pycql Documentation

Release 0.0.3

Fabian Schindler

Aug 30, 2021

Contents:

CHAPTER 1

Introduction

pycql is a pure python parser of the Common Query Language (CQL) defined in the OGC Catalogue specification.

The basic bare-bone functionality is to parse the given CQL to an abstract syntax tree (AST) representation. This AST can then be used to create filters for databases or search engines.

1.1 pycql license

CHAPTER 2

Installation

pycql can be installed using `pip` from the Python Package Index (PyPI):

```
pip install pycql
```

You can also install pycql from source:

```
cd path/to/pycql/
python setup.py install
```

CHAPTER 3

Usage

The basic functionality parses the input string to an abstract syntax tree (AST) representation. This AST can then be used to build database filters or similar functionality.

```
>>> import pycql
>>> ast = pycql.parse(filter_expression)
```

What is returned by the *pycql.parser.parse()* is the root *pycql.ast.Node* of the AST representation.

3.1 Inspection

The easiest way to inspect the resulting AST is to use the *pycql.ast.get_repr()* function, which returns a nice string representation of what was parsed:

```
>>> ast = pycql.parse('id = 10')
>>> print(pycql.get_repr(ast))
ATTRIBUTE id = LITERAL 10.0
>>>
>>>
>>> filter_expr = '(number BETWEEN 5 AND 10 AND string NOT LIKE "%B") OR␣
↪INTERSECTS(geometry, LINESTRING(0 0, 1 1))'
>>> print(pycql.get_repr(pycql.parse(filter_expr)))
(
    (
            ATTRIBUTE number BETWEEN LITERAL 5.0 AND LITERAL 10.0
    ) AND (
            ATTRIBUTE string NOT ILIKE LITERAL '%B'
    )
) OR (
    INTERSECTS(ATTRIBUTE geometry, LITERAL GEOMETRY 'LINESTRING(0 0, 1 1)')
)
```

3.2 Evaluation

In order to create useful filters from the resulting AST, it has to be evaluated. For the Django integration, this was done using a recursive descent into the AST, evaluating the subnodes first and constructing a *Q* object. Consider having a *filters* API (for an example look at the Django one) which creates the filter. Now the evaluator looks something like this:

```
from pycql.ast import *
from myapi import filters    # <- this is where the filters are created.
                             # of course, this can also be done in the
                             # evaluator itself
class FilterEvaluator:
    def __init__(self, field_mapping=None, mapping_choices=None):
        self.field_mapping = field_mapping
        self.mapping_choices = mapping_choices

    def to_filter(self, node):
        to_filter = self.to_filter
        if isinstance(node, NotConditionNode):
            return filters.negate(to_filter(node.sub_node))
        elif isinstance(node, CombinationConditionNode):
            return filters.combine(
                (to_filter(node.lhs), to_filter(node.rhs)), node.op
            )
        elif isinstance(node, ComparisonPredicateNode):
            return filters.compare(
                to_filter(node.lhs), to_filter(node.rhs), node.op,
                self.mapping_choices
            )
        elif isinstance(node, BetweenPredicateNode):
            return filters.between(
                to_filter(node.lhs), to_filter(node.low),
                to_filter(node.high), node.not_
            )
        elif isinstance(node, BetweenPredicateNode):
            return filters.between(
                to_filter(node.lhs), to_filter(node.low),
                to_filter(node.high), node.not_
            )

        # ... Some nodes are left out for brevity

        elif isinstance(node, AttributeExpression):
            return filters.attribute(node.name, self.field_mapping)

        elif isinstance(node, LiteralExpression):
            return node.value

        elif isinstance(node, ArithmeticExpressionNode):
            return filters.arithmetic(
                to_filter(node.lhs), to_filter(node.rhs), node.op
            )

        return node
```

As mentionend, the *to_filter* method is the recursion.

3.3 Django integration

For Django there is a default bridging implementation, where all the filters are translated to the Django ORM. In order to use this integration, we need two dictionaries, one mapping the available fields to the Django model fields, and one to map the fields that use `choices`. Consider the following example models:

```
from django.contrib.gis.db import models


optional = dict(null=True, blank=True)

class Record(models.Model):
    identifier = models.CharField(max_length=256, unique=True, null=False)
    geometry = models.GeometryField()

    float_attribute = models.FloatField(**optional)
    int_attribute = models.IntegerField(**optional)
    str_attribute = models.CharField(max_length=256, **optional)
    datetime_attribute = models.DateTimeField(**optional)
    choice_attribute = models.PositiveSmallIntegerField(choices=[
                                                                 (1, 'ASCENDING'),
                                                                 (2, 'DESCENDING'),],
                                                        **optional)


class RecordMeta(models.Model):
    record = models.ForeignKey(Record, on_delete=models.CASCADE, related_name='record_
↪metas')

    float_meta_attribute = models.FloatField(**optional)
    int_meta_attribute = models.IntegerField(**optional)
    str_meta_attribute = models.CharField(max_length=256, **optional)
    datetime_meta_attribute = models.DateTimeField(**optional)
    choice_meta_attribute = models.PositiveSmallIntegerField(choices=[
                                                                      (1, 'X'),
                                                                      (2, 'Y'),
                                                                      (3, 'Z')],
                                                             **optional)
```

Now we can specify the field mappings and mapping choices to be used when applying the filters:

```
FIELD_MAPPING = {
    'identifier': 'identifier',
    'geometry': 'geometry',
    'floatAttribute': 'float_attribute',
    'intAttribute': 'int_attribute',
    'strAttribute': 'str_attribute',
    'datetimeAttribute': 'datetime_attribute',
    'choiceAttribute': 'choice_attribute',

    # meta fields
    'floatMetaAttribute': 'record_metas__float_meta_attribute',
    'intMetaAttribute': 'record_metas__int_meta_attribute',
    'strMetaAttribute': 'record_metas__str_meta_attribute',
    'datetimeMetaAttribute': 'record_metas__datetime_meta_attribute',
    'choiceMetaAttribute': 'record_metas__choice_meta_attribute',
}
```

(continues on next page)

(continued from previous page)

```
MAPPING_CHOICES = {
    'choiceAttribute': dict(Record._meta.get_field('choice_attribute').choices),
    'choiceMetaAttribute': dict(RecordMeta._meta.get_field('choice_meta_attribute').
↪choices),
}
```

Finally we are able to connect the CQL AST to the Django database models. We also provide factory functions to parse the timestamps, durations, geometries and envelopes, so that they can be used with the ORM layer:

```
from pycql.integrations.django import to_filter, parse

cql_expr = 'strMetaAttribute LIKE "%parent%" AND datetimeAttribute BEFORE 2000-01-
↪01T00:00:01Z'

# NOTE: we are using the django integration `parse` wrapper here
ast = parse(cql_expr)
filters = to_filter(ast, mapping, mapping_choices)

qs = Record.objects.filter(**filters)
```

CHAPTER 4

API Documentation

4.1 pycql

4.2 pycql.ast

class pycql.ast.**ArithmeticExpressionNode**(*lhs*, *rhs*, *op*)
Node class to represent arithmetic operation expressions with two sub-expressions and an operator.

Variables

- **lhs** – the left hand side node of this arithmetic expression
- **rhs** – the right hand side node of this arithmetic expression
- **op** – the comparison type. One of `"+"`, `"-"`, `"*"`, `"/"`

get_sub_nodes()
Get a list of sub-node of this node.

Returns a list of all sub-nodes

Return type list[*Node*]

get_template()
Get a template string (using the `%` operator) to represent the current node and sub-nodes. The template string must provide a template replacement for each sub-node reported by *get_sub_nodes()*.

Returns the template to render

class pycql.ast.**AttributeExpression**(*name*)
Node class to represent attribute lookup expressions

Variables **name** – the name of the attribute to be accessed

class pycql.ast.**BBoxPredicateNode**(*lhs*, *minx*, *miny*, *maxx*, *maxy*, *crs=None*)
Node class to represent a bounding box predicate.

Variables

- **lhs** – the left hand side node of this predicate
- **minx** – the minimum X value of the bounding box
- **miny** – the minimum Y value of the bounding box
- **maxx** – the maximum X value of the bounding box
- **maxx** – the maximum Y value of the bounding box
- **crs** – the coordinate reference system identifier for the CRS the BBox is expressed in

get_sub_nodes()
Get a list of sub-node of this node.

Returns a list of all sub-nodes

Return type list[*Node*]

get_template()
Get a template string (using the `%` operator) to represent the current node and sub-nodes. The template string must provide a template replacement for each sub-node reported by *get_sub_nodes()*.

Returns the template to render

class pycql.ast.**BetweenPredicateNode**(*lhs*, *low*, *high*, *not_*)
Node class to represent a BETWEEN predicate: to check whether an expression value within a range.

Variables

- **lhs** – the left hand side node of this comparison
- **low** – the lower bound of the clause
- **high** – the upper bound of the clause
- **not** – whether the predicate shall be negated

get_sub_nodes()
Get a list of sub-node of this node.

Returns a list of all sub-nodes

Return type list[*Node*]

get_template()
Get a template string (using the `%` operator) to represent the current node and sub-nodes. The template string must provide a template replacement for each sub-node reported by *get_sub_nodes()*.

Returns the template to render

class pycql.ast.**CombinationConditionNode**(*lhs*, *rhs*, *op*)
Node class to represent a condition to combine two other conditions using either AND or OR.

Variables

- **lhs** – the left hand side node of this combination
- **rhs** – the right hand side node of this combination
- **op** – the combination type. Either `"AND"` or `"OR"`

get_sub_nodes()
Get a list of sub-node of this node.

Returns a list of all sub-nodes

Return type list[*Node*]

get_template()
Get a template string (using the % operator) to represent the current node and sub-nodes. The template string must provide a template replacement for each sub-node reported by *get_sub_nodes()*.

Returns the template to render

class pycql.ast.**ComparisonPredicateNode**(*lhs*, *rhs*, *op*)
Node class to represent a comparison predicate: to compare two expressions using a comparison operation.

Variables

- **lhs** – the left hand side node of this comparison
- **rhs** – the right hand side node of this comparison
- **op** – the comparison type. One of "=", "<>", "<", ">", "<=", ">="

get_sub_nodes()
Get a list of sub-node of this node.

Returns a list of all sub-nodes

Return type list[*Node*]

get_template()
Get a template string (using the % operator) to represent the current node and sub-nodes. The template string must provide a template replacement for each sub-node reported by *get_sub_nodes()*.

Returns the template to render

class pycql.ast.**ConditionNode**
The base class for all nodes representing a condition

class pycql.ast.**ExpressionNode**
The base class for all nodes representing expressions

class pycql.ast.**InPredicateNode**(*lhs*, *sub_nodes*, *not_*)
Node class to represent list checking predicate.

Variables

- **lhs** – the left hand side node of this predicate
- **sub_nodes** – the list of sub nodes to check the inclusion against
- **not** – whether the predicate shall be negated

get_sub_nodes()
Get a list of sub-node of this node.

Returns a list of all sub-nodes

Return type list[*Node*]

get_template()
Get a template string (using the % operator) to represent the current node and sub-nodes. The template string must provide a template replacement for each sub-node reported by *get_sub_nodes()*.

Returns the template to render

class pycql.ast.**LikePredicateNode**(*lhs*, *rhs*, *case*, *not_*)
Node class to represent a wildcard sting matching predicate.

Variables

- **lhs** – the left hand side node of this predicate

- **rhs** – the right hand side node of this predicate
- **case** – whether the comparison shall be case sensitive
- **not** – whether the predicate shall be negated

get_sub_nodes()
Get a list of sub-node of this node.

Returns a list of all sub-nodes

Return type list[*Node*]

get_template()
Get a template string (using the `%` operator) to represent the current node and sub-nodes. The template string must provide a template replacement for each sub-node reported by *get_sub_nodes()*.

Returns the template to render

class pycql.ast.**LiteralExpression**(*value*)
Node class to represent literal value expressions

Variables **value** – the value of the literal

class pycql.ast.**Node**
The base class for all other nodes to display the AST of CQL.

get_sub_nodes()
Get a list of sub-node of this node.

Returns a list of all sub-nodes

Return type list[*Node*]

get_template()
Get a template string (using the `%` operator) to represent the current node and sub-nodes. The template string must provide a template replacement for each sub-node reported by *get_sub_nodes()*.

Returns the template to render

class pycql.ast.**NotConditionNode**(*sub_node*)
Node class to represent a negation condition.

Variables **sub_node** – the condition node to be negated

get_sub_nodes()
Returns the sub-node for the negated condition.

get_template()
Get a template string (using the `%` operator) to represent the current node and sub-nodes. The template string must provide a template replacement for each sub-node reported by *get_sub_nodes()*.

Returns the template to render

class pycql.ast.**NullPredicateNode**(*lhs*, *not_*)
Node class to represent null check predicate.

Variables

- **lhs** – the left hand side node of this predicate
- **not** – whether the predicate shall be negated

get_sub_nodes()
Get a list of sub-node of this node.

Returns a list of all sub-nodes

Return type list[*Node*]

get_template()
Get a template string (using the % operator) to represent the current node and sub-nodes. The template string must provide a template replacement for each sub-node reported by *get_sub_nodes()*.

Returns the template to render

class pycql.ast.**PredicateNode**
The base class for all nodes representing a predicate

class pycql.ast.**SpatialPredicateNode**(*lhs*, *rhs*, *op*, *pattern=None*, *distance=None*, *units=None*)
Node class to represent spatial relation predicate.

Variables

- **lhs** – the left hand side node of this comparison
- **rhs** – the right hand side node of this comparison
- **op** – the comparison type. One of "INTERSECTS", "DISJOINT", "CONTAINS", "WITHIN", "TOUCHES", "CROSSES", "OVERLAPS", "EQUALS", "RELATE", "DWITHIN", "BEYOND"
- **pattern** – the relationship patter for the "RELATE" operation
- **distance** – the distance for distance related operations
- **units** – the units for distance related operations

get_sub_nodes()
Get a list of sub-node of this node.

Returns a list of all sub-nodes

Return type list[*Node*]

get_template()
Get a template string (using the % operator) to represent the current node and sub-nodes. The template string must provide a template replacement for each sub-node reported by *get_sub_nodes()*.

Returns the template to render

class pycql.ast.**TemporalPredicateNode**(*lhs*, *rhs*, *op*)
Node class to represent temporal predicate.

Variables

- **lhs** – the left hand side node of this comparison
- **rhs** – the right hand side node of this comparison
- **op** – the comparison type. One of "BEFORE", "BEFORE OR DURING", "DURING", "DURING OR AFTER", "AFTER"

get_sub_nodes()
Get a list of sub-node of this node.

Returns a list of all sub-nodes

Return type list[*Node*]

get_template()
Get a template string (using the % operator) to represent the current node and sub-nodes. The template string must provide a template replacement for each sub-node reported by *get_sub_nodes()*.

Returns the template to render

pycql.ast.**get_repr**(*node*, *indent_amount=0*, *indent_incr=4*)

Get a debug representation of the given AST node. `indent_amount` and `indent_incr` are for the recursive call and don't need to be passed.

Parameters

- **node** (Node) – the node to get the representation for
- **indent_amount** (*int*) – the current indentation level
- **indent_incr** (*int*) – the indentation incrementation per level

Returns the represenation of the node

Return type str

4.3 pycql.lexer

4.4 pycql.parser

pycql.parser.**parse**(*cql*, *geometry_factory=<class 'pycql.values.Geometry'>*, *bbox_factory=<class 'pycql.values.BBox'>*, *time_factory=<class 'pycql.values.Time'>*, *duration_factory=<class 'pycql.values.Duration'>*)

Parses the passed CQL to its AST interpretation.

Parameters

- **cql** (*str*) – the CQL expression string to parse
- **geometry_factory** – the geometry parsing function: it shall parse the given WKT geometry string the relevant type
- **bbox_factory** – the bbox parsing function: it shall parse the given BBox tuple the relevant type.
- **time_factory** – the timestamp parsing function: it shall parse the given ISO8601 timestamp string tuple the relevant type.
- **duration_factory** – the duration parsing function: it shall parse the given ISO8601 furation string tuple the relevant type.

Returns the parsed CQL expression as an AST

Return type *Node*

4.5 pycql.util

pycql.util.**parse_duration**(*value*)

Parses an ISO 8601 duration string into a python timedelta object. Raises a `ValueError` if a conversion was not possible.

Parameters **value** (*str*) – the ISO8601 duration string to parse

Returns the parsed duration

Return type datetime.timedelta

4.6 pycql.values

4.7 pycql.integrations.django.evaluate

4.8 pycql.integrations.django.filters

4.9 pycql.integrations.django.parser

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

N

P

S

T