In [30]:
```
import os
print(os.getcwd())
from cactvs import *
```

```
/users/wdi
```

In [31]:
```
cactvs['sql_dialect'] = 'postgres'
```

# Storing and querying chemical structure data

A most versatile method to store information are generic SQL databases. However, chemists have specific requirements for queries of chemical structure (and reaction) data, in addition to standard database queries.

- Full-structure queries (modified by isotope/stereochemistry/parent compound... relevance criteria)
- Similarity queries
- Substructure queries
- Superstructure queries
- Formula queries

And do not forget reaction data.

# Database cartridges

For the last decades, database **cartridges** have been the technological solution of choice. They add chemistry-specific functions to standard databases.

In [32]:
```
db=Dbase(dbtype='postgres',database='chembl_27',socket='/tmp')
```

In [42]:
```
print(db.itemquery('select count(*) from compound_query where match_substructure(
```

```
12781
```

This works pretty well and is an industry standard.

Is there a problem?

# New data worlds

Chemistry databases get bigger every year. They have exceeded the storage and maintenance capabilities of local sites.

Fortunately, there are vendors like Google or Amazon, which provide hosted cloud SQL databases of near unlimited scalability, at a very affordable price (Google BigQuery, Amazon Aurora)

So why are we not happy?

***There are no cartridges in the cloud.*** (Well, at least not without vendor vetting, see Aurora/RDkit)

And without cartridges, no chemistry queries.

# Coming from a different direction

We do scripting for chemisty. Our scripting environment includes a quite sophisticated query processor.

Query specification parsers:

- SMARTS
- QuerySLN
- MDL Query Molfiles
- PATRAN
- QueryCDX/CDXML
- EliLilly Cerius-Class queries
- Arbitrary assembly of multiple query components by scripting

Query patterns are read into common structure objects, regardless of encoding. Structure and reaction query instructions can be linked via boolean (and other) operators to complex query expressions.

Integrated query optimizer, and hooks into file I/O module accelerator functions.

Apply to single objects, in-memory datasets, plain and accelerated structure and reaction data files, virtual files.

In [43]:
```
e=Ens('C2(=CC(=O)C1=CC=CC=C1C2=O)SC3=CC=CC=C3')
print(e.scan('structure >= c1ccccc1S'))
e.delete()
```

```
True
```

Out[43]: 1

In [44]:
```
d=Dataset('C2(=CC(=O)C1=CC=CC=C1C2=O)SC3=CC=CC=C3'
'C12=C(ON=C1C(=CC=C2)CC(=O)O)C3=CC=C(Cl)C=C3'
'C12=C(ON=C1C(=CC=C2)CC(=O)O)C3=CC=CC=C3',
'ClC1=CC=C(C=C1)C3=C2C=C(C=CC2=NO3)C=O',
'BrC2=CC1=C(ON=C1C=C2)C3=CC(=C(OC)C(=C3)OC)OC',
'O1N=C3C(=C1C2=CC(=CC=C2)OC)C=C(C=C3)C(=O)\\C=C\\N(C)C',
'O1N=C3C(=C1C2=CC=CC=C2)C=C(C=C3)C(=O)\\C=C\\N(C)C',
'O1N=C3C(=C1C2=CC=C(OC)C=C2)C=C(C=C3)C(=O)C',
'O1N=C5C(=C1C2=CC=CC=C2)C=C(C(=O)N3CCN(CC3)C4=CC=CC=C4)C=C5')
print(d.scan('structure ~>= O1N=C3C(=C1C2=CC=C(OC)C=C2)C=C(C=C3)C(=O)C 90','prope
d.delete()
```

```
[[3, 90], [5, 100]]
```

Out[44]: 1

In [45]:
```
f=Molfile('/home/wdi/CACTVS/TESTFILES/VCH100.sdf')
print(f.scan('structure >= CN(=O)=O','recordlist'))
f.close()
```

```
[87, 169, 170, 171, 172, 207, 209, 423, 469, 585, 586, 880, 944, 978]
```

Out[45]: 1

In [13]:
```
# Molfile.Scan("<pubchem>", .....)
```

# The destination

Can we extend this query capability to work seamlessly also on external database - local ones, or in the cloud, to support universal scripting from single molecule tests to sifting through billions of compounds?

Idea: Translate these queries into pure SQL. Difficulty ranges from trivial (identity queries) to pretty tough (sub/superstructure queries).

No limits on client-side preprocessing. No toys like a SMILES parser in SQL. You are not expected to write these queries yourself.

If we are going to use pure SQL, databases tables with chemistry data will need to be basic, and the SQL portable.

Requirements:

- *bit_count()* function and bit operators on blobs or bitvectors, not just ints, for screening and similarity
- Basic array support, including the *value*=any(*array*) construct
- Robust support for chained and nested (and potentially recursive) CTEs
- Requirements are met by primary target cloud DB systems
- Infrastructure support for efficient data transfers (e.g. AVRO table file I/O support for Google BigQuery)

# The simple parts

Identity and similarity queries

In [46]:
```
q=Query('structure {= stereo} C1(N(N=CC(N1)=O)C3=CC(=C(C(C2=C(Cl)C=CC=C2)=O)C=C3)
q.set({'stereohash_property':'E_STEREO_HASH128','stereohash_field':'stereohash'})
print(q.sqlquery)
print(db.itemquery('select molregno from base where '+q.sqlquery))
# print(db.itemquery('select molregno from compound_query where stereohash = ens_stri
```

```
(stereohash = '309725E567E8CF85B68EA15C48C3F0C1')
1
```

In [47]:
```
q=Query('structure ~>= C1(N(N=CC(N1)=O)C3=CC(=C(C(C2=C(Cl)C=CC=C2)=O)C=C3)C)=O 95
print(q.query)
```

```
('prop', {'id': 100, 'level': 0, 'complexity': 20, 'negated': False, 'fieldname':
'E_SCREEN', 'threshold': 95, 'cmpflags': frozenset({'approximate', 'tanimoto'}),
'property': E_SCREEN, 'value': 1100000001111011001100000000000000000100000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000110000011000001000000000000000
0000000000000000000000000000000000000000000000101000000000000000000000000111100
00000100001100000000000000000000000000000001100000011101100000110011000001001000
0110011110000001000001011000000000000000000000101010000000001010110101011100110
10100000000000010010010000001000000000000100101000001110000000000011110100010000
00000010000000001100110110010001000100000100000001100101100000110010111100100011
00001000010000100001000011110001001110100100010010010001100110110100111000110001
0001001110000001001111010001000000000000000000010000000000000000010001000000000
0000000000100000000000000000000100000000000000000100010000000000000000000000000
000000000000000000000000000000, 'operator': 'ge'})
```

In [48]:
```
q.set('similarityscreen_field','simscreen')
sql1=q.sqlquery
print(sql1)
```

```
(bit_count(simscreen & B'110000000111101100110000000000000000010000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000110000011000001000000000000000000000
0000000000000000000000000000000000000000010100000000000000000000000000111100000001
0000110000000000000000000000000000000110000001110110000011001100000100100001100 1
1110000001000001011000000000000000000000010101000000000010101101010111001101010 00
0000000001001001000000100000000000010010100000111000000000001111010001000000000
1000000000110011011001000100010000010000000110010110000011001011110010001100001 0
0001000010000100001111000100111010010001001001000110011011010011100011000100100
1110000001001111010001000000000000000000000100000000000000000100010000000000000
0000100000000000000000000100000000000000000010001000000000000000000000000000000
0000000000000000000000000')*100/(bit_count(simscreen)+154-bit_count(simscreen &
B'110000000111101100110000000000000000010000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000011000001100000100000000000000000000000000000000000000000000
0000000000000000010100000000000000000000000001111000000010000110000000000000000
0000000000000011000000111011000001100110000010010000110011110000001000001011000
0000000000000000001010100000000001010110101011100110101000000000000100100100000 1
0000000000001001010000011100000000000011110100010000000000100000000001100110110010
0010001000001000000011001011000001100101111001000110000100001000010000100001111 0
0010011101001000100100100011001101101001110001100010001001110000001001111010001 0
0000000000000000000100000000000000000010001000000000000000000000100000000000000
```

In [49]:
```
q.set('flags','+ignorethresholds')
sql2=q.sqlquery
print(db.query('select molregno,'+sql2+' from base where'+sql1))
```

```
(('1', '100'), ('5', '96'), ('54', '96'), ('76', '95'), ('78', '95'), ('79', '95
'))
```

# And now for the heavy lifting...

**Reasonably efficient** substructure search in pure SQL.

Has this been done before?

Yes: *J. Chem. Inf. Model. 2009, 49, 1, 22–27*

What can we learn from it?

*How **not** to approach it.*

The straightforward (and published) method does not work for real-life queries and database structures! Not even if formulated in a bond-centric fashion and with structure or substructure segment equivalence preprocessing.

```
select a1.label,a2.label,...
   from atoms as a1, atoms as a2, bonds as b1 ... where
   a1.element=6 and a2.element=5 and b1=bond(a1.label,a2.label)
   and b.order=1...
```

This has a runtime of (m*n)**n (m=#structureatoms, n=#substructureatoms) without magic optimization (and it does not happen in the databases which I tested).

The test data set in the publication are 1000 cpds. And the sample substructures are small.

# A better way?

Perform the atom-by-atom matching by an explicitly scripted path. SQL is about tables, so let's do it with (temporary, transient) tables.

- Get all the matches of a (preferably exotic) substructure atom on the atoms for the current test structure in a table
- Use this table to match all possibilities to match one of the atoms connected to the first, together with the bond attributes of the link bond, generating a table of two-atom matches
- Repeat with other atoms connected to the first (yielding a 3, 4, ... column match table), and when these are all handled, that of the second etc. matched atom which have not yet been treated by neighbor expansion, until all atoms and bonds have been matched
- Special consideration for ring links, and disconnected query fragments
- This maps well to SQL CTE expressions
- Can be elegantly implemented in a Recursive CTE, or more efficiently as a sequence of #ssatom+2 chained simple CTEs

It is a non-recursive breadth-first algorithm, which is not optimal if you are only interested in a yes/no match, but does not hurt too often for real-life queries. On the other hand, it can directly provide a count of possible matches, which can be of interest.

In [52]:
```
q=Query('structure >= c1ccccc1C(=O)c1c(Br)cccc1')
q.set('keycolumn','molregno')
q.set('substructurescreen_property','E_QUERY_SCREEN','substructurescreen_field','
print(q.sqlquery)
```

```
((bit_count(screen & B'0000100001000010000100000000001000000000110100011000000001
1100110100100000000101000010000001010000000000000100000001000000000100001100100000
0100001000000000100000000000000110000000000000001010001000101000011000010000000000
0001001101001000001100000100000000000100000010001000010100001000000000000000000000
0010100000000000011000000110100100010001001100000001001100110000000000000000010000
0000011100101000110001000000000001000000000000000001100010001000001000100001101000
0000010010100000000000010000001000000010000010110000010000101100011101000000000001
1010000000000001000010000001000000000000001010001000000000000000100000000001001000
1000010010000000010000001100001010001000000100100100000010100000000000010100000000
0001000000000101000010100000100000000000110100000000100000000001001001001100100000
0001001000000000000100000000100000110000010000010001000000000001001100100000001000
0001011000000001100000001100100000000000000100010000100000000001000000001000000000
0000000000000010000000001000000000000100000000001000000010000010000000000000000000
0001000000000000001001110010000110000000000100000000001010000000000001010000100000
0010010001100100011000001000010100101110000000001000000010100000000100101000001000
0000000000010000000000000000001001100010000000000100000000100100000010000000010000
0001010001000000000001000000000000000100000000010000000000000011000011010001001110010
1001000000000000101100010010010000000100000100100000000000100000000000001000010000100
1100000000000000100100011000100000001001010000000010000000100000011000000000010000000
```

And now - a pure SQL substructure query on 1.8 mil cpds, with multiple results in less than a second:

In [51]:
```
print(db.itemquery('select count(*) from base where '+q.sqlquery))
```

```
36
```

In some cases, this can actually be *faster* than a cartridge. The toolkit query optimizer merges the screens of the two small substructures into a single, much more selective screen. That is an optimization which is not accessible to the SQL query optimizer which does not know about screen relationships.

In [38]:
```
q=Query('and {structure >= C#N} {structure >= c[SH]}')
q.set('keycolumn','molregno')
q.set('substructurescreen_property','E_QUERY_SCREEN','substructurescreen_field','
```

```
# print(db.itemquery("select count(*) from compound_query where match_substructure(sc
print(q.sqlquery)
```

```
(((bit_count(screen & B'0000000000000010000000000000000000000000000000100000000
0000000000000100000000000000000000000000000100000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000100
0000000000000010000010000000000000000000000100000000000000000000000000000000000
0000100000000000001000001000000000000000100001000000000000000000000000000001000
0000000001000000000000000000000000110000000000000000000000000000000000000000000
0000000001010000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000100010000000000000000000000000000000000
0000000000000000000000001001000010000000000000000000000000000000000100000000000
0000000000000100000000000000000000000000000100000000000100000000000000000000000
0000000000000000000000000000000000000000000011000000000000000000000000000000000
0000000000000000000000001000000000000000000000000000000000000000000000000000000
0000000000000010000000000000000000000000000000000000000000000000000000001000000
0000000000000000000000000000000000000000100000000000000000000000000000000000000
0000001000000000000000000000000000000000000000100000000000000000000000100000000
0000000000000000000000000000000000000000100010000000000000100000000100000000000
0000000000000000000000000000000000000000000000000000000000100000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000001000000
1000000000000000000000000000000000000000100000000000100100000000000000000000000
0000000000000000000000000000000000000000100000000000000000000000000000000000000
0000000000100000000000000100000000000000000000000000000000000000000000000000000
0000000000000000000010000010000000000000000000000000000000100000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000010
0000000000000000000000000000000000000000001101000000000000000000000000000000000
0000000000000000000010000000000000000000000000000000100000')=59) and (with tempa a
s (select * from atom where base.molregno=atom.molregno),tempb as (select * from
bond where base.molregno=bond.molregno), match1(l1) as (select a1.label from tem
pa as a1 where a1.element=7),match2(l1,l2) as (select l1,a2.label from match1, te
mpa as a2, tempa as a1, tempb as b1 where a1.label=l1 and a2.element=6 and l1=any
(a2.ligands) and a2.label<>l1 and (b1.label=any(a1.bonds) and b1.label=any(a2.bo
nds) and b1.order=3) limit 1) select count(*) from match2)>0)) and (((with tempa
as (select * from atom where base.molregno=atom.molregno),tempb as (select * from
bond where base.molregno=bond.molregno), match1(l1) as (select a1.label from tem
pa as a1 where a1.element=16),match2(l1,l2) as (select l1,a2.label from match1, t
empa as a2, tempa as a1, tempb as b1 where a1.label=l1 and a2.element=6 and a2.ar
oringcount>0 and l1=any(a2.ligands) and a2.label<>l1 and (b1.label=any(a1.bonds)
and b1.label=any(a2.bonds) and (b1.order=1 or b1.arocount>0))),match3(l1,l2,l3)
as (select l1,l2,a3.label from match2, tempa as a3, tempa as a2, tempb as b2 wher
e a2.label=l1 and a3.element=1 and l1=any(a3.ligands) and a3.label<>l1 and a3.lab
el<>l2 and (b2.label=any(a2.bonds) and b2.label=any(a3.bonds) and (b2.order=1 or
b2.arocount>0)) limit 1) select count(*) from match3)>0))
```

# Performance

Screening performance is almost identical to cartridge version.

Single-thread atom-by-atom match performance on a recent workstation CPU: 2.5-10K complete substructure match attempts per second.

Performance loss factor vs.cartridge: 1-15, depending on query.

For interactive queries (reasonable but not great screening efficiency, do not want to see more than 10K hits, and an answer in 30 secs) this translates to 5-20 mil cpds per table shard and thread.

For scripted queries (willing to wait a few mins, but not hours), table shards of up to 100 mil cpds are reasonable.

# Implementation status

- Fullstructure with/without stereochemistry, isotopes, tautomers
- Substructure (simple hit check)
- Superstructure
- Formula (extended CONQUEST syntax)
- Similarity (not just Tanimoto)
- Near complete SMARTS support (and indirectly of other query spec formats)

Yet missing but probably important functionality:

- Stereochemistry matching
- Recursive SMARTS
- Substructure match mode manipulation
- Reaction queries
- R-group variability

(these are all supported in the non-SQL query system)

In [15]:
```
Dbase.Close('all')
Query.Delete('all')
```

Out[15]:
```
2
```

In [ ]: