# Evaluation of QL programs

A QL program is evaluated in a number of different steps.

## Process

When a QL program is run against a database, it is compiled into a variant of the logic programming language Datalog. It is optimized for performance, and then evaluated to produce results.

These results are sets of ordered tuples. An ordered tuple is a finite, ordered sequence of values. For example, `(1, 2, "three")` is an ordered tuple with two integers and a string. There may be intermediate results produced while the program is being evaluated: these are also sets of tuples.

A QL program is evaluated from the bottom up, so a predicate is usually only evaluated after all the predicates it depends on are evaluated.

The database includes sets of ordered tuples for the built-in predicates and external predicates. Each evaluation starts from these sets of tuples. The remaining predicates and types in the program are organized into a number of layers, based on the dependencies between them. These layers are evaluated to produce their own sets of tuples, by finding the least fixed point of each predicate. (For example, see "Recursion.")

The program's queries determine which of these sets of tuples make up the final results of the program. The results are sorted according to any ordering directives (`order by`) in the queries.

For more details about each step of the evaluation process, see the "QL language specification."

## Validity of programs

The result of a query must always be a **finite** set of values, otherwise it can't be evaluated. If your QL code contains an infinite predicate or query, the QL compiler usually gives an error message, so that you can identify the error more easily.

Here are some common ways that you might define infinite predicates. These all generate compilation errors:

- The following query conceptually selects all values of type `int`, without restricting them. The QL compiler returns the error `'i' is not bound to a value`:

```
from int i
select i
```

- The following predicate generates two errors: `'n' is not bound to a value` and `'result' is not bound to a value`:

```
int timesTwo(int n) {
  result = n * 2
}
```

- The following class `Person` contains all strings that start with `"Peter"`. There are infinitely many such strings, so this is another invalid definition. The QL compiler gives the error message `'this' is not bound to a value`:

```
class Person extends string {
  Person() {
    this.matches("Peter%")
  }
}
```

To fix these errors, it’s useful to think about **range restriction**: A predicate or query is **range-restricted** if each of its variables has at least one binding occurrence. A variable without a binding occurrence is called **unbound**. Therefore, to perform a range restriction check, the QL compiler verifies that there are no unbound variables.

## Binding

To avoid infinite relations in your queries, you must ensure that there are no unbound variables. To do this, you can use the following mechanisms:

1. **Finite types**: Variables of a finite type are bound. In particular, any type that is not primitive is finite. To give a finite type to a variable, you can declare it with a finite type, use a cast, or use a type check.
2. **Predicate calls**: A valid predicate is usually range-restricted, so it binds all its arguments. Therefore, if you call a predicate on a variable, the variable becomes bound.

> **Important**
>
> If a predicate uses non-standard binding sets, then it does **not** always bind all its arguments. In such a case, whether the predicate call binds a specific argument depends on which other arguments are bound, and what the binding sets say about the argument in question. For more information, see “Binding sets.”

3. **Binding operators**: Most operators, such as the arithmetic operators, require that all their operands are bound. For example, you can’t add two variables in QL unless you have a finite set of possible values for both of them.

   However, there are some built-in operators that can bind their arguments. For example, if one side of an equality check (using `=`) is bound and the other side is a variable, then the variable becomes bound too. See the table below for examples.

Intuitively, a binding occurrence restricts the variable to a finite set of values, while a non-binding occurrence doesn’t. Here are some examples to clarify the difference between binding and non-binding occurrences of variables:

| Variable occurrence | Details |
|---|---|
| `x = 1` | Binding: restricts `x` to the value `1` |

| Variable occurrence | Details |
|---|---|
| `x != 1`, `not x = 1` | Not binding |
| `x = 2 + 3`, `x + 1 = 3` | Binding |
| `x in [0 .. 3]` | Binding |
| `p(x, _)` | Binding, since `p()` is a call to a predicate. |
| `x = y`, `x = y + 1` | Binding for `x` if and only if the variable `y` is bound. Binding for `y` if and only if the variable `x` is bound. |
| `x = y * 2` | Binding for `x` if the variable `y` is bound. Not binding for `y`. |
| `x > y` | Not binding for `x` or `y` |
| `"string".matches(x)` | Not binding for `x` |
| `x.matches(y)` | Not binding for `x` or `y` |
| `not (... x ...)` | Generally non-binding for `x`, since negating a binding occurrence typically makes it non-binding. There are certain exceptions: `not not x = 1` is correctly recognized as binding for `x`. |
| `sum(int y \| y = 1 and x = y \| y)` | Not binding for `x`. `strictsum(int y \| y = 1 and x = y \| y)` would be binding for `x`. Expressions in the body of an aggregate are only binding outside of the body if the aggregate is *strict*. |
| `x = 1 or y = 1` | Not binding for `x` or for `y`. The first subexpression, `x = 1`, is binding for `x`, and the second subexpression, `y = 1`, is binding for `y`. However, combining them with disjunction is only binding for variables for which **all** disjuncts are binding—in this case, that's no variable. |

While the occurrence of a variable can be binding or non-binding, the variable's property of being "bound" or "unbound" is a global concept—a single binding occurrence is enough to make a variable bound.

Therefore, you could fix the "infinite" examples above by providing a binding occurrence. For example, instead of `int timesTwo(int n) { result = n * 2 }`, you could write:

```
int timesTwo(int n) {
  n in [0 .. 10] and
  result = n * 2
}
```

The predicate now binds `n`, and the variable `result` automatically becomes bound by the computation `result = n * 2`.