Constraint Query Language
A high level interface to SQL databases

Mike Elston
Matt Lilley
E-Mail: matt.s.lilley@gmail.com

February 18, 2018

Abstract

CQL is a high level Prolog interface to SQL databases. It is inspired by the work of Christoph Draxler [Draxler, 1991] in the sense that SQL queries are generated but unlike Draxler's work, database tables are not mapped to Prolog predicates, but database queries are described by Prolog terms. These terms allow for accessing table columns by name and provide access to several aspects of SQL that have no natural Prolog equivalent, such as outer joins, inserts, etc.

Contents

1 library(cql/cql): CQL - Constraint Query Language

Note that CQL is currently in a state of flux. Features may be dropped in future releases, and the generated SQL may change between releases. In particular, *runtime* mode is deprecated.

CQL is a Prolog interface to SQL databases. There are two modes: *fully compiled* and *runtime*. The *fully compiled* mode should be used if possible due to the far greater compile time checking it provides.

1.1 Warnings

1.1.1 Comparisons with NULL

CQLv2 correctly compiles equality comparisons with NULL into the appropriate expression at run-time. In CQLv1, executing

```
A={null}, {[A], foo :: [a-A]}
```

would never succeed, regardless of the value of foo.a. This is no longer the case: If A is `{null}` then this will execute as `SELECT .... WHERE a IS NULL` and if A is not {null}, it will execute as `SELECT .... WHERE a = ?`

See the section *Removing null comparisions* for the dealing with the common requirement to ignore comparisons with null.

1.1.2 Avoid `setof/3` and `bagof/3`

It is generally not a good idea to wrap CQL inside a `setof/3` or a `bagof/3` ... unless you are prepared to declare all the CQL variables that are neither bound nor mentioned in the setof/bagof template. If you want to sort, use `findall/3` followed by `sort/2`. Note that `sort/2` (like `setof/3`) removes duplicates. If you don't want to remove duplicates, use `msort/2`.

1.2 Retrieved nulls have special logic to handle outer joins

In the course of executing a select query, the following rules are applied:

1. Any selected attribute that is null does not bind its associated variable.
2. Just before returning from the query any select variables that are still free are bound to {null}.

This is so we can handle outer joins. Consider this:

```
x :: [a-A] *== y :: [a-A]
```

Assume x.a binds A to a non-null value. If there is no matching row in `y`, then `y.a = null`. If variable *A* was truly shared the query could never succeed. By not binding the variable associated with `y.a` the query can succeed (rule 1) and *A* will be bound to the value in `x.a`.

1.3 Getting Started Quickly

Here is a simple example of a SQL SELECT from the table `se_lt_x`

```
test(A) :-
  format('About to call CQL with A=~w', [A]),
  {[],
  se_lt_x :: [a-A,
              b-B,
              c-C]},
  format('B=~w, C=~w', [B, C]).
```

- The CQL is distinguished from the ordinary Prolog by appearing in curly brackets
- Prolog variables which are ground when the CQL is executed will appear in the resulting SQL as part of the WHERE clause

Comparisons can be done in-line e.g.

```
[a-'ELSTON_M']
```

or with the == operator e.g.

```
[a-A], A == 'ELSTON_M'.
```

The single = operator means unify, not compare. Use = for unification, not comparison
FIXME: Unification is deprecated.
The operators =:= and \== are also available for numerical value comparisons (they just translate to SQL = and <>, so in fact you could use them for string comparisons)

1.4 Debugging

You can debug CQL using the meta-predicates `?/1`, `??/2` and `???/3`:

```
???{[], se_lt_x :: [a-A, b-_], A == 'ELSTON_M'}.
```

`?/1` Display a summary form of the generated SQL before and after the goal is called.

```
[main]  CALL   SELECT slx_2.b, slx_2.a  FROM se_lt_x AS slx_2 WHERE slx_2.a =
[main]  EXIT   SELECT slx_2.b, slx_2.a  FROM se_lt_x AS slx_2 WHERE slx_2.a =
```

`??/1` Display the exact query (and results) in a format which can be executed directly by the DBMS (In this case, SQL Server) The generated SQL may be significantly more complicated than expected, and this can be used to debug the CQL compiler itself

```
[main]  CALL
DECLARE @P0 VARCHAR(50);
SET @P0 = 'ELSTON_M';
SELECT slx_450.b,
       slx_450.a
FROM se_lt_x AS slx_450
WHERE slx_450.a = @P0 AND slx_450.a COLLATE Latin1_General_CS_AS = @P0
Result: se_lt_x.b = {null}
        se_lt_x.a = 'ELSTON_M'
 (0.003304s, 0.00cpu, 359 inferences)
```

`???/1` Display simplified SQL before the goal is called and display the results afterwards

```
[main]  CALL
SELECT slx_450.b,
       slx_450.a
FROM se_lt_x AS slx_450
WHERE slx_450.a = 'ELSTON_M'
Result: se_lt_x.b = {null}
        se_lt_x.a = 'ELSTON_M'
 (0.003304s, 0.00cpu, 359 inferences)
```

1.5 Prolog Variables

A Prolog variable can be simultaneously a *SELECT* variable, a *JOIN* variable and a *WHERE* variable as A is in the following example:

```
{[],
 se_lt_x :: [a-A, c-C]
 =*=
 se_lt_y :: [d-A, f-F],
 A == 'A4'}
```

which generates the following SQL

```
SELECT
  x_192.a, x_192.c, y_73.d, y_73.f
 FROM
  se_lt_x x_192 INNER JOIN se_lt_y y_73 ON y_73.d=x_192.a
 WHERE   x_192.a = ? and y_73.d = ?
```

Note how **all the variables referenced in the query** are retrieved in the SELECT. This is done to make the query *Prolog-like*. This means the retrieved row should behave like a Prolog fact so that when a query succeeds all the variables become instantiated.

There is one notable exception however: **WHERE variables and JOIN variables are not bound in aggregation selections**

FIXME: Is this still the case?

```
sum_test :-
  {[],
   #se_lt_x :: [a-ValueA,
                sum(b)-Summation]
   =*=
   #se_lt_y :: [e-ValueB],

   ValueA == ValueB,   % Explicit join point

   group_by([ValueA])},

  writeln(ValueA-ValueB-Summation).
```

```
'ELSTON_M'-_G375971-99450
true ;
```

1.6 Special Attributes

The following attributes are automatically provided i.e if the attribute is present in the table, CQL will automatically fill in the value:

1. **generation_** Set to 0 on INSERT and incremented by 1 on each update
2. **inserted_** Set to the current time at the time of the INSERT transaction
3. **inserted_by_** Set to the user ID corresponding to the access token supplied to the transaction
4. **updated_** Set to the current time at the time of the UPDATE transaction. Note that updated_ is also set by an INSERT
5. **updated_by_** Set to the user ID corresponding to the access token supplied to the transaction. Note that updated_by_ is also set by an INSERT
6. **transaction_id_** Set to the transaction ID

All the special attributes can be overridden by supplying the attribute-value pair explicitly.

1.7 Examples

Rather than provide an abstract description of CQL syntax here is a set of examples that show how to use it.

1.7.1 Simple INSERT

```
{[],
 insert(se_lt_x, [a-'A', b-'B', c-100])}
```

1.7.2 Simple INSERT with retrieval of identity of the inserted

```
{[],
 insert(se_lt_x, [a-'A', b-'B', c-100]),
 identity(I)}
```

1.7.3 Simple DELETE

```
{[],
 delete(se_lt_x, [x_pk-I])}
```

Note that the WHERE clause is part of the `delete/2` term unlike *update* where the WHERE clause is defined outside the `update/2` term. I could have made delete consisent with update, but this would have required the @ alias in the delete WHERE clause to identify the table where the rows are to be deleted). This seems like overkill because a delete can in fact refer to only one table anyway i.e. you can't identify rows to delete via a JOIN.

1.7.4 Simple SELECT

```
{[],
 se_lt_x :: [a-A, b-B]}
```

This query will either:

- If A is bound, and B are bound, fail if there are no such rows, or succeed (without binding anything) the same number of times as there are matching rows in se_lt_x.
- If A is bound and B is unbound, bind B to each of the values in `se_lt_x.b` where `se_lt_x.a = A`
- If B is bound and A is unbound, bind A to each of the values in `se_lt_x.a` where `se_lt_x.b = B`
- If *A* and *B* are both unbound, bind *A* and *B* to each of the tuples in `se_lt_x`

1.7.5 Simple UPDATE

```
{[],
 update(se_lt_x, [c-100]),
 @ :: [a-'A1'],
 row_count(N)}
```

This corresponds to `UPDATE se_lt_x SET c=100 WHERE se_lt_x.a='A1'`. The '@' is a special alias referring to the table that is being updated. The `row_count/1` term gives the number or rows updated.

1.7.6 WHERE with arithmetic comparison

```
{[],
 se_lt_x :: [a-A, c-C],
 C > 10}
```

1.7.7 Simple INNER JOIN

```
{[],
 se_lt_x :: [a-J1, c-C]
  =*=
 se_lt_y :: [d-J1, f-F]}
```

The join is `se_lt_x.a = se_lt_y.d` because of the shared variable *J1*. `se_lt_x.c` will be returned in *C* and `se_lt_y.f` will be returned in *F*

1.7.8 Arithmetic UPDATE with an INNER JOIN and a WHERE restriction

```
{[],
 update(se_lt_x, [c-(C + 2 * F + 20)]),
 @ :: [a-A, c-C] =*= se_lt_y :: [d-A, f-F],
 C < 100}
```

This joins the table being updated (table `se_lt_x`) on table `se_lt_y` where `se_lt_x.a = se_lt_y.a` and where `se_lt_x.c < 200` then updates each identified row `se_lt_x.c` with the specified expression.

1.7.9 Confirm row does not exist

```
\+ exists {[], se_lt_x :: [a-'Z']}
```

1.7.10 Aggregation - Count

```
{[],
 se_lt_x :: [count(c)-C]}
```

This will count the rows in table se_lt_x

1.7.11 Aggregation - Sum

```
{[],
 se_lt_x :: [sum(c)-C]}
```

Sum the values of attribute c in table se_lt_x

1.7.12 Aggregation - Average

```
{[],
 se_lt_x :: [avg(c)-C]}
```

Calculate the mean of the values of attribute c in table se_lt_x

1.7.13 Maximum Value

```
{[],
 se_lt_x :: [max(c)-C]}
```

Calculate the maximum of the values of attribute c in table se_lt_x

1.7.14 Minimum Value

```
{[],
 se_lt_x :: [min(c)-C]}
```

Calculate the minimum of the values of attribute c in table se_lt_x

1.7.15 Aggregation requiring GROUP BY

```
{[],
 se_lt_z :: [g-G, sum(i)-I],
 group_by([G])}
```

This will generate the `GROUP BY SQL` and sum `se_lt_z.i` for each value of `se_lt_z.g`

1.7.16 INNER JOIN with an aggregation sub-query where the sub-query is constrained by a shared variable from the main query

```
{[],
 se_lt_x :: [b-J1, a-A]
   =*=
 se_lt_z :: [h-J1, i-I, g-Z],
 I > min(Y, se_lt_y :: [f-Y, d-Z])}
```

The main query and the sub-query share variable Z. The generated SQL is:

```
SELECT
  x37.a, z4.i, z4.g
 FROM
  se_lt_x x37 INNER JOIN se_lt_z z4 ON x37.b=z4.h and z4.h=x37.b
 WHERE
  z4.i > (SELECT min(y11.f) FROM se_lt_y y11 WHERE z4.g=y11.d)
```

1.7.17 INNER JOIN in an aggregation sub-query

```
{[],
 se_lt_y :: [d-D,f-F],
 F < sum(I,
         se_lt_x :: [b-J1]
           =*=
         se_lt_z :: [h-J1, i-I])}
```

1.7.18 Negation

```
{[],
 se_lt_x :: [a-A, b-B],
 \+ exists se_lt_y :: [d-A]}
```

The generated SQL is:

```
SELECT
  x39.a, x39.b
 FROM
  se_lt_x x39
 WHERE NOT EXISTS (SELECT * FROM se_lt_y y13 WHERE x39.a = y13.d)
```

1.7.19 EXISTS

An exists restriction translates to a `WHERE` sub-query and is used to say that ”each row returned in the main query must satisfy some condition expressed by another query”.

Example

```
{[],
 se_lt_x :: [a-A, b-B],
 exists se_lt_y :: [d-A]}
```

compiles to:

```
SELECT
  x.b, x.a
FROM
  se_lt_x x
WHERE
  EXISTS (SELECT * FROM se_lt_y WHERE x.a = y.d)
```

1.7.20 Left Outer Join

```
se_lt_x :: [a-J1, b-B]
  *==
se_lt_y :: [d-J1, e-E]}
```

1.7.21 List-based Restrictions

CQL supports query restrictions based on lists. Note that in both cases **`\== []` and `== []` are equivalent** despite the obvious logical inconsistency.

FIXME: Can we make this behaviour be controlled by a flag? It IS quite useful, even if it is completely illogical

```
{[], se_lt_x :: [a-Bar], Bar == []}
```

and

```
{[], se_lt_x :: [a-Bar], Bar \== []}
```

both do **exactly** the same thing - they will not restrict the query based on Bar. The second case seems to be logically consistent - all things are not in the empty list.

1.7.22 Compile time in-list constraint

If your list is bound at compile-time, you can simply use it as the attribute value in CQL, for example:

```
{[], se_lt_x :: [a-['ELSTON_M', 'LILLEY_N']]}
```

This does not require the list to be **ground**, merely **bound**. For example, this is not precluded:

```
foo(V1, V2):-
    {[], se_lt_x :: [a-[V1, V2]]}.
```

If, however, your list is not bound at compile-time, you must wrap the variable in `list/1`:

```
Bar = [a,b,c],
{[], se_lt_x :: [bar-list(Bar)]}
```

If you write

```
foo(V1):-
    {[], se_lt_x :: [a-V1]}.
```

and at runtime call `foo([value1])`, you will get a type error.

Remember: If the list of IN values is *empty* then no restriction is generated i.e.

```
{[], se_lt_x :: [a-[], b-B}

is the exactly the same as

{[], se_lt_x :: [b-B}
```

1.7.23 Disjunction resulting in OR in WHERE clause

```
{[],
 se_lt_x :: [a-A, b-B, c-C],
 (C == 10 ; B == 'B2', C < 4)}
```

The generated SQL is:

```
SELECT
  x.a, x.b, x.c
 FROM
  se_lt_x x
 WHERE
  ((x.b = ? AND x.c < ?) OR x.c = ?)
```

1.7.24 Disjunction resulting in different joins (implemented as a SQL UNION)

```
{[],
 se_lt_x :: [a-A, c-C]
 =*=
 (se_lt_y :: [d-A] ; se_lt_z :: [g-A])}
```

The generated SQL is:

```
SELECT
  x43.c
 FROM
  (se_lt_x x43 INNER JOIN se_lt_z z6 ON x43.a=z6.g AND z6.g=x43.a)

UNION

SELECT
  x44.c
 FROM
  (se_lt_x x44 INNER JOIN se_lt_y y16 ON x44.a=y16.d AND y16.d=x44.a)
```

1.7.25 Disjunction resulting in different SELECT attributes (implemented as separate ODBC queries)

```
{[],
 (se_lt_x :: [a-A, c-10]
 ;
 se_lt_y :: [d-A, f-25])}
```

The output variable A is bound to the value from two different attributes and so the query is implemented as two separate ODBC queries

1.7.26 ORDER BY

```
{[],
 se_lt_z :: [g-G, h-H],
 order_by([-G])}
```

The order_by specification is a list of "signed" variables. The example above will order by se_lt_z.g descending

1.7.27 DISTINCT

Use `distinct(ListOfVars)` to specify which attributes you want to be distinct:

```
test_distinct :-
  findall(UserName,
          {[],
           se_lt_x :: [a-UserName,
                       c-Key],
           Key >= 7,
           distinct([UserName])},
          L),
  length(L, N),
  format('~w solutions~n', [N]).

CALL  : user:test_distinct/0
26 solutions
EXIT  : user:test_distinct/0 (0.098133s, 0.00cpu, 1,488 inferences)
```

1.7.28 SELECT with NOT NULL restriction

```
{[],
 se_lt_z :: [i-I, j-J],
 J \== {null}}
```

1.7.29 First N

```
{[],
 N = 3,
 se_lt_z :: [i-I],
 top(N),
 order_by([+I])}
```

This generates a TOP clause in SQL Server, and LIMIT clauses for PostgreSQL and SQLite

1.7.30 Self JOIN

```
{[],
 se_lt_z :: [h-H, i-I1]
  =*=
 se_lt_z :: [h-H, i-I2],
 I1 \== I2}
```

1.7.31 Removing null comparisions

Use the ignore_if_null wrapper in your CQL to 'filter out' null input values. This is a useful extension for creating user-designed searches.

```
{[],
 se_lt_x :: [a-UserName,
             b-ignore_if_null(SearchKey),
             ...]}
```

At runtime, if SearchKey is bound to a value other than {null} then the query will contain `WHERE ... b = ?`. If, however, SearchKey is bound to `{null}`, then this comparison will be omitted.

Disjunctions

In general, don't use ignore_if_null in disjunctions. Consider this query:

```
SearchKey = '%ELSTON%',
{[],
 se_lt_x :: [a-UserName,
             b-RealName],
 ( RealName =~ SearchKey
 ; UserName =~ SearchKey)}
```

The query means "find a user where the UserName contains ELSTON OR the RealName contain ELSTON". If !SearchKey is {null} then RealName=~ {null} will fail, which is correct. If ignore_if_null was used, the test would *succeed*, which means the disjunction would always succeed i.e. the query would contain no restriction, which is clearly not the intended result. FIXME: Mike, what is this all about?

1.7.32 Three table JOIN

```
{[],
 se_lt_x :: [a-A, c-C]
  =*=
 se_lt_y :: [d-A, f-F]
  =*=
 se_lt_z :: [i-F, g-G]}
```

The shared variable A joins `se_lt_x` and `se_lt_y`; the shared variable *F* joins `se_lt_y` and `se_lt_z`

1.7.33 Three table JOIN with NOLOCK locking hint

```
{[],
 se_lt_x :: [a-A, c-C]
  =*=
 #se_lt_y :: [d-A, f-F]
  =*=
 #se_lt_z :: [i-F, g-G]}
```

The hash operator indicates the table that should be accessed WITH (NOLOCK)

1.7.34 SELECT with LIKE

```
{[],
 se_lt_z :: [g-G, i-I],
 G =~ 'A_'}
```

The operator =~ means LIKE. If you are using PostgreSQL, it means ILIKE.

1.7.35 Writing exceptions directly to the database

You can write an exception term directly to a varchar-type column in the database. Note that it will be rendered as text using ~p, and truncated if necessary - so you certainly can't read it out again and expect to get an exception! Example code:

```
catch(process_message(Message),
      Exception,
      {[],
      update(some_table, [status-'ERROR',
                          status_comment-Exception]),
      @ :: [some_table_primary_key-PrimaryKey]}).
```

FIXME: This code is specific to my usage of CQL

1.7.36 TOP N is Parametric

You can pass the "N" is TOP N as a parameter (Subject to DBMS compatibility. This works in SQL Server 2005 and later, and PostgreSQL 9 (possibly earlier versions) and SQLite3.

```
N = 3,
findall(I,
        {[],
         se_lt_z :: [i-I], top(N), order_by([+I])},
        L)
```

1.7.37 Using `compile_time_goal/1`

You can include `compile_time_goal(Goal)` in your CQL. If you specify a module, it will be used, otherwise the goal will be called in the current module. Note that the goal is executed in-order - if you want to use the bindings in your CQL, you must put the compile_time_goal before them.

Example 1

```
{[],
 se_lt_x :: [a-UserName,
             b-RealName,
             d-FavouriteColour],
   compile_time_goal(standard_batch_size_for_search(StandardBatchSize)),
   top(StandardBatchSize),
   order_by([+UserName]}
```

Example 2

```
excellent_colours(['RED', 'BLUE']).

{[],
 se_lt_x :: [a-UserName,
             b-RealName,
             d-FavouriteColour],
 compile_time_goal(excellent_colours(Colours)),
 FavouriteColour == Colours}
```

1.7.38 ON

CQL supports both constant and shared variable join specifications. This is particularly useful when specifying outer joins.

Example

```
{[],
 se_lt_x :: [a-UserNameA,
             b-RealName,
             d-FavouriteColour]
 *==
 se_lt_x :: [a-UserNameB,
             e-FavouriteFood] on( UserNameA == UserNameB,
                                  FavouriteColour == FavouriteFood,
                                  FavouriteFood == 'ORANGE')}
```

All the CQL comparison operators, `<`, `=<`, `==`, `=~`, `\=~`, `\==`, `>=`, `>` can be used in ON specifications.

For example:

```
{[],
 se_lt_z :: [i-J1, k-K]
 *==
 se_lt_x :: [c-J1, a-A, b-B] on A \== 'A1'},
```

1.7.39 Expressions In Where Restrictions

Expressions in WHERE restrictions are supported, for example:

```
{[],
 se_lt_n :: [i-I, j-J, k-K],
 J > 10 * (K / I) + 15},
```

1.7.40 Explicitly avoid the "No WHERE restriction" message

To avoid accidentally deleting or updating all rows in a table CQL raises an exception if there is no WHERE restriction.

Sometimes however you really do need to delete or update all rows in a table.

To support this requirement in a disciplined way (and to avoid the creation of "dummy" WHERE restrictions) the keyword **absence_of_where_restriction_is_deliberate** has been added. For example:

```
{[],
 update(se_lt_x, [c-10]),
        @ :: [],
        absence_of_where_restriction_is_deliberate}
```

1.7.41 HAVING

HAVING restrictions can be specified. For example:

```
{[],
 se_lt_z :: [sum(i)-I,
             g-G],
 group_by([G]),
 having(I > 30)}
```

For a description of HAVING see http://en.wikipedia.org/wiki/Having_(SQL)

There is one important difference between SQL HAVING and SQL WHERE clauses. The SQL WHERE clause condition is tested against **each and every** row of data, while the SQL HAVING clause condition is tested against the *groups and/or aggregates specified in the SQL GROUP BY clause and/or the SQL SELECT column list*.

1.7.42 INSERT and UPDATE value in-line formatting

INSERT and UPDATE values can be formatted in-line at runtime. For example:

```
Suffix = 'NOGG',
cql_transaction(Schema, UserId,
                {[],
                insert(se_lt_x, [a-'A', b-'B', c-100, d-format('EGG_~w', [Suffix]
```

will insert 'EGG_NOGG' into attribute 'd'.

1.7.43 Negations in WHERE Clauses

You can specify negations in CQL WHERE clauses e.g.

```
{[],
 se_lt_z :: [g-G, h-H, i-I],
 \+((G == 'A1', H == 'B1' ; G == 'D1', H == 'B3'))},
```

Note that, just like in Prolog, `\+` is a unary operator hence the "double" brackets in the example above.

1.7.44 Predicate-generated Attribute Values

It is possible to generate **compile time** attribute values by specifying a *predicate* which is executed when the CQL statement is compiled.

The predicate must return the value you want as its last argument. You specify the predicate where you would normally put the attribute value. The predicate is specified *with its output argument missing*.

Example - Using domain allowed values in a query.

In the following CQL statement the predicate `cql_domain_allowed_value/3` is called within `findall/3` **at compile time** to generate a list of domain values that restrict favourite_colour to be 'ORANGE' or 'PINK' or 'BLUE', or 'GREEN'.

```
colour('ORANGE').
colour('PINK').
colour('BLUE').
colour('GREEN').

{[],
 se_lt_x :: [d-findall(Value,
                       permissible_colour(Value)),
             a-UserName]}
```

Note how `findall/3` is actually called by specifying `findall/2`.

There is not much point using predicate-generated attribute values in compile-at-runtime CQL as you can always call the predicate to generate the required values *outside* the CQL statement.

1.7.45 INSERT from SELECT

INSERT from SELECT is supported:

```
Constant = 'MIKE',
{[],
 insert(se_lt_x1, [x_pk-Pk, a-A, b-B, c-C, d-Constant]),
 se_lt_x :: [x_pk-Pk, a-A, b-B, c-C, as(d)-Constant]}
```

which generates the following SQL:

```
INSERT INTO se_lt_x1 (x_pk, a, b, c, d)
SELECT se_lt_x_955.x_pk, se_lt_x_955.a, se_lt_x_955.b, se_lt_x_955.c, ? AS d
  FROM se_lt_x lt_x_955
```

Note the use of the `as(d)` construct in the SELECT part of the CQL to make the constant **'MIKE'** appear to come from the SELECT thus setting `lt_x1.d` to **'MIKE'** in every row inserted.

1.8 Hooks

CQL provides a large number of hooks to fine-tune behaviour and allow for customization. These are:

1.8.1 Generated Code Hooks

- cql:`cql_dependency_hook(+EntitySet, +Module)` can be defined to be notified when a given Module references a list of database entities. This can be used to manage meta-data/code dependency
- cql:`cql_generated_sql_hook(+Filename, +LineNumber, +Goals)` can be defined to examine generated SQL. Use `cql_sql_clause(+Goals, -SQL, -Parameters)` to examine the goals
- cql:`cql_index_suggestion_hook(+Index)` can be defined if you are interested in proposed indices for your schema. Note that this is not very mature (yet)

1.8.2 Data Representation Hooks

- cql:`cql_atomic_value_check_hook(+Value)` can be defined to declare new 'atomic' types (That is, types which can be written directly to the database), such as a representation like `boolean(true)` for 1.
- cql:`cql_check_value_hook(+Value)` can be used to check that a value is legal
- cql:`application_value_to_odbc_value_hook(+OdbcDataType, +Schema, +TableName, +C`
- cql:`odbc_value_to_application_value_hook(+OdbcDataType, +Schema, +TableName, +C`

1.8.3 Application Integration

- cql:`cql_access_token_hook(+AccessToken, -UserId)` can be defined to map the generic 'AccessToken' passed to `cql_transaction/3` to a user ID. If not defined, the AccessToken is assumed to be the user ID. This UserID is used in logging.
- cql:log_selects can be defined if you want to receive logging information about selects. By default only update, delete and insert are logged
- cql:`cql_execution_hook(+Statement, +OdbcParameters, +OdbcParameterDataTypes, -R` can be defined if you want to implement the exeuction yourself (for example, to add extra debugging)
- cql:`cql_log_hook(+Topics, +Level, +Format, +Args)` can be defined to redirect CQL logging.
  - Levels is one of informational, warning, or error
  - Topics is a list of topics. Currently the only lists possible are `[]` and `[debug(deadlocks)]`
- cql:`sql_gripe_hook(+Level, +Format, +Args)` is called when suspect SQL is found by the SQL parser
- cql:`cql_normalize_atom_hook(+DBMS, +ApplciationAtom, -DBMSAtom)` can be used to create a map for atoms in a specific DBMS. For example, your schema may have arbitrarily long table names, but your DBMS may only allow names up to 64 bytes long. In this case, you can create a scheme for mapping the application-level atom to the DBMS. Other uses include deleting or normalizing illegal characters in names
- cql:`cql_error_hook(+ErrorId, +Format, +Args)` can be defined to generate a specific exception term from the given arguments. If not defined (or if it does not throw an exception, or fails), you will get `cql_error(ErrorId, FormattedMessage)`.
- cql:`cql_max_db_connections_hook(-Max)` can be defined to limit the number of simultaneous connections each thread will attempt to have
- cql:`odbc_connection_complete_hook(+Schema, +Details, +Connection)` can be hooked if you want to know every time a connection is made
- cql:`cql_transaction_info_hook(+AccessToken, +Connection, +DBMS, +Goal, -Info)` can be defined if you want to define any application-defined information on a per-transaction level. This can be recovered via `database_transaction_query_info(?ThreadId, ?Goal, ?Info)`.

1.8.4 Inline values

`cql :` `cql_inline_domain_value_hook`*(+DomainName, +Value)*
can be defined if you want the given value to be 'inlined' into the CQL (ie not supplied as a parameter). Great care must be taken to avoid SQL injection attacks if this is used.

1.8.5 Schema

These define the schema. You MUST either define them, or include `library(cql/cql_autoschema)` and add two directives to build the schema automatically:

- `:-register_database_connection_details(+Schema, +ConnectionInfo).`
- `:-build_schema(+Schema).`

Otherwise, you need to define at least `cql:default_schema/1` and `cql:dbms/2`, and then as many of the other facts as needed for your schema.

- cql:`default_schema(-Schema)` MUST be defined. CQL autoschema will define this for you if you use it.
- cql:`dbms(+Schema, -DBMS)` MUST be defined for every schema you use. CQL autoschema will define this for you if you use it. DBMS must be one of the following:
  - 'Microsoft SQL Server'
  - 'PostgreSQL'
  - 'SQLite'
- cql:`odbc_data_type(+Schema, +TableName, +ColumnName, +OdbcDataType).`
- cql:`primary_column_name(+Schema, +Tablename, +ColumnName).`
- cql:`database_attribute(+EntityType:table/view, +Schema:atom, +EntityName:atom,`
- cql:`database_domain(+DomainName, +OdbcDataType).`
- cql:`routine_return_type(+Schema, +RoutineName, +OdbcDataType).`
- cql:`database_constraint(+Schema, +EntityName, +ConstraintName, +Constraint).`

1.8.6 Event Processing and History

CQL provides hooks for maintaining detailed history of data in the database.

The hook predicates are:

- cql:`cql_event_notification_table(+Schema, +TableName)`
- cql:`cql_history_attribute(+Schema, +TableName, +ColumnName)`
- cql:`cql_update_history_hook(+Schema, +TableName, +ColumnName, +PrimaryKeyColumn`
- cql:`process_database_events(+Events)`

Event Processing and History recording can be suppressed for a particular update/insert/delete statement by including the _no_state_change_actions_9 directive.

For example

```
{[],
 update(se_lt_x, [f-'LILAC']
 @ :: [a-'ELSTON_M'],
 no_state_change_actions,   % Don't want history to record this change
 row_count(RowCount)}
```

1.8.7 Statistical Hooks

CQL has hooks to enable in-memory statistics to be tracked for database tables. Using this hook, it's possible to monitor the number of rows in a table with a particular value in a particular column.

Often the kind of statistics of interest are 'how many rows in this table are in ERROR' or 'how many in this table are at NEW'? While it may be possible to maintain these directly in any code which updates tables, it can be difficult to ensure all cases are accounted for, and requires developers to remember which attributes are tracked.

To ensure that all (CQL-originated) updates to statuses are captured, it's possible to use the CQL hook system to update them automatically. Define add a fact like:

```
cql_statistic_monitored_attribute_hook(my_schema, my_table,
                                       my_table_status_column).
```

This will examine the domain for the column 'my_table_status_column', and generate a statistic for each of my_table::`my_table_status_column(xxx)`, where xxx is each possible allowed value for the domain. Code will be automatically generated to trap updates to this specific column, and maintain the state. This way, if you are interested in the number of rows in my_table which have a status of 'NEW', you can look at my_table::`my_table_status_column('NEW')`, without having to manage the state directly. CQL update statements which affect the status will automatically maintain the statistics.

The calculations are vastly simpler than the history mechanism, so as to keep performance as high as possible. For inserts, there is no cost to monitoring the table (the insert simply increments the statistic if the transaction completes). For deletes, the delete query is first run as a select, aggregating on the monitored columns to find the number of deletes for each domain allowed value. This means that a delete of millions of rows might requires a select returning only a single row for statistics purposes. For updates, the delete code is run, then the insert calculation is done, multiplied by the number of rows affected by the update.

In all cases, CQL ends up calling `cql_statistic_monitored_attribute_change_hook/5`, where the last argument is a signed value indicating the number of changes to that particular statistic.

cql_set_module_default_schema(+*Schema*)
Set the *Schema* for a module

cql_get_module_default_schema(+*Module, ?ModuleDefaultSchema*)

cql_goal_expansion(*?Schema, ?Cql, ?GoalExpansion*)
Expand at compile time if the first term is a list of unbound input variables

Expand at runtime if the first term is compile_at_runtime

cql_runtime(+*Schema, +IgnoreIfNullVariables, +CqlA, +CqlB, +VariableMap, +FileName, +LineNumber*)

cql_temporary_column_name(*?Schema, ?DataType, ?ColumnName, ?Type*)

cql_show(*:Goal, +Mode*)
Called when `?/1`, `??/1`, and `???/1` applied to CQL

Arguments	
Goal	goal term
Mode	minimal ; explicit ; full

statistic_monitored_attribute(+*Schema, +TableName, +ColumnName*)

dbms(+*Schema, -DBMSName*) *[multifile]*
Determine the DBMS for a given *Schema*. Can be autoconfigured.

odbc_data_type(+*Schema, +TableSpec, +ColumnName, ?OdbcDataType*) *[multifile]*
OdbcDataType must be a native SQL datatype, such as `varchar(30)` or `decimal(10, 5)`
Can be autoconfigured.

primary_key_column_name(+*Schema, +TableName, -PrimaryKeyAttributeName*) *[multifile]*
Can be autoconfigured.

database_attribute(*?EntityType:table/view, ?Schema:atom, ?EntityName:atom, ?ColumnName:atom, ?DomainOrNative*
Can be autoconfigured.

routine_return_type(*?Schema:atom, ?EntityName:atom, ?OdbcType*) *[multifile]*
Can be autoconfigured

database_constraint(*?Schema:atom, ?EntityName:atom, ?ConstraintName:atom, ?Constraint*)*[nondet,multifile]*
Constraint is one of:

- `primary_key(ColumnNames:list)`
- `foreign_key(ForeignTableName:atom, ForeignColumnNames:list, ColumnNames:li`
- `unique(ColumnNames:list)`
- `check(CheckClause)`

In theory this can be autoconfigured too, but I have not written the code for it yet

attribute_domain(+*Schema, +TableName, +ColumnName, -Domain*)

database_identity(*?Schema:atom, ?EntityName:atom, ?ColumnName:atom*)

database_key(*?Schema:atom, ?EntityName:atom, ?ConstraintName:atom, ?KeyColumnNames:list, ?KeyType*)

Arguments	
KeyColumnNames	list of *atom* in database-supplied order
KeyType	*identity* ; *'primary key'* ; *unique*

cql_event_notification_table(*+Schema, +TableName*) *[multifile]*

cql_history_attribute(*+Schema, +TableName, +ColumnName*) *[multifile]*

sql_gripe_hook(*+Level, +Format, +Args*) *[multifile]*

Called when something dubious is found by the SQL parser.

cql_normalize_name(*+DBMS, +Name, -NormalizedName*)

Normalize a name which is potentially longer than the *DBMS* allows to a unique truncation

register_database_connection_details(*+Schema:atom, +ConnectionDetails*) *[det]*

This should be called once to register the database connection details.

Arguments		
ConnectionDetails	`driver_string(DriverString)`	or
	`dsn(Dsn, Username, Password)`	

References

[Draxler, 1991] C. Draxler. Accessing relational and NF^2 databases through database set predicates. In Geraint A. Wiggins, Chris Mellish, and Tim Duncan, editors, *ALPUK91: Proceedings of the 3rd UK Annual Conference on Logic Programming, Edinburgh 1991*, Workshops in Computing, pages 156–173. Springer-Verlag, 1991.

Index