# ChemicaLite Documentation

***Release 0.2.0***

**Riccardo Vianello**

August 22, 2016

Contents

Contents:

CHAPTER 1

# Building the extension

## 1.1 Dependencies

- SQLite (devel package too)
- RDKit

Support for loading extensions is often disabled in the *sqlite3* package that is provided by the python standard library. Using the extension from Python may therefore require a proper build of `pysqlite` (python2.7 only), or or `APSW`. APWS is also required for running the python tests.

## 1.2 Configure and build

Default linux build:

```
$ cd build/dir
$ cmake path/to/chemicalite/dir -DRDKit_DIR=path/to/rdkit/lib/dir
$ make
$ LD_LIBRARY_PATH=. make test
```

Building with tests disabled:

```
$ cmake path/to/chemicalite/dir \
    -DRDKit_DIR=path/to/rdkit/lib/dir -DCHEMICALITE_ENABLE_TESTS=OFF
```

Or only python tests disabled:

```
$ cmake path/to/chemicalite/dir \
    -DRDKit_DIR=path/to/rdkit/lib/dir -DCHEMICALITE_ENABLE_PYTHON_TESTS=OFF
```

CHAPTER 2

# ChemicaLite Tutorial

## 2.1 Building a database

This tutorial is based on a similar one which is included with the RDKit PostgreSQL Cartridge documentation and it will guide you through the construction of a chemical SQLite database and the execution of some simple queries. Python will be used in illustrating the various operations, but almost any other programming language could be used instead (as long as SQLite drivers are available).

Download a copy of the ChEMBLdb database and decompress it:

```
$ gunzip chembl_20_chemreps.txt.gz
```

Creating a database and initializing its schema requires just a few statements:

```
import apsw

# the extension is usually loaded right after the connection to the
# database
connection = apsw.Connection('chembldb.sql')
connection.enableloadextension(True)
connection.loadextension(path_to_chemicalite_ext)
connection.enableloadextension(False)

cursor = connection.cursor()

# the SQLite memory page size affects the configuration of the
# substructure search index tree. this operation must be performed
# at database creation, before the first CREATE TABLE.
cursor.execute("PRAGMA page_size=2048")

# our database will consist of a single table, containing a subset of the
# columns from the ChEMBLdb database. The molecular structure is inserted
# as a binary blob of the pickled molecule.
cursor.execute("CREATE TABLE chembl(id INTEGER PRIMARY KEY, " +
               "chembl_id TEXT, smiles TEXT, molecule MOL)")

# finally, this statement will create and configure an index
# associated to the 'molecule' column of the 'chembl' table.
cursor.execute("SELECT create_molecule_rdtree('chembl', 'molecule')")
```

Support for custom indexes in SQLite is a bit different than other database engines. The data structure of a custom index is in fact wrapped behind the implementation of a "virtual table", an object that exposes an interface that is almost identical to that of a regular SQL table, but whose implementation can be customized.

The above call to the *create_molecule_rdtree* function creates a virtual table with SQL name *str_idx_chembl_molecule* and a few triggers that connect the manipulation of the *molecule* column of the *chembl* table with the management of the tree data structure wrapped behind *str_idx_chembl_molecule*.

For example, each time a new record is inserted into the *chembl* table, a bitstring signature of the involved molecule is computed and inserted into *str_idx_chembl_molecule*.

Join operations involving the *chembl* and *str_idx_chembl_molecule* tables can this way use the tree data structure to strongly reduce the number of *chembl* records that are checked during a substructure search.

The ChEMBLdb data can be parsed with a python generator function similar to the following:

```
def chembl(path):
    """Extract the chembl_id and SMILES fields"""
    with open(path, 'rb') as inputfile:
        reader = csv.reader(inputfile, delimiter='\t',
                            skipinitialspace=True)
        reader.next() # skip header line

        for chembl_id, smiles, inchi, inchi_key in reader:
            # check the SMILES and skip problematic compounds
            # [...]
            yield chembl_id, smiles
```

And the database is loaded with loop like this:

```
cursor.execute('BEGIN')
for chembl_id, smiles in chembl(chembl_path):
    cursor.execute("INSERT INTO chembl(chembl_id, smiles, molecule) "
                   "VALUES(?, ?, mol(?))", (chembl_id, smiles, smiles))
cursor.execute('COMMIT')
```

Please note that loading the whole ChEMBLdb is going to take a substantial amount of time and the resulting file will require about 1.5GB of disk space.

A python script implementing the full schema creation and database loading procedure as a single command line tool is available in the *docs* directory of the source code distribution:

```
# This will create the molecular database as a file named 'chembldb.sql'
$ ./create_chembldb.py /path/to/chemicalite.so chembl_20_chemreps.txt
```

## 2.2 Substructure Searches

A search for substructures could be performed with a query like the following:

```
SELECT COUNT(*) FROM chembl WHERE mol_is_substruct(molecule, 'c1ccnnc1');
```

but this would check every single record of the *chembl* table, resulting very inefficient. Performances can strongly improve if the index table is joined:

```
SELECT COUNT(*) FROM chembl, str_idx_chembl_molecule AS idx WHERE
    chembl.id = idx.id AND
    mol_is_substruct(chembl.molecule, 'c1ccnnc1') AND
    idx.id MATCH rdtree_subset(mol_bfp_signature('c1ccnnc1'));
```

A python script executing this second query is available in the *docs* directory of the source code distribution:

```
# returns the number of structures containing the query fragment.
$ ./match_count.py /path/chemicalite.so /path/to/chembldb.sql c1ccnnc1
```

And here are some example queries:

```
$ ./match_count.py /path/chemicalite.so chembldb.sql c1cccc2c1nncc2
searching for substructure: c1cccc2c1nncc2
Found 285 matches in 0.580219984055 seconds

$ ./match_count.py /path/chemicalite.so chembldb.sql c1ccnc2c1nccn2
searching for substructure: c1ccnc2c1nccn2
Found 707 matches in 0.415385007858 seconds

$ ./match_count.py /path/chemicalite.so chembldb.sql Nc1ncnc\(N\)n1
searching for substructure: Nc1ncnc(N)n1
Found 4564 matches in 1.44142603874 seconds

$ ./match_count.py /path/chemicalite.so chembldb.sql c1scnn1
searching for substructure: c1scnn1
Found 11235 matches in 2.81160211563 seconds

$ ./match_count.py /path/chemicalite.so chembldb.sql c1cccc2c1ncs2
searching for substructure: c1cccc2c1ncs2
Found 13521 matches in 5.35551190376 seconds

$ ./match_count.py /path/chemicalite.so chembldb.sql c1cccc2c1CNCCN2
searching for substructure: c1cccc2c1CNCCN2
Found 1210 matches in 15.256114006 seconds
```

*Note*: Execution times are only provided for reference and may vary depending on the available computational power. Moreover, and especially for larger database files, timings appear to be quite sensitive to the behavior of the operating system disk cache. Should you happen to observe anything like a 10-50x difference between the execution times for the first and the second run of the same query, please try bringing the sqlite file into the OS disk cache and see if it helps (something like *cat chembldb.sql > /dev/null* should do).

A second script is provided with the documentation and it's designed to only return the first results (sometimes useful for queries that return a large number of matches):

```
$ ./substructure_search.py /path/chemicalite.so chembldb.sql c1cccc2c1CNCCN2
searching for substructure: c1cccc2c1CNCCN2
CHEMBL323692 C1CNc2ccccc2CN1
CHEMBL1458895 COC(=O)CN1CCN(C(=O)c2ccc(F)cc2)c3ccccc3C1
CHEMBL1623831 C(C1CNc2ccccc2CN1)c3ccccc3
[...]
CHEMBL270270 NCCCCC1NC(=O)c2ccc(Cl)cc2N(Cc3ccccc3)C1=O
CHEMBL233255 Oc1ccc(C[C@@H]2NC(=O)c3ccccc3NC2=O)cc1
Found 25 matches in 0.536008834839 seconds
```

## 2.3 Similarity Searches

Fingerprint data for similarity searches is conveniently stored into indexed virtual tables, as illustrated by the following statements:

```
import apsw

connection = apsw.Connection(chembldb_path)
connection.enableloadextension(True)
```

```
connection.loadextension(chemicalite_path)
connection.enableloadextension(False)

cursor = connection.cursor()

# create a virtual table to be filled with morgan bfp data
cursor.execute("CREATE VIRTUAL TABLE morgan USING\n" +
               "rdtree(id, bfp bytes(64))");

# compute and insert the fingerprints
cursor.execute("INSERT INTO morgan(id, bfp)\n" +
               "SELECT id, mol_morgan_bfp(molecule, 2) FROM chembl")
```

Once again, a script file implementing the above commands is provided:

```
$ ./create_bfp_data.py /path/to/chemicalite.so /path/to/chembldb.sql
```

A search for similar structures is therefore based on filtering this newly created table. The following statement would for example return the number of compounds with a Tanimoto similarity greater than or equal to the threshold value (see also the *tanimoto_count.py* file for a complete script):

```
count = c.execute("SELECT count(*) FROM "
                  "morgan as idx WHERE "
                  "idx.id match rdtree_tanimoto(mol_morgan_bfp(?, 2), ?)",
                  (target, threshold)).fetchone()[0]
```

A sorted list of SMILES strings identifying the most similar compounds is for example produced by the following query:

```
rs = c.execute(
    "SELECT c.chembl_id, c.smiles, bfp_tanimoto(mol_morgan_bfp(c.molecule, 2), mol_morgan_bfp(?, 2))
    "FROM "
    "chembl as c JOIN "
    "(SELECT id FROM morgan WHERE id match rdtree_tanimoto(mol_morgan_bfp(?, 2), ?)) as idx "
    "USING(id) ORDER BY t DESC",
    (target, target, threshold)).fetchall()
```

Finally, these last two examples were executed using the *tanimoto_search.py* script, which is based on the previous query:

```
$ ./tanimoto_search.py /path/to/chemicalite.so /path/to/chembldb.sql "Cc1ccc2nc(-c3ccc(NC(C4N(C(c5cc
searching for target:  Cc1ccc2nc(-c3ccc(NC(C4N(C(c5cccs5)=O)CCC4)=O)cc3)sc2c1
CHEMBL467428 Cc1ccc2nc(sc2c1)c3ccc(NC(=O)C4CCN(CC4)C(=O)c5cccs5)cc3 0.772727272727
CHEMBL461435 Cc1ccc2nc(sc2c1)c3ccc(NC(=O)C4CCCN(C4)S(=O)(=O)c5cccs5)cc3 0.657534246575
CHEMBL460340 Cc1ccc2nc(sc2c1)c3ccc(NC(=O)C4CCN(CC4)S(=O)(=O)c5cccs5)cc3 0.647887323944
CHEMBL460588 Cc1ccc2nc(sc2c1)c3ccc(NC(=O)C4CCN(C4)S(=O)(=O)c5cccs5)cc3 0.638888888889
CHEMBL1608585 Clc1ccc2nc(NC(=O)[C@@H]3CCCN3C(=O)c4cccs4)sc2c1 0.623188405797
[...]
CHEMBL1325810 Cc1ccc(NC(=O)N2CCCC2C(=O)NCc3cccs3)cc1 0.5
CHEMBL1864141 Clc1ccc(NC(=O)[C@@H]2CCCN2C(=O)c3cccs3)cc1S(=O)(=O)N4CCOCC4 0.5
CHEMBL1421062 COc1cc(Cl)c(C)cc1NC(=O)[C@@H]2CCCN2C(=O)c3cccs3 0.5
Found 66 matches in 1.53940916061 seconds
```

```
$ ./tanimoto_search.py /path/to/chemicalite.so /path/to/chembldb.sql "Cc1ccc2nc(N(C)CC(=O)O)sc2c1" 0
searching for target: Cc1ccc2nc(N(C)CC(=O)O)sc2c1
CHEMBL394654 CN(CCN(C)c1nc2ccc(C)cc2s1)c3nc4ccc(C)cc4s3 0.692307692308
CHEMBL491074 CN(CC(=O)O)c1nc2cc(ccc2s1)[N+](=O)[O-] 0.583333333333
CHEMBL1617304 CN(C)CCCN(C(=O)C)c1nc2ccc(C)cc2s1 0.571428571429
CHEMBL1350062 Cl.CN(C)CCCN(C(=O)C)c1nc2ccc(C)cc2s1 0.549019607843
```

```
[...]
CHEMBL1610437 Cl.CN(C)CCCN(C(=O)CS(=O)(=O)c1ccccc1)c2nc3ccc(C)cc3s2 0.5
CHEMBL1351385 Cl.CN(C)CCCN(C(=O)CCc1ccccc1)c2nc3ccc(C)cc3s2 0.5
CHEMBL1622712 CN(C)CCCN(C(=O)COc1ccc(Cl)cc1)c2nc3ccc(C)cc3s2 0.5
CHEMBL1591601 Cc1ccc2nc(sc2c1)N(Cc3cccnc3)C(=O)Cc4ccccc4 0.5
Found 18 matches in 1.39061594009 seconds
```

CHAPTER 3

# Extended Tutorial

<TODO>

CHAPTER 4

# API Reference

## 4.1 Data types

- *mol*: an rdkit molecule. Can be created from a SMILES using the *mol* function, for example: *mol('c1ccccc1')* creates a molecule from the SMILES *'c1ccccc1'*
- *qmol*: an rdkit molecule containing query features (i.e. constructed from SMARTS). Can be created from a SMARTS using the *qmol* function, for example: *qmol('c1cccc[c,n]1')* creates a query molecule from the SMARTS *'c1cccc[c,n]1'*
- *bfp*: a bit vector fingerprint

In most places where a *mol* or *qmol* is expected, a text string can be used instead and is implicitly interpreted as a SMILES. Use an explicit cast in case a SMARTS is being passed.

*Note*: All of the above data types are serialized and stored in the form of binary blobs. No attempt is made at implementing any sort of type system that could allow the code to recognize if the wrong data is passed as input to a procedure. Some care is therefore recommended.

## 4.2 Operators

### 4.2.1 Substructure searches

Substructure searches are performed constraining the selection on a column of *mol* data with a *WHERE* clause based on the return value of function *mol_is_substruct*. This can be optionally (and preferably) joined with a *MATCH* constraint on an *rdtree* index, using the match object returned by *rdtree_subset*:

```
SELECT * FROM mytable, str_idx_mytable_molcolumn AS idx WHERE
    mytable.id = idx.id AND
    mol_is_substruct(mytable.molcolumn, 'c1ccnnc1') AND
    idx.id MATCH rdtree_subset(mol_bfp_signature('c1ccnnc1'));
```

### 4.2.2 Similarity searches

Similarity search on *rdtree* virtual tables of binary fingerprint data are supported by means of the match object returned by the *rdtree_tanimoto* factory function:

```
SELECT c.smiles, bfp_tanimoto(mol_morgan_bfp(c.molecule, 2), mol_morgan_bfp(?, 2)) as t
    FROM mytable as c JOIN (SELECT id FROM morgan WHERE id match rdtree_tanimoto(mol_morgan_bfp(?, 2)
    USING(id) ORDER BY t DESC;
```

## 4.3 Functions

### 4.3.1 Molecule

- *mol(string)*
- *qmol(string)*
- *mol_smiles(mol)*
- *mol_is_substruct(mol, mol)*
- *mol_is_superstruct(mol, mol)*
- *mol_cmp(mol, mol)*

### 4.3.2 Descriptors

- *mol_hba(mol)*
- *mol_hbd(mol)*
- *mol_mw(mol)*
- *mol_logp(mol)*
- *mol_tpsa(mol)*
- *mol_num_atms(mol)*
- *mol_num_hvyatms(mol)*
- *mol_num_rotatable_bnds(mol)*
- *mol_num_hetatms(mol)*
- *mol_num_rings(mol)*
- *mol_chi0v(mol) - mol_chi4v(mol)*
- *mol_chi0n(mol) - mol_chi4n(mol)*
- *mol_kappa1(mol) - mol_kappa3(mol)*

### 4.3.3 Fingerprints

- *mol_layered_bfp(mol)*
- *mol_rdkit_bfp(mol)*
- *mol_atom_pairs_bfp(mol)*
- *mol_topological_torsion_bfp(mol)*
- *mol_maccs_bfp(mol)*
- *mol_morgan_bfp(mol, int)*

- *mol_feat_morgan_bfp(mol, int)*
- *mol_bfp_signature(mol)*
- *bfp_tanimoto(bfp, bfp)*
- *bfp_dice(bfp, bfp)*
- *bfp_length(bfp)*
- *bfp_weight(bfp)*

CHAPTER 5

# Links and resources

- ChemicaLite on GitHub
- SQLite C/C++ API reference
- The Virtual Table Mechanism of SQLite
- The SQLite R*Tree Module
- The RD-tree: an index structure for sets
- RDKit: Cheminformatics and Machine Learning Software
- ChEMBLdb

CHAPTER 6

# Indices and tables

- genindex
- modindex
- search