django-rdkit Documentation

Release 0.0.5

Riccardo Vianello

Aug 16, 2018

Contents

Contents:

CHAPTER 1

About django-rdkit

CHAPTER 2

Tutorial

This tutorial will try to reproduce the operations described in the RDKit PostgreSQL cartridge documentation, but within the context of a django project.

Some familiarity with django and the django database api is assumed (excellent documentation about these is available from the django web site).

PostgreSQL and the RDKit cartridge should be installed and running on the system. A database should be created with appropriate access privileges to be used by the tutorial project. Minimally, this requires running the following command:

```
$ createdb django_rdkit_tutorial
```

2.1 Creation of the tutorial project

Create a new skeleton django project named `tutorial_project`:

```
$ django-admin startproject tutorial_project
```

Change working directory to the `tutorial_project` directory (where the `manage.py` file is located) and open the `tutorial_project/settings.py` module with your favourite text editor.

Replace the default database settings with those appropriate to the created PostgreSQL database:

```
DATABASES={
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'django_rdkit_tutorial',
        'USER': '',
        'PASSWORD': '',
        'HOST': '',
        'PORT': '',
    }
}
```

And extend the INSTALLED_APPS list to include the django_rdkit application:

```
INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'django_rdkit',
)
```

Finally, initialize the database:

```
$ python manage.py migrate
```

The migrate command above configures the database for the installed applications. The inclusion of django_rdkit in the INSTALLED_APP is not strictly required, but allows integrating the creation of the RDKit extension with the management of the django project, as evidenced by using sqlmigrate:

```
$ python manage.py sqlmigrate django_rdkit 0001
BEGIN;
CREATE EXTENSION IF NOT EXISTS rdkit;

COMMIT;
```

The correct configuration of the project may be quickly verified at this stage by running a direct SQL query using the database connection that is created by django:

```
$ python manage.py shell
[...]
In [1]: from django.db import connection

In [2]: with connection.cursor() as cursor:
   ...:     cursor.execute("SELECT mol_amw('C')")
   ...:     print(cursor.fetchone()[0])
   ...:
16.043
```

2.2 Creation of a django application

The additional functionalities developed in the context of this tutorial will be contained in a so-called django application. We'll call this application tutorial_application:

```
$ python manage.py startapp tutorial_application
```

The list of INSTALLED_APPS in the tutorial_project/settings.py module must be extended to include the new application:

```
INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
```

(continues on next page)

(continued from previous page)

```
    'django.contrib.staticfiles',
    'django_rdkit',
    'tutorial_application',
)
```

We'll use this application to manage a collection of compound structures. In order to do so, edit the `tutorial_applications/models.py` module so that it looks like the following:

```
from django_rdkit import models

class Compound(models.Model):

    name = models.CharField(max_length=256)
    molecule = models.MolField()
```

Please note that we import `models` from the `django_rdkit` package, instead of from `django.db` as we would usually do. This makes the `MolField` and the other functionalities that are specific the RDKit cartridge available, together with the rest of the usual fields and functions that are usually availble from `django.db`.

In order to extend the schema of the PostgreSQL database to include this model, we now need to create and apply a corresponding migration:

```
$ python manage.py makemigrations tutorial_application
Migrations for 'tutorial_application':
  0001_initial.py:
    - Create model Compound
$ python manage.py migrate tutorial_application
Operations to perform:
  Apply all migrations: tutorial_application
Running migrations:
  Rendering model states... DONE
  Applying tutorial_application.0001_initial... OK
```

We can immediately try adding data to this model using again the python shell:

```
$ python manage.py shell
[...]
In [1]: from tutorial_application.models import Compound

In [2]: Compound.objects.create(name='benzene', molecule='c1ccccc1')
Out[2]: <Compound: Compound object>

In [3]: from django_rdkit.models import *

In [4]: for compound in Compound.objects.annotate(amw=AMW('molecule')):
   ...:     print(compound.name, compound.amw)
   ...:
benzene 78.114
```

We can now delete this sample compound, more data will be imported in the next section of this tutorial:

```
In [5]: Compound.objects.all().delete()
```

2.3 Structures import and substructure queries

To display the use of structure searches we'll use a copy of the ChEMBL data. Download a copy of the `chembl_20_chemreps.txt` which is available from here and place it into a suitable directory.

The initial import may therefore be performed with code similar to the following:

```
$ python manage.py shell
[...]
In [1]: path = '../../chembl/chembl_20_chemreps.txt'

In [2]: from rdkit import Chem

In [3]: def chembl(path, limit=None):
   ...:     count = 0
   ...:     with open(path, 'rt') as f:
   ...:         next(f) # skip header
   ...:         for line in f:
   ...:             name, smiles = line.split()[:2]
   ...:             molecule = Chem.MolFromSmiles(smiles)
   ...:             if molecule:
   ...:                 yield name, molecule
   ...:                 count += 1
   ...:             if limit and count == limit:
   ...:                 break
   ...:

In [4]: from tutorial_application.models import Compound

In [5]: for name, molecule in chembl(path, limit=None):
   ...:     smiles = Chem.MolToSmiles(molecule)
   ...:     test_molecule = Chem.MolFromSmiles(smiles)
   ...:     if not test_molecule:
   ...:         print('smiles-mol-smiles roundtrip issue:', name)
   ...:     else:
   ...:         Compound.objects.create(name=name, molecule=molecule)
   ...:
```

The import loop may take some time, consider using the `limit` parameter to shorten the duration of this step. Once the import has completed one can easily verify the number of available compounds:

```
In [8]: Compound.objects.count()
Out[8]: 1455712
```

In order to efficiently perform structural queries on the imported compounds, a database index must be created. This operation can be implemented with a database migration. Execute the following command to create an empty skeleton for this migration:

```
$ python manage.py makemigrations --empty --name create_compound_molecule_index␣
↪tutorial_application
Migrations for 'tutorial_application':
  0002_create_compound_molecule_index.py:
```

Now open the file `tutorial_application\migrations\0002_create_compound_molecule_index.py` with a text editor and edit a couple of lines in order to import the `GiSTIndex` operation and apply it. The resulting migration module should look similar to the following:

```
from django.db import models, migrations
from django_rdkit.operations import GiSTIndex

class Migration(migrations.Migration):

    dependencies = [
        ('tutorial_application', '0001_initial'),
    ]

    operations = [
        GiSTIndex('Compound', 'molecule')
    ]
```

When done, save your changes and run the migration (depending on the number of structures imported into the model, the indexing may take quite some time to complete):

```
$ python manage.py migrate tutorial_application
Operations to perform:
  Apply all migrations: tutorial_application
Running migrations:
  Rendering model states... DONE
  Applying tutorial_application.0002_create_compound_molecule_index...
```

Finally, following the original tutorial, we can now perform a few example substructure queries:

```
In [1]: from django_rdkit.models import *

In [2]: from tutorial_application.models import *

In [3]: def smiles_substructure_query(substructure):
   ....:     query = Compound.objects.filter(molecule__hassubstruct=substructure)
   ....:     for cmpd in query.annotate(smiles=MOL_TO_SMILES('molecule'))[:5]:
   ....:         print(cmpd.name, cmpd.smiles)
   ....:
```

The above code uses the `hassubstruct` lookup operator, which is specific to the `MolField` field, and also uses the `MOL_TO_SMILES` database function to convert the selected molecules and annotate the model instance with a smiles string. Both functionalities are provided by the RDKit cartridge.

```
In [4]: smiles_substructure_query('c1cccc2c1nncc2')
CHEMBL113970 CCCCn1c(=O)c2cc(OC)c(OC)cc2c2nnc3cc4c(cc3c21)OCO4
CHEMBL113470 COc1cc2c(cc1OC)c1nnc3cc4c(cc3c1n(C(C)CN(C)C)c2=O)OCO4
CHEMBL12112 CC(C)Sc1ccc(CC2CCN(C3CCN(C(=O)c4cnnc5ccccc54)CC3)CC2)cc1
CHEMBL71086 COc1cc2c(cc1OC)c1nnc3cc4c(cc3c1n(CCN(C)C)c2=O)OCO4
CHEMBL89981 c1ccc(CN2CCC(CCNc3cc4ccc5ccccc5c4nn3)CC2)cc1

In [5]: smiles_substructure_query('c1ccnc2c1nccn2')
CHEMBL110168 CCOC(=O)Nc1cc(NC(C)CCCN(CC)CC)c2nc(-c3ccccc3)c(-c3ccccc3)nc2n1
CHEMBL50456 Clc1ccc(CN2CCN(c3nc4cccnc4n4cccc34)CC2)c(Cl)c1
CHEMBL107535 O=c1c2cccn2c2ncccc2n1CCNC(=S)Nc1ccc(Br)cn1
CHEMBL51225 c1cc2c(N3CCN(c4ccccc4)CC3)nc3cccnc3n2c1
CHEMBL54246 Cc1ccnc2c1nc(N1CCN(Cc3ccccc3)CC1)c1cccn12
```

2.3.1 SMARTS-based queries

Similarly, substructure queries can use a SMARTS string as argument:

```
In [20]: def smarts_substructure_query(substructure):
   ....:     query = Compound.objects.filter(molecule__
↪hassubstruct=QMOL(Value(substructure)))
   ....:     for cmpd in query.annotate(smiles=MOL_TO_SMILES('molecule'))[:5]:
   ....:         print(cmpd.name, cmpd.smiles)
   ....:
```

The lookup api expects a SMILES string by default, so a query molecule must be created explicitly, using the `QMOL` constructor, which is exposed as a database function. Please note that database functions execute on the backend, and by default assume their argument to resolve to a database column. Since a literal SMARTS string is used, it must be wrapped inside a call to `Value()` (the query expression api was introduced in django 1.8, for further details about this see the official documentation.

```
In [21]: smarts_substructure_query('c1[o,s]ncn1')
CHEMBL52013 C[C@@H](NC(=O)c1nsc(-c2ccc(Cl)cc2)n1)[C@](O)(Cn1cncn1)c1ccc(F)cc1F
CHEMBL48759 CCN(CC)C(=O)N1Cc2c(-c3noc(C4CC4)n3)ncn2-c2ccccc21
CHEMBL48839 CCSC(=O)N1Cc2c(-c3noc(C4CC4)n3)ncn2-c2ccccc21
CHEMBL105111 COc1ccc(-c2noc(CN3C(=O)c4ccccc4C3=O)n2)cc1
CHEMBL105112 Cc1ccccc1-c1noc(CN2C(=O)c3ccccc3C2=O)n1
```

2.3.2 Using stereochemistry

By default stereochemistry is not taken into account when performing substructure queries:

```
In [42]: smiles_substructure_query('NC(=O)[C@H]1CCCN1C=O')
CHEMBL118176␣
↪CC(C)[C@@H](NC(=O)COc1ccc(OCC(=O)O)cc1)C(=O)N1CCC[C@H]1C(=O)N[C@H](C(=O)c1nc2ccccc2o1)C(C)C
CHEMBL117981 O=C(CCCc1ccccc1)N1CCC[C@H]1C(=O)N1CCC[C@H]1C(=O)c1ccccn1
CHEMBL117920 O=C(CCCc1ccccc1)N1CCC[C@H]1C(=O)N1CCC[C@H]1C(=O)c1ccccnc1
CHEMBL117024␣
↪Cc1ccc(C[C@H](NC(=O)c2ccc(C)c(O)c2C)[C@H](O)C(=O)N2C[C@@H](Cl)C[C@H]2C(=O)NC(C)(C)C)cc1
CHEMBL117088␣
↪Cc1cc(O)c(C)c(C(=O)N[C@@H](Cc2cccc(C(F)(F)F)c2)[C@H](O)C(=O)N2C[C@@H](Cl)C[C@H]2C(=O)NC(C)(C)C)c1
```

As described in the RDKit documentation, the cartridge defines a set of configuration parameters that allow controlling this and other aspects. These parameters are exposed as attributes of a `config` object:

```
In [43]: from django_rdkit.config import config
```

In particular, the effect of stereochemistry on the results returned by substructure searches is changed using the `do_chiral_sss` configuration variable:

```
In [44]: config.do_chiral_sss = True

In [45]: smiles_substructure_query('NC(=O)[C@H]1CCCN1C=O')
CHEMBL100712␣
↪N=C(N)NCCC[C@H]1NC(=O)[C@H]2CCCN2C(=O)[C@H](Cc2ccccc2)NC(=O)CCCCCCCCCCNC(=O)C1=O
CHEMBL98474 Cc1ccccc1S(=O)(=O)NC(=O)N1CCC[C@@H]1C(=O)NCCC(=O)NC(Cc1c[nH]cn1)C(=O)O
CHEMBL2369135␣
↪CC[C@H](C)[C@@H]1NC(=O)[C@@H]([C@H](C)c2c(C)cc(OC)cc2C)NC(=O)[C@H](N)C(C)(C)SSC[C@@H]2NC(=O)[C@@H]
CHEMBL2369136␣
↪CC[C@H](C)[C@@H]1NC(=O)[C@H]([C@@H](C)c2c(C)cc(OC)cc2C)NC(=O)[C@H](N)C(C)(C)SSC[C@@H]2NC(=O)[C@@H]
CHEMBL98856␣
↪N=C(N)NCCC[C@H]1NC(=O)[C@H]2CCCN2C(=O)[C@H](Cc2ccccc2)NC(=O)CCCCCNC(=O)C1=O
```

2.4 Similarity queries

Open the file `tutorial_application/models.py` for editing again, and extend the `Compound` model with some fingerprint fields, as displayed below:

```
from django_rdkit import models

class Compound(models.Model):

    name = models.CharField(max_length=256)
    molecule = models.MolField()

    torsionbv = models.BfpField(null=True)
    mfp2 = models.BfpField(null=True)
    ffp2 = models.BfpField(null=True)
```

(please note that the new fields are defined as nullable so that we can alter the existing database table adding initially empty columns).

Create a corresponding schema migration:

```
$ python manage.py makemigrations tutorial_application --name add_compound_
↪fingerprint_fields
Migrations for 'tutorial_application':
  0003_add_compound_fingerprint_fields.py:
    - Add field ffp2 to compound
    - Add field mfp2 to compound
    - Add field torsionbv to compound
```

And finally, apply it to the current schema:

```
$ python manage.py migrate tutorial_application
Operations to perform:
  Apply all migrations: tutorial_application
Running migrations:
  Rendering model states... DONE
  Applying tutorial_application.0003_add_compound_fingerprint_fields...
```

The fingerpring columns may be filled with data that is computed with an update query:

```
$ python manage.py shell
[...]
In [1]: from django_rdkit.models import *

In [2]: from tutorial_application.models import Compound

In [3]: Compound.objects.update(
   ...: torsionbv=TORSIONBV_FP('molecule'),
   ...: mfp2=MORGANBV_FP('molecule'),
   ...: ffp2=FEATMORGANBV_FP('molecule'),
   ...: )
Out[3]: 1455712
```

Once this query has completed, an index must still be added on the column (or columns) that will be frequently used to perform similarity queries. This database administration step may be again integrated into the management of the django project by means of a custom migration. First create an empty migration:

```
$ python manage.py makemigrations --empty --name create_compound_mfp2_index tutorial_
↪application
Migrations for 'tutorial_application':
  0004_create_compound_mfp2_index.py:
```

Edit the file `tutorial_application/migrations/0004_create_compound_mfp2_index.py` to add the creation of a GiST index on the `mfp2` column:

```
from django.db import models, migrations
from django_rdkit.operations import GiSTIndex


class Migration(migrations.Migration):

    dependencies = [
        ('tutorial_application', '0003_add_compound_fingerprint_fields'),
    ]

    operations = [
        GiSTIndex('Compound', 'mfp2')
    ]
```

And then run the migration to complete the preparation of the database:

```
$ python manage.py migrate tutorial_application
Operations to perform:
  Apply all migrations: tutorial_application
Running migrations:
  Rendering model states... DONE
  Applying tutorial_application.0004_create_compound_mfp2_index...
```

The following demonstrate a basic similarity search:

```
In [1]: from django_rdkit.models import *

In [2]: from tutorial_application.models import *

In [3]: smiles = 'Cc1ccc2nc(-c3ccc(NC(C4N(C(c5cccs5)=O)CCC4)=O)cc3)sc2c1'

In [4]: value = MORGANBV_FP(Value(smiles))

In [5]: Compound.objects.filter(mfp2__tanimoto=value).count()Out[6]: 67
```

Following the original tutorial from the RDKit documentation, the next step consists in implementing a query to return the sorted list of neighbors along with the accompanying SMILES:

```
In [8]: def get_mfp2_neighbors(smiles):
   ...:     value = MORGANBV_FP(Value(smiles))
   ...:     queryset = Compound.objects.filter(mfp2__tanimoto=value)
   ...:     queryset = queryset.annotate(smiles=MOL_TO_SMILES('molecule'))
   ...:     queryset = queryset.annotate(sml=TANIMOTO_SML('mfp2', value))
   ...:     queryset = queryset.order_by(TANIMOTO_DIST('mfp2', value))
   ...:     queryset = queryset.values_list('name', 'smiles', 'sml')
   ...:     return queryset
   ...:
```

The function wraps a non-trivial database api expression, but the generated SQL query can be easily displayed for a sample queryset:

```
In [22]: qs = get_mfp2_neighbors('c1ccccc1')

In [23]: print(qs.query)
SELECT "tutorial_application_compound"."name",
mol_to_smiles("tutorial_application_compound"."molecule") AS "smiles",
tanimoto_sml("tutorial_application_compound"."mfp2", morganbv_fp(c1ccccc1)) AS "sml"
FROM "tutorial_application_compound" WHERE
"tutorial_application_compound"."mfp2" % (morganbv_fp(c1ccccc1)) ORDER BY
("tutorial_application_compound"."mfp2" <%> morganbv_fp(c1ccccc1)) ASC
```

You can use the `get_mfp2_neighbors` function to perform some sample queries:

```
In [9]: for name, smiles, sml in get_mfp2_neighbors('Cc1ccc2nc(-
↪c3ccc(NC(C4N(C(c5cccs5)=O)CCC4)=O)cc3)sc2c1')[:10]:
    print(name, smiles, sml)
   ...:
CHEMBL467428 Cc1ccc2nc(-c3ccc(NC(=O)C4CCN(C(=O)c5cccs5)CC4)cc3)sc2c1 0.772727272727273
CHEMBL461435 Cc1ccc2nc(-c3ccc(NC(=O)C4CCCN(S(=O)(=O)c5cccs5)C4)cc3)sc2c1 0.
↪657534246575342
CHEMBL460340 Cc1ccc2nc(-c3ccc(NC(=O)C4CCN(S(=O)(=O)c5cccs5)CC4)cc3)sc2c1 0.
↪647887323943662
CHEMBL460588 Cc1ccc2nc(-c3ccc(NC(=O)C4CCN(S(=O)(=O)c5cccs5)C4)cc3)sc2c1 0.
↪638888888888889
CHEMBL1608585 O=C(Nc1nc2ccc(Cl)cc2s1)[C@@H]1CCCN1C(=O)c1cccs1 0.623188405797101
CHEMBL1327784 COc1ccc2nc(NC(=O)[C@@H]3CCCN3C(=O)c3cccs3)sc2c1 0.619718309859155
CHEMBL518028 Cc1ccc2nc(-c3ccc(NC(=O)C4CN(S(=O)(=O)c5cccs5)C4)cc3)sc2c1 0.
↪611111111111111
CHEMBL1316870 Cc1ccc(NC(=O)C2CCCN2C(=O)c2cccs2)cc1C 0.606060606060606
CHEMBL1309021 O=C(Nc1ccc(S(=O)(=O)N2CCCC2)cc1)C1CCCN1C(=O)c1cccs1 0.602941176470588
CHEMBL1706764 Cc1ccc(NC(=O)C2CCCN2C(=O)c2cccs2)c(C)c1 0.597014925373134

In [10]: for name, smiles, sml in get_mfp2_neighbors('Cc1ccc2nc(N(C)CC(=O)O)sc2c1
↪')[:10]:
   ....:     print(name, smiles, sml)
   ....:
CHEMBL394654 Cc1ccc2nc(N(C)CCN(C)c3nc4ccc(C)cc4s3)sc2c1 0.692307692307692
CHEMBL491074 CN(CC(=O)O)c1nc2cc([N+](=O)[O-])ccc2s1 0.583333333333333
CHEMBL1617304 CC(=O)N(CCCN(C)C)c1nc2ccc(C)cc2s1 0.571428571428571
CHEMBL1350062 CC(=O)N(CCCN(C)C)c1nc2ccc(C)cc2s1.Cl 0.549019607843137
CHEMBL1621941 Cc1ccc2nc(N(CCN(C)C)C(=O)c3cc(Cl)sc3Cl)sc2c1 0.518518518518518
CHEMBL1626442 Cc1ccc2nc(N(CCCN(C)C)C(=O)CS(=O)(=O)c3ccccc3)sc2c1 0.517857142857143
CHEMBL1617545 Cc1ccc2nc(N(CCCN(C)C)C(=O)CCc3ccccc3)sc2c1 0.517857142857143
CHEMBL406760 Cc1ccc2nc(NC(=O)CCC(=O)O)sc2c1 0.510204081632653
CHEMBL1624740 Cc1ccc(S(=O)(=O)CC(=O)N(CCCN(C)C)c2nc3ccc(C)cc3s2)cc1 0.509090909090909
CHEMBL1620007 Cc1ccc2nc(N(CCN(C)C)C(=O)c3ccc4ccccc4c3)sc2c1 0.509090909090909
```

2.4.1 Adjusting the similarity cutoff

```
In [11]: print(get_mfp2_neighbors('Cc1ccc2nc(N(C)CC(=O)O)sc2c1').count())
18

In [12]: from django_rdkit.config import config

In [13]: config.tanimoto_threshold = 0.7
```

(continues on next page)

(continued from previous page)

```
In [14]: print(get_mfp2_neighbors('Cc1ccc2nc(N(C)CC(=O)O)sc2c1').count())
0

In [15]: config.tanimoto_threshold = 0.6

In [16]: print(get_mfp2_neighbors('Cc1ccc2nc(N(C)CC(=O)O)sc2c1').count())
1

In [17]: config.tanimoto_threshold = 0.5

In [18]: print(get_mfp2_neighbors('Cc1ccc2nc(N(C)CC(=O)O)sc2c1').count())
18
```

CHAPTER 3

Database setup

The RDKit extension for PostgreSQL must be compiled and installed.

Conda packages for the main RDKit releases, including the postgres cartridge for the linux platform, may be found from the RDKit binstar channel or built using the recipes available from the conda-rdkit repository.

Please refer to the RDKit documentation for general instructions regarding building the database cartridge from a source code distribution.

The details of the PostgreSQL database creation and configuration may vary depending on the deployment strategy and the application-specific needs, but no additional requirements exist for using the RDKit PostgreSQL cartridge in a django project. For general advice please refer to the official PostgreSQL and django documentation.

CHAPTER 4

Project configuration and management

Django projects integrating the functionalities provided by `django_rdkit` should configure their settings to use the PostgreSQL database backend:

```
DATABASES={
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        # [...],
    }
}
```

Two additional operations are then to be performed on the database in order to use the RDKit cartridge.

Firstly, it is necessary to have the cartidge installed in the configured database. This operation corresponds to executing the following SQL statement:

```
CREATE EXTENSION IF NOT EXISTS rdkit;
```

One simple way to integrate this operation within a django project consists in installing the `django_rdkit` package as a django application:

```
INSTALLED_APPS = (
    # [...]
    'django_rdkit',
)
```

A migration will be this way automatically included in the database configuration, ensuring that that the RDKit extension is created (please note that creating an extension requires database superuser privileges).

Moreover, efficient execution of structure and similarity searches requires the creation of an additional GiST index:

```
CREATE INDEX index_name ON table_name USING GIST (column_name);
```

The creation of this custom index is also supported with a migration operation. Users should include this operation in the implementation of a custom migration for the fields that may require it.

4.1 Migration Operations

class django_rdkit.operations.**RDKitExtension**

An `Operation` subclass that will install the RDKit cartridge (install `django_rdkit` as a django application to include a migration that wraps this operation).

class django_rdkit.operations.**GiSTIndex**(*model_name*, *name*, *index_name=None*)

An `Operation` subclass that wraps the management of a GiST index for field `name` of model `model_name`. The optional `index_name` parameter allows customizing the name of the created index.

CHAPTER 5

Model field reference

A few custom model fields are provided, mapping to the chemical data types defined by the RDKit cartridge.

When fetched form the database, the python type of the model data attributes will match the corresponding RDKit class types, but for some fields additional data types may be used in assignment. For example, the value attributed to a `MolField` will be a `Mol` instance, and the value attributed to a `ChemicalReaction` will be a `ChemicalReaction` instance, but the values for these fields may be also assigned using a SMILES representation.

In trasferring data to and from the database, a binary representation is used for the fields supporting it.

5.1 Field types

5.1.1 `MolField`

class django_rdkit.models.**MolField**(***options*)

A field representing an RDKit molecule. It may be assigned using a `Mol` instance or with a SMILES string. Molecules values can be also created using one of the database functions implemented by the RDKit cartridge.

5.1.2 `RxnField`

class django_rdkit.models.**RxnField**(***options*)

A field storing a `ChemicalReaction` instance. It is assigned and returned from the database as a SMILES reactions string.

5.1.3 `BfpField`

class django_rdkit.models.**BfpField**(***options*)

A bit vector fingerprint. It may be assigned using an `ExplicitBitVect` instance or with an update query using one of the implemented fingerprint functions.

5.1.4 SfpField

class django_rdkit.models.**SfpField**(***options*)

A sparse count vector fingerprint (SparseIntVect). Direct assignment of the SfpField field on the client side is not supported, the most practical way to assign values to the mapped table column is using an update query.

CHAPTER 6

Field Lookups

6.1 MolField

6.1.1 Lookup operators

- `hassubstruct`
- `issubstruct`
- `exact`

6.1.2 Descriptor transforms

Most of the molecular descriptor functions defined by the cartridge are also available as transform operators. To ease the mnemonics, the name of these operators is based on the original function name, deprived of the `mol_` prefix (`mol_hba` becomes `hba`) and following the usual django conventions all names are lowercase. For example:

```
# count all compounds with AMW above a provided threshold value
CompoundModel.objects.filter(molecule__amw__gt=threshold).count()
```

- `hba`
- `hbd`
- `numatoms`
- `numheavyatoms`
- `numrotatablebonds`
- `numheteroatoms`
- `numrings`
- `numaromaticrings`

- numaliphaticrings
- numsaturatedrings
- numaromaticheterocycles
- numaliphaticheterocycles
- numsaturatedheterocycles
- numaromaticcarbocycles
- numaliphaticcarbocycles
- numsaturatedcarbocycles
- amw
- logp
- tpsa
- fractioncsp3
- chi0v
- chi1v
- chi2v
- chi3v
- chi4v
- chi0n
- chi1n
- chi2n
- chi3n
- chi4n
- kappa1
- kappa2
- kappa3
- murckoscaffold

6.2 RxnField

6.2.1 Lookup operators

- hassubstruct
- hassubstructfp
- issubstruct
- issubstructfp

6.2.2 Descriptor transforms

- `numreactants`
- `numproducts`
- `numagents`

6.3 BfpField and SfpField

6.3.1 Lookup operators

- `tanimoto`
- `dice`

CHAPTER 7

Database functions

Most of the database functions implemented by the cartridge are also exposed through the django api and may be used to convert and create values, or in annotation and aggregate expressions:

```
result = MoleculeModel.objects.aggregate(avg_amw=Avg(AMW('molecule')))
```

For consistency, all function names are defined as uppercase (this is not probably the prettiest solution, but it's easy to remember and unlikely to produce name clashes).

7.1 Functions

- HBA
- HBD
- NUMATOMS
- NUMHEAVYATOMS
- NUMROTATABLEBONDS
- NUMHETEROATOMS
- NUMRINGS
- NUMAROMATICRINGS
- NUMALIPHATICRINGS
- NUMSATURATEDRINGS
- NUMAROMATICHETEROCYCLES
- NUMAROMATICCARBOCYCLES
- NUMALIPHATICCARBOCYCLES
- NUMSATURATEDCARBOCYCLES

- AMW
- LOGP
- TPSA
- FRACTIONCSP3
- CHI0V
- CHI1V
- CHI2V
- CHI3V
- CHI4V
- CHI0N
- CHI1N
- CHI2N
- CHI3N
- CHI4N
- KAPPA1
- KAPPA2
- KAPPA3
- MURCKOSCAFFOLD
- MOL
- MOL_FROM_SMILES
- MOL_FROM_SMARTS
- MOL_FROM_CTAB
- QMOL
- QMOL_FROM_SMILES
- QMOL_FROM_SMARTS
- MOL_TO_SMILES
- MOL_TO_SMARTS
- MOL_TO_CTAB
- MOL_INCHI
- MOL_INCHIKEY
- MOL_FORMULA
- IS_VALID_SMILES
- IS_VALID_SMARTS
- IS_VALID_CTAB
- NUMREACTANTS
- NUMPRODUCTS

- NUMAGENTS
- REACTION
- REACTION_FROM_SMILES
- REACTION_FROM_SMARTS
- REACTION_FROM_CTAB
- REACTION_TO_SMILES
- REACTION_TO_SMARTS
- REACTION_TO_CTAB
- REACTION_DIFFERENCE_FP
- REACTION_STRUCTURAL_BFP
- MORGAN_FP
- MORGANBV_FP
- FEATMORGAN_FP
- FEATMORGANBV_FP
- RDKIT_FP
- ATOMPAIR_FP
- ATOMPAIRBV_FP
- TORSION_FP
- TORSIONBV_FP
- LAYERED_FP
- MACCS_FP
- TANIMOTO_SML
- DICE_SML
- TVERSKY_SML
- TANIMOTO_DIST
- DICE_DIST

7.2 Aggregates

- FMCS

CHAPTER 8

Configuration parameters

The RDKit PostgreSQL cartridge defines a set of configuration parameters fine-tuning some of the implemented functions. From the SQL interface these parameters can be manipulated using `set` and `show` statements:

```
my_database=# set rdkit.do_chiral_sss=true;
```

but in the `django_rdkit` package they are also exposed as attributes of a `config` object:

```
In [43]: from django_rdkit.config import config
```

so that their values can be set and queried without leaving the python domain:

```
In [44]: config.do_chiral_sss = True

In [45]: print(config.tanimoto_threshold)
0.5
```

As you may notice from the examples, the main difference compared to the RDKit cartridge should consist in a minor change in the naming convention. The cartridge defines these parameters with a name starting with an `rdkit.` prefix. In naming the corresponding attributes of the `config` object this prefix is dropped.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index