Bigmetadata Documentation

Release 0.0.1

CARTO

Jul 02, 2018

Contents

1 Quickstart **3**
- 1.1 Requirements . . . 3
- 1.2 Clone & configure . . . 3
- 1.3 Start . . . 3
- 1.4 Run . . . 4

2 Example ETL/metadata pipeline **7**
- 2.1 1. Import libraries . . . 8
- 2.2 2. Download the data . . . 8
- 2.3 3. Import data into PostgreSQL . . . 9
- 2.4 4. Preprocess data in PostgreSQL . . . 10
- 2.5 5. Write metadata . . . 11
- 2.6 6. Populate output table . . . 14

3 Development **19**
- 3.1 Utility Functions . . . 19
- 3.2 Abstract classes . . . 19
- 3.3 Batteries included . . . 19
- 3.4 Running and Re-Running Pieces of the ETL . . . 20

4 Convenience tasks **21**
- 4.1 Makefile . . . 21
- 4.2 Tasks . . . 22
- 4.3 Functions . . . 22

5 Metadata model **23**
- 5.1 Relational Diagram . . . 23
- 5.2 Manually generated entities . . . 24
- 5.3 Autogenerated entities . . . 25

6 Validating your code **29**
- 6.1 Best practices . . . 29
- 6.2 Making sure ETL code works right . . . 32
- 6.3 Making sure metadata works right . . . 33
- 6.4 Regenerate and look at the Catalog . . . 36
- 6.5 Upload to a test CARTO server . . . 36

7 Testing your data **37**
- 7.1 ETL unit tests . . . 37

7.2 Metadata integration tests . . . 37
7.3 API unit tests . . . 37
7.4 Integration tests . . . 38
7.5 Diagnosing common issues in integration tests . . . 38

8 Deploying the Observatory **39**

9 Indices and tables **41**

All data for CARTO's Data Observatory is obtained through tasks built subclassing Bigmetadata ETL classes.

The classes themselves are derived from Luigi tasks.

By performing the ETL using these classes, we gain a few guarantees:

- Reproduceability, and avoidance of duplicate work
- Generation of high-quality metadata consumable by the Observatory API
- Scalability across multiple processes

Contents:

CHAPTER 1

Quickstart

Requirements

You'll need:

- git
- docker 1.13.1+
- docker-compose 1.18.0+

You should also install make to get access to convenience commands, if you don't have it already.

You'll want at least 2GB of memory available on the host machine.

You'll want at least 30GB of disk space available on the host machine to work comfortably with data and get everything running. If you want to install an existing database dump, you will need more like 120GB of space. If you want to install, say, the entire American Community Survey, you will want more like 1TB of space.

Clone & configure

Once your prerequisites are set up, clone the repo:

```
git clone https://github.com/cartodb/bigmetadata.git
cd bigmetadata
touch .env
```

The last line sets up an empty conviguration. If you want to upload to your CARTO account from the ETL, you'll then need to configure `CARTODB_API_KEY` and `CARTODB_URL` in the `.env` file.

If you're on Linux instead of Mac, you may want to give your existing user docker (which is equivalent to root) privileges:

```
sudo gpasswd -a $(whoami) docker
```

Then log out, and log in.

Start

Before running tasks the first time, you'll need to download and start the containers.

```
docker-compose up -d
```

Once the containers are up, you need to confirm that the Postgres container has started.

```
make psql
```

This will attempt to launch into an interactive session with the container Postgres. If it doesn't work, wait a little bit and try again. The database takes some time to get running initially.

Run

You now should be able to run a task.

```
make -- run es.ine.FiveYearPopulation
```

Note: The first time you run it, that command will download a few Docker images. Depending on the speed of your connection, it could take ten or fifteen minutes. Grab a coffee!

That will run `FiveYearPopulation`. This includes downloading all the source data files if they don't already exist locally, and generating all the metadata necessary to make this dataset work with observatory-extension functions.

You can take a look at the data:

```
make psql

gis=# select count(*) from observatory.obs_column;
 count
-------
   169
(1 row)

gis=# select id, name, type, aggregate from observatory.obs_column where name ilike
↪'population%';

             id            |            name             |  type    | aggregate
-------------------------+----------------------------+---------+-----------
 es.ine.pop_0_4           | Population age 0 to 4       | Numeric | sum
 es.ine.pop_5_9           | Population age 5 to 9       | Numeric | sum
 es.ine.pop_10_14         | Population age 10 to 14     | Numeric | sum
 es.ine.pop_15_19         | Population age 15 to 19     | Numeric | sum
 es.ine.pop_20_24         | Population age 20 to 24     | Numeric | sum
 es.ine.pop_25_29         | Population age 25 to 29     | Numeric | sum
 es.ine.pop_30_34         | Population age 30 to 34     | Numeric | sum
 es.ine.pop_35_39         | Population age 35 to 39     | Numeric | sum
 es.ine.pop_40_44         | Population age 40 to 44     | Numeric | sum
 es.ine.pop_45_49         | Population age 45 to 49     | Numeric | sum
 es.ine.pop_50_54         | Population age 50 to 54     | Numeric | sum
 es.ine.pop_55_59         | Population age 55 to 59     | Numeric | sum
 es.ine.pop_60_64         | Population age 60 to 64     | Numeric | sum
 es.ine.pop_65_69         | Population age 65 to 69     | Numeric | sum
 es.ine.pop_70_74         | Population age 70 to 74     | Numeric | sum
 es.ine.pop_75_79         | Population age 75 to 79     | Numeric | sum
 es.ine.pop_80_84         | Population age 80 to 84     | Numeric | sum
 es.ine.pop_85_89         | Population age 85 to 89     | Numeric | sum
```

```
 es.ine.pop_90_94         | Population age 90 to 94     | Numeric | sum
 es.ine.pop_95_99         | Population age 95 to 99     | Numeric | sum
 es.ine.pop_100_more      | Population age 100 or more | Numeric | sum
(21 rows)

gis=# select * from observatory.obs_column_to_column where source_id in (select id␣
↪from observatory.obs_column where name ilike 'population%');

          source_id       |  target_id  |   reltype
-------------------------+-------------+-------------
 es.ine.pop_0_4           | es.ine.t1_1 | denominator
 es.ine.pop_5_9           | es.ine.t1_1 | denominator
 es.ine.pop_10_14         | es.ine.t1_1 | denominator
 es.ine.pop_15_19         | es.ine.t1_1 | denominator
 es.ine.pop_20_24         | es.ine.t1_1 | denominator
 es.ine.pop_25_29         | es.ine.t1_1 | denominator
 es.ine.pop_30_34         | es.ine.t1_1 | denominator
 es.ine.pop_35_39         | es.ine.t1_1 | denominator
 es.ine.pop_40_44         | es.ine.t1_1 | denominator
 es.ine.pop_45_49         | es.ine.t1_1 | denominator
 es.ine.pop_50_54         | es.ine.t1_1 | denominator
 es.ine.pop_55_59         | es.ine.t1_1 | denominator
 es.ine.pop_60_64         | es.ine.t1_1 | denominator
 es.ine.pop_65_69         | es.ine.t1_1 | denominator
 es.ine.pop_70_74         | es.ine.t1_1 | denominator
 es.ine.pop_75_79         | es.ine.t1_1 | denominator
 es.ine.pop_80_84         | es.ine.t1_1 | denominator
 es.ine.pop_85_89         | es.ine.t1_1 | denominator
 es.ine.pop_90_94         | es.ine.t1_1 | denominator
 es.ine.pop_95_99         | es.ine.t1_1 | denominator
 es.ine.pop_100_more      | es.ine.t1_1 | denominator
(21 rows)

gis=# select id, name, type, aggregate from observatory.obs_column where id = 'es.ine.
↪t1_1';
     id      |       name       |  type   | aggregate
-------------+------------------+---------+-----------
 es.ine.t1_1 | Total population | Numeric | sum
(1 row)
```

CHAPTER 2

Example ETL/metadata pipeline

This is a quick guide to building a full ETL pipeline, along with associated metadata, for the Data Observatory.

As an example, we will bring in the Quarterly Census of Employment and Wages (QCEW), a product of the Bureau of Labor Statistics. This dataset tracks the number of employees, firms, and average wages across the full gamut of North American Industry Classification System (NAICS) industries.

QCEW is, of course, a quarterly release, and counties are the smallest geography considered.

The process of building a Python module to bring a new dataset into the Data Observatory can be broadly divided into six steps:

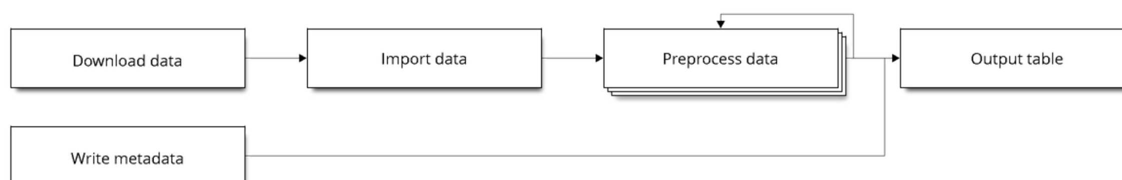
- *1. Import libraries*
- *2. Download the data*
- *3. Import data into PostgreSQL*
- *4. Preprocess data in PostgreSQL*
- *5. Write metadata*
- *6. Populate output table*

We use Luigi to isolate each step into a `Task`. A `Task` has well-defined inputs (other tasks) and outputs (files, tables on disk, etc.) In a nutshell:

- a task cannot be run if it is complete
- if all of a `Task`'s outputs exist, then it is complete
- in order to run, all of a `Task`'s requirements must be complete

Each of the steps except (1) corresponds to a `Task`.

The actual flow of `Task` dependencies could be charted like this:



Name	Description
Download data	DownloadUnzipTask, Task
Import data	CSV2TempTableTask, Shp2TempTableTask, TempTableTask
Preprocess data	TempTableTask
Write metadata	ColumnsTask
Output table	TableTask

Where each step should be a `Task` subclassed from the noted Bigmetadata utility class.

We use a set of utility classes to avoid writing repetitive code.

To get started, make sure you're running the IPython notebook container.

```
docker-compose up -d ipython
```

Then, get the port of the running IPython notebook container:

```
make ps
```

And navigate to it in your browser.

1. Import libraries

```
# Import a test runner

from tests.util import runtask
```

```
# We'll need these basic utility classes and methods

from tasks.util import underscore_slugify, shell, classpath
from tasks.base_tasks import (TempTableTask, TableTask, ColumnsTask,
                              DownloadUnzipTask, CSV2TempTableTask)
from tasks.meta import current_session, DENOMINATOR

# We like OrderedDict because it makes it easy to pass dicts
# like {column name : column definition, ..} where order still
# can matter in SQL
from collections import OrderedDict
from luigi import IntParameter, Parameter
import os
```

```
# These imports are useful for checking the database

from tasks.meta import OBSTable, OBSColumn, OBSTag
```

```
# We'll also want these tags for metadata

from tasks.tags import SectionTags, SubsectionTags, UnitTags
```

2. Download the data

The first step of most ETLs is going to be downloading the source and saving it to a temporary folder.

`DownloadUnzipTask` is a utility class that handles the file naming and unzipping of the temporary output for you. You just have to write the code which will do the download to the output file name.

```
class DownloadQCEW(DownloadUnzipTask):

    year = IntParameter()

    URL = 'http://www.bls.gov/cew/data/files/{year}/csv/{year}_qtrly_singlefile.zip'

    def download(self):
        shell('wget -O {output}.zip {url}'.format(
           output=self.output().path,
           url=self.URL.format(year=self.year)
        ))
```

Within the IPython environment, we can create and run the task within a sandbox.

We have to specify the year, since it's specified as a parameter without a default.

```
download_task = DownloadQCEW(year=2014)
runtask(download_task)
```

Provided the output folder exists, the `DownloadQCEW` task for 2014 will not run again.

```
download_task.output().path
```

```
'tmp/tmp/DownloadQCEW_2014_cfabf27024'
```

```
download_task.output().exists()
```

```
True
```

3. Import data into PostgreSQL

A lot of processing can be done in PostgreSQL quite easily. We have utility classes to more easily bring both Shapefiles and CSVs into PostgreSQL.

For `CSV2TempTableTask`, we only have to define an `input_csv` method that will return a path (or iterable of paths) to the CSV(s). The header row will automatically be checked and used to construct a schema to bring the data in.

The standard `requires` method of Luigi is used here, too. This requires that the `DownloadQCEW` task for the same year must be run beforehand; the `output` from that task is now accessible as the `input` of this one.

```
class RawQCEW(CSV2TempTableTask):

    year = IntParameter()

    def requires(self):
        return DownloadQCEW(year=self.year)

    def input_csv(self):
        return os.path.join(self.input().path,'{}.q1-q4.singlefile.csv'.format(self.
↪year))
```

Run the task. If the table exists and has more than 0 rows, it will not be run again.

```
current_session().rollback()
raw_task = RawQCEW(year=2014)

runtask(raw_task)
```

Confirm the task has completed successfully.

```
raw_task.complete()
```

```
True
```

Session can be used to execute raw queries against the table.

The output of a `TempTableTask` can be queried directly by using its `table` method, which is a string with the fully schema-qualified table name. We are guaranteed names that are unique to the module/task/parameters without having to come up with any names manually.

```
raw_task.output().table
```

```
'"tmp".RawQCEW_2014_cfabf27024'
```

```
session = current_session()
resp = session.execute('select count(*) from {}'.format(raw_task.output().table))
resp.fetchall()
```

```
[(14276508L,)]
```

4. Preprocess data in PostgreSQL

QCEW data has a lot of rows we don't actually need – these can be filtered out in SQL easily.

For QCEW, the download files are annual, but contain quarterly time periods. Output tables should be limited to a single point in time. We're also only interested in private employment (`own_code = '5'`) and county level aggregation by total (71), supersector (73), and NAICS sector (74).

```
class SimpleQCEW(TempTableTask):

    year = IntParameter()
    qtr = IntParameter()

    def requires(self):
        return RawQCEW(year=self.year)

    def run(self):
        session = current_session()
        session.execute("CREATE TABLE {output} AS "
                        "SELECT * FROM {input} "
                        "WHERE agglvl_code IN ('74', '73', '71') "
                        "  AND year = '{year}' "
                        "  AND qtr = '{qtr}' "
                        "  AND own_code = '5' ".format(
                            input=self.input().table,
                            output=self.output().table,
```

```
                            year=self.year,
                            qtr=self.qtr,
                       ))
```

Run the task and confirm it completed. We don't have to run each step as we write it, as requirements guarantee anything required will be run.

```
simple_task = SimpleQCEW(year=2014, qtr=4)
runtask(simple_task)
simple_task.complete()
```

```
True
```

```
simple_task.output().table
```

```
'"tmp".SimpleQCEW_4_2014_79152e4934'
```

```
resp = session.execute('select count(*) from {}'.format(simple_task.output().table))
resp.fetchall()
```

```
[(97167L,)]
```

5. Write metadata

We have to create metadata for the measures we're interested in from QCEW. Often metadata don't take parameters, but this one is, since we have to reorganize the table from one row per NAICS code to one column per NAICS code, which is easiest done programmatically.

The `ColumnsTask` provides a structure for generating metadata. The only required method is `columns`. What must be returned from that method is an `OrderedDict` whose values are all `OBSColumn` and whose keys are all strings. The keys may be used as human-readable column names in tables based off this metadata, although that is not always the case. If the `id` of the `OBSColumn` is left blank, the dict's key will be used to generate it (qualified by the module).

Also, conventionally there will be a `requires` method that brings in our standard tags: `SectionTags`, `SubsectionTags`, and `UnitTags`. This is an example of defining several tasks as prerequisites: the outputs of those tasks will be accessible via `self.input()[<key>]` in other methods.

```
from tasks.us.naics import (NAICS_CODES, is_supersector, is_sector,
                            get_parent_code)

class QCEWColumns(ColumnsTask):

    naics_code = Parameter()

    def requires(self):
        requirements = {
            'sections': SectionTags(),
            'subsections': SubsectionTags(),
            'units': UnitTags(),
        }
        parent_code = get_parent_code(self.naics_code)
        if parent_code:
```

```
            requirements['parent'] = QCEWColumns(naics_code=parent_code)

        return requirements

    def columns(self):
        cols = OrderedDict()
        code, name, description = self.naics_code, NAICS_CODES[self.naics_code], ''

        # This gives us easier access to the tags we defined as dependencies
        input_ = self.input()
        units = input_['units']
        sections = input_['sections']
        subsections = input_['subsections']
        parent = input_.get('parent')
        cols['avg_wkly_wage'] = OBSColumn(
            # Make sure the column ID is unique within this module
            # If left blank, will be taken from this column's key in the output␣
↪OrderedDict
            id=underscore_slugify(u'avg_wkly_wage_{}'.format(code)),
            # The PostgreSQL type of this column.  Generally Numeric for numbers and␣
↪Text
            # for categories.
            type='Numeric',
            # Human-readable name.  Will be used as header in the catalog
            name=u'Average weekly wage for {} establishments'.format(name),
            # Human-readable description.  Will be used as content in the catalog.
            description=u'Average weekly wage for a given quarter in the {name}␣
↪industry (NAICS {code}).'
                        u'{name} is {description}.'.format(name=name, code=code,␣
↪description=description),
            # Ranking of importance, sometimes used to favor certain measures in auto-
↪selection
            # Weight of 0 will hide this column from the user.  We generally use␣
↪between 0 and 10
            weight=5,
            # How this measure was derived, for example "sum", "median", "average",␣
↪etc.
            # In cases of "sum", this means functions downstream can construct␣
↪estimates
            # for arbitrary geographies
            aggregate='average',
            # Tags are our way of noting aspects of this measure like its unit, the␣
↪country
            # it's relevant to, and which section(s) of the catalog it should appear␣
↪in.
            tags=[units['money'], sections['united_states'], subsections['income']],
        )
        cols['qtrly_estabs'] = OBSColumn(
            id=underscore_slugify(u'qtrly_estabs_{}'.format(code)),
            type='Numeric',
            name=u'Establishments in {}'.format(name),
            description=u'Count of establishments in a given quarter in the {name}␣
↪industry (NAICS {code}).'
                        u'{name} is {description}.'.format(name=name, code=code,␣
↪description=description),
            weight=5,
            aggregate='sum',
            tags=[units['businesses'], sections['united_states'], subsections[
↪'commerce_economy']],
```

```
            targets={parent['qtrly_estabs']: DENOMINATOR} if parent else {},
        )
        cols['month3_emplvl'] = OBSColumn(
            id=underscore_slugify(u'month3_emplvl_{}'.format(code)),
            type='Numeric',
            name=u'Employees in {} establishments'.format(name),
            description=u'Number of employees in the third month of a given quarter␣
↪with the {name} '
                        u'industry (NAICS {code}). {name} is {description}.'.format(
                            name=name, code=code, description=description),
            weight=5,
            aggregate='sum',
            tags=[units['people'], sections['united_states'], subsections['employment
↪']],
        )
        cols['lq_avg_wkly_wage'] = OBSColumn(
            id=underscore_slugify(u'lq_avg_wkly_wage_{}'.format(code)),
            type='Numeric',
            name=u'Average weekly wage location quotient for {} establishments'.
↪format(name),
            description=u'Location quotient of the average weekly wage for a given␣
↪quarter relative to '
                        u'the U.S. (Rounded to the hundredths place) within the {name}
↪ industry (NAICS {code}).'
                        u'{name} is {description}.'.format(name=name, code=code,␣
↪description=description),
            weight=3,
            aggregate=None,
            tags=[units['ratio'], sections['united_states'], subsections['income']],
        )
        cols['lq_qtrly_estabs'] = OBSColumn(
            id=underscore_slugify(u'lq_qtrly_estabs_{}'.format(code)),
            type='Numeric',
            name=u'Location quotient of establishments in {}'.format(name),
            description=u'Location quotient of the quarterly establishment count␣
↪relative to '
                        u'the U.S. (Rounded to the hundredths place) within the {name}
↪ industry (NAICS {code}).'
                        u'{name} is {description}.'.format(name=name, code=code,␣
↪description=description),
            weight=3,
            aggregate=None,
            tags=[units['ratio'], sections['united_states'], subsections['commerce_
↪economy']],
        )
        cols['lq_month3_emplvl'] = OBSColumn(
            id=underscore_slugify(u'lq_month3_emplvl_{}'.format(code)),
            type='Numeric',
            name=u'Employment level location quotient in {} establishments'.
↪format(name),
            description=u'Location quotient of the employment level for the third␣
↪month of a given quarter '
                        u'relative to the U.S. (Rounded to the hundredths place)␣
↪within the {name} '
                        u'industry (NAICS {code}). {name} is {description}.'.format(
                            name=name, code=code, description=description),
            weight=3,
            aggregate=None,
```

```
            tags=[units['ratio'], sections['united_states'], subsections['employment
↪']],
        )
        return cols
```

We should never run metadata tasks on their own – they should be defined as requirements by `TableTask`, below – but it is possible to do so, as an example.

NAICS code '1025' is the supersector for eduction & health.

```
education_health_columns = QCEWColumns(naics_code='1025')
runtask(education_health_columns)
education_health_columns.complete()
```

```
True
```

Output from a `ColumnsTask` is an `OrderedDict` with the columns wrapped in `ColumnTargets`, which allow us to pass them around without immediately committing them to the database.

```
education_health_columns.output()
```

```
OrderedDict([('avg_wkly_wage', <tasks.targets.ColumnTarget at 0x7f12a40eead0>),
             ('qtrly_estabs', <tasks.targets.ColumnTarget at 0x7f12a5831c50>),
             ('month3_emplvl', <tasks.targets.ColumnTarget at 0x7f12a5831090>),
             ('lq_avg_wkly_wage', <tasks.targets.ColumnTarget at 0x7f12a4338b10>),
             ('lq_qtrly_estabs', <tasks.targets.ColumnTarget at 0x7f12a4d1fb50>),
             ('lq_month3_emplvl',
              <tasks.targets.ColumnTarget at 0x7f12a4525390>)])
```

We can check the `OBSColumn` table for evidence that our metadata has been committed to disk, since we ran the task.

```
[(col.id, col.name) for col in session.query(OBSColumn)[:5]]
```

```
[(u'tmp.avg_wkly_wage_10',
  u'Average weekly wage for Total, all industries establishments'),
 (u'tmp.qtrly_estabs_10', u'Establishments in Total, all industries'),
 (u'tmp.month3_emplvl_10',
  u'Employees in Total, all industries establishments'),
 (u'tmp.lq_avg_wkly_wage_10',
  u'Average weekly wage location quotient for Total, all industries establishments'),
 (u'tmp.lq_qtrly_estabs_10',
  u'Location quotient of establishments in Total, all industries')]
```

6. Populate output table

Now that we have our data in a format similar to what we'll need, and our metadata lined up, we can tie it together with a `TableTask`. Under the hood, `TableTask` handles the relational lifting between columns and actual data, and assigns a hash number to the dataset.

Several methods must be overriden for `TableTask` to work:

- `version()`: a version control number, which is useful for forcing a re-run/overwrite without having to track down and delete output artifacts.

- `table_timespan()`: the timespan (for example, '2014', or '2012Q4') that identifies the date range or point-in-time for this table.
- `columns()`: an OrderedDict of (colname, ColumnTarget) pairs. This should be constructed by pulling the desired columns from required `ColumnsTask` classes.
- `populate()`: a method that should populate (most often via) INSERT the output table.

```
# Since we have a column ('area_fips') that is a shared reference to
# geometries ('geom_ref') we have to import that column.
from tasks.us.census.tiger import GeoidColumns

class QCEW(TableTask):

    year = IntParameter()
    qtr = IntParameter()

    def version(self):
        return 1

    def requires(self):
        requirements = {
            'data': SimpleQCEW(year=self.year, qtr=self.qtr),
            'geoid_cols': GeoidColumns(),
            'naics': OrderedDict()
        }
        for naics_code, naics_name in NAICS_CODES.iteritems():
            # Only include the more general NAICS codes
            if is_supersector(naics_code) or is_sector(naics_code) or naics_code ==
↪'10':
                requirements['naics'][naics_code] = QCEWColumns(naics_code=naics_code)
        return requirements

    def table_timespan(self):
        return get_timespan('{year}Q{qtr}'.format(year=self.year, qtr=self.qtr))

    def columns(self):
        # Here we assemble an OrderedDict using our requirements to specify the
        # columns that go into this table.
        # The column name
        input_ = self.input()
        cols = OrderedDict([
            ('area_fips', input_['geoid_cols']['county_geoid'])
        ])
        for naics_code, naics_cols in input_['naics'].iteritems():
            for key, coltarget in naics_cols.iteritems():
                naics_name = NAICS_CODES[naics_code]
                colname = underscore_slugify(u'{}_{}_{}'.format(
                        key, naics_code, naics_name))
                cols[colname] = coltarget
        return cols

    def populate(self):
        # This select statement transforms the input table, taking advantage of our
        # new column names.
        # The session is automatically committed if there are no errors.
        session = current_session()
        columns = self.columns()
        colnames = columns.keys()
```

```
        select_colnames = []
        for naics_code, naics_columns in self.input()['naics'].iteritems():
            for colname, coltarget in naics_columns.iteritems():
                select_colnames.append('''MAX(CASE
                    WHEN industry_code = '{naics_code}' THEN {colname} ELSE NULL
                END)::Numeric'''.format(naics_code=naics_code,
                                        colname=colname
                                      ))
        insert = '''INSERT INTO {output} ({colnames})
                    SELECT area_fips, {select_colnames}
                    FROM {input}
                    GROUP BY area_fips '''.format(
                        output=self.output().table,
                        input=self.input()['data'].table,
                        colnames=', '.join(colnames),
                        select_colnames=', '.join(select_colnames),
                    )
        session.execute(insert)
```

On a fresh database, this should return False Will not run if it has been run before for this year & quarter combination.

```
table_task = QCEW(year=2014, qtr=4)
runtask(table_task)
table_task.complete()
```

```
True
```

The table should exist in metadata, as well as in data, with all relations well-defined.

Unlike the `TempTableTasks` above, the output of a `TableTask` is a postgrse table in the `observatory` schema, with a unique hash name.

```
table = table_task.output()
table.table
```

```
'observatory.obs_3dc49b70f71ed9bbf5b4a48773c860519af70e1e'
```

It's possible for us to peek at the output data.

```
session.execute('SELECT * FROM {} LIMIT 1'.format(table.table)).fetchall()
```

```
[(u'01001', None, None, None, None, None, None, Decimal('395'), Decimal('5'), Decimal(
↪'144'), Decimal('0.65'), Decimal('0.52'), Decimal('0.68'), Decimal('609'), Decimal(
↪'80'), Decimal('1024'), Decimal('0.96'), Decimal('0.66'), Decimal('0.74'), Decimal(
↪'364'), Decimal('68'), Decimal('368'), Decimal('0.79'), Decimal('0.95'), Decimal('1.
↪13'), Decimal('917'), Decimal('3'), Decimal('66'), Decimal('0.68'), Decimal('0.94
↪'), Decimal('1.00'), Decimal('2317'), Decimal('5'), Decimal('103'), Decimal('1.89
↪'), Decimal('3.26'), Decimal('2.45'), Decimal('914'), Decimal('77'), Decimal('426
↪'), Decimal('1.17'), Decimal('1.16'), Decimal('0.90'), Decimal('1231'), Decimal('33
↪'), Decimal('157'), Decimal('1.26'), Decimal('0.60'), Decimal('0.35'), Decimal('925
↪'), Decimal('20'), Decimal('198'), Decimal('1.14'), Decimal('1.66'), Decimal('1.30
↪'), Decimal('914'), Decimal('77'), Decimal('426'), Decimal('1.17'), Decimal('1.16
↪'), Decimal('0.90'), Decimal('1225'), Decimal('30'), Decimal('1347'), Decimal('1.45
↪'), Decimal('1.01'), Decimal('1.44'), Decimal('584'), Decimal('85'), Decimal('1168
↪'), Decimal('0.93'), Decimal('0.65'), Decimal('0.73'), Decimal('904'), Decimal('91
↪'), Decimal('380'), Decimal('0.98'), Decimal('0.61'), Decimal('0.25'), None, None, ␣
↪None, None, None, None, Decimal('433'), Decimal('149'), Decimal('1935'), Decimal('1.
↪13'), Decimal('1.63'), Decimal('1.57'), Decimal('1225'), Decimal('30'), Decimal(
↪'1347'), Decimal('1.45'), Decimal('1.01'), Decimal('1.44'), Decimal('274'), Decimal(
↪'66'), Decimal('1432'), Decimal('1.10'), Decimal('1.13'), Decimal('1.50'), Decimal(
↪'301'), Decimal('8'), Decimal('66'), Decimal('0.53'), Decimal('0.68'), Decimal('0.44
↪'), Decimal('620'), Decimal('15'), Decimal('127'), Decimal('0.97'), Decimal('0.73
↪'), Decimal('0.36'), None, None, None, None, None, None, Decimal('929'), Decimal('17
↪'), Decimal('132'), Decimal('2.13'), Decimal('1.91'), Decimal('1.53'), Decimal('677
```

CHAPTER 3

Development

Writing ETL tasks is pretty repetitive. In `tasks.util` are a number of functions and classes that are meant to make life easier through reusability.

- *Utility Functions*
- *Abstract classes*
- *Batteries included*
- *Running and Re-Running Pieces of the ETL*
  - *Using `--force` during development*
  - *Deleting byproducts to force a re-run of parts of ETL*
  - *Update the ETL & metadata through `version`*

Utility Functions

These functions are very frequently used within the methods of a new ETL task.

`tasks.meta.`**`current_session`**`()`

Returns the session relevant to the currently operating `Task`, if any. Outside the context of a `Task`, this can still be used for manual session management.

Abstract classes

These are the building blocks of the ETL, and should almost always be subclassed from when writing a new process.

Batteries included

Data comes in many flavors, but sometimes it comes in the same flavor over and over again. These tasks are meant to take care of the most repetitive aspects.

Running and Re-Running Pieces of the ETL

When doing local development, it's advisable to run small pieces of the ETL locally to make sure everything works correctly. You can use the `make -- run` helper, documented in *Run any task*. There are several methods for re-running pieces of the ETL depending on the task and are described below:

Using `--force` during development

When developing with abstract-classes that offer a `force` parameter, you can use it to re-run a task that has already been run, ignoring and overwriting all output it has already created. For example, if you have a tasks.base_tasks.TempTableTask that you've modified in the course of development and need to re-run:

```
from tasks.base_tasks import TempTableTask
from tasks.meta import current_session

class MyTempTable(TempTableTask):

    def run(self):
        session = current_session()
        session.execute('''
           CREATE TABLE {} AS SELECT 'foo' AS mycol;
        ''')
```

Running `make -- run path.to.module MyTempTable` will only work once, even after making changes to the `run` method.

However, running `make -- run path.to.module MyTempTable --force` will force the task to be run again, dropping and re-creating the output table.

Deleting byproducts to force a re-run of parts of ETL

In some cases, you may have a luigi.Task you want to re-run, but does not have a `force` parameter. In such cases, you should look at its `output` method and delete whatever files or database tables it created.

Utility classes will put their file byproducts in the `tmp` folder, inside a folder named after the module name. They will put database byproducts into a schema that is named after the module name, too.

Update the ETL & metadata through `version`

When you make changes and improvements, you can increment the `version` method of tasks.base_tasks.TableTask, tasks.base_tasks.ColumnsTask and tasks.base_tasks.TagsTask to force the task to run again.

CHAPTER 4

Convenience tasks

There are a number of tasks and functions useful for basic, repetitive operations like interacting with or uploading tables to CARTO.

- *Makefile*
  - *Run any task*
  - *Other tasks*
- *Tasks*
- *Functions*

Makefile

The Makefile makes it easier to run tasks.

Run any task

Any task can be run with:

```
make -- run path.to.module ClassName --param-name-1 value1 --param-name-2 value2
```

For example:

```
make -- run us.bls QCEW --year 2014 --qtr 4
```

Other tasks

- `make dump`: Runs `DumpS3`
- `make restore <path/to/dump>`: Restore database from a `dump`.
- `make sync-meta`: Runs `SyncMetadata`
- `make sync-data`: Runs `SyncAllData`
- `make sh`: Drop into an interactive shell in the Docker container

- `make psql`: Drop into an interactive psql session in the database
- `make kill`: Kill all Docker processes
- `make docs`: Regenerate all documentation
- `make catalog`: Regenerate the HTML catalog

Tasks

Functions

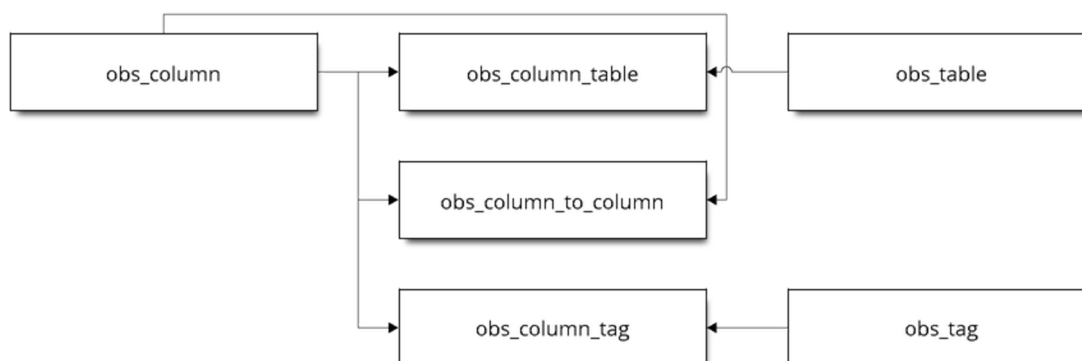
CHAPTER 5

Metadata model

Our metadata is contained in six highly related tables, which are defined using SQLAlchemy classes.

- *Relational Diagram*
- *Manually generated entities*
- *Autogenerated entities*

Relational Diagram



Name	Description
obs_column	`OBSColumn`
obs_column_table	`OBSColumnTable`
obs_column_to_column	`OBSColumnToColumn`
obs_column_tag	`OBSColumnTag`
obs_table	`OBSTable`
obs_tag	`OBSTag`

Manually generated entities

class `tasks.meta.`**`OBSColumn`**`(**kwargs)`

Describes the characteristics of data in a column, which can exist in multiple physical tables.

These should *only* be instantiated to generate the return value of `ColumnsTask.columns()`. Any other usage could have unexpected consequences.

`id`
The unique identifier for this column. Should be qualified by the module name of the class that created it. Is automatically generated by `ColumnsTask` if left unspecified, and automatically qualified by module name either way.

`type`
The type of this column – for example, `Text`, `Numeric`, `Geometry`, etc. This is used to generate a schema for any table using this column.

`name`
The human-readable name of this column. This is used in the catalog and API.

`description`
The human-readable description of this column. THis is used in the catalog.

`weight`
A numeric weight for this column. Higher weights are favored in certain rankings done by the API. Defaults to zero, which hides the column from catalog.

`aggregate`
How this column can be aggregated. For example, populations can be summed, so a population column should be `sum`. Should be left blank if it is not to aggregate.

`tables`
Iterable of all linked :class:‘~.meta.OBSColumnTable‘s, which could be traversed to find all tables with this column.

`tags`
Iterable of all linked :class:‘~.meta.OBSColumnTag‘s, which could be traversed to find all tags applying to this column.

`targets`
Dict of all related columns. Format is `<OBSColumn>: <reltype>`.

`version`
A version control number, used to determine whether the column and its metadata should be updated.

`extra`
Arbitrary additional information about this column stored as JSON.

`catalog_lonlat``()`
Return tuple (longitude, latitude) for the catalog for this measurement.

`denominators``()`
Return the `OBSColumn` denominator of this column.

`geom_timespans``()`
Return a dict of geom columns and timespans that this measure is available for.

`has_catalog_image``()`
Returns True if this column has a pre-generated image for the catalog.

`has_children``()`
Returns True if this column has children, False otherwise.

has_denominators()
Returns True if this column has no denominator, False otherwise.

is_cartographic()
Returns True if this column is a geometry that can be used for cartography.

is_geomref()
Returns True if the column is a geomref, else Null

is_interpolation()
Returns True if this column is a geometry that can be used for interpolation.

license_tags()
Return license tags.

source_tags()
Return source tags.

summable()
Returns True if we can sum this column and calculate for arbitrary areas.

unit()
Return the `OBSTag` unit of this column.

class `tasks.meta.`**OBSTag**(***kwargs*)
Tags permit arbitrary groupings of columns.

They should only be created as part of a `tags()` implementation.

id
The unique identifier for this table. Is always qualified by the module name of the `TagsTask` that created it, and should never be specified manually.

name
The name of this tag. This is exposed in the API and user interfaces.

type
The type of this tag. This is used to provide more information about what the tag means. Examples are `section`, `subsection`, `license`, `unit`, although any arbitrary type can be defined.

description
Description of this tag. This may be exposed in the API and catalog.

columns
An iterable of all the :class:‘~.meta.OBSColumn‘s tagged with this tag.

version
A version control number, used to determine whether the tag and its metadata should be updated.

Autogenerated entities

class `tasks.meta.`**OBSColumnToColumn**(***kwargs*)
Relates one column to another. For example, a `Text` column may contain identifiers that are unique to geometries in another `Geometry` column. In that case, an `OBSColumnToColumn` object of `reltype GEOM_REF` should indicate the relationship, and make relational joins possible between tables that have both columns with those that only have one.

These should *never* be created manually. Their creation should be handled automatically from specifying *`OBSColumn.targets`*.

These are unique on `(source_id,target_id)`.

source_id
ID of the linked source `OBSColumn`.

target_id
ID of the linked target `OBSColumn`.

reltype
required text specifying the relation type. Examples are `GEOM_REF` and *~.meta.DENOMINATOR*.

source
The linked source `OBSColumn`.

target
The linked target `OBSColumn`.

class `tasks.meta.`**`OBSColumnTable`**`(`***kwargs*`)`
Glues together `OBSColumn` and `OBSTable`. If this object exists, the related column should exist in the related table, and can be selected with *`colname`*.

Unique along both `(column_id,table_id)` and `(table_id,colname)`.

These should *never* be created manually. Their creation and removal is handled automatically as part of `targets.TableTarget.update_or_create_metadata()`.

column_id
ID of the linked `OBSColumn`.

table_id
ID of the linked `OBSTable`.

colname
Column name for selecting this column in a table.

column
The linked `OBSColumn`.

table
The linked `OBSTable`.

extra
Extra JSON information about the data. This could include statistics like max, min, average, etc.

class `tasks.meta.`**`OBSTable`**`(`***kwargs*`)`
Describes a physical table in our database.

These should *never* be instantiated manually. They are automatically created by `output()`. The unique key is :attr:~.meta.OBSTable.id:.

id
The unique identifier for this table. Is always qualified by the module name of the class that created it.

columns
An iterable of all the `OBSColumnTable` instances contained in this table.

tablename
The automatically generated name of this table, which can be used directly in select statements. Of the format `obs_<hash>`.

timespan
An OBSTimespan instance containing information about the timespan this table applies to. Obtained from `timespan()`.

the_geom
A simple geometry approximating the boundaries of the data in this table.

description
A description of the table. Not used.

version
A version control number, used to determine whether the table and its metadata should be updated.

geom_column()
Return the column geometry column for this table, if it has one.

Returns None if there is none.

geomref_column()
Return the geomref column for this table, if it has one.

Returns None if there is none.

class tasks.meta.**OBSColumnTag**(***kwargs*)
Glues together `OBSColumn` and `OBSTag`. If this object exists, the related column should be tagged with the related tag.

Unique along `(column_id,tag_id)`.

These should *never* be created manually. Their creation and removal is handled automatically as part of `targets.TableTarget.update_or_create_metadata()`.

column_id
ID of the linked `OBSColumn`.

tag_id
ID of the linked `OBSTag`.

column
The linked `OBSColumn`.

tag
The linked `OBSTag`.

CHAPTER 6

Validating your code

- *Best practices*
  - *Proper use of utility classes*
  - *Clearly documented command that runs a WrapperTask to create everything*
  - *Use parameters only when necessary*
  - *Use default parameter values sparingly*
  - *Keep fewer than 1000 columns per table*
  - *Each geometry column should have a unique `geom_ref` column with it*
  - *Specify section, subsection, source tags and license tags for all columns*
  - *Specify unit tags for all measure columns*
- *Making sure ETL code works right*
  - *Results and byproducts are being generated*
  - *Delete old data to start from scratch to make sure everything works*
- *Making sure metadata works right*
  - *Regenerate the `obs_meta` table*
- *Regenerate and look at the Catalog*
- *Upload to a test CARTO server*

Best practices

Writing ETL code is meant to be open-ended, but there are some standards that should be followed to help keep the code clean and clear.

Proper use of utility classes

There are extensive abstract-classes available for development. These can do things like download and unzip a file to disk (tasks.base_tasks.DownloadUnzipTask) and import a CSV on disk to a temporary table

(tasks.base_tasks.CSV2TempTableTask). These classes should be used when possible to minimize specialized ETL code. In particular, these tasks save output to well-known locations so as to avoid redundantly running the same tasks.

Clearly documented command that runs a WrapperTask to create everything

Oftentimes an ETL will have to loop over a parameter to get all the data – for example, if a dataset is available online year-by-year, it may make sense to write a single task that downloads one year's file, with an parameter specifying which year.

luigi.WrapperTask is a way to make sure such tasks are executed for every relevant parameter programmatically. A powerful example of this can be found with tasks.us.AllZillow, which executes a tasks.us.zillow.Zillow task once for each geography level, year, and month in that year.

A generic example of using a luigi.WrapperTask:

```
from luigi import WrapperTask, Task, Parameter

class MyTask(Task):
    '''
    This task needs to be run for each possible `geog`
    '''

    geog = Parameter()

    def run(self):
        pass

    def output(self):
        pass


class MyWrapperTask(WrapperTask):
    '''
    Execute `MyTask` once for each possible `geog`.
    '''

    def requires(self):
        for geog in ('state', 'county', 'city'):
            yield MyTask(geog=geog)
```

Use parameters only when necessary

Tasks are unique to their parameters. In other words, if a task is run once with a certain set of parameters, it will not be run again unless the output it generated is deleted.

Therefore it's very important to not have parameters available in a Task's definition that do not affect its result. If you have such extraneous parameters, it would be possible to run a task redundantly.

An example of this:

```
from tasks.base_tasks import DownloadUnzipTask

class MyBadTask(DownloadUnzipTask):

    goodparam = Parameter()
    badparam = Parameter()
```

```
    def url(self):
        return 'http://somesite/with/data/{}'.format(self.goodparam)
```

tasks.base_tasks.DownloadUnzipTask will generate the location for a unique output file automatically based off of all its params, but `badparam` above doesn't actually affect the file being downloaded. That means if we change `badparam` we'll download the same file twice.

Use default parameter values sparingly

The above bad practice is easily paired with setting default values for parameters. For example:

```
from tasks.base_tasks import DownloadUnzipTask

class MyBadTask(DownloadUnzipTask):
    '''
    My URL doesn't depend on `badparam`!
    '''

    goodparam = Parameter()
    badparam = Parameter(default='foo')

    def url(self):
        return 'http://somesite/with/data/{}'.format(self.goodparam)
```

Now it's easy to simply forget that `badparam` even exists! But it still affects the output filename, making it noisy and less clear which parameters actually matter.

Keep fewer than 1000 columns per table

Postgres has a hard limit on the number of columns. If you create a tasks.base_tasks.TableTask whose `columns` method returns a OrderedDict with much more than 1000 columns, the task will fail.

In such cases, you'll want to split your tasks.base_tasks.TableTask into several pieces, likely pulling columns from the same tasks.base_tasks.ColumnsTask. There is no limit on the number of columns in a tasks.base_tasks.ColumnsTask.

Each geometry column should have a unique `geom_ref` column with it

When setting up a tasks.base_tasks.ColumnsTask for Geometries, make sure that you store a meaningful and unique `geom_ref` from the same table.

- It is meaningful if it can be found as a way to refer to that geometry in data sources elsewhere – for example, FIPS codes are meaningful references to county geometries in the USA. However, the automatically generated serial `ogc_fid` column from a Shapefile is not meaningful.
- It is unique if that `geom_ref` column has an ID that is not duplicated by any other columns.

For example:

```
from tasks.base_tasks import ColumnsTask
from tasks.meta import OBSColumn, GEOM_REF
from luigi import Parameter

class MyGeoColumnsTask(ColumnsTask):
```

```
    resolution = Parameter()

    def columns(self):

        geom = OBSColumn(
          id=self.resolution,
          type='Geometry')

        geomref = OBSColumn(
          id=self.resolution + '_id',   # Make sure we have "+ '_id'"!
          type='Text',
          targets={geom: GEOM_REF})

        return OrderedDict([
          ('geom', geom),
          ('geomref', geomref)
        ])
```

No matter what `resolution` this Task is passed, it will generate a unique ID for both the `geom` and the `geomref`. If the `+ '+id'` concatenation were missing, it would mean that the metadata model would not properly link geomrefs to the geometries they refer to.

Specify section, subsection, source tags and license tags for all columns

When defining your tasks.meta.OBSColumn objects in a tasks.base_tasks.ColumnsTask class, make sure each column is assigned a tasks.meta.OBSTag of `type`, `section`, `subsection`, `source`, and `license`. Use shared tags from tasks.tags when possible, in particular for `section` and `subsection`.

Specify unit tags for all measure columns

When defining a tasks.meta.OBSColumn that will hold a measurement, make sure to define a `unit` using a tag. This could be something like `people`, `money`, etc. There are standard units accessible in tasks.tags.

Making sure ETL code works right

After having written an ETL, you'll want to double check all of the following to make sure the code is usable.

Results and byproducts are being generated

When you use *Run any task* to run individual components:

- Were any exceptions thrown? On what task were they thrown? With which arguments?
- Are appropriate files being generated in the `tmp` folder?
- Are tables being created in the relevant `tmp` schema?
- Are tables and columns being added to the `observatory.obs_table` and `observatory.obs_column` metadata tables?

Provided tasks.base_tasks.TableTask and tasks.base_tasks.ColumnTask classes were executed, it's wise to jump into the database and check to make sure entries were made in those tables.

```
make psql
```

```
SELECT COUNT(*) FROM observatory.obs_column WHERE id LIKE 'path.to.module.%';

SELECT COUNT(*) FROM observatory.obs_table WHERE id LIKE 'path.to.module.%';

SELECT COUNT(*) FROM observatory.obs_column_table
WHERE column_id LIKE 'path.to.module%'
  AND table_id  LIKE 'path.to.module%';
```

Delete old data to start from scratch to make sure everything works

When using the proper utility classes, your data on disk, for example from downloads that are part of the ETL, will be saved to a file or folder `tmp/module.name/ClassName_Args`.

In order to make sure the ETL is reproduceable, it's wise to delete this folder or move it to another location after development, and re-run to make sure that the whole process can still run from start to finish.

Making sure metadata works right

Checking the metadata works right is one of the more challenging components of QA'ing new ETL code.

Regenerate the `obs_meta` table

The `obs_meta` table is a denormalized view of the underlying metadata objects that you've created when running tasks.

You can force the regeneration of this table using tasks.carto.OBSMetaToLocal

```
make -- run carto OBSMetaToLocal --force
```

Once the table is generated, you can take a look at it in SQL:

```
make psql
```

If the metadata is working correctly, you should have more entries in `obs_meta` than before. If you were starting from nothing, there should be more than 0 rows in the table.

```
SELECT COUNT(*) FROM observatory.obs_meta;
```

If you already had data, you can filter `obs_meta` to look for new rows with a schema corresponding to what you added. For example, if you added metadata columns and tables in `tasks/mx/inegi`, you should look for columns with that schema:

```
SELECT COUNT(*) FROM observatory.obs_meta WHERE numer_id LIKE 'mx.inegi.%';
```

If nothing is appearing in `obs_meta`, chances are you are missing some metadata:

Have you defined and executed a proper tasks.base_tasks.TableTask?

You can check to see if these links exist by checking `obs_column_table`:

```
make psql
```

```
SELECT COUNT(*) FROM observatory.obs_column_table
WHERE column_id LIKE 'my.schema.%'
  AND table_id LIKE 'my.schema.%';
```

If they don't exist, make sure that your Python code roughly corresponds to:

```
from tasks.base_tasks import ColumnsTask, TableTask

class MyColumnsTask(ColumnsTask):

    def columns(self):
        # Return OrderdDict of columns here

class MyTableTask(TableTask):

    def table_timespan(self):
        # Return timespan here

    def requires(self):
        return {
            'columns': MyColumnsTask()
         }

    def columns(self):
        return self.input()['columns']

    def populate(self):
        # Populate the output table here
```

Unless the TableTask returns some of the columns from ColumnsTask in its own `columns` method, the links will not be initialized properly.

Finally, double check that you actually ran the TableTask using `make -- run my.schema MyTableTask`.

Are you defining `geom_ref` relationships properly?

In cases where a TableTask does not have its own geometries, at least one of the columns returned from its `columns` method needs to be in a `geom_ref` relationship. Here's an example:

```
from collections import OrderedDict

from tasks.base_tasks import ColumnsTask, TableTask
from tasks.meta import OBSColumn, GEOM_REF

class MyGeoColumnsTask(ColumnsTask):
    def columns(self):

        geom = OBSColumn(
          type='Geometry')

        geomref = OBSColumn(
          type='Text',
          targets={geom: GEOM_REF})

        return OrderedDict([
```

```
            ('geom', geom),
            ('geomref', geomref)
        ])

class MyColumnsTask(ColumnsTask):

    def columns(self):
        # Return OrderdDict of columns here

class MyTableTask(TableTask):

    def table_timespan(self):
        # Return timespan here

    def requires(self):
        return {
            'geom_columns': MyGeoColumnsTask(),
            'data_columns': MyColumnsTask()
         }

    def columns(self):
        cols = OrderedDict()
        cols['geomref'] = self.input()['geom_columns']['geomref']
        cols.update(self.input()['data_columns'])
        return cols

    def populate(self):
        # Populate the output table here
```

The above code would ensure that all columns existing inside `MyTableTask` would be appropriately linked to any geometries that connect to `geomref`.

Do you have both the data and geometries in your table?

You can check by running:

```
SELECT * FROM observatory.obs_table
WHERE id LIKE 'my.schema.%';
```

If there is only one table and it has a null "the_geom" boundary, then you are missing a geometry table. For example:

```
SELECT * from observatory.obs_table
WHERE id LIKE 'es.ine.five_year_population%';
```

```
                   id                   |                  tablename                    
↪ | timespan | the_geom | description | version
----------------------------------------+---------------------------------------------
↪-+----------+----------+-------------+---------
 es.ine.five_year_population_99914b932b | obs_
↪24b656e9e23d1dac2c8ab5786a388f9bf0f4e5ae | 2015     |          |             |         
↪ 5
(1 row)
```

Notice that the_geom is empty. You will need to write a second TableTask with the following structure:

```
class Geometry(TableTask):

     def table_timespan(self):
         # Return timespan here

     def requires(self):
        return {
            'meta': MyGeoColumnsTask(),
            'data': RawGeometry()
        }

    def columns(self):
        return self.input()['meta']

     def populate(self):
         # Populate the output table here
```

Regenerate and look at the Catalog

Once tasks.carto.OBSMetaToLocal has been run, you can generate the catalog.

```
make catalog
```

You can view the generated Catalog in a browser window by going to the IP and port address for the nginx process. The current processes are shown with `docker-compose ps` or `make ps`.

1. Are there any nasty typos or missing data?
   - Variable names should be unique, human-readable, and concise. If the variable needs more in-depth definition, this should go in the "description" of the variable.
2. Does the nesting look right? Are there columns not nested?
   - Variables that are denominators should also have subcolumns of direct nested variables.
   - There may be repetitive nesting if a variable is nested under two denominators, which is fine.
3. Are sources and licenses populated for all measures?
   - A source and license tasks.util.OBSTag must be written for new sources and licenses
4. Is a table with a boundary/timespan matrix appearing beneath each measure?
   - If not, hardcode the sample latitude and longitude in tasks.meta.catalog_lonlat.

Upload to a test CARTO server

If you set a `CARTODB_API_KEY` and `CARTODB_URL` in your `.env` file, in the format:

```
CARTODB_API_KEY=your_api_key
CARTODB_URL=https://username.carto.com
```

You will now be able to upload your data and metadata to CARTO for previewing.

```
make sync
```

CHAPTER 7

Testing your data

- *ETL unit tests*
- *Metadata integration tests*
- *API unit tests*
- *Integration tests*
- *Diagnosing common issues in integration tests*
  - *Cannot find point to test*

ETL unit tests

Unit tests ensure that there are no errors in the underlying utility classes that could cause errors in code you build on top of them.

Tests are run with:

```
make etl-unittest
```

Metadata integration tests

Integration tests make sure that the metadata being generated as part of your ETL will actually be queryable by the API. For example, if you have an ETL that ingests data but does not ..

API unit tests

API unit tests make sure the observatory-extension, which reads data and metadata from the ETL, are working right.

In order for this to function, you'll need to clone a copy of `observatory-extension` into the root of the `bigmetadata` repo.

```
git clone git@github.com:cartodb/observatory-extension
make extension-unittest
```

Integration tests

Integration tests ensure that the data from the ETL that is set for deployment is is able to return a measure for every piece of metadata.

As above, you'll need a copy of `observatory-extension` locally for this test to work.

```
git clone git@github.com:cartodb/observatory-extension
make extension-autotest
```

Diagnosing common issues in integration tests

Cannot find point to test

TODO

CHAPTER 8

Deploying the Observatory

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

A

aggregate (OBSColumn attribute), 24

C

catalog_lonlat() (tasks.meta.OBSColumn method), 24
colname (OBSColumnTable attribute), 26
column (OBSColumnTable attribute), 26
column (OBSColumnTag attribute), 27
column_id (OBSColumnTable attribute), 26
column_id (OBSColumnTag attribute), 27
columns (OBSTable attribute), 26
columns (OBSTag attribute), 25
current_session() (in module tasks.meta), 19

D

denominators() (tasks.meta.OBSColumn method), 24
description (OBSColumn attribute), 24
description (OBSTable attribute), 26
description (OBSTag attribute), 25

E

extra (OBSColumn attribute), 24
extra (OBSColumnTable attribute), 26

G

geom_column() (tasks.meta.OBSTable method), 27
geom_timespans() (tasks.meta.OBSColumn method), 24
geomref_column() (tasks.meta.OBSTable method), 27

H

has_catalog_image() (tasks.meta.OBSColumn method), 24
has_children() (tasks.meta.OBSColumn method), 24
has_denominators() (tasks.meta.OBSColumn method), 24

I

id (OBSColumn attribute), 24
id (OBSTable attribute), 26
id (OBSTag attribute), 25
is_cartographic() (tasks.meta.OBSColumn method), 25
is_geomref() (tasks.meta.OBSColumn method), 25
is_interpolation() (tasks.meta.OBSColumn method), 25

L

license_tags() (tasks.meta.OBSColumn method), 25

N

name (OBSColumn attribute), 24
name (OBSTag attribute), 25

O

OBSColumn (class in tasks.meta), 24
OBSColumnTable (class in tasks.meta), 26
OBSColumnTag (class in tasks.meta), 27
OBSColumnToColumn (class in tasks.meta), 25
OBSTable (class in tasks.meta), 26
OBSTag (class in tasks.meta), 25

R

reltype (OBSColumnToColumn attribute), 26

S

source (OBSColumnToColumn attribute), 26
source_id (OBSColumnToColumn attribute), 25
source_tags() (tasks.meta.OBSColumn method), 25
summable() (tasks.meta.OBSColumn method), 25

T

table (OBSColumnTable attribute), 26
table_id (OBSColumnTable attribute), 26
tablename (OBSTable attribute), 26
tables (OBSColumn attribute), 24
tag (OBSColumnTag attribute), 27
tag_id (OBSColumnTag attribute), 27
tags (OBSColumn attribute), 24
target (OBSColumnToColumn attribute), 26
target_id (OBSColumnToColumn attribute), 26
targets (OBSColumn attribute), 24
the_geom (OBSTable attribute), 26

timespan (OBSTable attribute), 26
type (OBSColumn attribute), 24
type (OBSTag attribute), 25

U

unit() (tasks.meta.OBSColumn method), 25

V

version (OBSColumn attribute), 24
version (OBSTable attribute), 27
version (OBSTag attribute), 25

W

weight (OBSColumn attribute), 24