Jinja SQL Implementation Example

In [1]:
```
from __future__ import absolute_import, division, print_function
import dataiku
from jinja2 import Environment, Template, FunctionLoader
```

DSS Internal Code

Specify DSS customized Jinja template class (required)

In [2]:
```
class SqlTemplate:
    """
    Defines a version of a Jinja template that is
    customized to SQL and Dataiku.
    """

    def __init__(self, sql_template, dss_globals):
        """
        Set up DSS customized environment and set the template
        """

        self.env = Environment(
            # loader enables user definition of project/instance macros
            loader=FunctionLoader(load_jinja_sql_template),
            variable_start_string='${', # DSS format
            variable_end_string='}',
            line_statement_prefix='#', # Support line delimiters
            line_comment_prefix='##',
            trim_blocks=True,
            lstrip_blocks=True,
            autoescape=False,
            # expression statement, break/continue in loops, {% debug %} tag
            extensions=['jinja2.ext.do', 'jinja2.ext.loopcontrols',
                        'jinja2.ext.debug']
            )

        # add DSS global macros to all sql templates
        self.sql_template = dss_globals + sql_template

    def render(self, *args, **kwargs):
        """
        Render template. This executes all Jinja statements.
        Can pass dictionaries and/or named arguments.
        """
        return self.env.from_string(self.sql_template).render(*args, **kwargs)
```

Enable execution of SQL queries from the Jinja template (optional)

In [3]:
```
class SqlConnection:
    """
    Wrap a DSS connection in a class that supports a
    method for executing SQL queries
    """
    def __init__(self, connection=None):
        self.connection = connection
    def execute(self, sql_statement=None):
        # Use SQLExecutor2 to execute sql_statement
        # For test, just return a constant set of values for one column

        # Could return only first column in a list or maybe a dict of lists
        return ['value1', 'value2', 'value3']
```

Specify "included with DSS" Jinja macros (optional)

In [4]:
```
# DSS provided macros (optional)
dss_globals = """
{% macro count_rows_duplicate_keys(table, keys) -%}

SELECT COUNT(*)
FROM (
SELECT
{% for key in keys %}
    ${key}${ "," if not loop.last else "" }
{% endfor %}
FROM ${table}
GROUP BY
{% for key in keys %}
    ${key}${ "," if not loop.last else "" }
{% endfor %}
HAVING COUNT(*) > 1
) T1

{%- endmacro -%}
"""
```

Render the SQL Jinja template to pure SQL (required)

In [5]:
```
# This functionality would be executed just prior to passing
# the SQL to the database for execution
def render_sql_recipe(recipe_script, project_variables, recipe_connection):
    conn_object = SqlConnection(recipe_connection)
    return SqlTemplate(recipe_script, dss_globals).render(
        project_variables, recipe_connection=conn_object)
```

Set project and recipe state variables

In [6]:
```
# Set values for testing
project_variables = {'beginDate': '2020-01-01', 'endDate': '2020-12-31'}
recipe_connection = 'NZ_Lab' # connection used to execute recipe
```

User Defined Project & Instance Macros

Enabling users to define their own macros for sharing within a project and across projects could accomplished in a number of ways.

The way shown here is to define templates inside a function. This function could be defined in a project library or a git sourced project library (for sharing across multiple users). Even better, one function could be for project use and another could be for shared use (ChoiceLoader could be used provide access to both shared and local functions).

In [7]:
```
def load_jinja_sql_template(template_name):
    template_source = None
    if template_name == 'my_macros':
        template_source = u"""
        # macro select_qualified_customers(month, type):
            SELECT CUST_ID
            FROM CUSTOMERS
            WHERE QUALIFY_MONTH = ${month}
            AND TYPE = ${type}
            AND PRODUCT_TYPE = 'V'
            AND ACTIVE = 1
        #- endmacro
        """
    return template_source
```

Recipe Examples

Example 1 - Local variables

In [8]:
```
recipe_script = """
# set top_n = 20

-- Both local Jinja variable and DSS project variables used here
(SELECT C1, C2 FROM TABLE1 LIMIT ${top_n} WHERE DT > '${beginDate}') UNION ALL
(SELECT C1, C2 FROM TABLE2 LIMIT ${top_n} WHERE DT > '${beginDate}') UNION ALL
(SELECT C1, C2 FROM TABLE3 LIMIT ${top_n} WHERE DT > '${beginDate}')
"""
```

In [9]:
```
print(render_sql_recipe(recipe_script, project_variables, recipe_connection))
```

```
-- Both local Jinja variable and DSS project variables used here
(SELECT C1, C2 FROM TABLE1 LIMIT 20 WHERE DT > '2020-01-01') UNION ALL
(SELECT C1, C2 FROM TABLE2 LIMIT 20 WHERE DT > '2020-01-01') UNION ALL
(SELECT C1, C2 FROM TABLE3 LIMIT 20 WHERE DT > '2020-01-01')
```

Example 2 - Loop through multiple items

In [10]:
```
recipe_script = """
# set joins = [('TBL1', 'COL1'), ('TBL2', 'COL2'), ('TBL3', 'COL3')]

SELECT *
FROM BASE_TABLE AS BT
# for join in joins:
    INNER JOIN ${join[0]} ON ${join[0]}.${join[1]} = BT.COL
# endfor
"""
```

In [11]:
```
print(render_sql_recipe(recipe_script, project_variables, recipe_connection))
```

```
SELECT *
FROM BASE_TABLE AS BT
    INNER JOIN TBL1 ON TBL1.COL1 = BT.COL
    INNER JOIN TBL2 ON TBL2.COL2 = BT.COL
    INNER JOIN TBL3 ON TBL3.COL3 = BT.COL
```

Example 3 - Locally defined macro

In [12]:
```
recipe_script = """
# macro join_to(table, column):
    INNER JOIN ${table} ON ${table}.${column} = BT.COL
#- endmacro

SELECT *
FROM BASE_TABLE AS BT
${join_to('TBL1', 'COL1')}
${join_to('TBL2', 'COL2')}
${join_to('TBL3', 'COL3')}
"""
```

In [13]:
```
print(render_sql_recipe(recipe_script, project_variables, recipe_connection))
```

```
SELECT *
FROM BASE_TABLE AS BT
    INNER JOIN TBL1 ON TBL1.COL1 = BT.COL
    INNER JOIN TBL2 ON TBL2.COL2 = BT.COL
    INNER JOIN TBL3 ON TBL3.COL3 = BT.COL
```

Example 4 - Project level macro

In [14]:
```
recipe_script = """
{% import 'my_macros' as projmacros %}

SELECT *
FROM CUSTOMERS
WHERE CUST_ID IN (${projmacros.select_qualified_customers("'2020-01-01'", 3)})
;
"""
```

In [15]:
```
print(render_sql_recipe(recipe_script, project_variables, recipe_connection))
```

```
SELECT *
FROM CUSTOMERS
WHERE CUST_ID IN (            SELECT CUST_ID
            FROM CUSTOMERS
            WHERE QUALIFY_MONTH = '2020-01-01'
            AND TYPE = 3
            AND PRODUCT_TYPE = 'V'
            AND ACTIVE = 1)
;
```

Example 5 - DSS macro

In [16]:
```
# Counting rows that have the same key values (duplicates)
# is something we often specify in a SQL Probe Metric .
# This is an example of a generically useful macro that could
# be included in DSS

recipe_script = "${count_rows_duplicate_keys('MYTABLE', ['KEY1', 'KEY2'])}"
```

In [17]:
```
print(render_sql_recipe(recipe_script, project_variables, recipe_connection))
```

```
SELECT COUNT(*)
FROM (
SELECT
    KEY1,
    KEY2
FROM MYTABLE
GROUP BY
    KEY1,
    KEY2
HAVING COUNT(*) > 1
) T1
```

Example 6 - Building SQL from values in a table

In [18]:
```
recipe_script = """
# set colvalues = recipe_connection.execute("SELECT C1 FROM TABLE")

SELECT
# for colvalue in colvalues:
    ${colvalue}${ "," if not loop.last else "" }
# endfor
FROM TABLE
WHERE DT >= '${beginDate}'
"""
```

In [19]:
```
print(render_sql_recipe(recipe_script, project_variables, recipe_connection))
```

```
SELECT
    value1,
    value2,
    value3
FROM TABLE
WHERE DT >= '2020-01-01'
```

Example 7 - Build SQL from columns in a table (such as a Feature Store)

In [20]:
```
recipe_script = """

# set cols = recipe_connection.execute("SELECT COL FROM _V_SYS_COLUMNS WHERE T
ABLE_NAME = 'FEATURE_STORE'")

# macro set_col_spec(col):
    COALESCE(FS.${col.upper()}, MISS.${col.upper()}) AS ${col.upper()}
#- endmacro


SELECT E.KEY,
# for col in cols:
    ${set_col_spec(col)}${ "," if not loop.last else "" }
# endfor
FROM EVENTS AS E
    LEFT JOIN FEATURE_STORE AS FS
    CROSS JOIN FEATURE_MISS_VALUES AS MISS
WHERE DT >= '${beginDate}'
"""
```

In [21]:
```
print(render_sql_recipe(recipe_script, project_variables, recipe_connection))
```

```
SELECT E.KEY,
        COALESCE(FS.VALUE1, MISS.VALUE1) AS VALUE1,
        COALESCE(FS.VALUE2, MISS.VALUE2) AS VALUE2,
        COALESCE(FS.VALUE3, MISS.VALUE3) AS VALUE3
FROM EVENTS AS E
    LEFT JOIN FEATURE_STORE AS FS
    CROSS JOIN FEATURE_MISS_VALUES AS MISS
WHERE DT >= '2020-01-01'
```