ChemicaLite Documentation

Release 2022.04.1

Riccardo Vianello

Apr 07, 2022

Contents

ChemicaLite is an SQLite extension for chemoninformatics applications.

Contents:

CHAPTER 1

Installation

Conda packages for the Linux and OSX operating systems are currently available from *conda-forge*.

To install the ChemicaLite extension using conda, just specify the *conda-forge* channel, either on the conda command line, or in your *.condarc* file.

For example:

```
$ conda install -c cond-forge chemicalite
```

CHAPTER 2

Building the extension

2.1 Dependencies

- SQLite (devel package too)
- RDKit
- Catch2 (to build the tests)

2.2 Configure and build

The supported build system uses *cmake*, and it requires RDKit 2020.09.1 or later.

Note: The code was built successfully on some Linux and OSX systems. Building on other operating systems (e.g. Windows) is probably possible, but it wasn't tested.

Default Linux build:

```
$ cd build/dir
$ cmake path/to/chemicalite/dir
$ make
$ LD_LIBRARY_PATH=$PWD/src make test
```

Building with tests disabled:

```
$ cmake path/to/chemicalite/dir -DCHEMICALITE_ENABLE_TESTS=OFF
```

CHAPTER 3

ChemicaLite Tutorial

3.1 Building a database

This tutorial is based on a similar one which is part of 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_28_chemreps.txt.gz
```

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

```
import sqlite3

connection = sqlite3.connect('chembldb.sql')

# the extension is usually loaded right after the connection to the
# database

# because this operation loads external code from the extension module into
# the program, it's considered potentially insecure, and it's by default disabled.
# it therefore needs to be explicitly enabled (and then disabled again).
connection.enable_load_extension(True)

# if the chemicalite extension is installed under the dynamic library search path
# for the system or running process, you can simply refer to it by name.
# otherwise you may need to pass the filesystem path of the loadable module file:
# connection.load_extension('/path/to/chemicalite.so')
connection.load_extension('chemicalite')

connection.enable_load_extension(False)
```

(continues on next page)

(continued from previous page)

```
# the database will mainly consist of one table, containing the
# compound structures from the ChEMBLdb data.
connection.execute(
    "CREATE TABLE chembl(id INTEGER PRIMARY KEY, chembl_id TEXT, molecule MOL)")
```

The ChEMBLdb compounds data is available as a simple tsv file, that can be parsed with a python generator function similar to the following:

```
import csv

def chembl(path):
    with open(path, 'rt') as inputfile:
        reader = csv.reader(inputfile, delimiter='\t')
        next(reader) # skip header line
        for chembl_id, smiles, *_ in reader:
            yield chembl_id, smiles
```

And the database table can be loaded with a statement like this:

```
with connection:
    connection.executemany(
        "INSERT INTO chembl(chembl_id, molecule) "
        "VALUES(?1, mol_from_smiles(?2))", chembl('chembl_28_chemreps.txt'))
```

A complete script implementing the full creation of this simple database of chemical structures is available from the examples directory of the source code distribution.

Note: Loading the entire collection of ChEMBLdb compounds may take some time and the resulting file will require several GB of disk space. The *create_chembldb.py* script supports a command line option that may help limiting the number of imported compounds during tests.

3.2 Substructure queries

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 kind of simple query would sequentially check every single molecule in the *chembl* table, resulting very inefficient.

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 the interface of a regular SQL table, but whose implementation can be customized.

ChemicaLite uses this virtual table mechanism to support indexing binary fingerprints in an RD-tree data structure, and this way improve the performances of substructure and similarity queries.

An RD-tree virtual table for substructure queries is created with a statement like the following:

```
connection.execute("CREATE VIRTUAL TABLE str_idx_chembl_molecule " +
                   "USING rdtree(id, fp bits(2048))")
```

And this index table is then filled with the structural fingerprint data generated from the *chembl* table:

```
with connection:
    connection.execute(
        "INSERT INTO str_idx_chembl_molecule(id, fp) " +
        "SELECT id, mol_pattern_bfp(molecule, 2048) FROM chembl " +
        "WHERE molecule IS NOT NULL")
```

The performances of the substructure query above can this way strongly improve if the index table is joined, and a substructure constraint is specified using an *rdtree_subset* match object:

```
SELECT COUNT(*) FROM chembl, str_idx_chembl_molecule AS idx WHERE
    chembl.id = idx.id AND
    mol_is_substruct(chembl.molecule, mol_from_smiles('c1ccnnc1')) AND
    idx.id MATCH rdtree_subset(mol_pattern_bfp(mol_from_smiles('c1ccnnc1'), 2048));
```

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

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

And here are some example queries:

```
$ ./match_count.py chembldb.sql c1cccc2c1nncc2
searching for substructure: c1cccc2c1nncc2
Found 525 matching structures in 0.226271390914917 seconds

$ ./match_count.py chembldb.sql c1ccnc2c1nccn2
searching for substructure: c1ccnc2c1nccn2
Found 1143 matching structures in 0.3587167263031006 seconds

$ ./match_count.py chembldb.sql Nc1ncnc\(N\)n1
searching for substructure: Nc1ncnc(N)n1
Found 8197 matching structures in 0.8730080127716064 seconds

$ ./match_count.py chembldb.sql c1scnn1
searching for substructure: c1scnn1
Found 17918 matching structures in 1.2525584697723389 seconds

$ ./match_count.py chembldb.sql c1cccc2c1ncs2
searching for substructure: c1cccc2c1ncs2
Found 23277 matching structures in 1.7844812870025635 seconds

$ ./match_count.py chembldb.sql c1cccc2c1CNCCN2
searching for substructure: c1cccc2c1CNCCN2
Found 1973 matching structures in 2.547306776046753 seconds
```

(Execution times are only provided for reference and may vary depending on the available computational resources).

A second script is available in the examples directory, and it illustrates how to return only the first results (sometimes useful for queries that return a large number of matches):

```
$ ./substructure_search.py chembldb.sql c1cccc2c1CNCCN2
searching for substructure: c1cccc2c1CNCCN2
CHEMBL7892 CC(=O)Nc1ccc2c(c1)C(=O)N(C(C(=O)NC1CCCCC1)c1ccc([N+](=O)[O-
↪])cc1)[C@@H](c1ccccc1)C(=O)N2
CHEMBL415394␣
↪CC(C)[C@H](NC(=O)[C@H](CCCN=C(N)N)NC(=O)[C@@H](N)CC(=O)O)C(=O)N[C@@H](Cc1ccc(O)cc1)C(=O)Nc1ccc2c(c1
CHEMBL8003 O=S(=O)(c1cc(Cl)ccc1Cl)N1Cc2ccccc2N(Cc2c[nH]cn2)C(CCc2ccccc2)C1
```

(continues on next page)

(continued from previous page)

```
[...]
CHEMBL53987 Cc1cccc(NCCNC(=O)c2ccc3c(c2)CN(C)C(=O)[C@H](CC(=O)O)N3)n1
CHEMBL53985 CN1Cc2cc(C(=O)NCc3c[nH]cn3)ccc2N[C@@H](CC(=O)O)C1=O
CHEMBL57915␣
↪CC(C)C[C@H]1C(=O)N2c3ccccc3[C@@](O)(C[C@@H]3NC(=O)c4ccccc4N4C(=O)c5ccccc5NC34)[C@H]2N1C(=O)CCC(=O)
↪].[Na+]
CHEMBL50075 CN1Cc2cc(C(=O)NCCNc3ccccc3)ccc2N[C@@H](CC(=O)O)C1=O
CHEMBL50257 CN1Cc2cc(C(=O)NCCc3cccc(N)n3)ccc2N[C@@H](CC(=O)O)C1=O
Found 25 matches in 0.08957481384277344 seconds
```

3.3 Similarity queries

In the same way, binary fingerprint data for similarity queries is conveniently stored and indexed into RD-tree virtual tables, as illustrated by the following statements:

```
# create a virtual table to be filled with morgan bfp data
connection.execute("CREATE VIRTUAL TABLE morgan_idx_chembl_molecule " +
            "USING rdtree(id, fp bits(1024))");

# compute and insert the fingerprints
with connection:
    connection.execute(
        "INSERT INTO morgan_idx_chembl_molecule(id, fp) " +
        "SELECT id, mol_morgan_bfp(molecule, 2, 1024) FROM chembl " +
        "WHERE molecule IS NOT NULL")
```

A search for similar structures is therefore based on filtering this new virtual 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* script for a complete example):

```
count = connection.execute(
    "SELECT count(*) FROM "
    "morgan_idx_chembl_molecule as idx WHERE "
    "idx.id match rdtree_tanimoto(mol_morgan_bfp(mol_from_smiles(?), 2, 1024), ?)",
    (target, threshold)).fetchall()[0][0]
```

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

```
rs = connection.execute(
    "SELECT c.chembl_id, mol_to_smiles(c.molecule), "
    "bfp_tanimoto(mol_morgan_bfp(c.molecule, 2, 1024), "
    "             mol_morgan_bfp(mol_from_smiles(?1), 2, 1024)) as t "
    "FROM "
    "chembl as c JOIN morgan_idx_chembl_molecule as idx USING(id) "
    "WHERE "
    "idx.id MATCH rdtree_tanimoto(mol_morgan_bfp(mol_from_smiles(?1), 2, 1024), ?2) "
    "ORDER BY t DESC",
    (target, threshold)).fetchall()
```

These last two examples show the output produced by the *tanimoto_search.py* script, which is based on the previous query:

```
$ ./tanimoto_search.py /path/to/chembldb.sql "Cc1ccc2nc(-
↪c3ccc(NC(C4N(C(c5cccs5)=O)CCC4)=O)cc3)sc2c1" 0.5
searching for target: Cc1ccc2nc(-c3ccc(NC(C4N(C(c5cccs5)=O)CCC4)=O)cc3)sc2c1
CHEMBL467428 Cc1ccc2nc(-c3ccc(NC(=O)C4CCN(C(=O)c5cccs5)CC4)cc3)sc2c1 0.
↪7611940298507462
CHEMBL461435 Cc1ccc2nc(-c3ccc(NC(=O)C4CCCN(S(=O)(=O)c5cccs5)C4)cc3)sc2c1 0.
↪6486486486486487
CHEMBL460340 Cc1ccc2nc(-c3ccc(NC(=O)C4CCN(S(=O)(=O)c5cccs5)CC4)cc3)sc2c1 0.
↪6301369863013698
[...]
CHEMBL218058 Cc1ccc2nc(-c3ccc(NC(=O)Nc4ccc(Cl)cc4)cc3)sc2c1 0.5
CHEMBL1317763 Cc1cc(C)c(NC(=O)CNC(=O)[C@@H]2CCCN2C(=O)c2cccs2)c(C)c1 0.5
Found 54 matches in 0.6459760665893555 seconds
```

```
$ ./tanimoto_search.py /path/to/chemicalite.so /path/to/chembldb.sql
↪"Cc1ccc2nc(N(C)CC(=O)O)sc2c1" 0.5
CHEMBL394654 Cc1ccc2nc(N(C)CCN(C)c3nc4ccc(C)cc4s3)sc2c1 0.6923076923076923
CHEMBL3928717 CN(CC(=O)O)c1nc2ccc([N+](=O)[O-])cc2s1 0.6739130434782609
CHEMBL491074 CN(CC(=O)O)c1nc2cc([N+](=O)[O-])ccc2s1 0.5833333333333334
[...]
CHEMBL1617545 Cc1ccc2nc(N(CCCN(C)C)C(=O)CCc3ccccc3)sc2c1 0.5087719298245614
CHEMBL1351385 Cc1ccc2nc(N(CCCN(C)C)C(=O)CCc3ccccc3)sc2c1.Cl 0.5
CHEMBL1418054 Cc1ccc2nc(N(CCN(C)C)C(=O)c3ccc4ccccc4c3)sc2c1.Cl 0.5
Found 12 matches in 1.2354457378387451 seconds
```

CHAPTER 4

API Reference

4.1 Data types

- *mol*: an RDKit molecule.
- *bfp*: a binary fingerprint.

The *mol* type is used to represent both a "regular" fully-specified molecule, and also a molecular structure that includes query features (e.g. built from SMARTS input).

No implicit conversion from text input formats to *mol* is supported. Passing a SMILES or SMARTS string where a *mol* argument is expected, should result in an error. The input textual representation is always required to be wrapped by a suitable conversion function (e.g. *mol_from_smiles*).

4.2 Functions

4.2.1 Molecule

- *mol_from_smiles(text) -> mol*
- *mol_from_smarts(text) -> mol*
- *mol_from_molblock(text) -> mol*
- *mol_from_binary_mol(blob) -> mol*
- *mol_to_smiles(mol) -> text*
- *mol_to_smarts(mol) -> text*
- *mol_to_molblock(mol) -> text*
- *mol_to_binary_mol(mol) -> blob*
- *mol_is_substruct(mol, mol) -> int*

- *mol_is_superstruct(mol, mol) -> int*
- *mol_cmp(mol, mol) -> int*
- *mol_hba(mol) -> int*
- *mol_hbd(mol) -> int*
- *mol_num_atms(mol) -> int*
- *mol_num_hvyatms(mol) -> int*
- *mol_num_rotatable_bnds(mol) -> int*
- *mol_num_hetatms(mol) -> int*
- *mol_num_rings(mol) -> int*
- *mol_num_aromatic_rings(mol) -> int*
- *mol_num_aliphatic_rings(mol) -> int*
- *mol_num_saturated_rings(mol) -> int*
- *mol_mw(mol) -> real*
- *mol_tpsa(mol) -> real*
- *mol_fraction_csp3(mol) -> real*
- *mol_chi0v(mol) - mol_chi4v(mol) -> real*
- *mol_chi0n(mol) - mol_chi4n(mol) -> real*
- *mol_kappa1(mol) - mol_kappa3(mol) -> real*
- *mol_logp(mol) -> real*
- *mol_formula(mol) -> text*
- *mol_hash_anonymousgraph(mol) -> text*
- *mol_hash_elementgraph(mol) -> text*
- *mol_hash_canonicalsmiles(mol) -> text*
- *mol_hash_murckoscaffold(mol) -> text*
- *mol_hash_extendedmurcko(mol) -> text*
- *mol_hash_molformula(mol) -> text*
- *mol_hash_atombondcounts(mol) -> text*
- *mol_hash_degreevector(mol) -> text*
- *mol_hash_mesomer(mol) -> text*
- *mol_hash_hetatomtautomer(mol) -> text*
- *mol_hash_hetatomprotomer(mol) -> text*
- *mol_hash_redoxpair(mol) -> text*
- *mol_hash_regioisomer(mol) -> text*
- *mol_hash_netcharge(mol) -> text*
- *mol_hash_smallworldindexbr(mol) -> text*
- *mol_hash_smallworldindexbrl(mol) -> text*

- *mol_hash_arthorsubstructureorder(mol) -> text*
- *mol_prop_list(mol) -> [text]*
- *mol_has_prop(mol, text) -> int*
- *mol_set_prop(mol, text, text\|real\|int) -> mol*
- *mol_get_text_prop(mol, text) -> text*
- *mol_get_int_prop(mol, text) -> int*
- *mol_get_float_prop(mol, text) -> real*
- *mol_delete_substructs() -> mol*
- *mol_replace_substructs() -> [mol]*
- *mol_replace_sidechains() -> mol*
- *mol_replace_core() -> mol*
- *mol_murcko_decompose() -> mol*
- *mol_find_mcs([mol]) -> mol*
- *mol_cleanup(mol, update_params=”) -> mol*
- *mol_normalize(mol, update_params=”) -> mol*
- *mol_reionize(mol, update_params=”) -> mol*
- *mol_remove_fragments(mol, update_params=”) -> mol*
- *mol_canonical_tautomer(mol, update_params=”) -> mol*
- *mol_tautomer_parent(mol, update_params=”, skip_standardize=false) -> mol*
- *mol_fragment_parent(mol, update_params=”, skip_standardize=false) -> mol*
- *mol_stereo_parent(mol, update_params=”, skip_standardize=false) -> mol*
- *mol_isotope_parent(mol, update_params=”, skip_standardize=false) -> mol*
- *mol_charge_parent(mol, update_params=”, skip_standardize=false) -> mol*
- *mol_super_parent(mol, update_params=”, skip_standardize=false) -> mol*

4.2.2 Binary Fingerprint

- *mol_layered_bfp(mol, int) -> bfp*
- *mol_rdkit_bfp(mol, int) -> bfp*
- *mol_atom_pairs_bfp(mol, int) -> bfp*
- *mol_topological_torsion_bfp(mol, int) -> bfp*
- *mol_pattern_bfp(mol, int) -> bfp*
- *mol_morgan_bfp(mol, int, int) -> bfp*
- *mol_feat_morgan_bfp(mol, int, int) -> bfp*
- *bfp_tanimoto(bfp, bfp) -> real*
- *bfp_dice(bfp, bfp) -> real*
- *bfp_length(bfp) -> int*

- *bfp_weight(bfp) -> int*

4.2.3 Utility

- *chemicalite_version() -> text*
- *rdkit_version() -> text*
- *rdkit_build() -> text*
- *boost_version() -> text*

4.3 Substructure and Similarity Queries

- *rdtree_subset(bfp) -> blob*
- *rdtree_tanimoto(bfp) -> blob*

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 (but 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, mol_from_smiles('c1ccnnc1')) AND
    idx.id MATCH rdtree_subset(mol_pattern_bfp(mol_from_smiles('c1ccnnc1'), 2048));
```

Similarity search queryes on *rdtree* virtual tables of binary fingerprint data are supported by 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), ?)) as idx
    USING(id) ORDER BY t DESC;
```

4.3.1 Molecular file format readers and writers

- *sdf_reader*
- *sdf_writer*
- *smi_reader*
- *smi_writer*

CHAPTER 5

Links and resources

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

CHAPTER 6

Pragma settings that may sometimes help

6.1 page_size

page_size settings impose an upper limit to the size of a node in the index tree, and therefore contribute to determining the maximum number of references to child nodes (fan-out). A larger page size, and therefore a larger fan-out, may help reduce the disk I/O and improve the performances of substructure or similarity queries.

This configuration parameter used to be more critically important a few years ago, when the default page size was almost always 1024 bytes, but beginning with SQLite v3.12.0 this value increased to 4096 bytes.

6.2 cache_size

The default cache_size is usually limited to 2000 KiB. Increasing this value may help keeping the data in memory and reduce the access to the disk.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search