pylytics Documentation

Release 1.2.2

onefinestay

July 14, 2016

Contents

CHAPTER 1

Introduction

This is a set of Python libraries that allow you to build and manage a star schema in MySQL.

The star schema is a simple approach to data warehousing, which is suited to mid-size data problems.

1.1 MySQL

Pylytics has been tested with MySQL versions 5.5.37 and 5.6.5. The recommended version is 5.6.5.

CHAPTER 2

Installation

Using a recent version of pip (tested with version 8.1.2):

```
pip install pylytics
```

CHAPTER 3

Documentation

The full documentation is available at readthedocs.org

CHAPTER 4

Contents

4.1 Getting Started

4.1.1 Introduction

What is pylytics?

pylytics is a tool for processing data into a star schema. The industry term for this is an ETL tool (extract, transform, load).

- Extract - pull the data from a range of different sources (SQL, flat files, APIs).
- Transform - clean the data, and prepare it for populating fact and dimension tables.
- Load - create the fact and dimension tables in a datawarehouse, and populate them with the cleaned data.

What are star schemas?

A star schema is a way of storing data which it is intended to be analysed.

There are many data visualisation tools on the market which work with star schemas. However, creating the star schema can be tricky in the first place, and this is what pylytics helps to make easier.

A star schema consists of two types of tables - *facts* and *dimensions*. Each fact table represents something you're interesting in measuring / recording. An example is sales in a retail store.

The things you want to record are known as *metric* columns in pylytics (though other tools may refer to them as *measures*). An example metric is sales amount. Metrics are numeric values, which will be summed, averaged, and counted by the visualisation tool.

Fact tables also consist of dimension columns, which contain foreign keys to dimension tables. Dimension tables describe the data being recorded. Examples are store, manager, and date. Dimensions are used to filter and group the metrics in useful ways.

A fact table will likely only contain one or two metric columns, but can contain dozens of dimension columns. This allows for powerful analysis.

Using the online store example, you can easily do queries such as 'show me the total sales of shampoo, for the New York store, during December, when the manager was Tom Plank'.

The more dimension tables you create, the easier it is to make future fact tables. A fact table could be created which records retail stock, and this could reuse a lot of the dimensions created for the retail sales fact (e.g. store, date).

4.1.2 Creating a new pylytics project

Create a virtualenv and activate it. Then install pylytics:

```
pip install pylytics
```

This installation adds *pylytics-admin.py* to the path. Use this to create a new pylytics project:

```
pylytics-admin.py my_project_name
```

This creates the my_project_name folder in the current directory, with a skeleton project inside.

Project structure

The project structure is as follows:

```
__init__.py
fact/
    __init__.py
    example_project
        __init__.py
        extract.py
        transform.py
        load.py
dimension/
    __init__.py
    example_project
        __init__.py
        extract.py
        transform.py
        load.py
shared/
    __init__.py
manage.py
settings.py
client.cnf
```

File purposes

- extract.py - contains all functions for pulling the raw data.
- transform.py - contains functions making up the 'expansion pipeline', which cleans and expands upon the raw data.
- load.py - contains fact and dimension definitions.

Why are facts and dimensions separated?

It's likely that your dimensions will be shared across several facts, which is why they're in separate folders.

However, there is no problem with declaring your dimensions in the same file as your facts.

The only constraint on project structure which must be followed is the facts you want to make available to the manage.py script have to be imported in fact/__init__.py.

4.1.3 Project settings

Make sure that the DATABASES dictionary in settings.py contains the details for all the databases you need.

At the moment, only MySQL databases are supported.

pylytics_db specifies which of these connections is your datawarehouse, which will be used for inserting facts and dimensions into.

client.cnf

This file is passed into the MySQL connector (Oracle's Python connector is used under the hood). It allows you to configure the MySQL connections you make from the client side. It's unlikely you'll need this, but it's useful for performance tweaking if required.

4.1.4 Writing facts and dimensions

Dimensions

Column Types

The two column types in a *Dimension* are *NaturalKey* and *Column*.

NaturalKey *NaturalKey* columns uniquely identify a row in a dimension table.

Examples of *natural keys* are telephone number, National Insurance number (UK), Social Security number (US), car number plate etc. They are values which are naturally unique, as opposed to a *surrogate key* which is an integer assigned to a row, but has no relationship to the underlying data in that row.

Taking the following as an example:

```python
# load.py

class Store(Dimension):

    __source__ = NotImplemented

    store_id = NaturalKey('store_id', int)
    store_shortcode = NaturalKey('store_shortcode', basestring)
    store_size = Column('store_size', basestring)
    employees = Column('employees', int)


class Sales(Fact):

    __source__ = NotImplemented

    sales = Metric('sales', int)
    store = DimensionKey('store', Store)
    ...
```

The NaturalKey has two arguments - the column name, and a Python type. The Python types are mapped to MySQL types when the tables are created. Here are some examples:

```
_type_map = {
   bool: "TINYINT",
   date: "DATE",
   datetime: "TIMESTAMP",
   Decimal: "DECIMAL(%s,%s)",
   float: "DOUBLE",
   int: "INT",
   long: "INT",
   timedelta: "TIME",
   time: "TIME",
   basestring: "VARCHAR(%s)",
   str: "VARCHAR(%s)",
   unicode: "VARCHAR(%s)",
   bytearray: "VARBINARY(%s)"
}
```

The Python type serves another purpose. Using the example above, if *Sales.store* = 'LON1', pylytics will try and find a matching *Store* row via the following process:

- Is there a *Store NaturalKey* with the same type?
- In this case yes - *Store.store_shortcode* is a *basestring*, which is a parent type of *str*.
- Find the surrogate key for the *Store* row where *Store.store_shortcode* == 'LON1'.
- Replace 'LON1' with the surrogate key, as a foreign key to the Store table.

Column *Column* has no special interactions in pylytics. It just represents a database column, which describes the dimension.

As an example, for a date dimension, we can potentially have many columns, which provide lots of ways to filter the fact which references it:

```
class Date(Dimension):

    __source__ = NotImplemented

    iso_date = NaturalKey('iso_date', basestring)  # 2000-01-01
    day = Column('day', int)  # 1 - 31
    day_name = Column('day_name', basestring)  # Wednesday
    day_of_week = Column('day_of_week', int)  # 1 - 7
    month_name = Column('month_name', basestring)  # December
    month_number = Column('month_number', int)  # 1 - 12
    year = Column('year', int)  # 2000
    quarter = Column('year', int)  # 1 - 4
```

Adding columns is an important part of making the star schema useful for analysis.

Facts

Column Types

Metric *Metric* columns store numeric values.

A *Fact* doesn't have to include *Metric* columns. It can just be a collection of *DimensionKey* columns.

DimensionKey A DimensionKey is defined as follows:

```
class Sales(Fact):

    __source__ = NotImplemented

    store = DimensionKey('store', Store)
    ...
```

The first argument is the name of the column to be created. The second argument is a Dimension subclass.

Another argument which can be important is *optional*.

By default, dimension keys cannot be null. However, for some use cases, this is not possible.

An example is a user submitted questionnaire, where a lot of the fields have been missed out. In this case, we make the dimension key optional:

```
class UserQuestionnaire(Fact):

    __source__ = NotImplemented

    rating = DimensionKey('rating', UserRating, optional=True)
    ...
```

DegenerateDimension A *DegenerateDimension* stores the dimension value in the fact itself, rather than using a foreign key to a dimension table, as is the case with *DimensionKey*.

Use *DegenerateDimension* when the dimension doesn't warrant its own table - for example, in a sales fact there might be an order_id. Having this in a separate table doesn't save any space, and results in an unneccessary join.

> class Sales(Fact):
>
> > __source__ = NotImplemented
> >
> > order_id = DegenerateDimension('order_id', basestring)

4.1.5 Transform

Introduction

Once data has been extracted from a `DatabaseSource` or `CallableSource`, it can be cleaned and expanded upon by 'expansions'.

Conceptually, pylytics is a pipeline from extract to load.

Each row which comes from the source is passed through this pipeline.

The pipeline consists of a number of expansions, which transform the data. At the end of the pipeline, the data is ready to the loaded.

The expansions are simple, testable functions. For example:

```
# transform.py

def convert_datetime_to_date(data):
    """ We're only interested in the created date, and not the time.
    """
    data['created_date'] = data['created_datetime'].date()
    del data['created_datetime']
```

```
def convert_str_to_int(data):
    """ The source returns integers as strings - convert them.
    """
    data['sales'] = int(data['sales'])
```

The `data` argument is a dictionary representing a single row.

Adding expansions to facts and dimensions

For example:

```
# load.py

from transform import convert_datetime_to_date, convert_str_to_int


class Sales(Fact):

    __source__ = DatabaseSource.define(
        database="sales",
        query="SELECT * FROM sales_table",
        expansions=[convert_datetime_to_date, convert_str_to_int]
    )

    ...
```

Both DatabaseSource and CallableSource accept the `expansions` argument.

The expansions are processed in the order they appear in the list.

4.1.6 Running scripts

The manage.py file in the root of the project directory is used for building and updating the star schema.

You can specify the facts to run. For example:

```
./manage.py {update,build,test,historical} fact_1 [fact_2]
```

Or run the command for all facts:

```
./manage.py {update,build,test,historical} all
```

Commands

build

This will make sure that the relevant tables have been created for the facts specified, as well as any dimensions that the fact requires.

update

This command automatically calls *build* before executing. It updates your fact and dimension tables.

You can just make the scheduled facts run as follows:

```
./manage.py {update,build,test,historical} scheduled
```

A common way of running pylytics in production is to setup a CRON job which calls *manage.py update scheduled* every 10 minutes.

historical

Facts are usually built each day by running *update*. However, in some cases it's useful to be able to rebuild the tables (for example, if the project is just starting off, or data loss has occurred).

When the *update* command is run, it gets data from the *__source__* property of the Fact and Dimension classes.

With the *historical* command, it first looks for a *__historical_source__* property of the Fact and Dimension classes. If it exists then it is used instead of *__source__*. Here is an example:

```
class Sales(Fact):

    __source__ = DatabaseSource.define(
        database="sales",
        query="SELECT * FROM sales_table WHERE created > NOW() - INTERVAL 1 DAY"
    )

    __historical_source__ = DatabaseSource.define(
        database="sales",
        query="SELECT * FROM sales_table"
    )
```

Specifying the settings file location

When a new pylytics project is created using pylytics-admin.py, a settings.py file is automatically added to the project directory.

However, when several pylytics projects are on a single server it sometimes makes sense to have a single settings.py file in a shared location, e.g. in /etc/pylytics/settings.py.

In this case, use:

```
./manage.py --settings='/etc/pylytics' {update,build,test,historical} fact_1 [fact_2]
```

4.1.7 Scheduling

Rather than maintaining several CRONs to update facts at certain times, pylytics contains basic scheduling capabilities.

Defining schedules

Here is an example fact which uses scheduling:

```
from datetime import timedelta

from pylytics.library.fact import Fact
from pylytics.library.schedule import Schedule
from pylytics.library.column import Metric, DimensionKey
from dimension.store import Manager, Store
from dimension.date import Date
from dimension.time import Time


class Sales(Fact):

    __source__ = DatabaseSource.define(
        database="sales",
        query="SELECT * FROM sales_table"
    )

    __schedule__ = Schedule(repeats=timedelta(hours=1))

    date = DimensionKey('date', Date)
    time = DimensionKey('time', Time)
    store = DimensionKey('store', Store)
    manager = DimensionKey('manager', Manager)
    sales_amount = Metric('sales_amount', int)
```

It will update every hour.

There are three arguments you can pass into Schedule:

- repeats
- starts
- ends
- timezone

repeats

This is a timedelta objects which specifies how frequently the fact updates.

If *starts* is 3pm, and *ends* is 4pm, and *repeats* is 30 minutes, then the fact is scheduled to run at 3pm, 3.30pm and 4pm.

This schedule would look like:

```
__schedule__ = Schedule(repeats=timedelta(minutes=30), starts=time(hour=3),
                        ends=time(hour=4))
```

The smallest permissible *repeats* value is 10 minutes. It's unlikely any fact will need to be updated more frequently than this.

Default schedule

If no `Schedule` is defined, the fact will just be scheduled to run at midnight every day.

4.1.8 Source Types

Each fact and dimension has to specify a source for extracting data from.

DatabaseSource

DatabaseSource makes use of connections defined in *DATABASES* in *settings.py* to make queries to a MySQL database.

Declaring

Here are some examples of how a *DatabaseSource* can be defined:

```
# load.py

class Manager(Dimension):

    __source__ = DatabaseSource.define(
        database="sales",
        query="SELECT name AS manager FROM managers"
    )

    manager = NaturalKey('manager', basestring)
```

CallableSource

CallableSource is the most common source used. The source data is any callable Python object.

This callable could generate the data programatically, pull from an API, make a query to a database, parse a flat file etc.

The only constraint is that the callable must return the data in a certain format - either as a sequence of tuples, or a sequence of dictionaries.

For example:

```
# extract.py

def my_simple_datasource():
    return ({'size': 'large'}, {'size': 'medium'}, {'size': 'small'})

# Or alternatively:

def my_simple_datasource():
    return (('size', 'large'), ('size', 'medium'), ('size': 'small'))
```

Declaring

Here are some examples of how a *CallableSource* can be defined:

```
# load.py

from extract import my_simple_datasource


class StoreSize(Dimension):

    __source__ = CallableSource.define(
        _callable=staticmethod(my_simple_datasource)
        )

    size = NaturalKey('size', basestring)


# For very simple callables, you can specify then as lambdas:

class StoreOpenWeekends(Dimension):

    __source__ = CallableSource.define(
        _callable=staticmethod(
            lambda: [{'open_weekends': True}, {'open_weekends': False}]
            )
        )

    open_weekends = NaturalKey('open_weekends', bool)
```

4.1.9 Mutable Dimensions

If you're new to pylytics, skip this section.

Introduction

Dimensions change over time. To use our store example:

```
class Store(Dimension):

    __source__ = NotImplemented

    store_id = NaturalKey('store_id', int)
    store_shortcode = NaturalKey('store_shortcode', basestring)
    store_size = Column('store_size', basestring)
    employees = Column('employees', int)
```

Over time, the number of `employees` might change, and so might the `store_size` if it gets an extension.

Fact and dimension tables in pylytics are idempotent, meaning you could run *manage.py historical fact_1*, and if the rows already exist they'll be left alone.

However, if any of the dimension columns change, a new row will be inserted. For example, if a store extension happens to 'LON1':

Store				
store_id	store_shortcode	store_size	employees	applicable_from
1	'LON1'	'small'	100	2010-01-01 00:00:00
1	'LON1'	'medium'	100	2014-01-01 00:00:00

Notice the 'applicable_from' column.

The next time the `Store` fact updates, it will refer to the latest version of the dimension, but existing fact rows will still point to the dimension row that was relevant when they were created.

4.1.10 Visualisation Tools

Mondrian

pylytics can export an XML schema definition for Mondrian.

Mondrian is an open source MDX engine, which powers Pentaho, Jaspersoft, and Saiku. These are all Business Intelligence products, which can visualise the data stored in star schemas.

Before using these tools you need to define an XML schema, which tells Mondrian about the structure of your star schema.

To export the XML use the following command:

```
./manage.py template Fact_1
```

This will export an entire schema definition for that fact (including dimension definitions).

The XML should be double checked for accuracy, because pylytics can't completely second guess the end requirements, but it's still a big time saver.