Datalog: Deductive Database Programming

Version 9.4.0.5

Jay McCarthy <jay@racket-lang.org>

November 14, 2026

Datalog is

- a declarative logic language in which each formula is a function-free Horn clause, and every variable in the head of a clause must appear in the body of the clause.
- a lightweight deductive database system where queries and database updates are expressed in the logic language.

The use of Datalog syntax and an implementation based on tabling intermediate results ensures that all queries terminate.

Contents

1 Datalog Module Language **3**

2 Tutorial **6**

3 Parenthetical Datalog Module Language **9**

4 Racket Interoperability **10**

5 Acknowledgments **14**

1 Datalog Module Language

`#lang datalog` package: `datalog`

In Datalog input, whitespace characters are ignored except when they separate adjacent tokens or when they occur in strings. Comments are also considered to be whitespace. The character `%` introduces a comment, which extends to the next line break. Comments do not occur inside strings.

A variable is a sequence of Unicode "Uppercase" and "Lowercase" letters, digits, and the underscore character. A variable must begin with a Unicode "Uppercase" letter.

An identifier is a sequence of printing characters that does not contain any of the following characters: `(`, `` ` ``, `'`, `)`, `=`, `:`, `.`, `~`, `?`, `"`, `%`, and space. An identifier must not begin with a Latin capital letter. Note that the characters that start punctuation are forbidden in identifiers, but the hyphen character is allowed.

A string is a sequence of characters enclosed in double quotes. Characters other than double quote, newline, and backslash may be directly included in a string. The remaining characters may be specified using escape characters, `\"`, `\ ` , and `\\` respectively.

A literal is a predicate symbol followed by an optional parenthesized list of comma separated terms, or it is an external query as described below. A predicate symbol is either an identifier or a string. A term is either a variable or a constant. A constant is an identifier, string, integer, or boolean, where booleans are written the same as the identifiers `true` and `false`, and integers are written the same as identifiers `0` or those with a nonempty sequence of digits, no leading zero, and optionally prefixed with `-`. As a special case, two terms separated by `=` (`!=`) is a literal for the equality (inequality) predicate. The following are literals:

```
parent(john, douglas)
zero-arity-literal
"="(3,3)
""(-0-0-0,&&&,***,"\00")
42
```

A clause is a head literal followed by an optional body. A body is a comma separated list of literals. A clause without a body is called a *fact*, and a rule when it has one. The punctuation `:-` separates the head of a rule from its body. A clause is safe if every variable in its head occurs in some literal in its body. The following are safe clauses:

```
parent(john, douglas)
ancestor(A, B) :-
    parent(A, B)
ancestor(A, B) :-
    parent(A, C),
```

```
ancestor(C, B)
```

A program is a sequence of zero or more statements. A statement is an assertion, a retraction, a query, or a requirement. An assertion is a clause followed by a `.`, and it adds the clause to the database if it is safe. A retraction is a clause followed by `~`, and it removes the clause from the database. A query is a literal followed by a `?`. A requirement is a `(`, then an identifier, then `)`, then `.`, and it imports functions that can be called as external queries.

A *external query* is a variable, then `:-`, then an identifier, then a parenthesized list of comma separated terms. Beware that an external query can break Datalog's termination guarantee.

The following BNF describes the syntax of Datalog.

⟨*program*⟩ ::= ⟨*statement*⟩*
⟨*statement*⟩ ::= ⟨*assertion*⟩
| ⟨*retraction*⟩
| ⟨*query*⟩
| ⟨*requirement*⟩
⟨*assertion*⟩ ::= ⟨*clause*⟩ `.`
⟨*retraction*⟩ ::= ⟨*clause*⟩ `~`
⟨*query*⟩ ::= ⟨*literal*⟩ `?`
⟨*requirement*⟩ ::= `(` ⟨*IDENTIFIER*⟩ `)` `.`
⟨*clause*⟩ ::= ⟨*literal*⟩ `:-` ⟨*body*⟩
| ⟨*literal*⟩
⟨*body*⟩ ::= ⟨*literal*⟩ `,` ⟨*body*⟩
| ⟨*literal*⟩
⟨*literal*⟩ ::= ⟨*predicate-sym*⟩ `(` `)`
| ⟨*predicate-sym*⟩ `(` ⟨*terms*⟩ `)`
| ⟨*predicate-sym*⟩
| ⟨*term*⟩ `=` ⟨*term*⟩
| ⟨*term*⟩ `!=` ⟨*term*⟩
| ⟨*VARIABLE*⟩ `:-` ⟨*external-sym*⟩ `(` ⟨*terms*⟩ `)`
⟨*predicate-sym*⟩ ::= ⟨*IDENTIFIER*⟩
| ⟨*STRING*⟩
⟨*terms*⟩ ::= ⟨*term*⟩
| ⟨*term*⟩ `,` ⟨*terms*⟩
⟨*term*⟩ ::= ⟨*VARIABLE*⟩
| ⟨*constant*⟩
⟨*constant*⟩ ::= ⟨*IDENTIFIER*⟩
| ⟨*STRING*⟩
| ⟨*INTEGER*⟩
| `true` | `false`

The effect of running a Datalog program is to modify the database as directed by its statements, and then to return the literals designated by the query. The modified database is provided as `theory`.

The following is a program:

```
#lang datalog
edge(a, b). edge(b, c). edge(c, d). edge(d, a).
path(X, Y) :- edge(X, Y).
path(X, Y) :- edge(X, Z), path(Z, Y).
path(X, Y)?
```

Here is a program that uses + and - from racket/base as external queries:

```
#lang datalog
(racket/base).

fib(0, 0).
fib(1, 1).

fib(N, F) :- N != 1,
             N != 0,
             N1 :- -(N, 1),
             N2 :- -(N, 2),
             fib(N1, F1),
             fib(N2, F2),
             F :- +(F1, F2).

fib(30, F)?
```

The Datalog REPL accepts new statements that are executed as if they were in the original program text.

2 Tutorial

Start DrRacket and type

```
#lang datalog
```

in the Definitions window. Click Run, then click in the REPL.

```
>
```

Facts are stored in tables. If the name of the table is `parent`, and `john` is the parent of `douglas`, store the fact in the database with this:

```
> parent(john, douglas).
```

Each item in the parenthesized list following the name of the table is called a *term*. A term can be either a logical `variable` or a `constant`. Thus far, all the terms shown have been constant terms.

A query can be used to see if a particular row is in a table. Type this to see if `john` is the parent of `douglas`:

```
> parent(john, douglas)?
```

```
parent(john, douglas).
```

Type this to see if `john` is the parent of `ebbon`:

```
> parent(john, ebbon)?
```

The query produced no results because `john` is not the parent of `ebbon`. Let's add more rows.

```
> parent(bob, john).
```

```
> parent(ebbon, bob).
```

Type the following to list all rows in the `parent` table:

```
> parent(A, B)?
```

```
parent(john, douglas).

parent(bob, john).

parent(ebbon, bob).
```

Type the following to list all the children of `john`:

```
> parent(john, B)?

parent(john, douglas).
```

A term that begins with a capital letter is a logical variable.When producing a set of answers, the Datalog interpreter lists all rows that match the query when each variable in the query is substituted for a constant. The following example produces no answers, as there are no substitutions for the variable `A` that produce a fact in the database. This is because no one is the parent of oneself.

```
> parent(A, A)?
```

A deductive database can use rules of inference to derive new facts. Consider the following rule:

```
> ancestor(A, B) :- parent(A, B).
```

The rule says that if A is the parent of B, then A is an ancestor of B. The other rule defining an ancestor says that if A is the parent of C, C is an ancestor of B, then A is an ancestor of B.

```
> ancestor(A, B) :-
    parent(A, C),
    ancestor(C, B).
```

In the interpreter, DrRacket knows that the clause is not complete, so by pressing Return, it doesn't interpret the line.

Rules are used to answer queries just as is done for facts.

```
> ancestor(A, B)?

ancestor(ebbon, bob).
```

```
ancestor(bob, john).

ancestor(john, douglas).

ancestor(bob, douglas).

ancestor(ebbon, john).

ancestor(ebbon, douglas).

> ancestor(X,john)?

ancestor(bob, john).

ancestor(ebbon, john).
```

A fact or a rule can be retracted from the database using tilde syntax:

```
> parent(bob, john)~

> parent(A, B)?

parent(john, douglas).

parent(ebbon, bob).

> ancestor(A, B)?

ancestor(ebbon, bob).

ancestor(john, douglas).
```

Unlike Prolog, the order in which clauses are asserted is irrelevant. All queries terminate, and every possible answer is derived.

```
> q(X) :- p(X).

> q(a).

> p(X) :- q(X).

> q(X)?

q(a).
```

3 Parenthetical Datalog Module Language

`#lang datalog/sexp` package: `datalog`

The semantics of this language is the same as the normal Datalog language, except it uses the parenthetical syntax described in §4 “Racket Interoperability”.

All identifiers in `racket/base` are available for use as predicate symbols or constant values. Top-level identifiers and datums are not otherwise allowed in the program. The program may contain `require` expressions.

The following is a program:

```
#lang datalog/sexp

(! (edge a b))
(! (edge b c))
(! (edge c d))
(! (edge d a))
(! (:- (path X Y)
       (edge X Y)))
(! (:- (path X Y)
       (edge X Z)
       (path Z Y)))
(? (path X Y))
```

This is also a program:

```
#lang datalog/sexp
(require racket/math)

(? (sqr 4 :- X))
```

The Parenthetical Datalog REPL accepts new statements that are executed as if they were in the original program text, except `require` is not allowed.

4 Racket Interoperability

`(require datalog)` package: `datalog`

The Datalog database can be directly used by Racket programs through this API.

Examples:

```
> (define family (make-theory))
> (datalog family
           (! (parent joseph2 joseph1))
           (! (parent joseph2 lucy))
           (! (parent joseph3 joseph2)))
'()
> (datalog family
           (? (parent X joseph2)))
'(#hasheq((X . joseph3)))
> (datalog family
           (? (parent X (string->symbol "joseph2"))))
'(#hasheq((X . joseph3)))
> (let ([atom 'joseph2])
    (datalog family
             (? (parent X #,atom))))
'(#hasheq((X . joseph3)))
> (let ([table 'parent])
    (datalog family
             (? (#,table X joseph2))))
'(#hasheq((X . joseph3)))
> (datalog family
           (? (parent joseph2 X)))
'(#hasheq((X . joseph1)) #hasheq((X . lucy)))
> (datalog family
           (? (parent joseph2 X))
           (? (parent X joseph2)))
'(#hasheq((X . joseph3)))
> (datalog family
           (! (:- (ancestor A B)
                  (parent A B)))
           (! (:- (ancestor A B)
                  (parent A C)
                  (= D C)
                  (ancestor D B))))
'()
> (datalog family
           (? (ancestor A B)))
'(#hasheq((A . joseph3) (B . joseph2))
```

```
  #hasheq((A . joseph2) (B . lucy))
  #hasheq((A . joseph2) (B . joseph1))
  #hasheq((A . joseph3) (B . lucy))
  #hasheq((A . joseph3) (B . joseph1)))
> (let ([x 'joseph2])
    (datalog family
             (? (parent x X))))
'(#hasheq((X . joseph1)) #hasheq((X . lucy)))
> (datalog family
           (? (add1 1 :- X)))
'(#hasheq((X . 2)))
> (datalog family
           (? (add1 X :- 2)))
'()
> (datalog family
           (? (#,(λ (x) (+ x 1)) 1 :- X)))
'(#hasheq((X . 2)))
```

```
theory/c : contract?
```

A contract for Datalog theories.

```
(make-theory) → theory/c
```

Creates a theory for use with `datalog`.

```
(write-theory t [out]) → void?
  t : theory/c
  out : output-port? = (current-output-port)
```

Writes *t* to *out*. Source location information is lost.

```
(read-theory [in]) → theory/c
  in : input-port? = (current-input-port)
```

Reads and returns a theory from *in*.

```
(datalog thy-expr
         stmt ...)

  thy-expr : theory/c
```

Executes the statements on the theory given by *thy-expr*. Returns the answers to the final query as a list of substitution dictionaries or returns `empty`.

```
(datalog! thy-expr
          stmt ...)

  thy-expr : theory/c
```

Executes the statements on the theory given by *thy-expr*. Prints the answers to every query in the list of statements. Returns (void).

Statements are either assertions, retractions, or queries.

```
(! clause)
```

Asserts the clause.

```
(~ clause)
```

Retracts the literal.

```
(:- literal question ...)
```

A conditional clause.

```
(? question)
```

Queries the literal and prints the result literals.

Questions are either literals or external queries.

Literals are represented as identifier or (table term ...).

A table is either an identifier or #,expr where expr evaluates to a symbol.

External queries are represented as (ext-table term ... :- term ...), where ext-table is an identifier bound to a procedure or #,expr where expr evaluates to a procedure that when given the first set of terms as arguments returns the second set of terms as values.

A term is either a non-capitalized identifiers for a constant symbol, a Racket expression for a constant datum, or a capitalized identifier for a variable symbol, or #,expr where expr evaluates to a constant datum. Bound identifiers in terms are treated as the datum they are bound to.

External queries fail if any logic variable is not fully resolved to a datum on the Datalog side. In other words, unbound logic variables never flow to Racket.

External queries invalidate Datalog's guaranteed termination. For example, this program does not terminate:

```
(datalog (make-theory)
         (! (:- (loop X)
                (add1 X :- Z)
                (loop Z)))
         (? (loop 1)))
```

5 Acknowledgments

This package was once structurally based on Dave Herman's `(planet dherman/javascript)` library and John Ramsdell's Datalog library.

The package uses the tabled logic programming algorithm described in Efficient Top-Down Computation of Queries under the Well-Founded Semantics by W. Chen, T. Swift, and D. S. Warren. Another important reference is Tabled Evaluation with Delaying for General Logic Programs by W. Chen and D. S. Warren. Datalog is described in What You Always Wanted to Know About Datalog (And Never Dared to Ask) by Stefano Ceri, Georg Gottlob, and Letizia Tanca.