# Documentation

☰ NAVIGATION

# The Bazel Query Reference

✎ Edit (https://github.com/bazelbuild/bazel/tree/master/site/docs/query.html)


- Examples (query.html#examples)
- Tokens: The Lexical Syntax (query.html#tokens)
- Bazel Query Language Concepts (query.html#concepts)
- Expressions: Syntax and Semantics of the Grammar (query.html#expressions)
- Functions (query.html#functions)
- Output Formats (query.html#output-formats)


When you use `bazel query` to analyze build dependencies, you use a little language, the *Bazel Query Language*. This document is the reference manual for that language. This document also describes the output formats `bazel query` supports.

## Examples

How do people use `bazel query`? Here are typical examples:

Why does the `//foo` tree depend on `//bar/baz`? Show a path:

```
somepath(foo/..., //bar/baz:all)
```

What C++ libraries do all the `foo` tests depend on that the `foo_bin` target does not?

```
kind("cc_library", deps(kind(".*test rule", foo/...)) except deps(//foo:foo_bin))
```

## Tokens: The Lexical Syntax

Expressions in the query language are composed of the following tokens:

- **Keywords**, such as `let`. Keywords are the reserved words of the language, and each of them is described below. The complete set of keywords is:

  ```
  except
  in
  ```

```
intersect
let
set
union
```

- **Words**, such as `foo/...` or `".*test rule"` or `//bar/baz:all`. If a character sequence is "quoted" (begins and ends with a single-quote `'`, or begins and ends with a double-quote `"`), it is a word. If a character sequence is not quoted, it may still be parsed as a word. Unquoted words are sequences of characters drawn from the set of alphabet characters, numerals, slash `/`, hyphen `-`, underscore `_`, star `*`, and period `.`. Unquoted words may not start with a hyphen or period.

  We chose this syntax so that quote marks aren't needed in most cases. The (unusual) `".*test rule"` example needs quotes: it starts with a period and contains a space. Quoting `"cc_library"` is unnecessary but harmless.

  Quoting *is* necessary when writing scripts that construct Bazel query expressions from user-supplied values.

  ```
  //foo:bar+wiz    # WRONG: scanned as //foo:bar + wiz.
  //foo:bar=wiz    # WRONG: scanned as //foo:bar = wiz.
  "//foo:bar+wiz"  # ok.
  "//foo:bar=wiz"  # ok.
  ```

  Note that this quoting is in addition to any quoting that may be required by your shell. e.g.

  ```
  bazel query ' "//foo:bar=wiz" '     # single-quotes for shell, double-quotes for Baze
  ```

  Keywords, when quoted, are treated as ordinary words, thus `some` is a keyword but `"some"` is a word. Both `foo` and `"foo"` are words.

- **Punctuation**, such as parens `()`, period `.` and comma `,`, etc. Words containing punctuation (other than the exceptions listed above) must be quoted.

Whitespace characters outside of a quoted word are ignored.

# Bazel Query Language Concepts

The Bazel query language is a language of expressions. Every expression evaluates to a **partially-ordered set** of targets, or equivalently, a **graph** (DAG) of targets. This is the only datatype.

In some expressions, the partial order of the graph is not interesting; In this case, we call the values "sets". In cases where the partial order of elements is significant, we call values "graphs". Note that both terms refer to the same datatype, but merely emphasize different aspects of it.

## Cycles in the dependency graph

Build dependency graphs should be acyclic. The algorithms used by the query language are intended for use in acyclic graphs, but are robust against cycles. The details of how cycles are treated are not specified and should not be relied upon.

## Implicit dependencies

In addition to build dependencies that are defined explicitly in BUILD files, Bazel adds additional *implicit* dependencies to rules. For example every Java rule implicitly depends on the JavaBuilder. Implicit dependencies are established using attributes that start with `$` and they cannot be overridden in BUILD files.

Per default `bazel query` takes implicit dependencies into account when computing the query result. This behavior can be changed with the `--[no]implicit_deps` option.

## Soundness

Bazel query language expressions operate over the build dependency graph, which is the graph implicitly defined by all rule declarations in all BUILD files. It is important to understand that this graph is somewhat abstract, and does not constitute a complete description of how to perform all the steps of a build. In order to perform a build, a *configuration* is required too; see the configurations (user-manual.html#configurations) section of the User's Guide for more detail.

The result of evaluating an expression in the Bazel query language is true *for all configurations*, which means that it may be a conservative over-approximation, and not exactly precise. If you use the query tool to compute the set of all source files needed during a build, it may report more than are actually necessary because, for example, the query tool will include all the files needed to support message translation, even though you don't intend to use that feature in your build.

## On the preservation of graph order

Operations preserve any ordering constraints inherited from their subexpressions. You can think of this as "the law of conservation of partial order". Consider an example: if you issue a query to determine the transitive closure of dependencies of a particular target, the resulting set is ordered according to the dependency graph. If you filter that set to include only the targets of `file` kind, the same *transitive* partial ordering relation holds between every pair of targets in the resulting subset—even though none of these pairs is actually directly connected in the original graph. (There are no file–file edges in the build dependency graph).

However, while all operators *preserve* order, some operations, such as the set operations don't *introduce* any ordering constraints of their own. Consider this expression:

```
deps(x) union y
```

The order of the final result set is guaranteed to preserve all the ordering constraints of its subexpressions, namely, that all the transitive dependencies of `x` are correctly ordered with respect to each other. However, the query guarantees nothing about the ordering of the targets in `y`, nor about the ordering of the targets in `deps(x)` relative to those in `y` (except for those targets in `y` that also happen to be in `deps(x)`).

Operators that introduce ordering constraints include: `allpaths`, `deps`, `rdeps`, `somepath`, and the target pattern wildcards `package:*`, `dir/...`, etc.

## Sky Query

Query has two different implementations, with slightly different features. The alternative one is called "Sky Query", and is activated by passing the following two flags: `--universe_scope` and `--order_output=no`. `--universe_scope=<target_pattern1>,...,<target_patternN>` tells query to preload the transitive closure of the target pattern specified by the target patterns, which can be both additive and subtractive. All queries are then evaluated in this "scope". In particular, the `allrdeps` and `rbuildfiles` operators only return results from this scope.

Sky Query has some advantages and disadvantages compared to the default query. The main disadvantage is that it cannot order its output according to graph order, and thus certain output formats are forbidden. Its advantages are that it provides two operators ( `allrdeps` and `rbuildfiles` ) that are not available in the default query. As well, Sky Query does its work by introspecting the Skyframe (https://bazel.build/designs/skyframe.html) graph, rather than creating a new graph, which is what the default implementation does. Thus, there are some circumstances in which it is faster and uses less memory.

## Expressions: Syntax and Semantics of the Grammar

This is the grammar of the Bazel query language, expressed in EBNF notation:

```
expr ::= word
       | let name = expr in expr
       | (expr)
       | expr intersect expr
       | expr ^ expr
       | expr union expr
       | expr + expr
       | expr except expr
       | expr - expr
       | set(word *)
       | word '(' int | word | expr ... ')'
```

We will examine each of the productions of this grammar in order.

### Target patterns

```
expr ::= word
```

Syntactically, a *target pattern* is just a word. It is interpreted as an (unordered) set of targets. The simplest target pattern is a label, which identifies a single target (file or rule). For example, the target pattern `//foo:bar` evaluates to a set containing one element, the target, the `bar` rule.

Target patterns generalize labels to include wildcards over packages and targets. For example, `foo/...:all` (or just `foo/...` ) is a target pattern that evaluates to a set containing all *rules* in every package recursively beneath the `foo` directory; `bar/baz:all` is a target pattern that evaluates to a set containing all the rules in the `bar/baz` package, but not its subpackages.

Similarly, `foo/...:*` is a target pattern that evaluates to a set containing all *targets* (rules *and* files) in every package recursively beneath the `foo` directory; `bar/baz:*` evaluates to a set containing all the targets in the `bar/baz` package, but not its subpackages.

Because the `:*` wildcard matches files as well as rules, it is often more useful than `:all` for queries. Conversely, the `:all` wildcard (implicit in target patterns like `foo/...`) is typically more useful for builds.

`bazel query` target patterns work the same as `bazel build` build targets do; refer to Target Patterns (bazel-user-manual.html#target-patterns) in the Bazel User Manual for further details, or type `bazel help target-syntax`.

Target patterns may evaluate to a singleton set (in the case of a label), to a set containing many elements (as in the case of `foo/...`, which has thousands of elements) or to the empty set, if the target pattern matches no targets.

All nodes in the result of a target pattern expression are correctly ordered relative to each other according to the dependency relation. So, the result of `foo:*` is not just the set of targets in package `foo`, it is also the *graph* over those targets. (No guarantees are made about the relative ordering of the result nodes against other nodes.) See the section on graph order for more details.

## Variables

```
expr ::= let name = expr1 in expr2
       | $name
```

The Bazel query language allows definitions of and references to variables. The result of evaluation of a `let` expression is the same as that of $expr_2$, with all free occurrences of variable *name* replaced by the value of $expr_1$.

For example, `let v = foo/... in allpaths($v, //common) intersect $v` is equivalent to the `allpaths(foo/..., //common) intersect foo/...`.

An occurrence of a variable reference `name` other than in an enclosing `let name = ...` expression is an error. In other words, toplevel query expressions cannot have free variables.

In the above grammar productions, `name` is like *word*, but with the additional constraint that it be a legal identifier in the C programming language. References to the variable must be prepended with the "$" character.

Each `let` expression defines only a single variable, but you can nest them.

(Both target patterns and variable references consist of just a single token, a word, creating a syntactic ambiguity. However, there is no semantic ambiguity, because the subset of words that are legal variable names is disjoint from the subset of words that are legal target patterns.)

(Technically speaking, `let` expressions do not increase the expressiveness of the query language: any query expressible in the language can also be expressed without them. However, they improve the conciseness of many queries, and may also lead to more efficient query evaluation.)

## Parenthesized expressions

```
expr ::= (expr)
```

Parentheses associate subexpressions to force an order of evaluation. A parenthesized expression evaluates to the value of its argument.

## Algebraic set operations: intersection, union, set difference

```
expr ::= expr intersect expr
       | expr ^ expr
       | expr union expr
       | expr + expr
       | expr except expr
       | expr - expr
```

These three operators compute the usual set operations over their arguments. Each operator has two forms, a nominal form such as `intersect` and a symbolic form such as `^`. Both forms are equivalent; the symbolic forms are quicker to type. (For clarity, the rest of this manual uses the nominal forms.) For example,

```
foo/... except foo/bar/...
```

evaluates to the set of targets that match `foo/...` but not `foo/bar/...`. Equivalently:

```
foo/... - foo/bar/...
```

The `intersect` (`^`) and `union` (`+`) operations are commutative (symmetric); `except` (`-`) is asymmetric. The parser treats all three operators as left-associative and of equal precedence, so you might want parentheses. For example, the first two of these expressions are equivalent, but the third is not:

```
x intersect y union z
(x intersect y) union z
x intersect (y union z)
```

(We strongly recommend that you use parentheses where there is any danger of ambiguity in reading a query expression.)

## Read targets from an external source: set

```
expr ::= set(word *)
```

The `set(a b c ...)` operator computes the union of a set of zero or more target patterns, separated by whitespace (no commas).

In conjunction with the Bourne shell's `$(...)` feature, `set()` provides a means of saving the results of one query in a regular text file, manipulating that text file using other programs (e.g. standard UNIX shell tools), and then introducing the result back into the query tool as a value for further processing. For example:

```
bazel query deps(//my:target) --output=label | grep ... | sed ... | awk ... > foo
bazel query "kind(cc_binary, set($(<foo)))"
```

In the next example, `kind(cc_library, deps(//some_dir/foo:main, 5))` is effectively computed by filtering on the `maxrank` values using an `awk` program.

```
bazel query 'deps(//some_dir/foo:main)' --output maxrank |
      awk '($1 < 5) { print $2;} ' > foo
bazel query "kind(cc_library, set($(<foo)))"
```

In these examples, `$(<foo)` is a shorthand for `$(cat foo)`, but shell commands other than `cat` may be used

too—such as the previous `awk` command.

Note, `set()` introduces no graph ordering constraints, so path information may be lost when saving and reloading sets of nodes using it. See the graph order section below for more detail.

# Functions

```
expr ::= word '(' int | word | expr ... ')'
```

The query language defines several functions. The name of the function determines the number and type of arguments it requires. The following functions are available:

`allpaths`
`attr`
`buildfiles`
`rbuildfiles`
`deps`
`filter`
`kind`
`labels`
`loadfiles`
`rdeps`
`allrdeps`
`siblings`
`some`
`somepath`
`tests`
`visible`

## Transitive closure of dependencies: deps

```
expr ::= deps(expr)
       | deps(expr, depth)
```

The `deps(x)` operator evaluates to the graph formed by the transitive closure of dependencies of its argument set *x*. For example, the value of `deps(//foo)` is the dependency graph rooted at the single node `foo`, including all its dependencies. The value of `deps(foo/...)` is the dependency graphs whose roots are all rules in every package beneath the `foo` directory. Please note that 'dependencies' means only rule and file targets in this context, therefore the BUILD, and Skylark files needed to create these targets are not included here. For that you should use the `buildfiles` operator.

The resulting graph is ordered according to the dependency relation. See the section on graph order for more details.

The `deps` operator accepts an optional second argument, which is an integer literal specifying an upper bound on the depth of the search. So `deps(foo:*, 1)` evaluates to all the direct prerequisites of any target in the `foo` package, and `deps(foo:*, 2)` further includes the nodes directly reachable from the nodes in `deps(foo:*, 1)`, and so on. (These numbers correspond to the ranks shown in the `minrank` output format.) If the *depth* parameter is omitted, the search is unbounded, i.e. it computes the reflexive transitive closure of prerequsites.

## Transitive closure of reverse dependencies: rdeps

```
expr ::= rdeps(expr, expr)
       | rdeps(expr, expr, depth)
```

The `rdeps(u, x)` operator evaluates to the reverse dependencies of the argument set *x* within the transitive closure of the universe set *u*.

The resulting graph is ordered according to the dependency relation. See the section on graph order for more details.

The `rdeps` operator accepts an optional third argument, which is an integer literal specifying an upper bound on the depth of the search. The resulting graph will only include nodes within a distance of the specified depth from any node in the argument set. So `rdeps(//foo, //common, 1)` evaluates to all nodes in the transitive closure of `//foo` that directly depend on `//common`. (These numbers correspond to the ranks shown in the `minrank` output format.) If the *depth* parameter is omitted, the search is unbounded.

## Transitive closure of all reverse dependencies: allrdeps

```
expr ::= allrdeps(expr)
       | allrdeps(expr, depth)
```

**Only available with Sky Query**
The `allrdeps` operator behaves just like the `rdeps` operator, except that the "universe set" is whatever the `--universe_scope` flag evaluated to, instead of being separately specified. Thus, if `--universe_scope=//foo/...` was passed, then `allrdeps(//bar)` is equivalent to `rdeps(//bar, //foo/...)`.

## Dealing with a target's package: siblings

```
expr ::= siblings(expr)
```

The `siblings(x)` operator evalutes to the full set of targets that are in the same package as a target in the argument set.

## Arbitrary choice: some

```
expr ::= some(expr)
```

The `some(x)` operator selects one target arbitrarily from its argument set *x*, and evaluates to a singleton set containing only that target. For example, the expression `some(//foo:main union //bar:baz)` evaluates to a set containing either `//foo:main` or `//bar:baz` —though which one is not defined.

If the argument is a singleton, then `some` computes the identity function: `some(//foo:main)` is equivalent to `//foo:main`. It is an error if the specified argument set is empty, as in the expression `some(//foo:main` `intersect //bar:baz)`.

## Path operators: somepath, allpaths

```
expr ::= somepath(expr, expr)
       | allpaths(expr, expr)
```

The `somepath(S, E)` and `allpaths(S, E)` operators compute paths between two sets of targets. Both queries accept two arguments, a set *S* of starting points and a set *E* of ending points. `somepath` returns the graph of nodes on *some* arbitrary path from a target in *S* to a target in *E*; `allpaths` returns the graph of nodes on *all* paths from any target in *S* to any target in *E*.

The resulting graphs are ordered according to the dependency relation. See the section on graph order for more details.

| `somepath(S1 + S2, E)`, one possible result. | `somepath(S1 + S2, E)`, another possible result. | `allpaths(S1 + S2, E)`. |
|---|---|---|

## Target kind filtering: kind

```
expr ::= kind(word, expr)
```

The `kind(pattern, input)` operator applies a filter to a set of targets, and discards those targets that are not of the expected kind. The *pattern* parameter specifies what kind of target to match.

- **file** patterns can be one of:
  - `source file`
  - `generated file`
- **rule** patterns can be one of:
  - *`ruletype`* `rule`
  - *`ruletype`*
    Where *ruletype* is a build rule. The difference between these forms is that including "rule" causes the regular expression match for *ruletype* to be anchored.
- **package group** patterns should simply be:
  - `package group`

For example, the kinds for the four targets defined by the BUILD file (for package `p`) shown below are illustrated in the table:

```
genrule(
    name = "a",
    srcs = ["a.in"],
    outs = ["a.out"],
    cmd = "...",
)
```

| Target | Kind |
|---|---|
| //p:a | genrule rule |
| //p:a.in | source file |
| //p:a.out | generated file |
| //p:BUILD | source file |

Thus, `kind("cc_.* rule", foo/...)` evaluates to the set of all `cc_library`, `cc_binary`, etc, rule targets beneath `foo`, and `kind("source file", deps(//foo))` evaluates to the set of all source files in the transitive closure of dependencies of the `//foo` target.

Quotation of the *pattern* argument is often required because without it, many regular expressions, such as `source file` and `.*_test`, are not considered words by the parser.

When matching for `package group`, targets ending in `:all` may not yield any results. Use `:all-targets` instead.

## Target name filtering: filter

```
expr ::= filter(word, expr)
```

The `filter(pattern, input)` operator applies a filter to a set of targets, and discards targets whose labels (in absolute form) do not match the pattern; it evaluates to a subset of its input.

The first argument, *pattern* is a word containing a regular expression over target names. A `filter` expression evaluates to the set containing all targets *x* such that *x* is a member of the set *input* and the label (in absolute form, e.g. `//foo:bar`) of *x* contains an (unanchored) match for the regular expression *pattern*. Since all target names start with `//`, it may be used as an alternative to the `^` regular expression anchor.

This operator often provides a much faster and more robust alternative to the `intersect` operator. For example, in order to see all `bar` dependencies of the `//foo:foo` target, one could evaluate

```
deps(//foo) intersect //bar/...
```

This statement, however, will require parsing of all BUILD files in the `bar` tree, which will be slow and prone to errors in irrelevant BUILD files. An alternative would be:

```
filter(//bar, deps(//foo))
```

which would first calculate the set of `//foo` dependencies and then would filter only targets matching the provided pattern—in other words, targets with names containing `//bar` as a substring.

Another common use of the `filter(pattern, expr)` operator is to filter specific files by their name or extension. For example,

```
filter("\.cc$", deps(//foo))
```

will provide a list of all `.cc` files used to build `//foo`.

## Rule attribute filtering: attr

```
expr ::= attr(word, word, expr)
```

The `attr(name, pattern, input)` operator applies a filter to a set of targets, and discards targets that are not rules, rule targets that do not have attribute *name* defined or rule targets where the attribute value does not match the provided regular expression *pattern*; it evaluates to a subset of its input.

The first argument, *name* is the name of the rule attribute that should be matched against the provided regular expression pattern. The second argument, *pattern* is a regular expression over the attribute values. An `attr` expression evaluates to the set containing all targets *x* such that *x* is a member of the set *input*, is a rule with the defined attribute *name* and the attribute value contains an (unanchored) match for the regular expression *pattern*. Please note, that if *name* is an optional attribute and rule does not specify it explicitly then default attribute value will be used for comparison. For example,

```
attr(linkshared, 0, deps(//foo))
```

will select all `//foo` dependencies that are allowed to have a linkshared attribute (e.g., `cc_binary` rule) and have it either explicitly set to 0 or do not set it at all but default value is 0 (e.g. for `cc_binary` rules).

List-type attributes (such as `srcs`, `data`, etc) are converted to strings of the form `[value`$_1$`, ..., value`$_n$`]`, starting with a `[` bracket, ending with a `]` bracket and using "`, `" (comma, space) to delimit multiple values. Labels are converted to strings by using the absolute form of the label. For example, an attribute `deps=[":foo", "//otherpkg:bar", "wiz"]` would be converted to the string `[//thispkg:foo, //otherpkg:bar, //thispkg:wiz]`. Brackets are always present, so the empty list would use string value `[]` for matching purposes. For example,

```
attr("srcs", "\[\]", deps(//foo))
```

will select all rules among `//foo` dependencies that have an empty `srcs` attribute, while

```
attr("data", ".{3,}", deps(//foo))
```

will select all rules among `//foo` dependencies that specify at least one value in the `data` attribute (every label is at least 3 characters long due to the `//` and `:`).

## Rule visibility filtering: visible

```
expr ::= visible(expr, expr)
```

The `visible(predicate, input)` operator applies a filter to a set of targets, and discards targets without the required visibility.

The first argument, *predicate*, is a set of targets that all targets in the output must be visible to. A *visible* expression evaluates to the set containing all targets *x* such that *x* is a member of the set *input*, and for all targets *y* in *predicate* *x* is visible to *y*. For example:

```
visible(//foo, //bar:*)
```

will select all targets in the package `//bar` that `//foo` can depend on without violating visibility restrictions.

## Evaluation of rule attributes of type label: labels

```
expr ::= labels(word, expr)
```

The `labels(attr_name, inputs)` operator returns the set of targets specified in the attribute *attr_name* of type "label" or "list of label" in some rule in set *inputs*.

For example, `labels(srcs, //foo)` returns the set of targets appearing in the `srcs` attribute of the `//foo` rule. If there are multiple rules with `srcs` attributes in the *inputs* set, the union of their `srcs` is returned.

## Expand and filter test_suites: tests

```
expr ::= tests(expr)
```

The `tests(x)` operator returns the set of all test rules in set *x*, expanding any `test_suite` rules into the set of individual tests that they refer to, and applying filtering by `tag` and `size`. By default, query evaluation ignores any non-test targets in all `test_suite` rules. This can be changed to errors with the `--strict_test_suite` option.

For example, the query `kind(test, foo:*)` lists all the `*_test` and `test_suite` rules in the `foo` package. All the results are (by definition) members of the `foo` package. In contrast, the query `tests(foo:*)` will return all of the individual tests that would be executed by `bazel test foo:*`: this may include tests belonging to other packages, that are referenced directly or indirectly via `test_suite` rules.

## Package definition files: buildfiles

```
expr ::= buildfiles(expr)
```

The `buildfiles(x)` operator returns the set of files that define the packages of each target in set *x*; in other words, for each package, its BUILD file, plus any files it references via `load`. Note that this also returns the BUILD files of the packages containing these `load`ed files.

This operator is typically used when determining what files or packages are required to build a specified target, often in conjunction with the `--output package` option, below). For example,

```
bazel query 'buildfiles(deps(//foo))' --output package
```

returns the set of all packages on which `//foo` transitively depends.

(Note: a naive attempt at the above query would omit the `buildfiles` operator and use only `deps`, but this yields an incorrect result: while the result contains the majority of needed packages, those packages that contain only files that are `load()`'ed will be missing.

## Package definition files: rbuildfiles

```
expr ::= rbuildfiles(expr)
```

**Only available with Sky Query**
The `rbuildfiles(x)` operator returns the set of "buildfiles" (BUILD and .bzl files) that depend on *x*, where *x* is a list

of path fragments for buildfiles. For instance, if `//foo` is a package, then `rbuildfiles(foo/BUILD)` will return the `//foo:BUILD` target. If the `foo/BUILD` file has `load('//bar:file.bzl'...` in it, then `rbuildfiles(bar/file.bzl)` will return the `//foo:BUILD` target and the `//bar:file.bzl` target, as well as the targets for any other BUILD files and .bzl files that load `//bar:file.bzl`

The scope of the rbuildfiles operator is the universe specified by the `--universe_scope` flag. Files that do not correspond directly to BUILD files and .bzl files do not affect the results. For instance, source files (like `foo.cc`) are ignored, even if they are explicitly mentioned in the BUILD file. Symlinks, however, are respected, so that if `foo/BUILD` is a symlink to `bar/BUILD`, then `rbuildfiles(bar/BUILD)` will include `//foo:BUILD` in its results.

The `rbuildfiles` operator is morally the inverse of the `buildfiles` operator. However, this moral inversion holds more strongly in one direction: the outputs of `rbuildfiles` are just like the inputs of `buildfiles`, since both are targets corresponding to packages and .bzl files. In the other direction, the correspondence is weaker. The outputs of the `buildfiles` operator are targets corresponding to all packages and .bzl files needed by a given input. However, the inputs of the `rbuildfiles` operator are not those targets, but rather the path fragments that correspond to those targets.

## Package definition files: loadfiles

```
expr ::= loadfiles(expr)
```

The `loadfiles(x)` operator returns the set of Skylark files that are needed to load the packages of each target in set *x*. In other words, for each package, it returns the .bzl files that are referenced from its BUILD files.

# Output Formats

`bazel query` generates a graph. You specify the content, format, and ordering by which `bazel query` presents this graph by means of the `--output` command-line option.

When running with Sky Query (sky-query), only output formats that are compatible with unordered output are allowed. Specifically, `graph`, `minrank`, and `maxrank` output formats are forbidden.

Some of the output formats accept additional options. The name of each output option is prefixed with the output format to which it applies, so `--graph:factored` applies only when `--output=graph` is being used; it has no effect if an output format other than `graph` is used. Similarly, `--xml:line_numbers` applies only when `--output=xml` is being used.

## On the ordering of results

Although query expressions always follow the "law of conservation of graph order", *presenting* the results may be done in either a dependency-ordered or unordered manner. This does **not** influence the targets in the result set or how the query is computed. It only affects how the results are printed to stdout. Moreover, nodes that are equivalent in the dependency order may or may not be ordered alphabetically. The `--order_output` flag can be used to control this behavior. (The `--[no]order_results` flag has a subset of the functionality of the `--order_output` flag and is deprecated.)

The default value of this flag is `auto`, which is equivalent to `full` for every output format except for `proto`, `graph`, `minrank`, and `maxrank`, for which it is equivalent to `deps`.

When this flag is `no` and `--output` is one of `build`, `label`, `label_kind`, `location`, `package`, `proto`, `record` or `xml`, the outputs will be printed in arbitrary order. **This is generally the fastest option**. It is not supported though when `--output` is one of `graph`, `min_rank` or `max_rank`: with these formats, bazel will always print results ordered by the dependency order or rank.

When this flag is `deps`, bazel will print results ordered by the dependency order. However, nodes that are unordered by the dependency order (because there is no path from either one to the other) may be printed in any order.

When this flag is `full`, bazel will print results ordered by the dependency order, with unordered nodes ordered alphabetically or reverse alphabetically, depending on the output format. This may be slower than the other options, and so should only be used when deterministic results are important — it is guaranteed with this option that running the same query multiple times will always produce the same output.

## Print the source form of targets as they would appear in BUILD

```
--output build
```

With this option, the representation of each target is as if it were hand-written in the BUILD language. All variables and function calls (e.g. glob, macros) are expanded, which is useful for seeing the effect of Skylark macros. Additionally, each effective rule is annotated with the name of the macro (if any, see `generator_name` and `generator_function`) that produced it.

Although the output uses the same syntax as BUILD files, it is not guaranteed to produce a valid BUILD file.

## Print the label of each target

```
--output label
```

With this option, the set of names (or *labels*) of each target in the resulting graph is printed, one label per line, in topological order (unless `--noorder_results` is specified, see notes on the ordering of results). (A topological ordering is one in which a graph node appears earlier than all of its successors.) Of course there are many possible topological orderings of a graph (*reverse postorder* is just one); which one is chosen is not specified. When printing the output of a `somepath` query, the order in which the nodes are printed is the order of the path.

Caveat: in some corner cases, there may be two distinct targets with the same label; for example, a `sh_binary` rule and its sole (implicit) `srcs` file may both be called `foo.sh`. If the result of a query contains both of these targets, the output (in `label` format) will appear to contain a duplicate. When using the `label_kind` (see below) format, the distinction becomes clear: the two targets have the same name, but one has kind `sh_binary rule` and the other kind `source file`.

## Print the label and kind of each target

```
--output label_kind
```

Like `label`, this output format prints the labels of each target in the resulting graph, in topological order, but it additionally precedes the label by the *kind* of the target.

## Print the label of each target, in rank order

```
--output minrank
--output maxrank
```

Like `label`, the `minrank` and `maxrank` output formats print the labels of each target in the resulting graph, but instead of appearing in topological order, they appear in rank order, preceded by their rank number. These are unaffected by the result ordering `--[no]order_results` flag (see notes on the ordering of results).

There are two variants of this format: `minrank` ranks each node by the length of the shortest path from a root node to it. "Root" nodes (those which have no incoming edges) are of rank 0, their successors are of rank 1, etc. (As always, edges point from a target to its prerequisites: the targets it depends upon.)

`maxrank` ranks each node by the length of the longest path from a root node to it. Again, "roots" have rank 0, all other nodes have a rank which is one greater than the maximum rank of all their predecessors.

All nodes in a cycle are considered of equal rank. (Most graphs are acyclic, but cycles do occur simply because BUILD files contain erroneous cycles.)

These output formats are useful for discovering how deep a graph is. If used for the result of a `deps(x)`, `rdeps(x)`, or `allpaths` query, then the rank number is equal to the length of the shortest (with `minrank`) or longest (with `maxrank`) path from `x` to a node in that rank. `maxrank` can be used to determine the longest sequence of build steps required to build a target.

Please note, the ranked output of a `somepath` query is basically meaningless because `somepath` doesn't guarantee to return either a shortest or a longest path, and it may include "transitive" edges from one path node to another that are not direct edges in original graph.

For example, the graph on the left yields the outputs on the right when `--output minrank` and `--output maxrank` are specified, respectively.

```
minrank       maxrank

0 //c:c       0 //c:c
1 //b:b       1 //b:b
1 //a:a       2 //a:a
2 //b:b.cc    2 //b:b.cc
2 //a:a.cc    3 //a:a.cc
```

## Print the location of each target

```
--output location
```

Like `label_kind`, this option prints out, for each target in the result, the target's kind and label, but it is prefixed by a string describing the location of that target, as a filename and line number. The format resembles the output of `grep`. Thus, tools that can parse the latter (such as Emacs or vi) can also use the query output to step through a series of matches, allowing the Bazel query tool to be used as a dependency-graph-aware "grep for BUILD files".

The location information varies by target kind (see the kind operator). For rules, the location of the rule's declaration within the BUILD file is printed. For source files, the location of line 1 of the actual file is printed. For a generated file, the location of the rule that generates it is printed. (The query tool does not have sufficient information to find the actual location of the generated file, and in any case, it might not exist if a build has not yet been performed.)

## Print the set of packages

```
--output package
```

This option prints the name of all packages to which some target in the result set belongs. The names are printed in lexicographical order; duplicates are excluded. Formally, this is a *projection* from the set of labels (package, target) onto packages.

Packages in external repositories are formatted as `@repo//foo/bar` while packages in the main repository are formatted as `foo/bar`.

In conjunction with the `deps(...)` query, this output option can be used to find the set of packages that must be checked out in order to build a given set of targets.

## Display a graph of the result

```
--output graph
```

This option causes the query result to be printed as a directed graph in the popular AT&T GraphViz format. Typically the result is saved to a file, such as `.png` or `.svg`. (If the `dot` program is not installed on your workstation, you can install it using the command `sudo apt-get install graphviz`.) See the example section below for a sample invocation.

This output format is particularly useful for `allpath`, `deps`, or `rdeps` queries, where the result includes a *set of paths* that cannot be easily visualized when rendered in a linear form, such as with `--output label`.

By default, the graph is rendered in a *factored* form. That is, topologically-equivalent nodes are merged together into a single node with multiple labels. This makes the graph more compact and readable, because typical result graphs contain highly repetitive patterns. For example, a `java_library` rule may depend on hundreds of Java source files all generated by the same `genrule`; in the factored graph, all these files are represented by a single node. This behavior may be disabled with the `--nograph:factored` option.

### `--graph:node_limit` *n*

The option specifies the maximum length of the label string for a graph node in the output. Longer labels will be truncated; -1 disables truncation. Due to the factored form in which graphs are usually printed, the node labels may be very long. GraphViz cannot handle labels exceeding 1024 characters, which is the default value of this option. This option has no effect unless `--output=graph` is being used.

### `--[no]graph:factored`

By default, graphs are displayed in factored form, as explained above. When `--nograph:factored` is specified, graphs are printed without factoring. This makes visualization using GraphViz impractical, but the simpler format may ease processing by other tools (e.g. grep). This option has no effect unless `--output=graph` is being used.

## XML

```
--output xml
```

This option causes the resulting targets to be printed in an XML form. The output starts with an XML header such as this

```
<?xml version="1.0" encoding="UTF-8"?>
<query version="2">
```

and then continues with an XML element for each target in the result graph, in topological order (unless unordered results are requested), and then finishes with a terminating

```
</query>
```

Simple entries are emitted for targets of `file` kind:

```
<source-file name='//foo:foo_main.cc' .../>
<generated-file name='//foo:libfoo.so' .../>
```

But for rules, the XML is structured and contains definitions of all the attributes of the rule, including those whose value was not explicitly specified in the rule's BUILD file.

Additionally, the result includes `rule-input` and `rule-output` elements so that the topology of the dependency graph can be reconstructed without having to know that, for example, the elements of the `srcs` attribute are forward dependencies (prerequisites) and the contents of the `outs` attribute are backward dependencies (consumers). `rule-input` elements for implicit dependencies are suppressed if `--noimplicit_deps` is specified.

```
<rule class='cc_binary rule' name='//foo:foo' ...>
  <list name='srcs'>
    <label value='//foo:foo_main.cc'/>
    <label value='//foo:bar.cc'/>
    ...
  </list>
  <list name='deps'>
    <label value='//common:common'/>
    <label value='//collections:collections'/>
    ...
  </list>
  <list name='data'>
    ...
  </list>
  <int name='linkstatic' value='0'/>
  <int name='linkshared' value='0'/>
  <list name='licenses'/>
  <list name='distribs'>
    <distribution value="INTERNAL" />
  </list>
  <rule-input name="//common:common" />
  <rule-input name="//collections:collections" />
  <rule-input name="//foo:foo_main.cc" />
  <rule-input name="//foo:bar.cc" />
  ...
</rule>
```

Every XML element for a target contains a `name` attribute, whose value is the target's label, and a `location` attribute, whose value is the target's location as printed by the `--output location` (output-location).

### `--[no]xml:line_numbers`

By default, the locations displayed in the XML output contain line numbers. When `--noxml:line_numbers` is specified, line numbers are not printed.

### `--[no]xml:default_values`

By default, XML output does not include rule attribute whose value is the default value for that kind of attribute (e.g. because it were not specified in the BUILD file, or the default value was provided explicitly). This option causes such attribute values to be included in the XML output.

## Querying with external repositories

If the build depends on rules from external repositories (defined in the WORKSPACE file) then query results will include these dependencies. For example, if `//foo:bar` depends on `//external:some-lib` and `//external:some-lib` is bound to `@other-repo//baz:lib`, then `bazel query 'deps(//foo:bar)'` will list both `@other-repo//baz:lib` and `//external:some-lib` as dependencies.

External repositories themselves are not dependencies of a build. That is, in the example above, `//external:other-repo` is not a dependency. It can be queried for as a member of the `//external` package, though, for example:

```
# Querying over all members of //external returns the repository.
bazel query 'kind(maven_jar, //external:*)'
//external:other-repo

# ...but the repository is not a dependency.
bazel query 'kind(maven_jar, deps(//foo:bar))'
INFO: Empty results
```

About

Who's using Bazel (https://github.com/bazelbuild/bazel/wiki/Bazel-Users)
Roadmap (https://www.bazel.build/roadmap.html)
Contribute (https://www.bazel.build/contributing.html)
Governance Plan (https://www.bazel.build/governance.html)

Support

Stack Overflow (http://stackoverflow.com/questions/tagged/bazel)
Issue Tracker (https://github.com/bazelbuild/bazel/issues)
Documentation (https://docs.bazel.build)
FAQ (https://www.bazel.build/faq.html)
Support Policy (https://www.bazel.build/support.html)

Stay Connected

Twitter (https://twitter.com/bazelbuild)
Blog (https://blog.bazel.build)
GitHub (https://github.com/bazelbuild/bazel)
Discussion group (https://groups.google.com/forum/#!forum/bazel-discuss)

© 2018 Google