Cassandra Dataset Manager Documentation

Release 1.0

Jon Haddad

April 29, 2016

Contents

Cassandra Dataset Manager, (cdm) is a tool to make it simple to start learning Apache Cassandra or Datastax Enterprise (DSE). This utility will provide a framework for building and installing datasets, which can then be explored via cqlsh, DevCenter, and the Jupyter notebooks that are included with datasets. In short, the focus of this tool is on the following:

Development of Datasets The CDM framework will provide a consistent experience for people interested in sharing public datasets.

Installation of Datasets It should take 15 minutes or less for a user to go from "I want to learn" to "I'm looking at data". The experience of loading sample data should be as simple as possible, with helpful error messages when things do go wrong or a requirement it not installed.

Visual Learning via Examples Several tools, such as Jupyter and Zeppelin notebooks, provide an elegant means of teaching database and data model concepts by interweaving explanatory text and executable code. By providing stable datasets through CDM people can create tutorials, blog posts, videos, and code examples without having to worry about creating new data models every time.

CHAPTER 1

Things this is not

1. A bulk loader
2. A way for you to manage your schema in your projects

Want to get up and running? See the QuickStart

CHAPTER 2

User Documentation

2.1 QuickStart

Make sure you have either open source Cassandra (2.1 or later) or DataStax Enterprise (4.8 or greater) installed.

Getting up and running with CDM is simple. In a virtualenv, run the following:

```
pip install cassandra-dataset-manager
```

You'll have a command line utility, `cdm`, installed in your virtualenv's bin directory. Update your local dataset list, then install the movielens-small dataset:

```
cdm update
cdm install movielens-small
```

You now have the *movielens-small* dataset installed in your local cassandra cluster.

Next, type *cqlsh* to start working with the Cassandra shell.

Once cqlsh starts, type *use movielens_small* then *desc tables* to see all the tables in the schema. Type the following to read some data:

2.2 Usage

Installing the cdm package will set up a `cdm` executable.

2.2.1 Installing Datasets

`cdm list` will provide a list of installable datasets and their descriptions.

`cdm install <dataset>` will install a specific dataset. Future versions will include support for DSE Search and DSE Graph.

2.2.2 Updating the local database

`cdm update` will update the local database.

2.3 Frequently Asked Questions

CHAPTER 3

Developer Documentation

3.1 Guide: Creating Datasets

This information is relevant only to developers wishing to create their own datasets for distribution.

3.1.1 What is a Dataset?

Think of a Dataset similar to a package managed by yum or apt. Instead of binaries and configuration files, installing a Dataset gives you a Cassandra schema, sample data, and a Jupyter notebook with tutorials on how to use that data.

3.1.2 Create a new project from the skeleton

Make sure CDM is installed. You will not be able to provide additional Python modules other than what CDM already provides (yet).

Create a new dataset with the `cdm new` command. It will generate a project skeleton for you. For example:

```
cdm new example-name
```

Installers are created by having a file called `install.py` in the top level of your dataset. The installer must subclass `cdm.installer.Installer`. The cdm utility will discover the Installer automatically so the name is somewhat arbitrary, however it should reflect the dataset's name as a convention.

3.1.3 Download resources and setup

Set up your `post_init()` hook. You should download and load any data into memory you'll need for all the various imports:

```
class MovieLensInstaller(Installer):
    def post_init(self):
        context = self.context
```

If you need to download any data (like a zip file of CSVs, etc), you can use `context.download(url)` which will download and cache the file at the URL return a file pointer. Caching is provided automatically.

If you download a zip file, the easiest way to access the data is using the built in Python ZipFile module:

```
fp = context.download("http://files.grouplens.org/datasets/movielens/ml-100k.zip")
zf = ZipFile(file=fp)
fp = zf.open("ml-100k/u.item")
```

You can use the file pointers returned from `ZipFile.open(name)` as normal pointers. If you're working with CSV data, it's recommended to use the Pandas library (provided by CDM):

```
movies = read_csv(fp, sep="|", header=None, index_col=0, names=["id", "name", "genre"]).fillna(0)
```

If you'd like to include your data with your dataset (a good idea of the dataset is small), you

You can see how it's pretty easy to use the `Context` to download and cache external files, then process and prepare using Pandas.

3.1.4 Set up Cassandra Schema

Next you'll want to set up a schema for Cassandra. There's a few options varying in complexity. Read up on the different options for configuring your Cassandra Schema.

3.1.5 Load Cassandra Data

Assuming you've loading some data into memory in the `post_init()`, you can now load data into your schema.

To load data, you'll want to use the `session` provided by the `Context`:

```
class MyInstaller(Installer):
    def install_cassandra(self):
        context = self.context
        session = context.session
        prepared = session.prepare("INSERT INTO data (key, value) VALUES (?, ?)")
        for row in self.data:
            session.execute(prepared, row.key, row.value)
```

3.1.6 Provided Libraries

Cassandra Driver The project would be useless without a driver, so it's included. We will stay reasonably up to date with current packages. It is always made available via the `/context` as the `session` variable.

Pandas Pandas is an excellent library for reading various raw formats such as CSV. It also provides facilities for data manipulation, which may be required to transform data.

Faker Faker makes for each generation of fake data. This is especially useful when you're dealing with an incomplete data model or one that has been anonymized.

3.1.7 Testing

Testing datasets is important. This project is leveraging features of py.test that make it easy to test datasets.

CDM will include a tool for testing a project. This runs all the projects unit tests as well as tests that verify project structure and conventions:

```
cdm test
```

All tests must pass `cdm test` for inclusion in the official Dataset repository.

3.2 Cassandra Schema

Working with a Cassandra schema is very flexible using CDM. There are several options available.

3.2.1 Using a schema file

This is useful if you have a schema somewhere already that you want to write to disk through cqlsh, and you don't wish to use CQLEngine models.

To easily use a schema file, make sure your `installer` subclasses `SimpleCQLSchema` *first*:

```
class MyInstaller(SimpleCQLSchema, Installer):
    pass
```

Put your schema in schema.cql, and it will automatically be picked up and loaded, splitting the statements on `;`.

3.2.2 CQLEngine Models

This is a convenient as you'll frequently want to leverage CQLEngine models for validating and inserting data. We'll use the `cassandra_schema()` hook to return the classes we want sync'ed to the database.

For example, in movielens-small, we define our `Movie` Model similar to this:

```
class Movie(Model):
    __table_name__ = 'movies'
    id = Integer(primary_key=True)
    name = Text()
    release_date = Date()
    video_release_date = Date()
    url = Text()
    avg_rating = Float()
    genres = Set(Text)
```

In our installer, we return a list of table models:

```
class MovieLensInstaller(Installer):
    def cassandra_schema(self):
        return [Movie]
```

3.2.3 Specifying a Schema Inline

This will be necessary for UDAs/UDFs as they aren't simply split on `;`. A future version of CDM may include a parser to properly support this but it's unlikely anytime soon. Until that day comes, it's possible to use fat strings to specify schema:

```
class MovieLensInstaller(Installer):
    def cassandra_schema(self):
        statements = ["""CREATE TABLE movies
                         (id uuid primary key,
                          name text)""",
                      """CREATE CUSTOM INDEX on movies(name)
                         USING 'org.apache.cassandra.index.sasi.SASIIndex'"""
        return statements
```

3.2.4 Mixed Mode

There are cases which are not handled with CQLEngine yet. Materialized views, SASI indexes, UDFs, UDAs are all difficult to express. Python allows us a lot of flexibility by allowing lists to contain objects of mixed types. We can leverage our CQLEngine models for our tables and provide fat strings for the rest of the schema:

```
class MovieLensInstaller(Installer):
    def cassandra_schema(self):
        statements = [Movie,
                      """CREATE CUSTOM INDEX on movies(name)
                         USING 'org.apache.cassandra.index.sasi.SASIIndex'"""]
```

This is cool because we can leverage CQLEngine for our database models but still get the flexibility of using any CQL that it doesn't support yet.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search