# git-range-diff

git-range-diff(1) ==========

NAME ---- git-range-diff - Compare two commit ranges (e.g. two versions of a branch)

SYNOPSIS --------

git range-diff [--color=[<when>]] [--no-color] [<diff-options>] [--no-dual-color] [--creation-factor=<factor>] [--left-only | --right-only] [--diff-merges=<format>] [--remerge-diff] [--no-notes | --notes[=<ref>]] ( <range1> <range2> | <rev1>...<rev2> | <base> <rev1> <rev2> ) [[--] <path>...]

DESCRIPTION -----------

This command shows the differences between two versions of a patch series, or more generally, two commit ranges (ignoring merge commits).

In the presence of *<path>* arguments, these commit ranges are limited accordingly.

To that end, it first finds pairs of commits from both commit ranges that correspond with each other. Two commits are said to correspond when the diff between their patches (i.e. the author information, the commit message and the commit diff) is reasonably small compared to the patches' size. See 'Algorithm' below for details.

Finally, the list of matching commits is shown in the order of the second commit range, with unmatched commits being inserted just after all of their ancestors have been shown.

There are three ways to specify the commit ranges:

- `<range1>` `<range2>`: Either commit range can be of the form `<base>..<rev>`, `<rev>^!` or `<rev>^-<n>`. See 'SPECIFYING RANGES' in linkgit:gitrevisions[7] for more details.
- `<rev1>…<rev2>`. This is equivalent to `<rev2>..<rev1>` `<rev1>..<rev2>`.
- `<base>` `<rev1>` `<rev2>`: This is equivalent to `<base>..<rev1>` `<base>..<rev2>`.

OPTIONS ------- `--no-dual-color`:: When the commit diffs differ, `git` `range-diff` recreates the original diffs' coloring, and adds outer -/+ diff markers with the **background** being red/green to make it easier to see e.g. when there was a change in what exact lines were added. Additionally, the commit diff lines that are only present in the first commit range are shown "dimmed" (this can be overridden using the `color.diff.<slot>` config setting where *<slot>* is one of `contextDimmed`, `oldDimmed` and `newDimmed`), and the commit diff lines that are only present in the

second commit range are shown in bold (which can be overridden using the config settings `color.diff.<slot>` with *<slot>* being one of `contextBold`, `oldBold` or `newBold`).
This is known to `range-diff` as "dual coloring". Use `--no-dual-color` to revert to color all lines according to the outer diff markers (and completely ignore the inner diff when it comes to color).

`--creation-factor=<percent>`
Set the creation/deletion cost fudge factor to *<percent>*. Defaults to 60. Try a larger value if `git range-diff` erroneously considers a large change a total rewrite (deletion of one commit and addition of another), and a smaller one in the reverse case. See the ’Algorithm’ section below for an explanation of why this is needed.

`--left-only`
Suppress commits that are missing from the first specified range (or the "left range" when using the `<rev1>…<rev2>` form).

`--right-only`
Suppress commits that are missing from the second specified range (or the "right range" when using the `<rev1>…<rev2>` form).

`--diff-merges=<format>`
Instead of ignoring merge commits, generate diffs for them using the corresponding `--diff-merges=<format>` option of linkgit:git-log[1], and include them in the comparison.

Note: In the common case, the `remerge` mode will be the most natural one to use, as it shows only the diff on top of what Git’s merge machinery would have produced. In other words, if a merge commit is the result of a non-conflicting `git merge`, the `remerge` mode will represent it with an empty diff.

`--remerge-diff`
Convenience option, equivalent to `--diff-merges=remerge`.

`--notes[=<ref>]`

`--no-notes`
This flag is passed to the `git log` program (see linkgit:git-log[1]) that generates the patches.

`<range1>` `<range2>`
Compare the commits specified by the two ranges, where *<range1>* is considered an older version of *<range2>*.

`<rev1>…<rev2>`
Equivalent to passing `<rev2>..<rev1>` and `<rev1>..<rev2>`.

`<base>` `<rev1>` `<rev2>`
Equivalent to passing `<base>..<rev1>` and `<base>..<rev2>`. Note that *<base>* does not need to be the exact branch point of the branches. Example: after rebasing a branch `my-topic`, `git range-diff my-topic@{u} my-topic@{1} my-topic` would show the differences introduced by the rebase.

`git` `range-diff` also accepts the regular diff options (see linkgit:git-diff[1]), most notably the `--color[=<when>]` and `--no-color` options. These options are used when generating the "diff between patches", i.e. to compare the author, commit message and diff of corresponding old/new commits. There is currently no means to tweak most of the diff options passed to `git` `log` when generating those patches.

OUTPUT STABILITY ----------

The output of the `range-diff` command is subject to change. It is intended to be human-readable porcelain output, not something that can be used across versions of Git to get a textually stable `range-diff` (as opposed to something like the `--stable` option to linkgit:git-patch-id[1]). There's also no equivalent of linkgit:git-apply[1] for `range-diff`, the output is not intended to be machine-readable.

This is particularly true when passing in diff options. Currently some options like `--stat` can, as an emergent effect, produce output that's quite useless in the context of `range-diff`. Future versions of `range-diff` may learn to interpret such options in a manner specific to `range-diff` (e.g. for `--stat` producing human-readable output which summarizes how the diffstat changed).

CONFIGURATION ---------- This command uses the `diff.color.*` and `pager.range-diff` settings (the latter is on by default). See linkgit:git-config[1].

EXAMPLES --------

When a rebase required merge conflicts to be resolved, compare the changes introduced by the rebase directly afterwards using:

```
$ git range-diff @{u} @{1} @
```

A typical output of `git` `range-diff` would look like this:

```
-:  ------- > 1:  0ddba11 Prepare for the inevitable!
1:  c0debee = 2:  cab005e Add a helpful message at the start
2:  f00dbal ! 3:  decafe1 Describe a bug
    @@ -1,3 +1,3 @@
     Author: A U Thor <author@example.com>

    -TODO: Describe a bug
    +Describe a bug
    @@ -324,5 +324,6
      This is expected.

    -+What is unexpected is that it will also crash.
    ++Unexpectedly, it also crashes. This is a bug, and the jury is
```

```
    ++still out there how to fix it best. See ticket #314 for details.


      Contact
3:  bedead < -:  ------- TO-UNDO
```

In this example, there are 3 old and 3 new commits, where the developer removed the 3rd, added a new one before the first two, and modified the commit message of the 2nd commit as well as its diff.

When the output goes to a terminal, it is color-coded by default, just like regular `git diff`'s output. In addition, the first line (adding a commit) is green, the last line (deleting a commit) is red, the second line (with a perfect match) is yellow like the commit header of `git show`'s output, and the third line colors the old commit red, the new one green and the rest like `git show`'s commit header.

A naive color-coded diff of diffs is actually a bit hard to read, though, as it colors the entire lines red or green. The line that added "What is unexpected" in the old commit, for example, is completely red, even if the intent of the old commit was to add something.

To help with that, `range` uses the `--dual-color` mode by default. In this mode, the diff of diffs will retain the original diff colors, and prefix the lines with -/+ markers that have their **background** red or green, to make it more obvious that they describe how the diff itself changed.

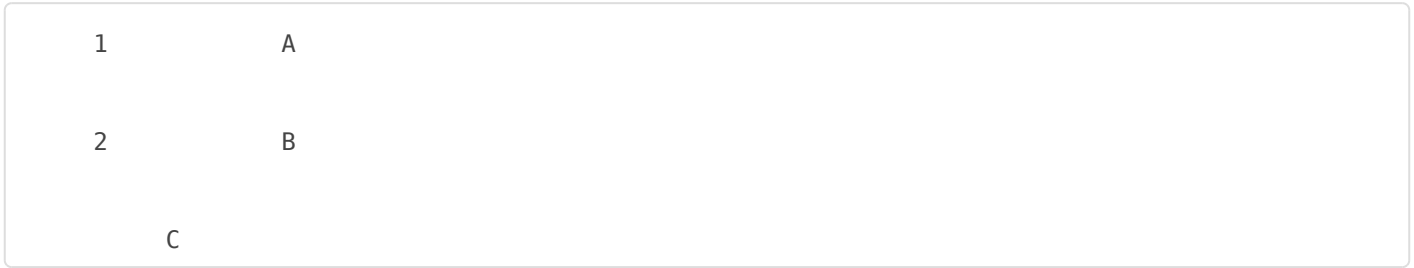
Algorithm ---------

The general idea is this: we generate a cost matrix between the commits in both commit ranges, then solve the least-cost assignment.

The cost matrix is populated thusly: for each pair of commits, both diffs are generated and the "diff of diffs" is generated, with 3 context lines, then the number of lines in that diff is used as cost.

To avoid false positives (e.g. when a patch has been removed, and an unrelated patch has been added between two iterations of the same patch series), the cost matrix is extended to allow for that, by adding fixed-cost entries for wholesale deletes/adds.

Example: Let commits `1–2` be the first iteration of a patch series and `A–C` the second iteration. Let's assume that `A` is a cherry-pick of `2`, and `C` is a cherry-pick of `1` but with a small modification (say, a fixed typo). Visualize the commits as a bipartite graph:



We are looking for a "best" explanation of the new series in terms of the old one. We can represent an "explanation" as an edge in the graph:

```
1           A
       /
2 --------'  B


     C
```

This explanation comes for "free" because there was no change. Similarly `C` could be explained using `1`, but that comes at some cost c>0 because of the modification:

```
1 ----.      A
  |    /
2 ----+---'  B
  |
  `----- C
  c>0
```

In mathematical terms, what we are looking for is some sort of a minimum cost bipartite matching; `1` is matched to `C` at some cost, etc. The underlying graph is in fact a complete bipartite graph; the cost we associate with every edge is the size of the diff between the two commits' patches. To explain also new commits, we introduce dummy nodes on both sides:

```
1 ----.      A
  |    /
2 ----+---'  B
  |
o     `----- C
  c>0
o            o


o            o
```

The cost of an edge `o–C` is the size of `C`'s diff, modified by a fudge factor that should be smaller than 100%. The cost of an edge `o–o` is free. The fudge factor is necessary because even if `1` and `C` have nothing in common, they may still share a few empty lines and such, possibly making the assignment `1–C`, `o–o` slightly cheaper than `1–o`, `o–C` even if `1` and `C` have nothing in common. With the fudge factor we require a much larger common part to consider patches as corresponding.

The overall time needed to compute this algorithm is the time needed to compute n+m commit diffs and then n*m diffs of patches, plus the time needed to compute the least-cost assignment

between n and m diffs. Git uses an implementation of the Jonker-Volgenant algorithm to solve the assignment problem, which has cubic runtime complexity. The matching found in this case will look like this:

```
    1 ----.      A
      |    /
    2 ----+---'  B
       .--+-----'
    o -'  `----- C
      c>0
    o ---------- o


    o ---------- o
```

SEE ALSO -------- linkgit:git-log[1]

GIT --- Part of the linkgit:git[1] suite

---

□□□ #1
□ □□□ □□□ 2026-10-31 20:12:14 UTC
□ □□□ □□□ 2026-10-31 20:12:14 UTC