Daylight Chemical Information Systems, Inc.
120 Vantis - Suite 550 - Aliso Viejo, CA 92656
tel +1 949-831-9990 - fax +1 949-831-9902 - www.daylight.com

Daylight Chemistry Cartridge for Postgresql

Table of Contents

Daylight Chemistry Cartridge for Postgresql

Daylight Version 4.9
Release Date 08/01/11

DAYLIGHT Chemical Information Systems, Inc. Laguna Niguel, CA USA

6. Function/Option Summary

1. Installation

A quick overview of the installation process can be found in the Daylight Installation Guide. This section is intended to give detailed installation instructions.

1.1 Installation of the Daylight Tar File

To install Daycart, the Daylight tar distribution must be unpacked. No other installation is required for Daycart. If one wishes to use other programs/tools within the Daylight distribution the relevant installation must be performed. Daylight is, by convention, unpacked into /usr/local/daylight/v495 and should be owned by the 'thor' or 'daylight' accounts. This location is defined as DY_ROOT. The Daylight distribution contains two directories that are relevant to the daycart install: 'pgcart', which has the installation SQL scripts and 'lib' which has the required shared object libraries.

1.2 Postgresql Installation

One can perform a generic Postgresql installation following the usual procedures. There are no specific configuration requirements of the Postgresql install, however there are two configuration settings which will make Daycart run more smoothly:

- Use C locale when initializing the database. This is necessary for Postgresql to perform text sorting and hence is used for indexed search terms: "like", ">", "&gt=;", "<", "&lt=;".

  The locale is set during the initdb step with: `initdb -D <database_directory> --locale=C`
- Set the "standard_conforming_strings" parameter to "on" in postgresql.conf. This parameter controls whether ordinary string literals treat backslashes ("\") literally, as specified in the SQL standard, or as escape characters, which was the historical usage.

  Prior to Postgres 9.1, the default setting was "off" which means that a backslash is treated as an escape character. Since the backslash is used in SMILES to indicate double-bond stereochemistry, it is preferred to disable the escaping of the backslash character and adopt the SQL standard behavior.

Note that this parameter does not apply to programatic interfaces such as JDBC, which have their own string handling/escaping conventions.

1.3 Schema and Function Creation

In order to use Daycart functions, each database within an instance must have the dcischem schema defined. The schema must contain a number of tables which control the licensing and behavior of the Daycart functions. Each Daycart user must have select privilege on the tables. It is convenient to define a new database role which grants the privileges as a set. Connect to the desired Postgresql database as the root user (typically postgres). Execute the '$DY_ROOT/pgcart/create_pg_schema.sql' script within the session:

```
db=# \i create_pg_schema.sql
```

Next, insert the Daycart license in the dcischem.license table. Contact Daylight at license@daylight.com if you don't have a license.

NOTE: The license key is required in order to run the installation of the Postgresql function creation.

Daycart include multiple versions of libpgcart.so in each release. One or more of the following files will be found in your Daylight distribution (depending on the operating system and version):

```
  $DY_ROOT/lib/libpgcart.so.84        32-bit ddlib.so for Postgresql 8.4.x
  $DY_ROOT/lib/libpgcart.so.90        32-bit ddlib.so for Postgresql 9.0.x
  $DY_ROOT/lib64/libpgcart.so.84      64-bit ddlib.so for Postgresql 8.4.x
  $DY_ROOT/lib64/libpgcart.so.90      64-bit ddlib.so for Postgresql 9.0.x
```

Run the command 'pg_config --pkglibdir'. This prints the default library location within your postgresql installation. Move the appropriate file into this directory as libpgcart.so.

Execute the script 'create_pg_functions.sql' as the root user in the desired database:

```
db=# \i create_pg_functions.sql
```

By default the create_pg_functions.sql script points to '$libdir/libpgcart.so' when defining the Daycart functions. The variable $libdir is specific to ones postgres installation and can be discovered by running 'pg_config --pkglibdir'. If one wishes to keep libpgcart.so in a different location, one must edit the create_pg_functions.sql file and change *all* references to $libdir/libpgcart.so to the desired path.

The only Postgresql-specific configuration needed is for handling of backslashes in psql. By default

1.4 Testing the Installation

Simple queries can be used to test the cartridge.

```
db=# select getinfo('daycart_version');

 getinfo
---------
 4.95

db=# select testlicense('daycart');

 testlicense
```

```
-------------
           1

db=# select smi2cansmi('NCC', 0);

 smi2cansmi
------------
 CCN
```

If the above queries don't give the expected results, see the section on troubleshooting.

1.5 Installation Troubleshooting

The Postgresql cartridge installation is fairly straightforward. The most common problem people run into is permissions; a non-privileged user doesn't have access to the tables within the dcischem schema. When the user attempts to run a function, one gets the error:

```
ERROR:  relation "dcischem.license" does not exist at character 67
QUERY:  SELECT expiration, key, product, organization, site, address
        from dcischem.license where expiration > current_date
```

In this case, verify that the schema dcischem exists *within the current database* and that the schema contains the license, options, and ptable tables. Furthermore, make sure that usage privileges have been granted on the schema to public (or to a named role) and that select privileges have been granted on the three tables to public (or to the named role). If using roles, verify that the current user has been granted to named role.

1.6 Verification

The subdirectory 'pgcart/TEST' includes a stand-alone script which tests the cartridge installation. The shell script 'test.sh' will execute the test.sql SQL script and compare the output to reference output file. Any discrepencies will be reported. The script 'test.sh' attempts to log in without specifying a userid/password with a default database. Depending on the local Postgresql configuration, one may need to edit the script and set the PG_HOME location, a username and database.

Note also that the script attepts to add data to the dcischem.options and dcischem.ptable tables in order to test their operation. If the user running test.sh does not have insert and delete privilege on these two tables then parts of the testing will fail. One will see the following output:

```
psql:test.sql:15: ERROR:  permission denied for relation options
psql:test.sql:18: ERROR:  permission denied for relation options
psql:test.sql:109: ERROR:  permission denied for relation ptable
psql:test.sql:130: ERROR:  permission denied for relation ptable
67a68
> INSERT 0 1
74c75
<        0
---
>        1
83c84
<  TRUE
---
>  FALSE
89a91
> DELETE 1
479a482
```

```
> INSERT 0 1
509a513
> DELETE 1
```

If these are the only failures then the installation is certainly fine. Granting insert and delete privilege on the two tables will allow the tests to complete.

Note that the SQL script gives useful examples of using the cartridge functionality in each category.

1.7 De-Installation

In order to cleanly de-install the cartridge, one simply runs the clean_pg.sql script. The script removes all cartridge functions. The dcischem schema and its three tables (license, options, ptable) are preserved for convenience.

It is safe to repeatedly run the scripts create_pg_functions.sql and clean_pg.sql.

When one runs create_pg_schema.sql after clean_pg.sql, one will see warnings that the tables license, options, and ptable already exist, however these warnings don't impact the outcome of the cartridge re-installation.

After running clean_pg.sql , the database will be left in its original pre-cartridge state. One can then delete the license, options, and ptable tables and dcischem schema, if desired, to eliminate all remnants of the cartridge.

2. SQL Functions

A number of user-defined SQL functions have been implemented for the cartridge. All of the functions are defined in the current database during the installation process.

The default installation (pg_create.sql) allows access to the functions by any user of the database.

2.1 String Data Handling

Daycart supports TEXT string datatypes for all string arguments. Postgresql will transparently convert VARCHAR and CHAR types into TEXT to execute Daycart functionality.

2.2 General Purpose Functions

Thg general purpose functions are typically used for debugging or setting session-level options.

geterrors

```
function geterrors (INTEGER level) => TEXT
```

Returns error strings from the error queue for previously failed functions or operators based on the level requested and clears the error queue. By default only message at the level of Errors or above are sent to the screen when a function fails.

```
0 - All messages
1 - Notes
2 - Warnings
3 - Errors
```

```
    4 - Fatal errors
```

getlog

```
function getlog => TEXT
```

Returns error strings from the log messages and clears the local buffer of log messages. Behavior is controlled by the session-level option 'log'.

In order to get log messages from the getlog() function the 'log' option must be set to either 'LOCAL' or 'BOTH'. From that point log messages will be kept and can be retrieved with the getlog() function. By default, the 'log' option is set to 'CENTRAL' and all log messages go to the central logfile, which by default is in the databases pg_log directory.

testlicense

```
function testlicense (TEXT product) => INTEGER
```

Checks the license. The license is contained in a special table is created at install time. Currently the only recognized value for product is 'daycart'. Returns 1 if the Daylight cartridge has a valid license and 0 if not.

getinfo

```
function getinfo (TEXT which) => TEXT
```

Returns informational strings from DayCart. Valid input parameters are listed below. In addition, getinfo can be used to find the value for any dayconvert option. See Section 2.3 Molecule / Reaction Functions for detailed descriptions of the dayconvert options.

`'toolkit_version'`

Returns the current Daylight Toolkit version.
`'daycart_version'`

Returns the cartridge executable version.
`'extproc_pid'`

Returns the extproc process ID for this DayCart session.
`'hostid'`

Returns the hardware hostid used for generation of the license key.
`'debug_level'`

Returns the debug level set for logging messages.
`'session_tag'`

Returns the session tag set for a given session. The session tag, if set, is included in any log messages and can be used to identify the session which generated a log entry.
`'log'`

Returns where the session log info is written. Choices are NONE, LOCAL, CENTRAL, and BOTH.

CENTRAL is the default, and refers to the value of DAYCART_LOGFILE (in the listener.ora) or a default of /tmp/extproc.log. LOCAL logging indicates that any log messages are stored for retrieval by the ddpackage.fgetlog() function.

`'default_delimiter'`

Returns the default delimiter used to separate lines of multi-line output.

`'force_delimiter'`

Returns TRUE or FALSE. When TRUE, the default delimiter is always used for multi-line output. When FALSE, Daycart may attempt to detect the delimiter to use (from other input).

setinfo

```
function setinfo (TEXT name_value_pair) => NUMBER
```

Allows the user to individually set session-level options in DayCart for the following parameters and all dayconvert options (see Section 2.3 Molecule / Reaction Functions).

`'debug_level={level}'` valid values are integers 0 - 9
`'session_tag={value}'` any string, truncated to 32 chars
`'log={place}'` valid values are NONE, LOCAL, CENTRAL, BOTH
`'default_delimiter={string}'`
`'force_delimiter={TRUE|FALSE}'`
`'options={class}'` valid values correspond to options table

The function setinfo can be used to globally set options on a session-level basis. An OPTIONS table which can be populated with user-defined sets of parameters is automatically created during installation. Executing setinfo using `'options={class}'` will reset all options to their default values and then will set the values associated with the given class. On session startup, the options with class of zero are set.

```
Name                                     Type
---------------------------------------- ----------------------------
NAME                                     VARCHAR(100)
VALUE                                    VARCHAR(100)
CLASS                                    INTEGER

NAME = parameter name as listed above or any of the dayconvert options
VALUE = new value
CLASS = set number for options.
```

2.3 Molecule / Reaction Functions

Functions and their respective operations relating to conversion, transformation. property values and normalization and of molecules and reactions are described below.

smi2cansmi

```
function smi2cansmi (TEXT smiles, INTEGER type) => TEXT
```

Returns a canonical SMILES string from an input SMILES. Type is either 0 or 1, for unique or absolute SMILES, respectively.

smi2xsmi

```
function smi2xsmi (TEXT smiles, INTEGER type, INTEGER explicit) => TEXT
```

Returns an exchange SMILES string which is semantically identical to the input SMILES but which does not use Daylight-specific aromaticity conventions. Type is either 0 or 1, for unique or absolute SMILES, respectively. When 'explicit' is 1, it supplies hydrogen count and other atomic properties explicitly for every atom. Note that exchange SMILES are not canonical; the same input molecule or reaction may return different exchange SMILES depending on the input order to this function.

smi2netch

```
function smi2netch (TEXT smiles) => INTEGER
```

Returns the net charge of the input molecule or reaction.

smi2hcount

```
function smi2hcount (TEXT smiles) => INTEGER
```

Returns the total hydrogen count for the input molecule or reaction.

smi2mf

```
function smi2mf (TEXT smiles) => TEXT
```

Returns the molecular formula string for the input molecule or reaction.

fsmi2amw

```
function smi2amw (TEXT smiles) => DOUBLE PRECISION
```

Returns the average molecular weight for the input molecule or reaction. The weight used for any atoms which do not have specified isotopes is the average atomic weight. The weight used for atoms with a specified isotope is the high precision molecular weight for that atom. For example, "c1ccccc1" returns 78.1184, while "[1H][12c]1[12c]([1H])[12c]([1H])[12c]([1H])[12c]([1H])[12c]1[1H]" returns 78.0469502.

smi2pmw

```
function smi2pmw (TEXT smiles) => DOUBLE PRECISION
```

Returns the high-precision molecular weight for the input molecule or reaction. The weight used for any atoms which do not have specified isotopes is the high precision weight for the most abundant isotope. The weight used for atoms with a specified isotope is the high precision molecular weight for that atomic isotope. For example, "c1ccccc1" returns 78.0469502, the high precision molecular weight. "BrBr" returns 157.836675, while "[81Br][81Br]" returns 161.832582.

Both smi2amw() and smi2pmw() return the same values for SMILES with fully specified isotopes; they only differ in their handling of SMILES with unspecified isotopic weights. Note also that the unique canonical SMILES (eg. smi2cansmi('smiles', 0)) returns the structure with all isotopic information removed; this is useful for consistent handling of partially specified isotopic information in conjunction with the two functions.

smi2graph

```
function smi2graph (TEXT smiles) => TEXT
```

Returns the hydrogen- and charge-suppressed canonical graph string for the input molecule or reaction.

vcs_desalt

```
function vcs_desalt(TEXT smiles, INTEGER type, INTEGER class) => TEXT
```

Removes molecule fragments found in the salts table from the input SMILES. Type is either 0 or 1, for unique or absolute SMILES, respectively. The class value is the class of salts entries used from the salts table. All of the structures in the salts table with the given class are checked against the input SMILES and if found, they are removed.

vcs_normalize

```
function vcs_normalize(TEXT smiles, INTEGER type, INTEGER class) => TEXT
```

Performs a SMIRKS-based structure normalization on the input SMILES. All of the SMIRKS from the transform table with the given class are applied to the input molecule. The resulting molecule is output as a canonical SMILES. Type is either 0 or 1 for unique or absolute SMILES, respectively.

gen_molecules
gen_reactions

```
function gen_molecules(TEXT smiles, TEXT smirks,
        INTEGER direction, INTEGER limit, INTEGER type) => TEXT
function gen_reactions(TEXT smiles, TEXT smirks,
        INTEGER direction, INTEGER limit, INTEGER type) => TEXT
```

Applies the transform created from the given SMIRKS to the input molecules. The argument direction indicates that the transform is applied in the forward (0) or reverse (1) direction. The limit parameter is the maximum count of specific molecules to return and 0 indicates no limit. The resulting molecules or reactions are output as a set of newline-delimited canonical SMILES. The gen_molecules() function and operator returns the molecules which result from a transformation; the gen_reactions() function and operator return the complete reaction. The type indicates either 0 or 1 for unique or absolute SMILES, respectively.

The results are returned as a single string. If more than one molecule or reaction is formed, then each is on a separate line of the output, delimited by the default_delimiter string (see info().

dayconvert

```
function dayconvert (TEXT data, TEXT ifmt, TEXT ofmt) => TEXT
function dayconvert (TEXT data, TEXT ifmt, TEXT ofmt,
    INTEGER ptable_class) => TEXT
```

Returns the input chemical information in a different format. See the Daylight Conversion Manual for additional information. Note: The 'ptable_class' parameters are optional as described below.

The 'ifmt' and 'ofmt' parameters are used to designate the input and output formats, respectively. Both parameters need to be identified by particular letter sequence. Valid combinations of input and output formats for conversion are as follows where tdtsma and tdtsmrk are tdt versions with smarts and smirks, respectively:

```
        smi ---> mol/sdf/rdf
```

```
tdt ---> mol/sdf/rdf
mol or sdf ---> smi/ism/sma/tdt/tdtsma
rdf ---> smi/ism/sma/smrk/tdt/tdtsma/tdtsmrk
```

Either mol or sdf can be used for rgfile input.

Delimiters for interpreting multi-line input data are detected from the input stream. Input delimiters may be the strings "\n", "\r", or "\r\n". On output, the delimiter chosen for multi-line output depends on the input delimiter detected and the settings of "force_delimiter" and "default_delimiter". See the description of these items in the documentation for the info(). function.

The ptable_class parameter is optional. If the 'ptable_class' parameter is provided, it indicates that valence and charge information in the user-defined PTABLE table information is to be used instead of that provided in the default ptable. The specific information to be used is based upon the class number supplied.

```
Name                                    Type
--------------------------------------- ----------------------------
AT_NO                                   INTEGER
SYMBOL                                  VARCHAR(8)
AT_MASS                                 INTEGER
VALENCE_CHARGE_LIST                     VARCHAR(4000)
CLASS                                   INTEGER

AT_NO = atomic number
SYMBOL = atomic symbol
AT_MASS = atomic mass
VALENCE_CHARGE_LIST = list of valence and charge, e.g., '2,-1,3,0'
        for valence 2 with -1 change or valence 3 with 0 charge
```

The following are a series of dayconvert related options. The value for any of these options can be found using the finfo() function. Values for these options can be changed on a per session basis for an individual option or a group of options using the fsetinfo() function as described in Section 2.2 General Purpose Functions.

'conv_smi_is_ism={TRUE|FALSE}'

Default is FALSE. When set to TRUE, conversions to tdt format include the isomeric SMILES in the root $SMI datatype

'conv_add_3d={TRUE|FALSE}'

Default is TRUE. When TRUE, conversion to tdt format includes 3D coordinates in the output file if present in the input.

'conv_add_2d={TRUE|FALSE}'

Default is TRUE. When TRUE, conversion to tdt format includes 2D coordinates in the output file if present in the input.

'conv_use_3d={TRUE|FALSE}'

Default is FALSE. When TRUE, conversion from tdt format includes 3D coordinates in the output if present in the input.

'conv_split_fields={TRUE|FALSE}'

Default is FALSE. Controls handling of multi-line data from sdf and tdt files. TRUE causes these to be

split into separate dataitems.
`'conv_id_field={id_field}'`

Sets the id field to be used. The default behavior for molecules being converted to tdt format is to use the first line of each header block as the id field. The default behavior for reactions is to use the value following $RIREG as the id.
`'conv_prefix={prefix}'`

Sets the designated prefix to be parsed from the datatype names for conversion of rdf files. The default is none.
`'conv_implicit_chirality={TRUE|FALSE}'`

Default is `FALSE`. Alters the way in which chirality is determined in order to detect implicit chiral centers. Useful for some natural products. For a bond A-hash-B, the interpretation is that B is below A from the perspective of A and A is above B from the perspective of B. `TRUE` causes both ends of the chiral bonds to be used in the determination of chiral centers.
`'conv_ring_cistrans={TRUE|FALSE}'`

Default is `FALSE`. Setting this option as `TRUE` ignores cis/trans stereochemistry for all ring double bonds.
`'conv_db_explicit_h={TRUE|FALSE}'`

Default is `FALSE`. When `TRUE` requires that double bonds have all hydrogens explicitly indicated for conversion from non-query MDL format to SMILES.
`'conv_chi_explicit_h={TRUE|FALSE}'`

Default is `FALSE`. When `TRUE` chiral atoms have all hydrogens explicitly indicated for conversion from non-query MDL format to SMILES.
`'conv_fix_radical_rings={TRUE|FALSE}'`

Default is `TRUE`. When `TRUE` allows for the certain types of five, six and seven-membered radical rings to be converted to aromatic. When `FALSE` keeps the ring as specified in the input MDL file. In order for a ring to be converted all atoms in the ring must be carbon and doublet radical. In addition no atom in the ring may have a charge.
`'conv_nametag={tag}'`

Sets the tdt datatype tag used for the id. Default is $NAM.
`'conv_comment_smi={TRUE|FALSE}'`

Default is `FALSE`. If `TRUE` output SMILES is written into the comment line of the header block in each connection table.
`'conv_smi_tuples={TRUE|FALSE}'`

Default is `TRUE`. When `FALSE` the tuple information associated with the isomeric SMILES field in a tdt is printed in the output MDL file.
`'conv_day_hcount={TRUE|FALSE}'`

Default is `TRUE`. When `TRUE` the program uses both explicit H and H-count field for the conversion of query MDL files.
`'conv_day_stereo={TRUE|FALSE}'`

Default is TRUE. When TRUE the program uses stereochemistry as specified for the conversion of query MDL files. When FALSE the program uses stereochemistry as specified and unspecified.
'conv_day_chih={TRUE|FALSE}'

Default is TRUE. When TRUE chiral hydrogens are required to be explicit for the conversion of query MDL files. When FALSE implicit hydrogens are used.

atomnorm
bondnorm
partnorm

```
function atomnorm (TEXT smiles, TEXT list, INTEGER ntuple,
    INTEGER isotype) => TEXT
function bondnorm (TEXT smiles, TEXT list, INTEGER ntuple,
    INTEGER isotype) => TEXT
function partnorm (TEXT smiles, TEXT list, INTEGER ntuple,
    INTEGER isotype) => TEXT
```

Returns a potentially reordered N-tuple string for the given list input parameter. The list string is a comma-separated list of data which is associated in order with the atoms, bonds or parts of the input SMILES. The list string is reordered based on the canonical atom, bond, or part ordering of the input SMILES. ntuple is the number of comma-separated values per atom, bond, or dot-separated part, and isotype is 0 for unique SMILES canonicalization and 1 for absolute SMILES canonicalization.

2.4 Fingerprint Functions

This section describes functions and their respective operations relating to generation of and information concerning fingerprints.

smi2fp

```
function smi2fp (TEXT smiles, INTEGER min, INTEGER max,
    NUMBER nbits) => TEXT
```

Returns the ASCII fingerprint for a given molecule or reaction. Min and max are the minimum and maximum pathlengths, respectively, and nbits is the number of bits in the fingerprint.

smi2xfp

```
function smi2xfp (TEXT smiles, INTEGER min, INTEGER max,
    NUMBER nbits) => TEXT
```

Returns the ASCII difference fingerprint for a given molecule or reaction. Min and max are the minimum and maximum pathlengths, respectively, and nbits is the number of bits in the fingerprint.

smi2screen

```
function smi2screen (TEXT smiles) => TEXT
```

Returns an ASCII string which contains screening data for a given molecule or reaction. The screen data can be used by the contains() and matches() functions for additional pre-screening.

foldfp

```
function foldfp (TEXT fpstr, INTEGER nbits, DOUBLE PRECISION dens) => TEXT
```

Folds the given fingerprint to the minimum appropriate size or density, whichever is limiting, and returns the new, folded fingerprint.

bitcount

```
function bitcount (TEXT fpstr) => INTEGER
```

Returns the number of bits on in the fingerprint.

nbits

```
function nbits (TEXT fpstr) => INTEGER
```

Returns the total size of the fingerprint, in bits. In the current Daylight toolkit this will always be a power of two.

isfp

```
function isfp (TEXT fpstr) => INTEGER
```

Returns 1 if the string is a fingerprint, 0 otherwise. The syntax of a fingerprint can never be confused with a valid SMILES. This is the bit of cleverness which allows us to overload the searching functions.

2.5 Comparison Functions

Functions and their respective operations relating to a variety of direct comparisons between input smiles, smarts, or fingerprints are described below.

graph

```
function graph (TEXT smiles1, TEXT smiles2) => INTEGER
```

Returns 1 if the two input SMILES share the same canonical graph, 0 otherwise.

tautomer

```
function tautomer (TEXT smiles1, TEXT smiles2) => INTEGER
```

Returns 1 if the two input SMILES share the same canonical graph, net charge, and total hydrogen count, 0 otherwise.

usmiles

```
function usmiles (TEXT smiles1, TEXT smiles2) => INTEGER
```

Returns 1 if the two input SMILES share the same unique canonical smiles, 0 otherwise.

asmiles

```
function asmiles (TEXT smiles1, TEXT smiles2) => INTEGER
```

Returns 1 if the two input SMILES share the same absolute canonical smiles, 0 otherwise.

component

```
function component (TEXT smiles1, TEXT smiles2) => INTEGER
```

Returns 1 if smiles2 (a molecule SMILES) is a component of smiles1 (any SMILES), otherwise returns 0. Also returns false if smiles2 is not a single-component query. The last example in the table below illustrates this problem. A component is a single dot-separated part of a larger molecule or reaction SMILES. Some examples follow:

smiles1	smiles2	Returns
CCC	CCC	1
CCC.CCCN	CCC	1
CCC.CCCN	CCCN	1
CCC>>CCCN	CCC	1
CCC>>CCCN	CCCN	1
CCC.CCCN	CCC.CCCN	0

reactant
agent
product

```
function reactant (TEXT smiles1, TEXT smiles2) => INTEGER
function agent (TEXT smiles1, TEXT smiles2) => INTEGER
function product (TEXT smiles1, TEXT smiles2) => INTEGER
```

Returns 1 if smiles2 (a molecule SMILES) is a component of smiles1 (a reaction SMILES) with the appropriate role, otherwise returns 0.

contains

```
function contains (TEXT smiles1, TEXT smiles2) => INTEGER
function contains (TEXT smiles1, TEXT smiles1_screen,
    TEXT smiles2) => INTEGER
```

Returns 1 if smiles1 contains smiles2; that is, smiles2, assuming opened valences for all hydrogens, is a substructure of smiles1.

The additional smiles1_screen argument can be used to pass in pre-computed screening data. The data should be computed and stored in advance of a search using the smi2screen() function. This data is used for fast searching of large tables as it allows Postgresql to perform some screening operations which speed up searching.

isin

```
function isin (TEXT smiles1, TEXT smiles2) => INTEGER
```

Returns 1 if smiles2 contains smiles1; that is, smiles1, assuming opened valences for all hydrogens, is a substructure of smiles2. This functionality is identical to 'contains()' with the arguments swapped.

matches

```
function matches (TEXT smiles, TEXT smarts) => INTEGER
function matches (TEXT smiles, TEXT smiles_screen, TEXT smarts) => INTEGER
```

Returns 1 if the smarts expression matches the given SMILES, 0 otherwise.

The additional smiles1_screen argument can be used to pass in pre-computed screening data. The data should be computed and stored in advance of a search using the smi2screen() function. This data is used for fast searching of large tables as it allows Postgresql to perform some screening operations which speed up searching.

matchcover

```
function matchcover (TEXT smiles, TEXT smarts) => DOUBLE PRECISION
```

Returns the ratio of atoms in the target SMILES matched by the given query to the number of atoms in the target as a number between zero and one.

euclid

```
function euclid (TEXT fp_or_smi1, TEXT fp_or_smi2) => DOUBLE PRECISION
```

Returns the euclidean distance between two fingerprints or SMILES. If both parameters are fingerprints and are not the same size (nbits()), then the larger will be folded automatically to match the size of the smaller before comparison. If one parameter is a SMILES, its fingerprint is generated automatically at the size of the other parameter. If both parameters are SMILES, then a fingerprint size of 512 bits is used. The returned value is a floating point number between 0.0 and 1.0.

tanimoto

```
function tanimoto (TEXT fp_or_smi1, TEXT fp_or_smi2) => DOUBLE PRECISION
```

Returns the tanimoto distance between two fingerprints or SMILES. If both parameters are fingerprints and are not the same size (nbits()), then the larger will be folded automatically to match the size of the smaller before comparison. If one parameter is a SMILES, its fingerprint is generated automatically at the size of the other parameter. If both parameters are SMILES, then a fingerprint size of 512 bits is used. The returned value is a floating point number between 0.0 and 1.0.

tversky

```
function tversky (TEXT fp_or_smi1, TEXT fp_or_smi2,
    DOUBLE PRECISION alpha, DOUBLE PRECISION beta) => DOUBLE PRECISION
```

Returns the tversky distance between two fingerprints or SMILES. If both parameters are fingerprints and are not the same size (nbits()), then the larger will be folded automatically to match the size of the smaller before comparison. If one parameter is a SMILES, its fingerprint is generated automatically at the size of the other parameter. If both parameters are SMILES, then a fingerprint size of 512 bits is used. The returned value is a floating point number between 0.0 and 1.0.

fingertest

```
function fingertest (TEXT fp_or_smi1, TEXT fp_or_smi2) => DOUBLE PRECISION
```

Returns 1 if all of the bits in fp_or_smi2 are also present in fp_or_smi1. That is, the fingerprint from fp_or_smi2 represents a possible substructure of fp_or_smi1. If both parameters are fingerprints and are not the same size (nbits()), then the larger will be folded automatically to match the size of the smaller before comparison. If one parameter is a SMILES, its fingerprint is generated automatically at the size of the other parameter. If both parameters are SMILES, then a fingerprint size of 512 bits is used. The returned value is a floating point number between 0.0 and 1.0.

similarity

```
function similarity (TEXT fp_or_smi1, TEXT fp_or_smi2,
    TEXT expression) => DOUBLE PRECISION
```

Returns the similarity or distance between two fingerprints or SMILES, based on the computed expression. If both parameters are fingerprints and are not the same size (nbits()), then the larger will be folded automatically to match the size of the smaller before comparison. If one parameter is a SMILES, its fingerprint is generated automatically at the size of the other parameter. If both parameters are SMILES, then a fingerprint size of 512 bits is used. The returned value is a floating point number.

The value of expression can be any legal expression based on the counts of bits in the corresponding fingerprints (see the man page for expression(5)). Optionally, the expression string can be preceded with either "distance=" or "similarity=", which are ignored. The following examples are all identical:

```
  similarity(smi1, smi2, 'TANIMOTO')
  similarity(smi1, smi2, 'tanimoto')
  similarity(smi1, smi2, 'similarity=tanimoto')
  similarity(smi1, smi2, 'c/(a+b+c)')
  similarity(smi1, smi2, 'similarity=c/(a+b+c)')
```

3. Searching:

3.1 General Comments

At present, Postgresql does not support a viable interface for adding custom index capabilities. It's index customization capabilities are cumbersome. Fortunately, calling individual custom functions within Postgresql is very fast, on the order of one million calls per second. This allows one to perform reasonable searching operations using very simple row-at-a-time SQL queries. This section discusses how to best implement various types of queries within Postgresql using a combination of indexes and row-at-a-time functions.

3.2 Exact Searching

An 'exact structure' search can be implemented as an index query against a canonical SMILES column. One should store structures and reactions in their canonical form, with isomeric information (eg. use smi2cansmi(smiles, 1)).

The following simple example creates a table with a SMILES column and a simple built-in text index on the SMILES column. The insert normalizes the SMILES to canonical form.

```
SQL> create table mytable (id number, smiles text);
```

```
SQL> create index mytable_smi_i on mytable(smiles);
SQL> insert into mytable (1, smi2cansmi('NCCC', 1));
```

In order to retrieve exact structure matches, a query of the following type is used:

```
SQL> select * from mytable where smiles = smi2cansmi(query, 1);
```

By canonicalizing both the stored SMILES and the query SMILES, a simple text lookup can be used to find exact SMILES matches. That being said, the next section describes the general approach of using a reduced graph as the index lookup key. The reduced graph is more flexible as it allows one to search for either exact matches, tautomers, or graphs with a common index and query idiom.

3.3 Graph and Tautomer Searching

The approach here is to store the canonical graph as an additional column for each table which contains either structures or reactions within the database. A query on the graph column will return the set of molecules or reactions which share the same graph. These can then be filtered with additional query terms to provide fast searching for isomeric and unique exact lookup, tautomer lookup, and graph lookup.

The following simple example creates a table with a SMILES column and an additional graph column. The graph column has a simple built-in text index. The insert normalizes the graph using smi2graph(), which generates a canonical reduced graph from a molecule or reaction SMILES.

```
SQL> create table mytable (id number, smiles text, graph text);
SQL> create index mytable_grf_i on mytable(graph);
SQL> insert into mytable (1, 'NCCC', smi2graph('NCCC'));
```

Note that these queries do not require that the SMILES column be canonicalized, nor do they use an index on the SMILES column. All of the following example queries use the index against the graph column plus row-at-a-time operations on the subset of rows which match the graph step.

The following example queries can be used against this pair of database tables:

```
-- Find same Isomeric SMILES
--
SQL> select * from mytable
      where graph = smi2graph(query)
      and asmiles(smiles, query) = 1;
--
-- Find same Unique SMILES
--
SQL> select * from mytable
      where graph = smi2graph(query)
      and usmiles(smiles, query) = 1;
--
-- Find same Tautomers
--
SQL> select * from mytable
      where graph = smi2graph(query)
      and tautomer(smiles, query) = 1;
--
-- Find same Graph
--
SQL> select * from mytable
      where graph = smi2graph(query);
```

A way to conceptualize these queries is as follows: for any query, one can quickly find the set of rows in the database with the same reduced graph. Once one has the set of rows with the same reduced graph, one can further prune that set to find the tautomers, isomers, etc.

3.4 SMARTS and Similarity Searching

The arguments to the tanimoto(), tversky(), euclid(), similarity(), and fingertest() functions can be either a fingerprint, screenfp, or SMILES. All of these functions actually perform their computations on Daylight fingerprints. When needed, the functions will use the input SMILES to create a normal fingerprint of the appropriate size for comparison. Fingerprint creation, however is expensive, so for efficient queries one should precompute and store the screenfp (which contains the fingerprint) with the SMILES and use the screenfp exclusively in queries.

For substructure search, using the contains() and matches() functions, there are special forms of these two functions which take an extra screenfp argument. This allows the search code to use the fingeprint and additional statistics contained in the screenfp data for filtering of the rows. This dramatically improves the efficiency of substructure searching.

```
SQL> create table mytable (id number, smiles text, screenfp text);
SQL> insert into mytable (1, 'NCCC', smi2screen('NCCC'));
```

Note that these queries do not require that the SMILES column be canonicalized, nor do they use an index at all. These are the most resource-intensive queries as they perform a full tablescan for every query (unless other non-chemistry query clauses can prune). Typically, tables of several hundred thousand rows are easily handled using this approach.

The following example queries can be used against this pair of database tables:

```
-- Find structures by tanimoto similarity:
--
SQL> select * from mytable
      where tanimoto(screenfp, smi2fp(query, 0, 7, 512)) > 0.9;
--
-- Find structures by euclidean distance:
--
SQL> select * from mytable
      where euclid(screenfp, smi2fp(query, 0, 7, 512)) <0.1;
--
-- Find matching structures of a SMILES query
--
SQL> select * from mytable
      where contains(smiles, screenfp, smiles_query) = 1;
--
-- Find matching structures of a SMARTS query
--
SQL> select * from mytable
      where matches(smiles, screenfp, smarts_query) = 1;
```

3.5 Best Practices

This section will describe a recommended organization of ones structure tables. It is devised to provide complete searching functionality with the minimum set of columns and indexes. It's a simple combination of the graph and screenfp cases described earlier in this section.

```
SQL> create table mytable (id number, smiles text,
                           graph text, smiscreen text);
SQL> create index mytable_grf_i on mytable(graph);
SQL> insert into mytable (1, newsmi',
                          smi2graph(newsmi), smi2screen(newsmi));
```

Optionally, one may wish to canonicalize the smiles column. In particular, if one wishes to use a database uniqueness constraint to prevent duplicate SMILES from being stored, one should always canonicalize the SMILES on insert and then set the uniqueness constraint on the SMILES column.

Under this organization, all of the queries described in the 'Graph and Tautomer Search' section, as well as all of the queries in the 'SMARTS and Similarity Search' section may be used. Note that the queries in the 'Exact Search' section are covered by the graph searches.

4. Error Codes

This section lists the possible error codes returned from the cartridge and diagnositic actions. Postgres may return additional diagnostic information in addition to the DayCart error. Also, the database log may contain additional information.

C$DCISCHEM-003: FAILED: License not available
: A cartridge function failed because of a license error. Check that the user has the daycart role granted to him, that the user has select privilege on the c$dcischem.license table, and that the license table has a valid cartridge license.

C$DCISCHEM-004: Return string too long
: For reaction transform functions, the enumerated set of molecules or reactions overflowed the returned string type. To avoid this error, coerce the function to return a CLOB type by calling it with a CLOB for the first parameter (smiles).

C$DCISCHEM-006: Can't read ptable
: The ptable is unable to be read for a conversion operation. Check for the existance and select permission on the c$dcischem.ptable table.

C$DCISCHEM-007: Conversion failed
: The conversion function failed.

C$DCISCHEM-021: Not a valid QUERY: %.*s
: The query string was not a valid SMARTS, SMILES, or fingerprint, as required.

C$DCISCHEM-022: Not a valid input SMILES: %.*s
: The SMILES parameter was not parseable as a valid molecule or reaction.

C$DCISCHEM-023: Not a valid fingerprint: %.*s
: The fingerprint parameter was not parseable as a valid molecule or reaction.

C$DCISCHEM-101: CARTRIDGE ERROR
: This is a generic error. Typically, there was an Oracle error during a cartridge function. The error stack should provide additional information.

C$DCISCHEM-102: CARTRIDGE WARNING
: This is a generic warning. Typically, there was an Oracle warning during a cartridge function. The error stack should provide additional information.

C$DCISCHEM-103: CARTRIDGE INFO
: This is a generic informational message. Typically, there was an Oracle informational message during a cartridge function. The error stack should provide additional information.

Previous Index

5. Release Notes

Version 4.95:

- The first fully supported version of the cartridge for Postgresql 8.4.x and 9.0.x.