Tech spec

The ODM sharing library aims to be able to read the most used file formats and databases, provide useful feedback to help build correct filters, and output easy-to-share result data.

Overview

A sharing schema (in CSV format) is used to define individual rules (as rows), which come together to form a data query.

The query rules are parsed into an abstract syntax tree (AST), which represents the structure of the query. It effectively separates rule-parsing from query-generation, making it more modular. The AST can then be used to generate concrete queries in SQL or other query languages, or it can be interpreted to perform data transformations directly using a library like Pandas.

The library will focus on using SQL as the query language, since it has wide support and is easy to generate. Using SQL will also provide a free performance boost when running queries on big indexed databases, as well as enable us to output the intermediate SQL to the user in case they want to inspect or execute the queries themselves.

CLI

Usage:

```
./share.py [--org=<name>] [--outdir=<path>] <schema> <input>
```

- org: the organization to output data for, or all if not specified
- outdir: output file directory
- schema: sharing schema file path
- input: spreadsheet file path or SQLAlchemy database url Examples:

```
./share.py --org=OHRI --outdir=~/ohri/ schema.csv data.xlsx
./share.py schema.csv mysql+pymysql://scott:tiger@localhost/foo
./share.py schema.csv mssql+pyodbc://user:pass@mydsn
```

API

Public modules

sharing

- high level:

```
extract(data_source: str, schema_file: str, org: str=''
        ) -> Dict[TableName, Dataframe]
```

    - data_source: a file path or database url (in SQLAlchemy format)
    - schema_file: rule schema file path
    - org: org to share with, or all orgs if empty
    - returns a Pandas Dataframe per table, for further processing
- low level:
    - connect(data_source: str) -> Connection
    - parse(schema_file: str) -> Dict[OrgName, Dict[TableName, Query]]
    - extract(Connection, Query) -> Dataframe
    - getCounts(Connection, Query) -> Dict[RuleId, int]
    - getColumns(Connection, Query) -> Tuple[RuleId, List[ColumnName]]

Private modules

cons

Data source connection abstraction, including intermediate copy of spreadsheet files to in-memory DBs.

- connect(data_source: str) -> Connection

rules

Loading of rule files.

- load(schema_file: str) -> Dict[RuleId, Rule]

trees

Parsing of rules into abstract syntax trees.

- parse(rules: Dict[RuleId, Rule] | List[Rule]) -> RuleTree

queries

(SQL) query generation from ASTs.

- generate(rt: RuleTree) -> Dict[OrgName, Query]

Examples

common definitions:

```
import pandas as pd

import sharing as s

data_file = 'data.xlsx'
rule_file = 'rules.csv'
org = 'OHRI'
```

high-level one-shot function:

```
results = s.extract(data_file, rules, org)
for table, data in results.items():
    data.to_csv(f'{org}-{table}.csv')
```

low-level sample code:

```
def describeTableQuery(con, rules, table, query):
    print(f'query table: {table}')

    (selectRuleId, columns) = s.getColumns(con, query)
    print(f'query columns (from rule {selectRuleId}):')
    print(','.join(columns))

    print('query counts per rule:')
    ruleCounts = s.getCounts(con, query)
    for ruleId, count in ruleCounts.items():
        r = rules[ruleId]
        ruleFilter = f'{r.key} {r.operator} {r.value}'
        print(f'{ruleId} | {count} | {ruleFilter}')

def extractFilteredData(con, table, query):
    data: pd.Dataframe = s.extract(con, query)
    data.to_csv(f'{org}-{table}.csv')

con = s.connect(data_file)
rules = s.load(rule_file)
ruleTree = s.parse(rules)
tableQueries = s.generate(ruleTree)[org]
for table, query in tableQueries.items():
    describeTableQuery(con, rules, table, query)
    extractFilteredData(con, table, query)
```

Rule schema parsing

1. open schema file
2. read lines
3. parse each line into a rule obj
4. add each rule obj to a dictionary with rule id as key

AST generation

Node kinds:

- **root**: AST root
- **share**: what to share with each org
- **table**: table name
- **select**: lists column name values

- **group**: groups filters together
- **filter**: defines a filter with operator, key and value
- **key**: field name
- **value**: generic

Node structure:

- (ruleId: int)
- kind: NodeKind
- value: str
- children: List[Node]

For each rule, a node is added based on its mode:

- select:
  - (**select**, ("" or "all")):
    - (**value**, column) for column in rule.key
- filter:
  - (**filter**, rule.operator):
    - (**key**, rule.key)
    - (**value**, x) for x in rule.value
- group:
  - (**group**, rule.operator):
    - nodes where node.ruleId in rule.value
- share:
  - (**root**, ""):
    - (**share**, org) for org in rule.key
      - (**table**, x) for x in select-rule-tables
        - filter-root node

Example rules with its generated tree:

```
ruleId,table,mode,key,operator,value,notes
10,measures,select,NA,NA,all,
11,measures,select,NA,NA,measure;value;unit;aggregation,
12,measures,filter,measure,=,mPox,
13,measures,filter,reportDate,in,2021-01-01;2021-12-31,
14,NA,group,NA,AND,12;13,
15,measures,filter,measure,=,cov,
16,measures,filter,reportDate,>=,2020-01-01,
17,NA,group,NA,AND,15;16,
18,NA,group,NA,OR,14;17,
19,NA,share,ohri,NA,11;18,
20,NA,share,other,NA,10,
```

```
(root, "")
    (share, "OHRI")
        (table, "measures")
            (select, "")
                (value, "measure")
                (value, "value")
                (value, "unit")
                (value, "aggregation")
            (group, "OR")
                (group, "AND")
                    (filter, "=")
                        (key, "measure")
                        (value, "mPox")
                    (filter, "in")
                        (key, "reportDate")
                        (value, "2021-01-01")
                        (value, "2021-12-31")
                (group, "AND")
                    (filter, "=")
                        (key, "measure")
                        (value, "cov")
                    (filter, ">=")
                        (key, "reportDate")
                        (value, "2020-01-01")
    (share, "other")
        (table, "measures")
            (select, "all")
```

SQL query generation

SQL queries are (recursively) generated from each table node of the AST. Values are separated from the query to prevent injections. Multiple kinds of queries can be generated for different purposes, including only getting the row-counts or column names.

- **table**:
  - "select " + recurse(select-child)
  - "from " + value/table
  - "where " + recurse(filter-child)
- **select**:
  - if "all" in values: " * "
  - else: join quoted child values with commas
- **group**:
  - op = value
  - fold children with value/operator
- **filter**:
  - recurse(key-child) + value/operator + recurse(value-child)
- **key**:
  - quote value/column
- **value**:
  - append to separate values

Example implementation of SQL-generation for a filter node:

```
keyNode = Node(kind: key, value: 'siteID')
valueNode = Node(kind: value, value: 'ottawa-1')

params = []
sql = genSql(keyNode, params) + ' = ' + genSql(valueNode, params)
assert sql == 'siteID = ?'
assert params == ['ottawa-1']
```

Data source connections

A connection can be established to multiple data sources, including databases and spreadsheet files.

Excel

Excel seems to be the main data source used by the ODM community.

Unofficial Excel plugins are available for SQLAlchemy, but users may not want to install them. The pyodbc library can also be used directly (instead of SQLAlchemy) together with the ODBC driver for Excel, but only on Windows.

SQLAlchemy Excel dialect plugin:

https://github.com/mclovinxie/dialect-pyexcel

Using the ODBC Excel driver on Windows:

https://github.com/mkleehammer/pyodbc/wiki/Connecting-to-Microsoft-Excel

As an alternative to the above, we can load the spreadsheet file into a temporary in-memory SQLite database and perform queries on that instead. It would work like the following:

1. read into memory (using pandas)
2. create in-memory sqlite db (using sqlalchemy)
3. copy data to db (using pandas)

Pandas and SQLAlchemy are both widely used and work well together.

Databases

The SQLAlchemy library is used for database-abstraction. (ODBC was initially considered as the database-connectivity library of choice, but seeing that most people want to use Excel documents as input files (which would only work on Windows or with unofficial plugins), it became clear that we needed an alternative.)

In this first version, we'll only accept sqlalchemy database urls. In future versions we may want to provide a more abstract and user friendly way.