
cmp-buffer

nvim-cmp source for buffer words.

Setup

```
1 require('cmp').setup({
2   sources = {
3     { name = 'buffer' },
4   },
5 })
```

Configuration

The below source configuration are available. To set any of these options, do:

```
1 cmp.setup({
2   sources = {
3     {
4       name = 'buffer',
5       option = {
6         -- Options go into this table
7       },
8     },
9   },
10  })
```

keyword_length (type: number)

Default: 3

The number of characters that need to be typed to trigger auto-completion.

keyword_pattern (type: string)

Default: `[[\%(-\?\d\+\%(\.\d\+)\)\?|\h\w*\%([\-\.\w\+])\]]`

A vim's regular expression for creating a word list from buffer content.

You can set this to `[[\k\+]]` if you want to use the `iskeyword` option for recognizing words. Lua's `[[]]` string literals are particularly useful here to avoid escaping all of the backslash (`\`) characters used for writing regular expressions.

NOTE: Be careful with where you set this option! You must do this:

```
1 cmp.setup({
2   sources = {
3     {
4       name = 'buffer',
5       -- Correct:
6       option = {
7         keyword_pattern = [[\k\+]],
8       }
9     },
10  },
11 })
```

Instead of this:

```
1 cmp.setup({
2   sources = {
3     {
4       name = 'buffer',
5       -- Wrong:
6       keyword_pattern = [[\k\+]],
7     },
8   },
9 })
```

The second notation is allowed by nvim-cmp (documented here), but it is meant for a different purpose and will not be detected by this plugin as the pattern for searching words.

get_bufnrs (type: fun(): number[])

Default: `function() return { vim.api.nvim_get_current_buf() } end`

A function that specifies the buffer numbers to complete.

You can use the following pre-defined recipes.

```
All buffers
1 cmp.setup {
2   sources = {
3     {
4       name = 'buffer',
5       option = {
6         get_bufnrs = function()
7           return vim.api.nvim_list_bufs()
8         end
9       }
10    }
11  }
```

```
12 }
```

Visible buffers

```
1 cmp.setup {
2   sources = {
3     {
4       name = 'buffer',
5       option = {
6         get_bufnrs = function()
7           local bufs = {}
8           for _, win in ipairs(vim.api.nvim_list_wins()) do
9             bufs[vim.api.nvim_win_get_buf(win)] = true
10          end
11          return vim.tbl_keys(bufs)
12        end
13      }
14    }
15  }
16 }
```

indexing_interval (type: number)

Default: 100

Optimization option. See the section Indexing.

indexing_batch_size (type: number)

Default: 1000

Optimization option. See the section Indexing.

max_indexed_line_length (type: number)

Default: 1024 * 40 (40 Kilobytes)

Optimization option. See the section Indexing.

Locality bonus comparator (distance-based sorting)

This source also provides a comparator function which uses information from the word indexer to sort completion results based on the distance of the word from the cursor line. It will also sort completion

results coming from other sources, such as Language Servers, which might improve accuracy of their suggestions too. The usage is as follows:

```
1 local cmp = require('cmp')
2 local cmp_buffer = require('cmp_buffer')
3
4 cmp.setup({
5   sources = {
6     { name = 'buffer' },
7     -- The rest of your sources...
8   },
9   sorting = {
10    comparators = {
11      function(...) return cmp_buffer:compare_locality(...) end,
12      -- The rest of your comparators...
13    }
14  }
15 })
```

Indexing and how to optimize it

When a buffer is opened, this source first has to scan all lines in the buffer, match all words and store all of their occurrences. This process is called *indexing*. When actually editing the text in the buffer, the index of words is kept up-to-date with changes to the buffer's contents, this is called *watching*. It is done by re-running the indexer on just the changed lines. Indexing happens completely asynchronously in background, unlike watching, which must be performed synchronously to ensure that the index of words is kept perfectly in-sync with the lines in the buffer. However, most of the time this will not be a problem since many typical text edit operations affect only one or two lines, unless you are pasting a 1000-line snippet.

Note that you can freely edit the buffer while it is being indexed, the underlying algorithm is written in such a way that your changes will not break the index or cause errors. If a crash does happen - it is a bug, so please report it.

The speed of indexing is configurable with two options: `indexing_interval` and `indexing_batch_size`. Essentially, when indexing, a timer is started, which pulls a batch of `indexing_batch_size` lines from the buffer, scans them for words, and repeats after `indexing_interval` milliseconds. Decreasing interval and/or increasing the batch size will make the indexer faster, but at the expense of higher CPU usage and more lag when editing the file while indexing is still in progress. Setting `indexing_batch_size` to a negative value will switch the indexer to the “synchronous” mode: this will process all lines in one go, take less time in total (since no other code will be running on the Lua thread), but with the obvious downside that the editor UI will be blocked.

The option `max_indexed_line_length` controls plugin's behavior in files with very long lines.

This is known to slow this source down significantly (see issue #13), so by default it will take only the first few kilobytes of the line it is currently on. In other words, very long lines are not ignored, but only a part of them is indexed.

Performance on large text files

This source has been tested on code files of a few megabytes in size (5-10) and contains optimizations for them, however, the indexed words can still take up tens of megabytes of RAM if the file is large. So, if you wish to avoid accidentally running this source on big files, you can tweak `get_bufnrs`, for example like this:

```
1 get_bufnrs = function()
2   local buf = vim.api.nvim_get_current_buf()
3   local byte_size = vim.api.nvim_buf_get_offset(buf,
4     vim.api.nvim_buf_line_count(buf))
5   if byte_size > 1024 * 1024 then -- 1 Megabyte max
6     return {}
7   end
8   return { buf }
```

Of course, this snippet can be combined with any other recipes for `get_bufnrs`.