Winnow Documentation

Release 0.0.1

Brian Schiller

Aug 31, 2017

Contents:

1 Quickstart **3**
- 1.1 Installation . . . 3
- 1.2 Define your Sources . . . 3
- 1.3 Create a Filter . . . 4
- 1.4 Get a SQL Query . . . 4

2 Value Types and Operators **5**
- 2.1 numeric . . . 5
- 2.2 string . . . 5
- 2.3 collection . . . 6
- 2.4 Datetime Operators . . . 6

3 Extending Winnow **9**
- 3.1 Playing with the column definition . . . 9
- 3.2 Referring to other tables . . . 9
- 3.3 Adding Operators . . . 11

4 Indices and tables **15**

Winnow is a Python package for safely building SQL where clauses from untrusted user input. It's designed to be expressive, extensible, and fast. Winnow's inputs look something like this:

```
{
    "logical_op": "&",
    "filter_clauses": [
        {
            "data_source": "Created",
            "operator": "before",
            "value": "2015-03-01"
        },
        {
            "data_source": "Owner",
            "operator": "any of",
            "value": [
                {"name": "Steven", "id": 23},
                {"name": "Margaret", "id": 41},
                {"name": "Evan", "id": 90}
            ]
        }
    ]
}
```

And its outputs looks like this:

```
(
  "WHERE created_date < %s::timestamp AND owner_id = ANY(VALUES (%s),(%s),(%s))",
  ('2015-03-01', 23, 41, 90)
)
```

CHAPTER 1

Quickstart

Installation

Winnow is available from PyPI as winnow-filters. I've been having some trouble downloading it from there though – I'm new to PyPI and it may be set up wrong. In the meantime, it can be installed with

```
pip install git+git://github.com/bgschiller/winnow.git@master
```

Define your Sources

Your sources will usually be a list of the columns you want to make available for filtering. Each source needs a display name and a list of the *value_types* it supports. See *Value Types and Operators* for a list of included value types.

```
sources = [
    {
        'display_name': 'Order Date',
        'column': 'order_date',
        'value_types': ['absolute_date', 'relative_date'] },
    {
        'display_name': 'Number Scoops',
        'column': 'num_scoops',
        'value_types': ['numeric', 'nullable'] },
    {
        'display_name': 'Flavor',
        'column': 'flavor',
        'value_types': ['collection'],
        'picklist_options': [
            'Mint Chocolate Chip',
            'Cherry Garcia',
            'Chocolate',
            'Cookie Dough',
            'Rocky Road',
```

```
            'Rainbow Sherbet',
            'Strawberry',
            'Vanilla',
            'Coffee',
         ]},
]
```

Create a Filter

Use the sources you defined to build a JSON filter. The value types specified on each source determine which operators are available.

```
ice_cream_filt = {
    'logical_op': '&',
    'filter_clauses': [
        {'data_source': 'Number Scoops', 'operator': '>=', 'value': '2'},
        {'data_source': 'Flavor', 'operator': 'any of', 'value': [
            'Strawberry',
            'Chocolate',
         ]}
    ]
}
```

Get a SQL Query

Now initialize a `Winnow()` instance using your sources, and the name of the table you're filtering against. Turn your filter into a query.

```
ice_cream_winnow = Winnow('ice_cream', sources)
query, params = ice_cream_filt.query(ice_cream_filt)
# query => SELECT * FROM ice_cream WHERE ((num_scoops >= %s) AND (flavor IN (%s,%s) ))
# params => (2, 'Strawberry', 'Chocolate')
```

CHAPTER 2

Value Types and Operators

A value type corresponds to the shape of the `value` key in a filter. It might be numeric (`3`), or string (`Mad-Eye Moody`) or relative_date (`last_7_days`). Each operator specifes the unique value type it operates on. Each data_source can list any number of value types it will support.

For example, "Order Date" source from our quickstart example lists that it supports both relative_date and absolute_date. The values for relative_date look like `last_month` or `next_7_days`. The values for absolute_date look like `2017-05-18T12:30`. Because both absolute_ and relative_date value_types are supported for "Order Date", we can use the relative_date operators (`within` and `outside of`) as well as the absolute_date operators (`before` and `after`).

The existing value types and operators can be easily extended by subclassing Winnow. We will see out to do this in *Extending Winnow*.

numeric

The numeric value type provides operators for >=, <=, >, <, `is`, and `is not`.

```
{
    'data_source': 'Number Scoops',
    'operator': '>=',
    'value': 3,
}
```

string

Data sources marked a supporting the string value type can use the `is`, `is not`, `contains`, `starts with`, `more than __ words`, and `fewer than __ words` operators.

```
{
    'data_source': "Scooper's Name",
```

```
    'operator': 'more than __ words',
    'value': 3,
},
{
    'data_source': "Scooper's Name",
    'operator': 'contains',
    'value': 'Heidi',
}
```

collection

To use the collection operators, a data source will usually need to provide a list of picklist_options to the client. It's fine to include those directly on the data source object:

```
{
    'display_name': 'Flavor',
    'column': 'flavor',
    'value_types': ['collection'],
    'picklist_options': [
        'Mint Chocolate Chip',
        'Cherry Garcia',
        'Chocolate',
        'Cookie Dough',
        'Rocky Road',
        'Rainbow Sherbet',
        'Strawberry',
        'Vanilla',
        'Coffee',
     ]
}
```

Collections have access to `any of` and `not any of` operators.

```
{
    'data_source': 'Flavor',
    'operator': 'any of',
    'value': ['Strawberry', 'Chocolate'],
}
```

Datetime Operators

Datetime operators are broken down into two sets, relative and absolute. Most timestamp sources will want to support both.

absolute_date

Absolute date values are ISO8601 strings, like `"2017-03-22T18:14:30"`. The supported operators are `before` and `after`.

```
{
    'data_source': 'Purchase Date',
```

```
    'operator': 'after',
    'value': '2017-03-22T18:14:30',
}
```

relative_date

Relative date values are also strings, but they're things like "last_30_days" and "current_month". I'm not very happy with how these are designed, so they will likely change in a future version. Please let me know if you have any advice. Maybe there's already a standard way to refer to intervals of time that aren't anchored to a particular day?

```
{
    'data_source': 'Purchase Date',
    'operator': 'within',
    'value': 'last_7_days',
}
```

The list of available values is found in relative_dates.py.

CHAPTER 3

Extending Winnow

Winnow provides some basic functionality, but it's expected that you will sometimes need to extend it to accomplish your filtering goals.

Playing with the column definition

Sometimes we need to do a little bit of processing to prepare data for filtering. In the winnow-demo project we're storing `cook_time` and `prep_time` as PostgreSQL interval columns, but we don't want to expect users to know how to enter a ISO 8601 formatted time interval. Instead, we'll define the source like this:

```
{
   "display_name": "Prep Time (minutes)",
   "column": "(EXTRACT(EPOCH FROM prep_time)::int / 60)",
   "value_types": ["numeric"]
}
```

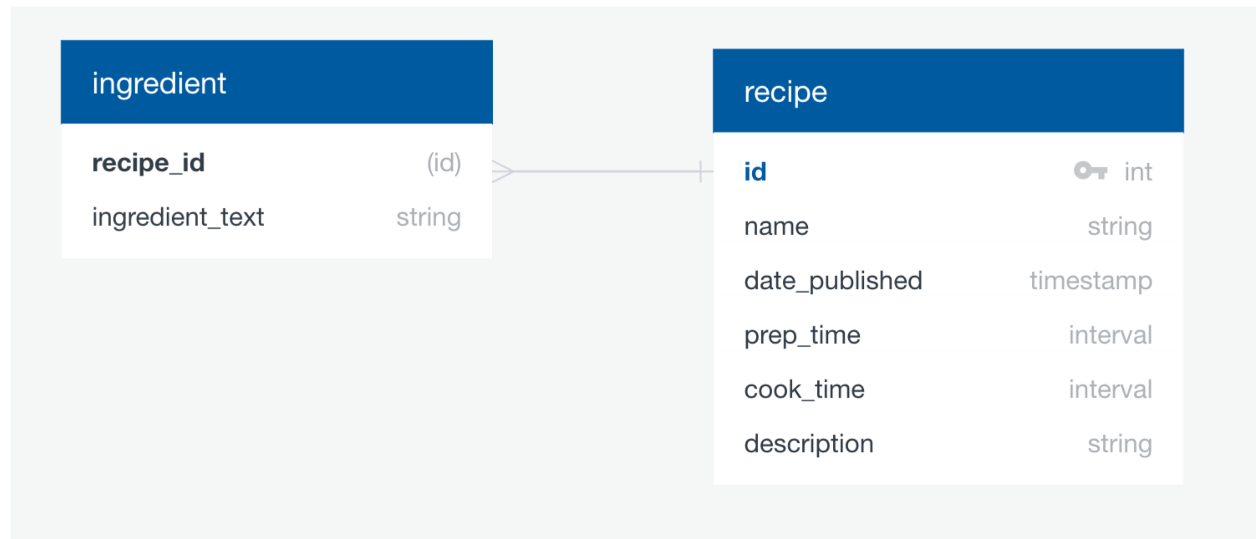
The `column` attribute of the source definition will be placed directly into the query without quoting, so we can use any valid SQL expression here – not just the name of a column. A filter clause like

```
{
    "data_source": "Prep Time (minutes)",
    "operator": "<=",
    "value": 5
}
```

will be turned into `(((EXTRACT(EPOCH FROM prep_time)::int / 60) <= 5))`.

Referring to other tables

In the winnow-demo project, we are filtering recipes. We'd like to filter by ingredient, but the ingredients are stored in another table. The schema looks roughly like this.



By default, if we try to write a filter like

```
// source definition
{
   "display_name": "Ingredients",
   "value_types": ["collection"]
}


// filter clause
{
    "data_source": "Ingredients",
    "operator": "any of",
    "value": ["Tomato"]
}
```

it will produce SQL like `WHERE ingredient IN ('Tomato')`. That won't work – there's no column called `ingredient`! Instead we'll have to override the handling of that data source.

```
@RecipeWinnow.special_case('Ingredients', 'collection')
def ingredients(rw, clause):
    return rw.sql.prepare_query(
        '''
        {% if not_any_of %}
            NOT
        {% endif %}
        id = ANY(
            SELECT recipe_id FROM ingredient
            WHERE ingredient_text = ANY({{ value }})
        )
        ''',
        value=clause['value_vivified'],
        not_any_of=clause['operator'] == 'not any of',
    )
```

With this change, we get SQL like

```
(
 id = ANY(
```

```
        SELECT recipe_id FROM ingredient
        WHERE ingredient_text = ANY(ARRAY['Tomato']))
)
```

That will work much better! You can see an example of how this works in recipe_winnow.py. The query is a bit more complicated there, because we're taking advantage of PostgreSQL's full-text-search capabilities. That way, a search for 'tomatoes' will also turn up results for 'tomato'.

Adding Operators

When searching recipes, it makes sense to ask for recipes that use *all of* a list of ingredients. We can accomplish that right now, but it's a bit awkward.

That's pretty painful to write. If your UI mirrors our json structure, it could be difficult for users to discover how to create a filter like that. Let's create an operator to handle this case.

First, we'll need to make a value type. We could piggyback on the 'collection' value type, but other collections don't necessarily know how to handle an 'all of' operator. For example, imagine we had a data source that was a star rating between 1 and 5. It wouldn't make sense to say "Rating is all of [3, 4]". By making a specific value type, we can allow each data source to decide whether or not to permit this operator. Let's call it 'collection_all'.

Now we'll add the operator to Winnow's list.

We'll also need to specify that the Ingredients data_source supports 'collection_all'.

```
// source definition
{
   "display_name": "Ingredients",
   "value_types": ["collection", "collection_all"]
}
```

Finally, we need to provide instructions for building SQL to answer that query. Let's do that using a special case handler to begin with, but we'll revisit this decision in the next section.

```
@RecipeWinnow.special_case('Ingredients', 'collection_all')
def ingredients(rw, clause):
    return rw.sql.prepare_query(
        '''
        id = ANY(
            {% for val in value %}
                (SELECT recipe_id FROM ingredient
                 WHERE ingredient_text = {{ val }})
                {% if not loop.last %}
                    INTERSECT
                {% endif %}
            {% endfor %}
        )
        ''',
        value=clause['value_vivified'],
    )
```

That's it! Now we can make queries like

```
{
    "logical_op": "&",
    "filter_clauses": [
```

```
        {
            "data_source": "Ingredients",
            "operator": "all of",
            "value": ["Tomato", "Basil", "Mozzarella"]
        }
    ]
}
```

to produce SQL like

```
id = ANY(

    (SELECT recipe_id FROM ingredient
     WHERE ingredient_text = 'Tomato')
        INTERSECT
    (SELECT recipe_id FROM ingredient
     WHERE ingredient_text = 'Basil')
        INTERSECT
    (SELECT recipe_id FROM ingredient
     WHERE ingredient_text = 'Mozzarella')

)
```

Including more data sources

In the last section, we said we would revisit the decision to use a special case handler. Well, that time has come! We have another data source where the 'all of' operator makes sense, and that's 'Suitable for Diet'. Our users are clamoring to be able to find recipes that are both gluten-free *and* vegetarian.

Now, for only two data sources, I would probably just use two special case handlers. But it's instructive to see how we might do things with 3, 4, 5, or more data sources. And since this *is* the documentation...

Let's override the definition of `Winnow.where_clause`

```
class RecipeWinnow(Winnow):

    def where_clause(self, data_source, operator, value):
        if op['value_type'] == 'collection_all':
            return self.collection_all_of(data_source, operator, value):
        return super().where_clause(data_source, operator, value)

    def collection_all_of(self, data_source, operator, value):
        return self.prepare_query(
            '''
            id = ANY(
                {% for val in values %}
                    (SELECT {{ foreign_key | sqlsafe }}
                     FROM {{ foreign_table | sqlsafe }}
                     WHERE {{ column | sqlsafe }} = {{ val }})
                    {% if not loop.last %} INTERSECT {% endif %}
                {% endfor %}
            )''',
            values=value,
            foreign_key=data_source['foreign_key'],
            foreign_table=data_source['foreign_table'],
            column=data_source['column'],
        )
```

Notice that there's a few more pieces of information that we're expecting from `data_source` now: `foreign_key`, `foreign_table`, and `column`. With 'Ingredients', all of this was just hard-coded. Now, since we want this to work for both 'Ingredients' and 'Suitable for Diet', we need to pass those values in as parameters. We'll store them on the data source:

```
// source definitions
{
   "display_name": "Ingredients",
   "value_types": ["collection", "collection_all"],
   "foreign_table": "ingredient",
   "foreign_key": "recipe_id",
   "column": "ingredient_text",

},
{
   "display_name": "Suitable for Diet",
   "value_types": ["collection", "collection_all"],
   "picklist_values": ["vegan", "vegetarian", "gluten-free", "halal", "kosher"],
   "foreign_table": "diet_suitability",
   "foreign_key": "recipe_id",
   "column": "diet",
}
```

CHAPTER 4

Indices and tables

- genindex
- modindex
- search