exit.

fix

Fix SQL files.

PATH is the path to a sql file or directory to lint. This can be either a file ('path/to/file.sql'), a path ('directory/of/sql/files'), a single ('-') character to indicate reading from *stdin* or a dot/blank ('./' ') which will be interpreted like passing the current working directory as a path argument.

```
sqlfluff fix [OPTIONS] [PATHS]...
```

Options

-n, --nocolor
No color - if this is set then the output will be without ANSI color codes.

-v, --verbose
Verbosity, how detailed should the output be. This is *stackable*, so *-vv* is more verbose than *-v*. For the most verbose option try *-vvvv* or *-vvvvv*.

--version
Show the version and exit.

--logger <logger>
Choose to limit the logging to one of the loggers.

Options parser|linter|rules

--bench
Set this flag to engage the benchmarking tool output.

--ignore <ignore>
Ignore particular families of errors so that they don't cause a failed run. For example *–ignore parsing* would mean that any parsing errors are ignored and don't influence the success or fail of a run. Multiple options are possible if comma seperated e.g. *–ignore parsing,templating*.

--exclude-rules <exclude_rules>
Exclude specific rules. For example specifying *–exclude-rules L001* will remove rule *L001* (Unnessesary trailing whitespace) from the set of considered rules. This could either be the whitelist, or the general set if there is no specific whitelist. Multiple rules can be specified with commas e.g. *–exclude-rules L001,L002* will exclude violations of rule *L001* and rule *L002*.

--rules <rules>
Narrow the search to only specific rules. For example specifying *–rules L001* will only search for rule *L001* (Unnessesary trailing whitespace). Multiple rules can be specified with commas e.g. *–rules L001,L002* will specify only looking for violations of rule *L001* and rule *L002*.

--templater <templater>
The templater to use (default=jinja)

--dialect <dialect>
The dialect of SQL to lint (default=ansi)

-f, --force
skip the confirmation prompt and go straight to applying fixes. **Use this with caution.**

--fixed-suffix <fixed_suffix>
An optional suffix to add to fixed files.

Arguments

PATHS
Optional argument(s)

lint

Lint SQL files via passing a list of files or using stdin.

PATH is the path to a sql file or directory to lint. This can be either a file ('path/to/file.sql'), a path ('directory/of/sql/files'), a single ('-') character to indicate reading from *stdin* or a dot/blank ('./' ') which will be interpreted like passing the current working directory as a path argument.

Linting SQL files:

sqlfluff lint path/to/file.sql sqlfluff lint directory/of/sql/files

Linting a file via stdin (note the lone '-' character):

cat path/to/file.sql | sqlfluff lint - echo 'select col from tbl' | sqlfluff lint -

```
sqlfluff lint [OPTIONS] [PATHS]...
```

Options

-n, --nocolor
No color - if this is set then the output will be without ANSI color codes.

-v, --verbose
Verbosity, how detailed should the output be. This is *stackable*, so *-vv* is more verbose than *-v*. For the most verbose option try *-vvvv* or *-vvvvv*.

--version
Show the version and exit.

--logger <logger>
Choose to limit the logging to one of the loggers.

Options parser|linter|rules

--bench
Set this flag to engage the benchmarking tool output.

--ignore <ignore>
Ignore particular families of errors so that they don't cause a failed run. For example *–ignore parsing* would mean that any parsing errors are ignored and don't influence the success or fail of a run. Multiple options are possible if comma seperated e.g. *–ignore parsing,templating*.

--exclude-rules <exclude_rules>
Exclude specific rules. For example specifying *–exclude-rules L001* will remove rule *L001* (Unnessesary trailing whitespace) from the set of considered rules. This could either be the whitelist, or the general set if there is no specific whitelist. Multiple rules can be specified with commas e.g. *–exclude-rules L001,L002* will exclude violations of rule *L001* and rule *L002*.

--rules <rules>
Narrow the search to only specific rules. For example specifying *–rules L001* will only search for rule *L001* (Unnessesary trailing whitespace). Multiple rules can be specified with commas e.g. *–rules L001,L002* will specify only looking for violations of rule *L001* and rule *L002*.

--templater <templater>
The templater to use (default=jinja)

--dialect <dialect>
The dialect of SQL to lint (default=ansi)

-f, --format <format>
What format to return the lint result in.

Options human|json|yaml

--nofail
If set, the exit code will always be zero, regardless of violations found. This is potentially useful during rollout.

--disregard-sqlfluffignores
Perform the operation regardless of .sqlfluffignore configurations

Arguments

PATHS
Optional argument(s)

parse

Parse SQL files and just spit out the result.

PATH is the path to a sql file or directory to lint. This can be either a file ('path/to/file.sql'), a path ('directory/of/sql/files'), a single ('-') character to indicate reading from *stdin* or a dot/blank ('./' ') which will be interpreted like passing the current working directory as a path argument.

```
sqlfluff parse [OPTIONS] PATH
```

Options

-n, --nocolor
No color - if this is set then the output will be without ANSI color codes.

-v, --verbose
Verbosity, how detailed should the output be. This is *stackable*, so *-vv* is more verbose than *-v*. For the most verbose option try *-vvvv* or *-vvvvv*.

--version
Show the version and exit.

--logger <logger>
Choose to limit the logging to one of the loggers.

Options parser|linter|rules

--bench
Set this flag to engage the benchmarking tool output.

--ignore <ignore>
Ignore particular families of errors so that they don't cause a failed run. For example *–ignore parsing* would mean that any parsing errors are ignored and don't influence the success or fail of a run. Multiple options are possible if comma seperated e.g. *–ignore parsing,templating*.

--exclude-rules <exclude_rules>
Exclude specific rules. For example specifying *–exclude-rules L001* will remove rule *L001* (Unnessesary trailing whitespace) from the set of considered rules. This could either be the whitelist, or the general set if there is no specific whitelist. Multiple rules can be specified with commas e.g. *–exclude-rules L001,L002* will exclude violations of rule *L001* and rule *L002*.

--rules <rules>
Narrow the search to only specific rules. For example specifying *–rules L001* will only search for rule *L001* (Unnessesary trailing whitespace). Multiple rules can be specified with commas e.g. *–rules L001,L002* will specify only looking for violations of rule *L001* and rule *L002*.

--templater <templater>
The templater to use (default=jinja)

--dialect <dialect>
The dialect of SQL to lint (default=ansi)

--recurse <recurse>
The depth to recursively parse to (0 for unlimited)

-c, --code-only
Output only the code elements of the parse tree.

-f, --format <format>
What format to return the parse result in.

Options human|json|yaml

--profiler
Set this flag to engage the python profiler.

--nofail
If set, the exit code will always be zero, regardless of violations found. This is potentially useful during rollout.

Arguments

PATH
Required argument

rules

Show the current rules in use.

```
sqlfluff rules [OPTIONS]
```

Options

-n, --nocolor
No color - if this is set then the output will be without ANSI color codes.

-v, --verbose
Verbosity, how detailed should the output be. This is *stackable*, so *-vv* is more verbose than *-v*. For the most verbose option try *-vvvv* or *-vvvvv*.

--version
Show the version and exit.

version

Show the version of sqlfluff.

```
sqlfluff version [OPTIONS]
```

Options

-n, --nocolor
No color - if this is set then the output will be without ANSI color codes.

-v, --verbose
Verbosity, how detailed should the output be. This is *stackable*, so *-vv* is more verbose than *-v*. For the most verbose option try *-vvvv* or *-vvvvv*.

--version
Show the version and exit.

2.11 API Reference

Sqlfluff exposes a public api for other python applications to use. A basic example of this usage is given here, with the documentation for each of the methods below.

```
"""This is an example of how to use the simple sqlfluff api."""

import sqlfluff

#  -------- LINTING ----------

my_bad_query = "SeLEct  *, 1, blah as  fOO  from myTable"

# Lint the given string and get a list of violations found.
result = sqlfluff.lint(my_bad_query, dialect="bigquery")

```

(continues on next page)

(continued from previous page)

```
# result =
# [
#     {"code": "L010", "line_no": 1, "line_pos": 1, "description": "Inconsistent␣
↪capitalisation of keywords."}
#     ...
# ]

#  -------- FIXING ----------

# Fix the given string and get a string back which has been fixed.
result = sqlfluff.fix(my_bad_query, dialect="bigquery")
# result = 'SELECT  *, 1, blah AS  foo  FROM mytable\n'

# We can also fix just specific rules.
result = sqlfluff.fix(my_bad_query, rules="L010")
# result = 'SELECT  *, 1, blah AS  fOO  FROM myTable'

# Or a subset of rules...
result = sqlfluff.fix(my_bad_query, rules=["L010", "L014"])
# result = 'SELECT  *, 1, blah AS  fOO  FROM mytable'

#  -------- PARSING ----------
# NOTE: sqlfluff is still in a relatively early phase of it's
# development and so until version 1.0.0 will offer no guarantee
# that the names and structure of the objects returned by these
# parse commands won't change between releases. Use with care
# and keep updated with the changelog for the project for any
# changes in this space.

parsed = sqlfluff.parse(my_bad_query)

# Get the structure of the query
structure = parsed.tree.to_tuple(show_raw=True, code_only=True)
# structure = ('file', (('statement', (('select_statement', (('select_clause', ((
↪'keyword', 'SeLEct'), ...

# Extract certain elements
keywords = [keyword.raw for keyword in parsed.tree.recursive_crawl("keyword")]
# keywords = ['SeLEct', 'as', 'from']
tbl_refs = [tbl_ref.raw for tbl_ref in parsed.tree.recursive_crawl("table_reference")]
# tbl_refs == ["myTable"]
```

2.11.1 Simple API commands

Sqlfluff is a SQL linter for humans.

fix(*sql*, *dialect='ansi'*, *rules=None*)
Fix a sql string or file.

Parameters

- **sql** (`str` or file-like object) – The sql to be linted either as a string or a subclass of `TextIOBase`.
- **dialect** (`str`, optional) – A reference to the dialect of the sql to be linted. Defaults to *ansi*.

- **rules** (str or iterable of str, optional) – A subset of rule reference to lint for.

Returns str for the fixed sql if possible.

lint(*sql*, *dialect='ansi'*, *rules=None*)
Lint a sql string or file.

Parameters

- **sql** (str or file-like object) – The sql to be linted either as a string or a subclass of TextIOBase.
- **dialect** (str, optional) – A reference to the dialect of the sql to be linted. Defaults to *ansi*.
- **rules** (str or iterable of str, optional) – A subset of rule reference to lint for.

Returns list of dict for each violation found.

parse(*sql*, *dialect='ansi'*)
Parse a sql string or file.

Parameters

- **sql** (str or file-like object) – The sql to be linted either as a string or a subclass of TextIOBase.
- **dialect** (str, optional) – A reference to the dialect of the sql to be linted. Defaults to *ansi*.

Returns ParsedString containing the parsed structure.

2.11.2 Advanced API usage

The simple API presents only a fraction of the functionality present within the core sqlfluff library. For more advanced use cases, users can import the Linter() and FluffConfig() classes from sqlfluff.core. As of version 0.4.0 this is considered as *experimental only* as the internals may change without warning in any future release. If you come to rely on the internals of sqlfluff, please post an issue on github to share what you're up to. This will help shape a more reliable, tidy and well documented public API for use.

The core elements of sqlfluff.

class FluffConfig(*configs: Optional[dict] = None*, *overrides: Optional[dict] = None*)
.The class that actually gets passed around as a config object.

diff_to(*other: sqlfluff.core.config.FluffConfig*) → dict
Compare this config to another.

Parameters **other** (*FluffConfig*) – Another config object to compare against. We will return keys from *this* object that are not in *other* or are different to those in *other*.

Returns A filtered dict of items in this config that are not in the other or are different to the other.

classmethod from_kwargs(*config: Optional[FluffConfig] = None*, *dialect: Optional[str] = None*, *rules: Optional[Union[str, List[str]]] = None*) → sqlfluff.core.config.FluffConfig
Instantiate a config from either an existing config or kwargs.

This is a convenience method for the ways that the public classes like Linter(), Parser() and Lexer() can be instantiated with a FluffConfig or with the convenience kwargs: dialect & rules.

classmethod from_path(*path: str*, *overrides: Optional[dict] = None*) → sqlfluff.core.config.FluffConfig
Loads a config object given a particular path.

classmethod from_root(*overrides: Optional[dict] = None*) → sqlfluff.core.config.FluffConfig
Loads a config object just based on the root directory.

get(*val: str*, *section: Union[str, Iterable[str]] = 'core'*)
Get a particular value from the config.

get_section(*section: Union[str, Iterable[str]]*) → Optional[dict]
Return a whole section of config as a dict.

If the element found at the address is a value and not a section, it is still returned and so this can be used as a more advanced from of the basic *get* method.

Parameters **section** – An iterable or string. If it's a string we load that root section. If it's an iterable of strings, then we treat it as a path within the dictionary structure.

iter_vals(*cfg: Optional[dict] = None*) → Iterable[tuple]
Return an iterable of tuples representing keys.

We show values before dicts, the tuple contains an indent value to know what level of the dict we're in. Dict labels will be returned as a blank value before their content.

make_child_from_path(*path: str*) → sqlfluff.core.config.FluffConfig
Make a new child config at a path but pass on overrides.

process_inline_config(*config_line: str*)
Process an inline config command and update self.

set_value(*config_path: Iterable[str]*, *val: Any*)
Set a value at a given path.

class Lexer(*config: Optional[sqlfluff.core.config.FluffConfig] = None*, *last_resort_lexer: Optional[sqlfluff.core.parser.lexer.SingletonMatcher] = None*, *dialect: Optional[str] = None*)
The Lexer class actually does the lexing step.

static enrich_segments(*segment_buff: Tuple[sqlfluff.core.parser.segments.base.BaseSegment, ...]*, *templated_file: sqlfluff.core.templaters.base.TemplatedFile*) → Tuple[sqlfluff.core.parser.segments.base.BaseSegment, ...]
Enrich the segments using the templated file.

We use the mapping in the template to provide positions in the source file.

lex(*raw: Union[str, sqlfluff.core.templaters.base.TemplatedFile]*) → Tuple[Tuple[sqlfluff.core.parser.segments.base.BaseSegment, ...], List[sqlfluff.core.errors.SQLLexError]]
Take a string or TemplatedFile and return segments.

If we fail to match the *whole* string, then we must have found something that we cannot lex. If that happens we should package it up as unlexable and keep track of the exceptions.

class Linter(*sql_exts=('.sql')*, *config=None*, *formatter=None*, *dialect=None*, *rules=None*, *user_rules=None*)
The interface class to interact with the linter.

static extract_ignore_from_comment(*comment*)
Extract ignore mask entries from a comment segment.

fix(*parsed*, *config=None*)
Fix a parsed file object.

get_ruleset(*config=None*)
Get hold of a set of rules.

lint(*parsed*, *config=None*)
Lint a parsed file object.

lint_path(*path*, *fix=False*, *ignore_non_existent_files=False*, *ignore_files=True*)
Lint a path.

lint_paths(*paths*, *fix=False*, *ignore_non_existent_files=False*, *ignore_files=True*)
Lint an iterable of paths.

lint_string(*in_str*, *fname='<string input>'*, *fix=False*, *config=None*)
Lint a string.

Returns an object representing that linted file.

Return type LintedFile

lint_string_wrapped(*string*, *fname='<string input>'*, *fix=False*)
Lint strings directly.

parse_path(*path*, *recurse=True*)
Parse a path of sql files.

NB: This a generator which will yield the result of each file within the path iteratively.

parse_string(*in_str*, *fname=None*, *recurse=True*, *config=None*)
Parse a string.

Returns

***ParsedString* of (*parsed*, *violations*, *time_dict*, *templated_file*).**

***parsed* is a segment structure representing the parsed file. If** parsing fails due to an inrecoverable violation then we will return None.

***violations* is a list of violations so far, which will either be** templating, lexing or parsing violations at this stage.

***time_dict* is a dict containing timings for how long each step** took in the process.

***templated_file* is a TemplatedFile containing the details** of the templated file.

paths_from_path(*path*, *ignore_file_name='.sqlfluffignore'*, *ignore_non_existent_files=False*, *ignore_files=True*, *working_path='/home/docs/checkouts/readthedocs.org/user_builds/sqlfluff/checkouts/0.4*
Return a set of sql file paths from a potentially more ambigious path string.

Here we also deal with the .sqlfluffignore file if present.

When a path to a file to be linted is explicitly passed we look for ignore files in all directories that are parents of the file, up to the current directory.

If the current directory is not a parent of the file we only look for an ignore file in the direct parent of the file.

rule_tuples()
A simple pass through to access the rule tuples of the rule set.

class Parser(*config: Optional[sqlfluff.core.config.FluffConfig] = None*, *dialect: Optional[str] = None*)
Instantiates parsed queries from a sequence of lexed raw segments.

parse(*segments: Tuple[BaseSegment, ...]*, *recurse=True*) → BaseSegment
Parse a series of lexed tokens using the current dialect.

2.12 SQLfluff in the Wild

Want to find other people who are using SQLfluff in production use cases? Want to brag about how you're using it? Just want to show solidartiy with the project and provide a testimonial for it?

Just add a section below by raising a PR on Github by editing this file .

- SQLfluff in production dbt projects at tails.com. We use the sqlfluff cli as part of our CI pipeline in codeship to enfore certain styles in our SQL codebase (with over 650 models) and keep code quality high. Contact @alanmcruickshank.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

INDEX

Symbols

B

C

D

E

F

G

I

L

M

P

R

S

T