



Full credit is given to the above companies including the Operating System (OS) that this PDF file was generated!

Rocky Enterprise Linux 9.2 Manual Pages on command 'git-rev-list.1'

\$ man git-rev-list.1

GIT-REV-LIST(1) Git Manual GIT-REV-LIST(1)

NAME

git-rev-list - Lists commit objects in reverse chronological order

SYNOPSIS

git rev-list [<options>] <commit>... [--] <path>...

DESCRIPTION

List commits that are reachable by following the parent links from the given commit(s), but exclude commits that are reachable from the one(s) given with a ^ in front of them.

The output is given in reverse chronological order by default.

You can think of this as a set operation. Commits reachable from any of the commits given on the command line form a set, and then commits reachable from any of the ones given with ^ in front are subtracted from that set. The remaining commits are what comes out in the command's output. Various other options and paths parameters can be used to further limit the result.

Thus, the following command:

```
$ git rev-list foo bar ^baz
```

means "list all the commits which are reachable from foo or bar, but not from baz".

A special notation "<commit1>..<commit2>" can be used as a short-hand for "^<commit1><commit2>". For example, either of the following may be used interchangeably:

```
$ git rev-list origin..HEAD
```

```
$ git rev-list HEAD ^origin
```

Another special notation is "<commit1>...<commit2>" which is useful for merges. The resulting set of commits is the symmetric difference between the two operands. The

following two commands are equivalent:

```
$ git rev-list A B --not $(git merge-base --all A B)
```

```
$ git rev-list A...B
```

rev-list is a very essential Git command, since it provides the ability to build and traverse commit ancestry graphs. For this reason, it has a lot of different options that enables it to be used by commands as different as git bisect and git repack.

OPTIONS

Commit Limiting

Besides specifying a range of commits that should be listed using the special notations explained in the description, additional commit limiting may be applied.

Using more options generally further limits the output (e.g. `--since=<date1>` limits to commits newer than `<date1>`, and using it with `--grep=<pattern>` further limits to commits whose log message has a line that matches `<pattern>`), unless otherwise noted.

Note that these are applied before commit ordering and formatting options, such as `--reverse`.

`-<number>`, `-n <number>`, `--max-count=<number>`

Limit the number of commits to output.

`--skip=<number>`

Skip number commits before starting to show the commit output.

`--since=<date>`, `--after=<date>`

Show commits more recent than a specific date.

`--until=<date>`, `--before=<date>`

Show commits older than a specific date.

`--max-age=<timestamp>`, `--min-age=<timestamp>`

Limit the commits output to specified time range.

`--author=<pattern>`, `--committer=<pattern>`

Limit the commits output to ones with author/committer header lines that match the specified pattern (regular expression). With more than one `--author=<pattern>`, commits whose author matches any of the given patterns are chosen (similarly for multiple `--committer=<pattern>`).

`--grep-reflog=<pattern>`

Limit the commits output to ones with reflog entries that match the specified pattern (regular expression). With more than one `--grep-reflog`, commits whose reflog message

matches any of the given patterns are chosen. It is an error to use this option unless `--walk-reflogs` is in use.

`--grep=<pattern>`

Limit the commits output to ones with log message that matches the specified pattern (regular expression). With more than one `--grep=<pattern>`, commits whose message matches any of the given patterns are chosen (but see `--all-match`).

`--all-match`

Limit the commits output to ones that match all given `--grep`, instead of ones that match at least one.

`--invert-grep`

Limit the commits output to ones with log message that do not match the pattern specified with `--grep=<pattern>`.

`-i, --regexp-ignore-case`

Match the regular expression limiting patterns without regard to letter case.

`--basic-regexp`

Consider the limiting patterns to be basic regular expressions; this is the default.

`-E, --extended-regexp`

Consider the limiting patterns to be extended regular expressions instead of the default basic regular expressions.

`-F, --fixed-strings`

Consider the limiting patterns to be fixed strings (don't interpret pattern as a regular expression).

`-P, --perl-regexp`

Consider the limiting patterns to be Perl-compatible regular expressions.

Support for these types of regular expressions is an optional compile-time dependency.

If Git wasn't compiled with support for them providing this option will cause it to die.

`--remove-empty`

Stop when a given path disappears from the tree.

`--merges`

Print only merge commits. This is exactly the same as `--min-parents=2`.

`--no-merges`

Do not print commits with more than one parent. This is exactly the same as

--max-parents=1.

--min-parents=<number>, --max-parents=<number>, --no-min-parents, --no-max-parents

Show only commits which have at least (or at most) that many parent commits. In particular, --max-parents=1 is the same as --no-merges, --min-parents=2 is the same as --merges. --max-parents=0 gives all root commits and --min-parents=3 all octopus merges.

--no-min-parents and --no-max-parents reset these limits (to no limit) again.

Equivalent forms are --min-parents=0 (any commit has 0 or more parents) and

--max-parents=-1 (negative numbers denote no upper limit).

--first-parent

Follow only the first parent commit upon seeing a merge commit. This option can give a better overview when viewing the evolution of a particular topic branch, because merges into a topic branch tend to be only about adjusting to updated upstream from time to time, and this option allows you to ignore the individual commits brought in to your history by such a merge.

--not

Reverses the meaning of the ^ prefix (or lack thereof) for all following revision specifiers, up to the next --not.

--all

Pretend as if all the refs in refs/, along with HEAD, are listed on the command line as <commit>.

--branches[=<pattern>]

Pretend as if all the refs in refs/heads are listed on the command line as <commit>.

If <pattern> is given, limit branches to ones matching given shell glob. If pattern lacks ?, *, or [, /* at the end is implied.

--tags[=<pattern>]

Pretend as if all the refs in refs/tags are listed on the command line as <commit>. If

<pattern> is given, limit tags to ones matching given shell glob. If pattern lacks ?, *, or [, /* at the end is implied.

--remotes[=<pattern>]

Pretend as if all the refs in refs/remotes are listed on the command line as <commit>.

If <pattern> is given, limit remote-tracking branches to ones matching given shell glob. If pattern lacks ?, *, or [, /* at the end is implied.

`--glob=<glob-pattern>`

Pretend as if all the refs matching shell glob `<glob-pattern>` are listed on the command line as `<commit>`. Leading `refs/` is automatically prepended if missing. If pattern lacks `?`, `*`, or `[, /*` at the end is implied.

`--exclude=<glob-pattern>`

Do not include refs matching `<glob-pattern>` that the next `--all`, `--branches`, `--tags`, `--remotes`, or `--glob` would otherwise consider. Repetitions of this option accumulate exclusion patterns up to the next `--all`, `--branches`, `--tags`, `--remotes`, or `--glob` option (other options or arguments do not clear accumulated patterns).

The patterns given should not begin with `refs/heads`, `refs/tags`, or `refs/remotes` when applied to `--branches`, `--tags`, or `--remotes`, respectively, and they must begin with `refs/` when applied to `--glob` or `--all`. If a trailing `/*` is intended, it must be given explicitly.

`--reflog`

Pretend as if all objects mentioned by reflogs are listed on the command line as `<commit>`.

`--alternate-refs`

Pretend as if all objects mentioned as ref tips of alternate repositories were listed on the command line. An alternate repository is any repository whose object directory is specified in `objects/info/alternates`. The set of included objects may be modified by `core.alternateRefsCommand`, etc. See `git-config(1)`.

`--single-worktree`

By default, all working trees will be examined by the following options when there are more than one (see `git-worktree(1)`): `--all`, `--reflog` and `--indexed-objects`. This option forces them to examine the current working tree only.

`--ignore-missing`

Upon seeing an invalid object name in the input, pretend as if the bad input was not given.

`--stdin`

In addition to the `<commit>` listed on the command line, read them from the standard input. If a `--` separator is seen, stop reading commits and start reading paths to limit the result.

`--quiet`

Don't print anything to standard output. This form is primarily meant to allow the caller to test the exit status to see if a range of objects is fully connected (or not). It is faster than redirecting stdout to /dev/null as the output does not have to be formatted.

--disk-usage

Suppress normal output; instead, print the sum of the bytes used for on-disk storage by the selected commits or objects. This is equivalent to piping the output into `git cat-file --batch-check='% (objectsize:disk)'`, except that it runs much faster (especially with `--use-bitmap-index`). See the CAVEATS section in `git-cat-file(1)` for the limitations of what "on-disk storage" means.

--cherry-mark

Like `--cherry-pick` (see below) but mark equivalent commits with `=` rather than omitting them, and inequivalent ones with `+`.

--cherry-pick

Omit any commit that introduces the same change as another commit on the other side when the set of commits are limited with symmetric difference.

For example, if you have two branches, A and B, a usual way to list all commits on only one side of them is with `--left-right` (see the example below in the description of the `--left-right` option). However, it shows the commits that were cherry-picked from the other branch (for example, `?3rd on b?` may be cherry-picked from branch A).

With this option, such pairs of commits are excluded from the output.

--left-only, --right-only

List only commits on the respective side of a symmetric difference, i.e. only those which would be marked `<` resp. `>` by `--left-right`.

For example, `--cherry-pick --right-only A...B` omits those commits from B which are in A or are patch-equivalent to a commit in A. In other words, this lists the `+` commits from `git cherry A B`. More precisely, `--cherry-pick --right-only --no-merges` gives the exact list.

--cherry

A synonym for `--right-only --cherry-mark --no-merges`; useful to limit the output to the commits on our side and mark those that have been applied to the other side of a forked history with `git log --cherry upstream...mybranch`, similar to `git cherry upstream mybranch`.

-g, --walk-reflogs

Instead of walking the commit ancestry chain, walk reflog entries from the most recent one to older ones. When this option is used you cannot specify commits to exclude (that is, ^commit, commit1..commit2, and commit1...commit2 notations cannot be used).

With --pretty format other than oneline and reference (for obvious reasons), this causes the output to have two extra lines of information taken from the reflog. The reflog designator in the output may be shown as ref@{Nth} (where Nth is the reverse-chronological index in the reflog) or as ref@{timestamp} (with the timestamp for that entry), depending on a few rules:

1. If the starting point is specified as ref@{Nth}, show the index format.
2. If the starting point was specified as ref@{now}, show the timestamp format.
3. If neither was used, but --date was given on the command line, show the timestamp in the format requested by --date.
4. Otherwise, show the index format.

Under --pretty=oneline, the commit message is prefixed with this information on the same line. This option cannot be combined with --reverse. See also git-reflog(1).

Under --pretty=reference, this information will not be shown at all.

--merge

After a failed merge, show refs that touch files having a conflict and don't exist on all heads to merge.

--boundary

Output excluded boundary commits. Boundary commits are prefixed with -.

--use-bitmap-index

Try to speed up the traversal using the pack bitmap index (if one is available). Note that when traversing with --objects, trees and blobs will not have their associated path printed.

--progress=<header>

Show progress reports on stderr as objects are considered. The <header> text will be printed with each progress update.

History Simplification

Sometimes you are only interested in parts of the history, for example the commits modifying a particular <path>. But there are two parts of History Simplification, one part is selecting the commits and the other is how to do it, as there are various strategies to

simplify the history.

The following options select the commits to be shown:

<paths>

Commits modifying the given <paths> are selected.

--simplify-by-decoration

Commits that are referred by some branch or tag are selected.

Note that extra commits can be shown to give a meaningful history.

The following options affect the way the simplification is performed:

Default mode

Simplifies the history to the simplest history explaining the final state of the tree.

Simplest because it prunes some side branches if the end result is the same (i.e. merging branches with the same content)

--show-pulls

Include all commits from the default mode, but also any merge commits that are not TREESAME to the first parent but are TREESAME to a later parent. This mode is helpful for showing the merge commits that "first introduced" a change to a branch.

--full-history

Same as the default mode, but does not prune some history.

--dense

Only the selected commits are shown, plus some to have a meaningful history.

--sparse

All commits in the simplified history are shown.

--simplify-merges

Additional option to --full-history to remove some needless merges from the resulting history, as there are no selected commits contributing to this merge.

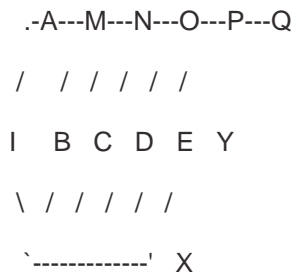
--ancestry-path

When given a range of commits to display (e.g. commit1..commit2 or commit2 ^commit1), only display commits that exist directly on the ancestry chain between the commit1 and commit2, i.e. commits that are both descendants of commit1, and ancestors of commit2.

A more detailed explanation follows.

Suppose you specified foo as the <paths>. We shall call commits that modify foo !TREESAME, and the rest TREESAME. (In a diff filtered for foo, they look different and equal, respectively.)

In the following, we will always refer to the same example history to illustrate the differences between simplification settings. We assume that you are filtering for a file foo in this commit graph:



The horizontal line of history A---Q is taken to be the first parent of each merge. The commits are:

- ? I is the initial commit, in which foo exists with contents ?asdf?, and a file quux exists with contents ?quux?. Initial commits are compared to an empty tree, so I is !TREESAME.
- ? In A, foo contains just ?foo?.
- ? B contains the same change as A. Its merge M is trivial and hence TREESAME to all parents.
- ? C does not change foo, but its merge N changes it to ?foobar?, so it is not TREESAME to any parent.
- ? D sets foo to ?baz?. Its merge O combines the strings from N and D to ?foobarbaz?; i.e., it is not TREESAME to any parent.
- ? E changes quux to ?xyzyz?, and its merge P combines the strings to ?quux xyzyz?. P is TREESAME to O, but not to E.
- ? X is an independent root commit that added a new file side, and Y modified it. Y is TREESAME to X. Its merge Q added side to P, and Q is TREESAME to P, but not to Y.

rev-list walks backwards through history, including or excluding commits based on whether --full-history and/or parent rewriting (via --parents or --children) are used. The

following settings are available.

Default mode

Commits are included if they are not TREESAME to any parent (though this can be changed, see --sparse below). If the commit was a merge, and it was TREESAME to one parent, follow only that parent. (Even if there are several TREESAME parents, follow only one of them.) Otherwise, follow all parents.

This results in:

```

.-A---N---O
 /   /   /
I-----D

```

Note how the rule to only follow the TREESAME parent, if one is available, removed B from consideration entirely. C was considered via N, but is TREESAME. Root commits are compared to an empty tree, so I is !TREESAME.

Parent/child relations are only visible with --parents, but that does not affect the commits selected in default mode, so we have shown the parent lines.

--full-history without parent rewriting

This mode differs from the default in one point: always follow all parents of a merge, even if it is TREESAME to one of them. Even if more than one side of the merge has commits that are included, this does not imply that the merge itself is! In the example, we get

```

I A B N D O P Q

```

M was excluded because it is TREESAME to both parents. E, C and B were all walked, but only B was !TREESAME, so the others do not appear.

Note that without parent rewriting, it is not really possible to talk about the parent/child relationships between the commits, so we show them disconnected.

--full-history with parent rewriting

Ordinary commits are only included if they are !TREESAME (though this can be changed, see --sparse below).

Merges are always included. However, their parent list is rewritten: Along each parent, prune away commits that are not included themselves. This results in

```

.-A---M---N---O---P---Q
 /   /   /   /   /
I   B / D /
 \   /   /   /
 `-----'

```

Compare to --full-history without rewriting above. Note that E was pruned away because it is TREESAME, but the parent list of P was rewritten to contain E's parent I. The same happened for C and N, and X, Y and Q.

In addition to the above settings, you can change whether TREESAME affects inclusion:

--dense

Commits that are walked are included if they are not TREESAME to any parent.

--sparse

All commits that are walked are included.

Note that without --full-history, this still simplifies merges: if one of the parents is TREESAME, we follow only that one, so the other sides of the merge are never walked.

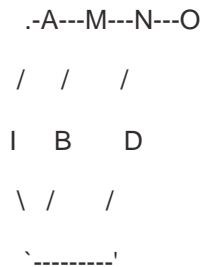
--simplify-merges

First, build a history graph in the same way that --full-history with parent rewriting does (see above).

Then simplify each commit C to its replacement C' in the final history according to the following rules:

- ? Set C' to C.
- ? Replace each parent P of C' with its simplification P'. In the process, drop parents that are ancestors of other parents or that are root commits TREESAME to an empty tree, and remove duplicates, but take care to never drop all parents that we are TREESAME to.
- ? If after this parent rewriting, C' is a root or merge commit (has zero or >1 parents), a boundary commit, or !TREESAME, it remains. Otherwise, it is replaced with its only parent.

The effect of this is best shown by way of comparing to --full-history with parent rewriting. The example turns into:



Note the major differences in N, P, and Q over --full-history:

- ? N's parent list had I removed, because it is an ancestor of the other parent M. Still, N remained because it is !TREESAME.
- ? P's parent list similarly had I removed. P was then removed completely, because it had one parent and is TREESAME.
- ? Q's parent list had Y simplified to X. X was then removed, because it was a TREESAME root. Q was then removed completely, because it had one parent and is

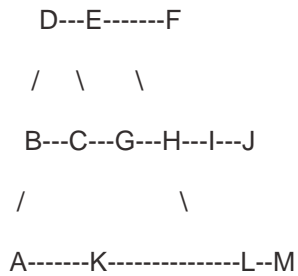
TREESAME.

There is another simplification mode available:

--ancestry-path

Limit the displayed commits to those directly on the ancestry chain between the ?from? and ?to? commits in the given commit range. I.e. only display commits that are ancestor of the ?to? commit and descendants of the ?from? commit.

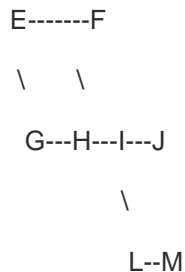
As an example use case, consider the following commit history:



A regular `D..M` computes the set of commits that are ancestors of M, but excludes the ones that are ancestors of D. This is useful to see what happened to the history leading to M since D, in the sense that ?what does M have that did not exist in D?.

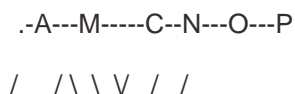
The result in this example would be all the commits, except A and B (and D itself, of course).

When we want to find out what commits in M are contaminated with the bug introduced by D and need fixing, however, we might want to view only the subset of `D..M` that are actually descendants of D, i.e. excluding C and K. This is exactly what the `--ancestry-path` option does. Applied to the `D..M` range, it results in:



Before discussing another option, `--show-pulls`, we need to create a new example history.

A common problem users face when looking at simplified history is that a commit they know changed a file somehow does not appear in the file?s simplified history. Let?s demonstrate a new example and show how options such as `--full-history` and `--simplify-merges` works in that case:



```

I  B \ R-'-Z' /
\ /  V    /
\ /  ^    /
`---X--' `---Y--'

```

For this example, suppose I created file.txt which was modified by A, B, and X in different ways. The single-parent commits C, Z, and Y do not change file.txt. The merge commit M was created by resolving the merge conflict to include both changes from A and B and hence is not TREESAME to either. The merge commit R, however, was created by ignoring the contents of file.txt at M and taking only the contents of file.txt at X. Hence, R is TREESAME to X but not M. Finally, the natural merge resolution to create N is to take the contents of file.txt at R, so N is TREESAME to R but not C. The merge commits O and P are TREESAME to their first parents, but not to their second parents, Z and Y respectively. When using the default mode, N and R both have a TREESAME parent, so those edges are walked and the others are ignored. The resulting history graph is:

```
I---X
```

When using --full-history, Git walks every edge. This will discover the commits A and B and the merge M, but also will reveal the merge commits O and P. With parent rewriting, the resulting graph is:

```

.-A---M-----N---O---P
/  /\ \ V / /
I  B \ R-'--' /
\ /  V    /
\ /  ^    /
`---X--' `-----'

```

Here, the merge commits O and P contribute extra noise, as they did not actually contribute a change to file.txt. They only merged a topic that was based on an older version of file.txt. This is a common issue in repositories using a workflow where many contributors work in parallel and merge their topic branches along a single trunk: many unrelated merges appear in the --full-history results.

When using the --simplify-merges option, the commits O and P disappear from the results. This is because the rewritten second parents of O and P are reachable from their first parents. Those edges are removed and then the commits look like single-parent commits that are TREESAME to their parent. This also happens to the commit N, resulting in a history

view as follows:

```
  .-A---M--.
 /   /   \
I   B   R
 \   /   /
  \ /   /
   `---X--'
```

In this view, we see all of the important single-parent changes from A, B, and X. We also see the carefully-resolved merge M and the not-so-carefully-resolved merge R. This is usually enough information to determine why the commits A and B "disappeared" from history in the default view. However, there are a few issues with this approach.

The first issue is performance. Unlike any previous option, the `--simplify-merges` option requires walking the entire commit history before returning a single result. This can make the option difficult to use for very large repositories.

The second issue is one of auditing. When many contributors are working on the same repository, it is important which merge commits introduced a change into an important branch. The problematic merge R above is not likely to be the merge commit that was used to merge into an important branch. Instead, the merge N was used to merge R and X into the important branch. This commit may have information about why the change X came to override the changes from A and B in its commit message.

`--show-pulls`

In addition to the commits shown in the default history, show each merge commit that is not TREESAME to its first parent but is TREESAME to a later parent.

When a merge commit is included by `--show-pulls`, the merge is treated as if it "pulled" the change from another branch. When using `--show-pulls` on this example (and no other options) the resulting graph is:

```
I---X---R---N
```

Here, the merge commits R and N are included because they pulled the commits X and R into the base branch, respectively. These merges are the reason the commits A and B do not appear in the default history.

When `--show-pulls` is paired with `--simplify-merges`, the graph includes all of the necessary information:

```
.-A---M--. N
```

```

    /  /  \ /
    I  B  R
    \  /  /
    \ /  /
    `---X--'

```

Notice that since M is reachable from R, the edge from N to M was simplified away.

However, N still appears in the history as an important commit because it "pulled" the change R into the main branch.

The `--simplify-by-decoration` option allows you to view only the big picture of the topology of the history, by omitting commits that are not referenced by tags. Commits are marked as `!TREESAME` (in other words, kept after history simplification rules described above) if (1) they are referenced by tags, or (2) they change the contents of the paths given on the command line. All other commits are marked as `TREESAME` (subject to be simplified away).

Bisection Helpers

`--bisect`

Limit output to the one commit object which is roughly halfway between included and excluded commits. Note that the bad bisection ref `refs/bisect/bad` is added to the included commits (if it exists) and the good bisection refs `refs/bisect/good-*` are added to the excluded commits (if they exist). Thus, supposing there are no refs in `refs/bisect/`, if

```
$ git rev-list --bisect foo ^bar ^baz
```

outputs `midpoint`, the output of the two commands

```
$ git rev-list foo ^midpoint
```

```
$ git rev-list midpoint ^bar ^baz
```

would be of roughly the same length. Finding the change which introduces a regression is thus reduced to a binary search: repeatedly generate and test new `'midpoint?'`s until the commit chain is of length one.

`--bisect-vars`

This calculates the same as `--bisect`, except that refs in `refs/bisect/` are not used, and except that this outputs text ready to be evaluated by the shell. These lines will assign the name of the midpoint revision to the variable `bisect_rev`, and the expected number of commits to be tested after `bisect_rev` is tested to `bisect_nr`, the expected

number of commits to be tested if bisect_rev turns out to be good to bisect_good, the expected number of commits to be tested if bisect_rev turns out to be bad to bisect_bad, and the number of commits we are bisecting right now to bisect_all.

--bisect-all

This outputs all the commit objects between the included and excluded commits, ordered by their distance to the included and excluded commits. Refs in refs/bisect/ are not used. The farthest from them is displayed first. (This is the only one displayed by --bisect.)

This is useful because it makes it easy to choose a good commit to test when you want to avoid to test some of them for some reason (they may not compile for example).

This option can be used along with --bisect-vars, in this case, after all the sorted commit objects, there will be the same text as if --bisect-vars had been used alone.

Commit Ordering

By default, the commits are shown in reverse chronological order.

--date-order

Show no parents before all of its children are shown, but otherwise show commits in the commit timestamp order.

--author-date-order

Show no parents before all of its children are shown, but otherwise show commits in the author timestamp order.

--topo-order

Show no parents before all of its children are shown, and avoid showing commits on multiple lines of history intermixed.

For example, in a commit history like this:

```
---1---2---4---7
 \       \
  3---5---6---8---
```

where the numbers denote the order of commit timestamps, git rev-list and friends with --date-order show the commits in the timestamp order: 8 7 6 5 4 3 2 1.

With --topo-order, they would show 8 6 5 3 7 4 2 1 (or 8 7 4 2 6 5 3 1); some older commits are shown before newer ones in order to avoid showing the commits from two parallel development track mixed together.

--reverse

Output the commits chosen to be shown (see Commit Limiting section above) in reverse order. Cannot be combined with --walk-reflogs.

Object Traversal

These options are mostly targeted for packing of Git repositories.

--objects

Print the object IDs of any object referenced by the listed commits. --objects foo ^bar thus means ?send me all object IDs which I need to download if I have the commit object bar but not foo?.

--in-commit-order

Print tree and blob ids in order of the commits. The tree and blob ids are printed after they are first referenced by a commit.

--objects-edge

Similar to --objects, but also print the IDs of excluded commits prefixed with a ?-? character. This is used by git-pack-objects(1) to build a ?thin? pack, which records objects in deltified form based on objects contained in these excluded commits to reduce network traffic.

--objects-edge-aggressive

Similar to --objects-edge, but it tries harder to find excluded commits at the cost of increased time. This is used instead of --objects-edge to build ?thin? packs for shallow repositories.

--indexed-objects

Pretend as if all trees and blobs used by the index are listed on the command line. Note that you probably want to use --objects, too.

--unpacked

Only useful with --objects; print the object IDs that are not in packs.

--object-names

Only useful with --objects; print the names of the object IDs that are found. This is the default behavior.

--no-object-names

Only useful with --objects; does not print the names of the object IDs that are found. This inverts --object-names. This flag allows the output to be more easily parsed by commands such as git-cat-file(1).

--filter=<filter-spec>

Only useful with one of the `--objects*`; omits objects (usually blobs) from the list of printed objects. The `<filter-spec>` may be one of the following:

The form `--filter=blob:none` omits all blobs.

The form `--filter=blob:limit=<n>[kmg]` omits blobs larger than `n` bytes or units. `n` may be zero. The suffixes `k`, `m`, and `g` can be used to name units in KiB, MiB, or GiB. For example, `blob:limit=1k` is the same as `blob:limit=1024`.

The form `--filter=object:type=(tag|commit|tree|blob)` omits all objects which are not of the requested type.

The form `--filter=sparse:oid=<blob-ish>` uses a sparse-checkout specification contained in the blob (or blob-expression) `<blob-ish>` to omit blobs that would not be required for a sparse checkout on the requested refs.

The form `--filter=tree:<depth>` omits all blobs and trees whose depth from the root tree is `>= <depth>` (minimum depth if an object is located at multiple depths in the commits traversed). `<depth>=0` will not include any trees or blobs unless included explicitly in the command-line (or standard input when `--stdin` is used). `<depth>=1` will include only the tree and blobs which are referenced directly by a commit reachable from `<commit>` or an explicitly-given object. `<depth>=2` is like `<depth>=1` while also including trees and blobs one more level removed from an explicitly-given commit or tree.

Note that the form `--filter=sparse:path=<path>` that wants to read from an arbitrary path on the filesystem has been dropped for security reasons.

Multiple `--filter=` flags can be specified to combine filters. Only objects which are accepted by every filter are included.

The form `--filter=combine:<filter1>+<filter2>+...<filterN>` can also be used to combined several filters, but this is harder than just repeating the `--filter` flag and is usually not necessary. Filters are joined by `+` and individual filters are %-encoded (i.e. URL-encoded). Besides the `+` and `%` characters, the following characters are reserved and also must be encoded: `~!@#$$%^&*()[]{}|\";,<>?'`` as well as all characters with ASCII code `<= 0x20`, which includes space and newline.

Other arbitrary characters can also be encoded. For instance, `combine:tree:3+blob:none` and `combine:tree%3A3+blob%3Anone` are equivalent.

`--no-filter`

Turn off any previous `--filter=` argument.

`--filter-provided-objects`

Filter the list of explicitly provided objects, which would otherwise always be printed even if they did not match any of the filters. Only useful with `--filter=`.

`--filter-print-omitted`

Only useful with `--filter=`; prints a list of the objects omitted by the filter. Object IDs are prefixed with a `?~?` character.

`--missing=<missing-action>`

A debug option to help with future "partial clone" development. This option specifies how missing objects are handled.

The form `--missing=error` requests that `rev-list` stop with an error if a missing object is encountered. This is the default action.

The form `--missing=allow-any` will allow object traversal to continue if a missing object is encountered. Missing objects will silently be omitted from the results.

The form `--missing=allow-promisor` is like `allow-any`, but will only allow object traversal to continue for EXPECTED promisor missing objects. Unexpected missing objects will raise an error.

The form `--missing=print` is like `allow-any`, but will also print a list of the missing objects. Object IDs are prefixed with a `???` character.

`--exclude-promisor-objects`

(For internal use only.) Prefilter object traversal at promisor boundary. This is used with partial clone. This is stronger than `--missing=allow-promisor` because it limits the traversal, rather than just silencing errors about missing objects.

`--no-walk[=(sorted|unsorted)]`

Only show the given commits, but do not traverse their ancestors. This has no effect if a range is specified. If the argument `unsorted` is given, the commits are shown in the order they were given on the command line. Otherwise (if `sorted` or no argument was given), the commits are shown in reverse chronological order by commit time. Cannot be combined with `--graph`.

`--do-walk`

Overrides a previous `--no-walk`.

Commit Formatting

Using these options, `git-rev-list(1)` will act similar to the more specialized family of commit log tools: `git-log(1)`, `git-show(1)`, and `git-whatchanged(1)`

`--pretty[=<format>], --format=<format>`

Pretty-print the contents of the commit logs in a given format, where `<format>` can be one of `oneline`, `short`, `medium`, `full`, `fuller`, `reference`, `email`, `raw`, `format:<string>` and `tformat:<string>`. When `<format>` is none of the above, and has `%placeholder` in it, it acts as if `--pretty=tformat:<format>` were given.

See the "PRETTY FORMATS" section for some additional details for each format. When `=<format>` part is omitted, it defaults to `medium`.

Note: you can specify the default pretty format in the repository configuration (see `git-config(1)`).

`--abbrev-commit`

Instead of showing the full 40-byte hexadecimal commit object name, show a prefix that names the object uniquely. "`--abbrev=<n>`" (which also modifies diff output, if it is displayed) option can be used to specify the minimum length of the prefix.

This should make "`--pretty=oneline`" a whole lot more readable for people using 80-column terminals.

`--no-abbrev-commit`

Show the full 40-byte hexadecimal commit object name. This negates `--abbrev-commit`, either explicit or implied by other options such as "`--oneline`". It also overrides the `log.abbrevCommit` variable.

`--oneline`

This is a shorthand for "`--pretty=oneline --abbrev-commit`" used together.

`--encoding=<encoding>`

Commit objects record the character encoding used for the log message in their encoding header; this option can be used to tell the command to re-code the commit log message in the encoding preferred by the user. For non plumbing commands this defaults to UTF-8. Note that if an object claims to be encoded in X and we are outputting in X, we will output the object verbatim; this means that invalid sequences in the original commit may be copied to the output. Likewise, if `iconv(3)` fails to convert the commit, we will quietly output the original object verbatim.

`--expand-tabs=<n>, --expand-tabs, --no-expand-tabs`

Perform a tab expansion (replace each tab with enough spaces to fill to the next display column that is multiple of `<n>`) in the log message before showing it in the output. `--expand-tabs` is a short-hand for `--expand-tabs=8`, and `--no-expand-tabs` is a

short-hand for `--expand-tabs=0`, which disables tab expansion.

By default, tabs are expanded in pretty formats that indent the log message by 4 spaces (i.e. `medium`, which is the default, `full`, and `fuller`).

`--show-signature`

Check the validity of a signed commit object by passing the signature to `gpg --verify` and show the output.

`--relative-date`

Synonym for `--date=relative`.

`--date=<format>`

Only takes effect for dates shown in human-readable format, such as when using `--pretty`. `log.date` config variable sets a default value for the `log` command's `--date` option. By default, dates are shown in the original time zone (either committer's or author's). If `-local` is appended to the format (e.g., `iso-local`), the user's local time zone is used instead.

`--date=relative` shows dates relative to the current time, e.g. `?2 hours ago?`. The

`-local` option has no effect for `--date=relative`.

`--date=local` is an alias for `--date=default-local`.

`--date=iso` (or `--date=iso8601`) shows timestamps in a ISO 8601-like format. The differences to the strict ISO 8601 format are:

? a space instead of the T date/time delimiter

? a space between time and time zone

? no colon between hours and minutes of the time zone

`--date=iso-strict` (or `--date=iso8601-strict`) shows timestamps in strict ISO 8601 format.

`--date=rfc` (or `--date=rfc2822`) shows timestamps in RFC 2822 format, often found in email messages.

`--date=short` shows only the date, but not the time, in `YYYY-MM-DD` format.

`--date=raw` shows the date as seconds since the epoch (1970-01-01 00:00:00 UTC), followed by a space, and then the timezone as an offset from UTC (a + or - with four digits; the first two are hours, and the second two are minutes). I.e., as if the timestamp were formatted with `strftime("%s %z")`. Note that the `-local` option does not affect the seconds-since-epoch value (which is always measured in UTC), but does switch the accompanying timezone value.

--date=human shows the timezone if the timezone does not match the current time-zone, and doesn't print the whole date if that matches (ie skip printing year for dates that are "this year", but also skip the whole date itself if it's in the last few days and we can just say what weekday it was). For older dates the hour and minute is also omitted.

--date=unix shows the date as a Unix epoch timestamp (seconds since 1970). As with --raw, this is always in UTC and therefore -local has no effect.

--date=format:... feeds the format ... to your system strftime, except for %z and %Z, which are handled internally. Use --date=format:%c to show the date in your system locale's preferred format. See the strftime manual for a complete list of format placeholders. When using -local, the correct syntax is --date=format-local:....

--date=default is the default format, and is similar to --date=rfc2822, with a few exceptions:

- ? there is no comma after the day-of-week
- ? the time zone is omitted when the local time zone is used

--header

Print the contents of the commit in raw-format; each record is separated with a NUL character.

--no-commit-header

Suppress the header line containing "commit" and the object ID printed before the specified format. This has no effect on the built-in formats; only custom formats are affected.

--commit-header

Overrides a previous --no-commit-header.

--parents

Print also the parents of the commit (in the form "commit parent.."). Also enables parent rewriting, see History Simplification above.

--children

Print also the children of the commit (in the form "commit child.."). Also enables parent rewriting, see History Simplification above.

--timestamp

Print the raw commit timestamp.

--left-right

Mark which side of a symmetric difference a commit is reachable from. Commits from the left side are prefixed with < and those from the right with >. If combined with --boundary, those commits are prefixed with -.

For example, if you have this topology:

```
    y---b---b branch B
   / \
  /   .
 /   \ \
o---x---a---a branch A
```

you would get an output like this:

```
$ git rev-list --left-right --boundary --pretty=oneline A...B
>bbbbbbb... 3rd on b
>bbbbbbb... 2nd on b
<aaaaaaa... 3rd on a
<aaaaaaa... 2nd on a
-yyyyyyy... 1st on b
-xxxxxxx... 1st on a
```

--graph

Draw a text-based graphical representation of the commit history on the left hand side of the output. This may cause extra lines to be printed in between commits, in order for the graph history to be drawn properly. Cannot be combined with --no-walk.

This enables parent rewriting, see History Simplification above.

This implies the --topo-order option by default, but the --date-order option may also be specified.

--show-linear-break[=<barrier>]

When --graph is not used, all history branches are flattened which can make it hard to see that the two consecutive commits do not belong to a linear branch. This option puts a barrier in between them in that case. If <barrier> is specified, it is the string that will be shown instead of the default one.

--count

Print a number stating how many commits would have been listed, and suppress all other output. When used together with --left-right, instead print the counts for left and right commits, separated by a tab. When used together with --cherry-mark, omit patch

equivalent commits from these counts and print the count for equivalent commits separated by a tab.

PRETTY FORMATS

If the commit is a merge, and if the pretty-format is not oneline, email or raw, an additional line is inserted before the Author: line. This line begins with "Merge: " and the hashes of ancestral commits are printed, separated by spaces. Note that the listed commits may not necessarily be the list of the direct parent commits if you have limited your view of history: for example, if you are only interested in changes related to a certain directory or file.

There are several built-in formats, and you can define additional formats by setting a pretty.<name> config option to either another format name, or a format: string, as described below (see git-config(1)). Here are the details of the built-in formats:

? oneline

<hash> <title line>

This is designed to be as compact as possible.

? short

commit <hash>

Author: <author>

<title line>

? medium

commit <hash>

Author: <author>

Date: <author date>

<title line>

<full commit message>

? full

commit <hash>

Author: <author>

Commit: <committer>

<title line>

<full commit message>

? fuller

commit <hash>

Author: <author>

AuthorDate: <author date>

Commit: <committer>

CommitDate: <committer date>

<title line>

<full commit message>

? reference

<abbrev hash> (<title line>, <short author date>)

This format is used to refer to another commit in a commit message and is the same as `--pretty='format:%C(auto)%h (%s, %ad)'`. By default, the date is formatted with `--date=short` unless another `--date` option is explicitly specified. As with any format: with format placeholders, its output is not affected by other options like `--decorate` and `--walk-reflogs`.

? email

From <hash> <date>

From: <author>

Date: <author date>

Subject: [PATCH] <title line>

<full commit message>

? mboxrd

Like email, but lines in the commit message starting with "From " (preceded by zero or more ">") are quoted with ">" so they aren't confused as starting a new commit.

? raw

The raw format shows the entire commit exactly as stored in the commit object.

Notably, the hashes are displayed in full, regardless of whether `--abbrev` or `--no-abbrev` are used, and parents information show the true parent commits, without taking grafts or history simplification into account. Note that this format affects the way commits are displayed, but not the way the diff is shown e.g. with `git log --raw`. To get full object names in a raw diff format, use `--no-abbrev`.

? format:<string>

The `format:<string>` format allows you to specify which information you want to show.

It works a little bit like `printf` format, with the notable exception that you get a newline with `%n` instead of `\n`.

E.g, format:"The author of %h was %an, %ar%nThe title was >>%s<<%n" would show something like this:

The author of fe6e0ee was Junio C Hamano, 23 hours ago

The title was >>t4119: test autocomputing -p<n> for traditional diff input.<<

The placeholders are:

? Placeholders that expand to a single literal character:

%n

newline

%%

a raw %

%x00

print a byte from a hex code

? Placeholders that affect formatting of later placeholders:

%Cred

switch color to red

%Cgreen

switch color to green

%Cblue

switch color to blue

%Creset

reset color

%C(...)

color specification, as described under Values in the "CONFIGURATION FILE" section of git-config(1). By default, colors are shown only when enabled for log output (by color.diff, color.ui, or --color, and respecting the auto settings of the former if we are going to a terminal). %C(auto,...) is accepted as a historical synonym for the default (e.g., %C(auto,red)). Specifying %C(always,...) will show the colors even when color is not otherwise enabled (though consider just using --color=always to enable color for the whole output, including this format and anything else git might color). auto alone (i.e. %C(auto)) will turn on auto coloring on the next placeholders until the color is switched again.

%m

left (<), right (>) or boundary (-) mark

`%w([<w>[,<i1>[,<i2>]])`

switch line wrapping, like the `-w` option of `git-shortlog(1)`.

`%<(<N>[,trunc|ltrunc|ltrunc])`

make the next placeholder take at least N columns, padding spaces on the right if necessary. Optionally truncate at the beginning (`ltrunc`), the middle (`mtrunc`) or the end (`trunc`) if the output is longer than N columns. Note that truncating only works correctly with `N >= 2`.

`%<|(<N>)`

make the next placeholder take at least until Nth columns, padding spaces on the right if necessary

`%>(<N>), %>|(<N>)`

similar to `%<(<N>)`, `%<|(<N>)` respectively, but padding spaces on the left

`%>>(<N>), %>>|(<N>)`

similar to `%>(<N>)`, `%>|(<N>)` respectively, except that if the next placeholder takes more spaces than given and there are spaces on its left, use those spaces

`%><(<N>), %><|(<N>)`

similar to `%<(<N>)`, `%<|(<N>)` respectively, but padding both sides (i.e. the text is centered)

? Placeholders that expand to information extracted from the commit:

`%H`

commit hash

`%h`

abbreviated commit hash

`%T`

tree hash

`%t`

abbreviated tree hash

`%P`

parent hashes

`%p`

abbreviated parent hashes

%an

author name

%aN

author name (respecting .mailmap, see git-shortlog(1) or git-blame(1))

%ae

author email

%aE

author email (respecting .mailmap, see git-shortlog(1) or git-blame(1))

%al

author email local-part (the part before the @ sign)

%aL

author local-part (see %al) respecting .mailmap, see git-shortlog(1) or git-blame(1))

%ad

author date (format respects --date= option)

%aD

author date, RFC2822 style

%ar

author date, relative

%at

author date, UNIX timestamp

%ai

author date, ISO 8601-like format

%al

author date, strict ISO 8601 format

%as

author date, short format (YYYY-MM-DD)

%ah

author date, human style (like the --date=human option of git-rev-list(1))

%cn

committer name

%cN

committer name (respecting .mailmap, see git-shortlog(1) or git-blame(1))

`%ce`

committer email

`%cE`

committer email (respecting `.mailmap`, see `git-shortlog(1)` or `git-blame(1)`)

`%cl`

committer email local-part (the part before the `@` sign)

`%cL`

committer local-part (see `%cl`) respecting `.mailmap`, see `git-shortlog(1)` or `git-blame(1)`)

`%cd`

committer date (format respects `--date=` option)

`%cD`

committer date, RFC2822 style

`%cr`

committer date, relative

`%ct`

committer date, UNIX timestamp

`%ci`

committer date, ISO 8601-like format

`%cl`

committer date, strict ISO 8601 format

`%cs`

committer date, short format (YYYY-MM-DD)

`%ch`

committer date, human style (like the `--date=human` option of `git-rev-list(1)`)

`%d`

ref names, like the `--decorate` option of `git-log(1)`

`%D`

ref names without the "`"`, "`"`" wrapping.

`%(describe[:options])`

human-readable name, like `git-describe(1)`; empty string for undecidable commits. The describe string may be followed by a colon and zero or more

comma-separated options. Descriptions can be inconsistent when tags are added

or removed at the same time.

? match=<pattern>: Only consider tags matching the given glob(7) pattern, excluding the "refs/tags/" prefix.

? exclude=<pattern>: Do not consider tags matching the given glob(7) pattern, excluding the "refs/tags/" prefix.

%S

ref name given on the command line by which the commit was reached (like git log --source), only works with git log

%e

encoding

%s

subject

%f

sanitized subject line, suitable for a filename

%b

body

%B

raw body (unwrapped subject and body)

%GG

raw verification message from GPG for a signed commit

%G?

show "G" for a good (valid) signature, "B" for a bad signature, "U" for a good signature with unknown validity, "X" for a good signature that has expired, "Y" for a good signature made by an expired key, "R" for a good signature made by a revoked key, "E" if the signature cannot be checked (e.g. missing key) and "N" for no signature

%GS

show the name of the signer for a signed commit

%GK

show the key used to sign a signed commit

%GF

show the fingerprint of the key used to sign a signed commit

%GP

show the fingerprint of the primary key whose subkey was used to sign a signed commit

%GT

show the trust level for the key used to sign a signed commit

%gD

reflog selector, e.g., refs/stash@{1} or refs/stash@{2 minutes ago}; the format follows the rules described for the -g option. The portion before the @ is the refname as given on the command line (so git log -g refs/heads/master would yield refs/heads/master@{0}).

%gd

shortened reflog selector; same as %gD, but the refname portion is shortened for human readability (so refs/heads/master becomes just master).

%gn

reflog identity name

%gN

reflog identity name (respecting .mailmap, see git-shortlog(1) or git-blame(1))

%ge

reflog identity email

%gE

reflog identity email (respecting .mailmap, see git-shortlog(1) or git-blame(1))

%gs

reflog subject

%(trailers[:options])

display the trailers of the body as interpreted by git-interpret-trailers(1).

The trailers string may be followed by a colon and zero or more comma-separated options. If any option is provided multiple times the last occurrence wins.

The boolean options accept an optional value [=<BOOL>]. The values true, false, on, off etc. are all accepted. See the "boolean" sub-section in "EXAMPLES" in git-config(1). If a boolean option is given with no value, it's enabled.

- ? `key=<K>`: only show trailers with specified key. Matching is done case-insensitively and trailing colon is optional. If option is given multiple times trailer lines matching any of the keys are shown. This option automatically enables the `only` option so that non-trailer lines in the trailer block are hidden. If that is not desired it can be disabled with `only=false`. E.g., `%(trailers:key=Reviewed-by)` shows trailer lines with key `Reviewed-by`.
- ? `only[=<BOOL>]`: select whether non-trailer lines from the trailer block should be included.
- ? `separator=<SEP>`: specify a separator inserted between trailer lines. When this option is not given each trailer line is terminated with a line feed character. The string `SEP` may contain the literal formatting codes described above. To use comma as separator one must use `%x2C` as it would otherwise be parsed as next option. E.g.,
`%(trailers:key=Ticket,separator=%x2C)` shows all trailer lines whose key is "Ticket" separated by a comma and a space.
- ? `unfold[=<BOOL>]`: make it behave as if `interpret-trailer?s --unfold` option was given. E.g., `%(trailers:only,unfold=true)` unfolds and shows all trailer lines.
- ? `keyonly[=<BOOL>]`: only show the key part of the trailer.
- ? `valueonly[=<BOOL>]`: only show the value part of the trailer.
- ? `key_value_separator=<SEP>`: specify a separator inserted between trailer lines. When this option is not given each trailer key-value pair is separated by `:`. Otherwise it shares the same semantics as `separator=<SEP>` above.

Note

Some placeholders may depend on other options given to the revision traversal engine.

For example, the `%g*` relog options will insert an empty string unless we are traversing relog entries (e.g., by `git log -g`). The `%d` and `%D` placeholders will use the "short" decoration format if `--decorate` was not already provided on the command line.

If you add a `+` (plus sign) after `%` of a placeholder, a line-feed is inserted immediately before the expansion if and only if the placeholder expands to a non-empty string.

If you add a - (minus sign) after % of a placeholder, all consecutive line-feeds immediately preceding the expansion are deleted if and only if the placeholder expands to an empty string.

If you add a ` ` (space) after % of a placeholder, a space is inserted immediately before the expansion if and only if the placeholder expands to a non-empty string.

? tformat:

The tformat: format works exactly like format:, except that it provides "terminator" semantics instead of "separator" semantics. In other words, each commit has the message terminator character (usually a newline) appended, rather than a separator placed between entries. This means that the final entry of a single-line format will be properly terminated with a new line, just as the "oneline" format does. For example:

```
$ git log -2 --pretty=format:%h 4da45bef \  
| perl -pe '$_ .= " -- NO NEWLINE\n" unless /\n/'  
4da45be
```

```
7134973 -- NO NEWLINE
```

```
$ git log -2 --pretty=tformat:%h 4da45bef \  
| perl -pe '$_ .= " -- NO NEWLINE\n" unless /\n/'  
4da45be
```

```
7134973
```

In addition, any unrecognized string that has a % in it is interpreted as if it has tformat: in front of it. For example, these two are equivalent:

```
$ git log -2 --pretty=tformat:%h 4da45bef
```

```
$ git log -2 --pretty=%h 4da45bef
```

EXAMPLES

? Print the list of commits reachable from the current branch.

```
git rev-list HEAD
```

? Print the list of commits on this branch, but not present in the upstream branch.

```
git rev-list @{upstream}..HEAD
```

? Format commits with their author and commit message (see also the porcelain git-log(1)).

```
git rev-list --format=medium HEAD
```

? Format commits along with their diffs (see also the porcelain git-log(1), which can do

this in a single process).

```
git rev-list HEAD |
```

```
git diff-tree --stdin --format=medium -p
```

- ? Print the list of commits on the current branch that touched any file in the Documentation directory.

```
git rev-list HEAD -- Documentation/
```

- ? Print the list of commits authored by you in the past year, on any branch, tag, or other ref.

```
git rev-list --author=you@example.com --since=1.year.ago --all
```

- ? Print the list of objects reachable from the current branch (i.e., all commits and the blobs and trees they contain).

```
git rev-list --objects HEAD
```

- ? Compare the disk size of all reachable objects, versus those reachable from reflogs, versus the total packed size. This can tell you whether running `git repack -ad` might reduce the repository size (by dropping unreachable objects), and whether expiring reflogs might help.

```
# reachable objects
```

```
git rev-list --disk-usage --objects --all
```

```
# plus reflogs
```

```
git rev-list --disk-usage --objects --all --reflog
```

```
# total disk size used
```

```
du -c .git/objects/pack/*.pack .git/objects/??/*
```

```
# alternative to du: add up "size" and "size-pack" fields
```

```
git count-objects -v
```

- ? Report the disk size of each branch, not including objects used by the current branch.

This can find outliers that are contributing to a bloated repository size (e.g., because somebody accidentally committed large build artifacts).

```
git for-each-ref --format='%(%(refname))' |
```

```
while read branch
```

```
do
```

```
    size=$(git rev-list --disk-usage --objects HEAD..$branch)
```

```
    echo "$size $branch"
```

```
done |
```

sort -n

? Compare the on-disk size of branches in one group of refs, excluding another. If you co-mingle objects from multiple remotes in a single repository, this can show which remotes are contributing to the repository size (taking the size of origin as a baseline).

```
git rev-list --disk-usage --objects --remotes=$suspect --not --remotes=origin
```

GIT

Part of the git(1) suite

Git 2.34.1

07/07/2023

GIT-REV-LIST(1)