Performance considerations on TCGAome

Pablo Riesgo

September 20, 2016

Motivation

We will test the performance of the common operations that we require to map terms to genes and vice versa: basic operations on data tables and specially set operations are critical (i.e.: union, intersection, etc.). For this purpose we will compare two object types the native data.frame and the data.frame subclass data.table https://cran.r-project.org/web/packages/data.table/index.html. We will also compare the library fastmatch https://cran.r-project.org/web/packages/fastmatch/index.html and its native counterpart match. Also, we will evaluate the improvement obtained by using functions compiled in byte code.

Critical components that are executed several thousand times, like the functions that compute similarity between terms, have been implemented as functions avoiding S4 method dispatching as it adds an undesired overhead.

No effort has been done on parallelization or implementation in C/C++.

The following benchmarks have been performed on fixed and relatively small datasets, we did not analyse how these techniques behave when data scales up.

data.table

As stated in its CRAN site "Fast aggregation of large data (e.g. 100GB in RAM), fast ordered joins, fast add/modify/delete of columns by group using no copies at all, list columns and a fast file reader (fread).". It is very likely that the datasets under analysis are not big enough to observe the advantages of using data.table. Furthermore, as we will see using data.table adds an overhead when dealing with the dataset under analysis.

fastmatch

fastmatch relies on keeping the hash table in memory, thus critical improvement in performance will only be observed when accessing repeteadly the same data. This justifies the dispersed distributions on the execution time that we will observe as compared to native match. For this reason the median will be used to compare native match and fastmatch performance.

Test data

The test data is based on the Gene Ontology annotations for biological process. We will compute our tests on the data structure that associates terms to genes (i.e.: term2gene).

```
## Loads raw annotations
raw_annotations <- TCGAome::load_goa(ontology = "BP")@raw_annotations
```

```
## INFO [2016-09-22 19:58:33] Loading human GOA...
## INFO [2016-09-22 19:59:05] Loaded 14291 GO terms and 16536 genes
```

```
# Creates the data.frame
term2gene_df <- aggregate(data = raw_annotations,
                          Gene ~ Term, c)
rownames(term2gene_df) <- term2gene_df$Term


# Creates the data.table and sets the key of the table
term2gene_dt <- data.table::data.table(
                   aggregate(data = raw_annotations,
                                Gene ~ Term, c))
data.table::setkey(term2gene_dt, Term)
rownames(term2gene_dt) <- term2gene_dt$Term

# Function to select random terms
get_random_term <- function(raw_annotations) {
    random_term <- raw_annotations$Term[runif(
        1,
        max=length(raw_annotations$Term))]
    return(random_term)
}
```

Additionally, we will use two sets of 1000 elements for the set operations.

```
first_set = sample(10000, size = 1000, replace = FALSE)
second_set = sample(10000, size = 1000, replace = FALSE)
```

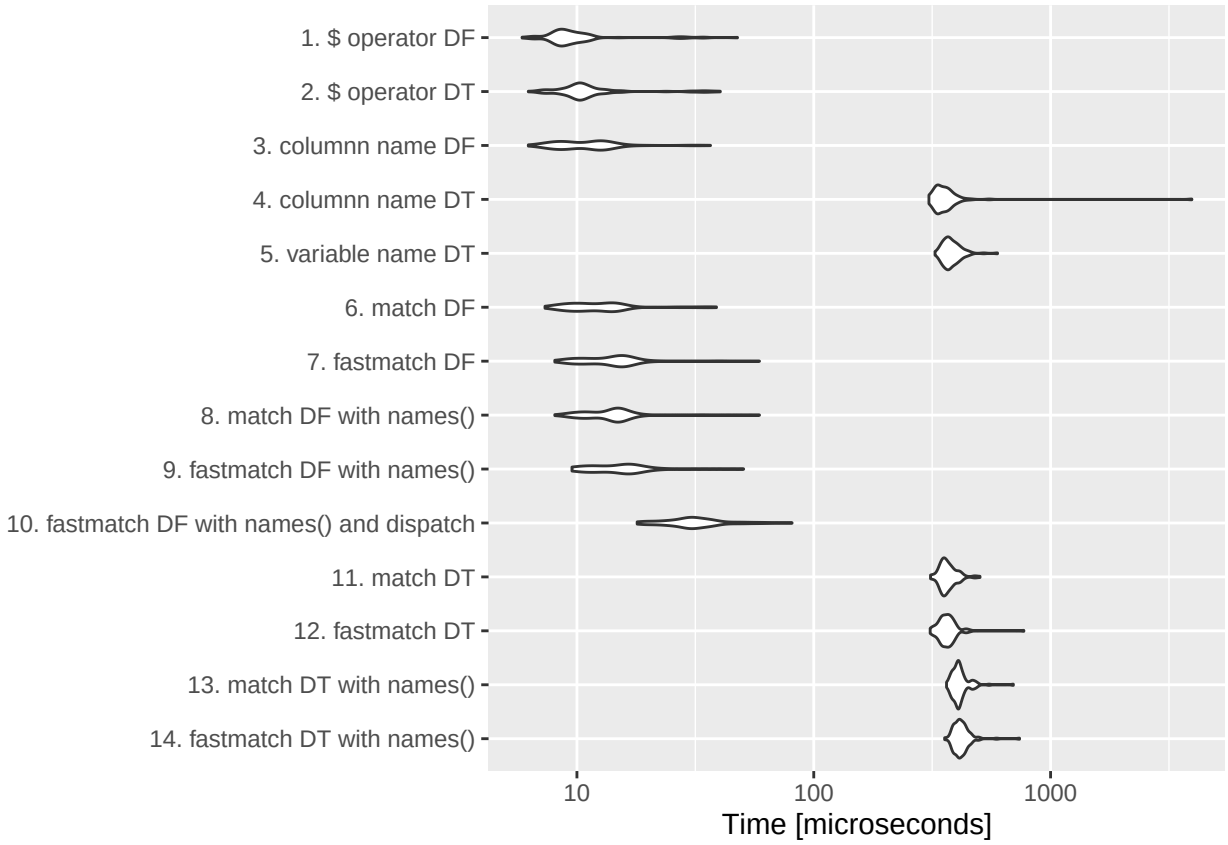
Single column selection

Evaluate the performance of different methods for column selection.

1. Select a column using the $ operator on a data.frame
2. Select a column using the $ operator on a data.table
3. Select a column using the column name on a data.frame
4. Select a column using the column name on a data.table (returns a data.table instead of a vector)
5. Select a column using the data.table column variables
6. Select a column using native match on the column name on a data.frame
7. Select a column using fastmatch on the column name on a data.frame
8. Select a column using native match on the column name on a data.frame (considering call to names())
9. Select a column using fastmatch on the column name on a data.frame (considering call to names())
10. Select a column using fastmatch on the column name on a data.frame (considering call to names() and package resolution)
11. Select a column using native match on the column name on a data.table
12. Select a column using fastmatch on the column name on a data.table
13. Select a column using native match on the column name on a data.table (considering call to names())
14. Select a column using fastmatch on the column name on a data.table (considering call to names())

```
column_names <- names(term2gene_df)
res <- microbenchmark(
    "1" = term2gene_df$Term,
    "2" = term2gene_dt$Term,
```

```
    "3" = term2gene_df[, c("Term")],
    "4" = term2gene_dt[, c("Term"), with = FALSE],
    "5" = term2gene_dt[, Term],
    "6" = term2gene_df[, match("Term", column_names)],
    "7" = term2gene_df[, fmatch("Term", column_names)],
    "8" = term2gene_df[, match("Term", names(term2gene_df))],
    "9" = term2gene_df[, fmatch("Term", names(term2gene_df))],
    "10" = term2gene_df[, fastmatch::fmatch("Term", names(term2gene_df))],
    "11" = term2gene_dt[, match("Term", column_names), with = FALSE],
    "12" = term2gene_dt[, fmatch("Term", column_names), with = FALSE],
    "13" = term2gene_dt[, match("Term", names(term2gene_df)), with = FALSE],
    "14" = term2gene_dt[, fmatch("Term", names(term2gene_df)), with = FALSE]
)
```




```
## Unit: microseconds
##  expr     min       lq      mean   median       uq      max neval cld
##     1   5.865   8.4300  10.90117   8.9805  10.2640   47.650   100 a
##     2   6.232   9.5300  12.15833  10.2640  11.5460   40.319   100 a
##     3   6.232   8.6140  11.38867  11.1800  12.8290   36.654   100 a
##     4 306.420 330.6115 391.44481 350.9535 373.4960 3950.839   100  bc
##     5 325.114 358.6510 383.48744 374.0455 399.1530  597.080   100  bc
##     6   7.331   9.7135  13.17732  11.3630  14.6615   38.853   100 a
##     7   8.064  10.9970  15.19323  14.8450  16.1280   59.012   100 a
##     8   8.065  11.3630  15.14567  14.2955  15.3950   59.012   100 a
##     9   9.530  11.7300  16.14992  15.7610  17.2280   50.582   100 a
```

```
##    10  17.960  26.2075  32.90040  30.7890  35.9200   81.004   100 a
##    11 311.185 348.0220 369.65464 361.7675 383.7595  503.615   100  b
##    12 310.086 346.1895 369.05339 364.3325 382.2925  771.182   100  b
##    13 363.599 392.3720 415.71288 408.3170 423.8935  696.409   100  bc
##    14 357.002 398.0535 422.29571 413.9970 434.5230  739.661   100   c
```

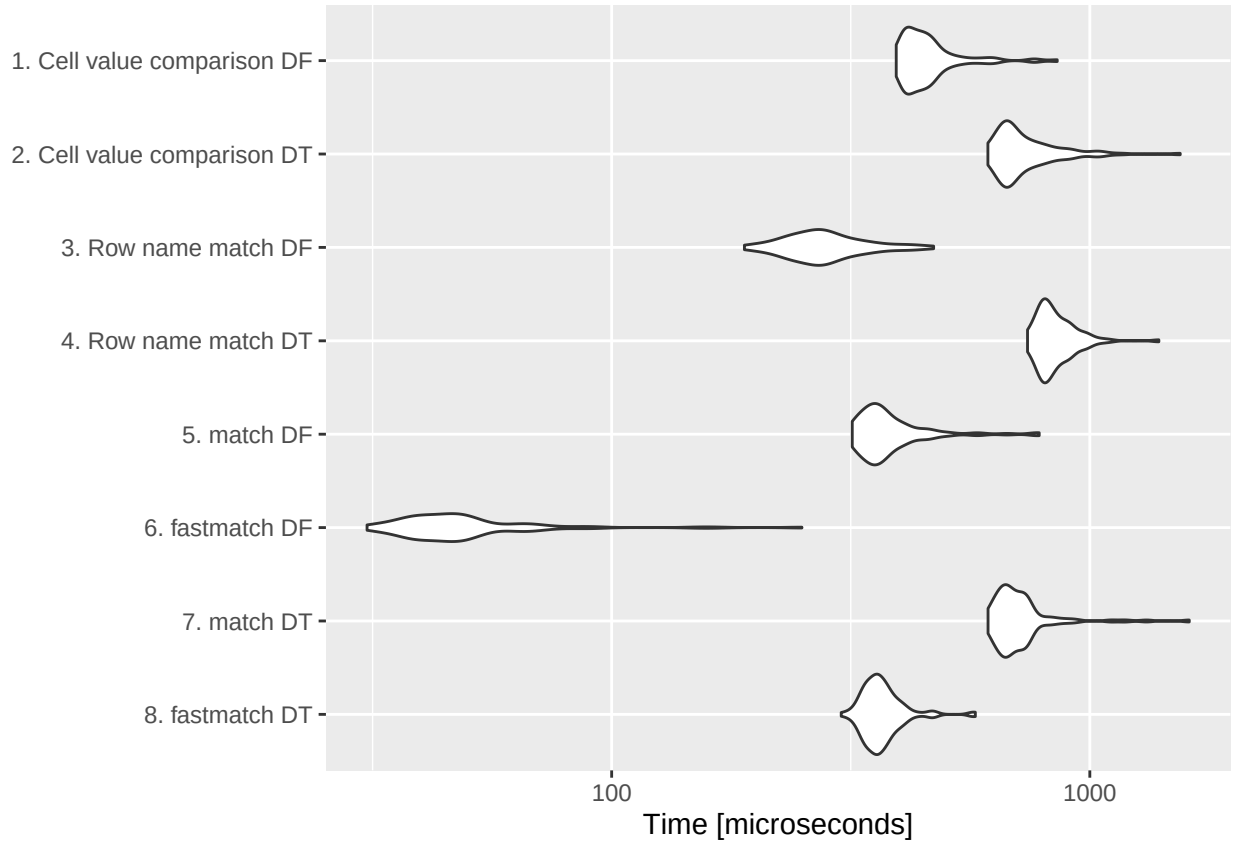
The conclusion is that when using a data.frame or a data.table the **fastest method is the $ operator**. Beware that package resolution is a big burden on performance.

Row selection

Evaluate the performance of different methods for row selection. Usually number of rows >> number of columns and thus the performance when selecting rows is a stronger limitation. In this case we will use as before two objects data.frame and data.table, but also combined with the library fastmatch.

1. Cell value comparison in a data.frame
2. Cell value comparison in a data.table
3. Row name match in a data.frame
4. Row name match in a data.table
5. Cell value native match in a data.frame
6. Cell value fastmatch in a data.frame
7. Cell value native match in a data.table
8. Cell value fastmatch in a data.table

```
random_term <- get_random_term(raw_annotations)
column_values <- term2gene_df$Term
res <- microbenchmark(
    "1" = term2gene_df[term2gene_df$Term == random_term, ],
    "2" = term2gene_dt[term2gene_dt$Term == random_term, ],
    "3" = term2gene_df[random_term, ],
    "4" = term2gene_dt[random_term, ],
    "5" = term2gene_df[match(random_term, column_values), ],
    "6" = term2gene_df[fmatch(random_term, column_values), ],
    "7" = term2gene_dt[match(random_term, column_values), ],
    "8" = term2gene_dt[fmatch(random_term, column_values), ]
    )
```




```
## Unit: microseconds
##  expr     min       lq      mean   median       uq      max neval    cld
##     1 393.655 413.9980 463.29956 443.5030 471.5425  854.751   100    d
##     2 612.108 664.3385 741.23652 690.7290 778.8795 1545.296   100     e
##     3 189.497 242.2775 283.38040 273.2495 307.8865  471.360   100  b
##     4 740.760 795.3730 851.28384 821.9465 892.1375 1394.285   100      f
##     5 318.516 344.3565 386.87055 365.4320 400.2525  783.645   100   c
##     6  30.789  39.2190  51.67776  45.8175  51.8645  249.975   100 a
##     7 612.474 654.4420 722.99787 687.2460 735.8120 1614.204   100     e
##     8 302.022 345.4560 369.48593 361.0335 380.4600  576.187   100   c
```

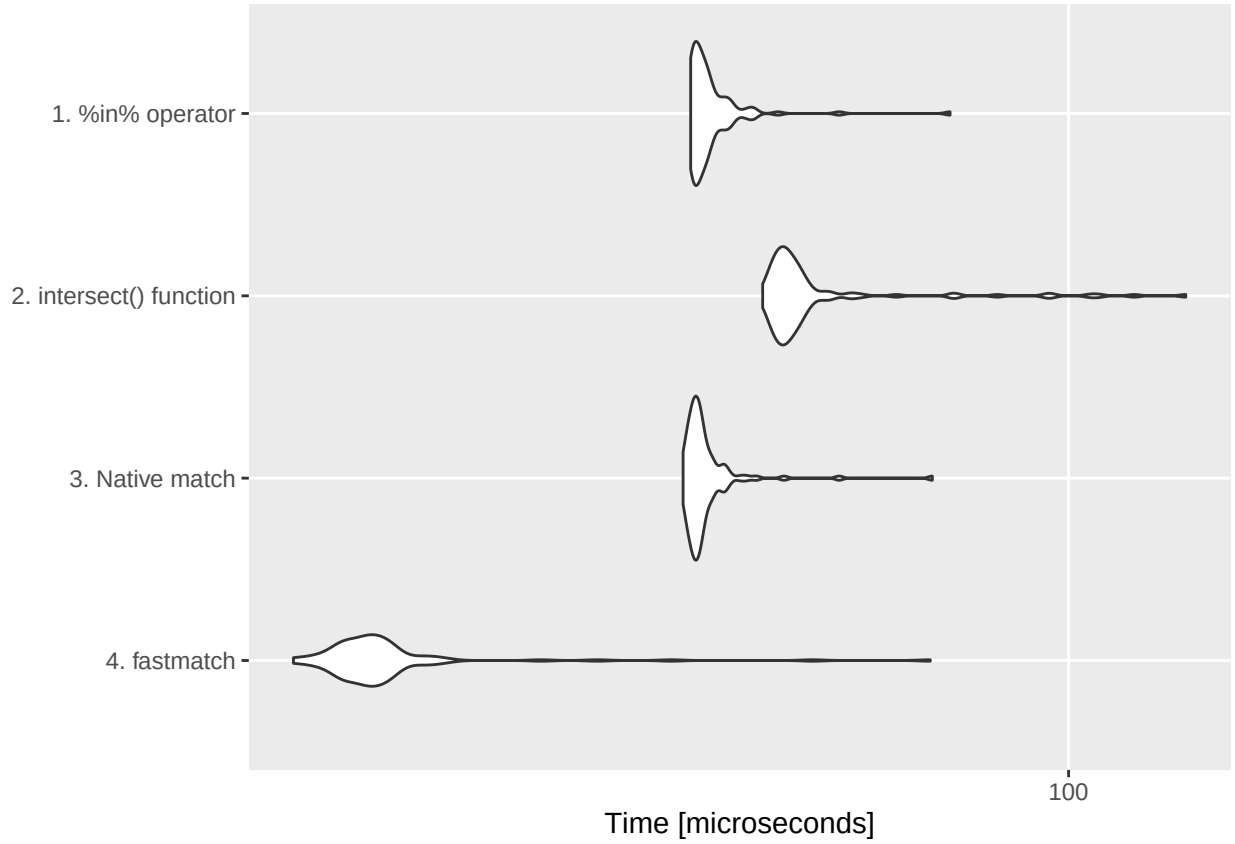
There are two alternatives with very close times, only when looking at the median we can conclude that the **fastest row selection is by row name matching on a data.frame**. The implementation using **fastmatch on a data.frame** gets almost the same average execution time.

Intersection

Evaluate the performance of intersection between sets in a character vector.

1. Intersect using the %in% operator
2. Intersect using the intersect() function
3. Intersect using native match
4. Intersect using fastmatch

```
res <- microbenchmark(
    "1" = first_set[first_set %in% second_set],
    "2" = intersect(first_set, second_set),
    "3" = first_set[!is.na(match(first_set, second_set))],
    "4" = first_set[!is.na(fmatch(first_set, second_set))]
)
```



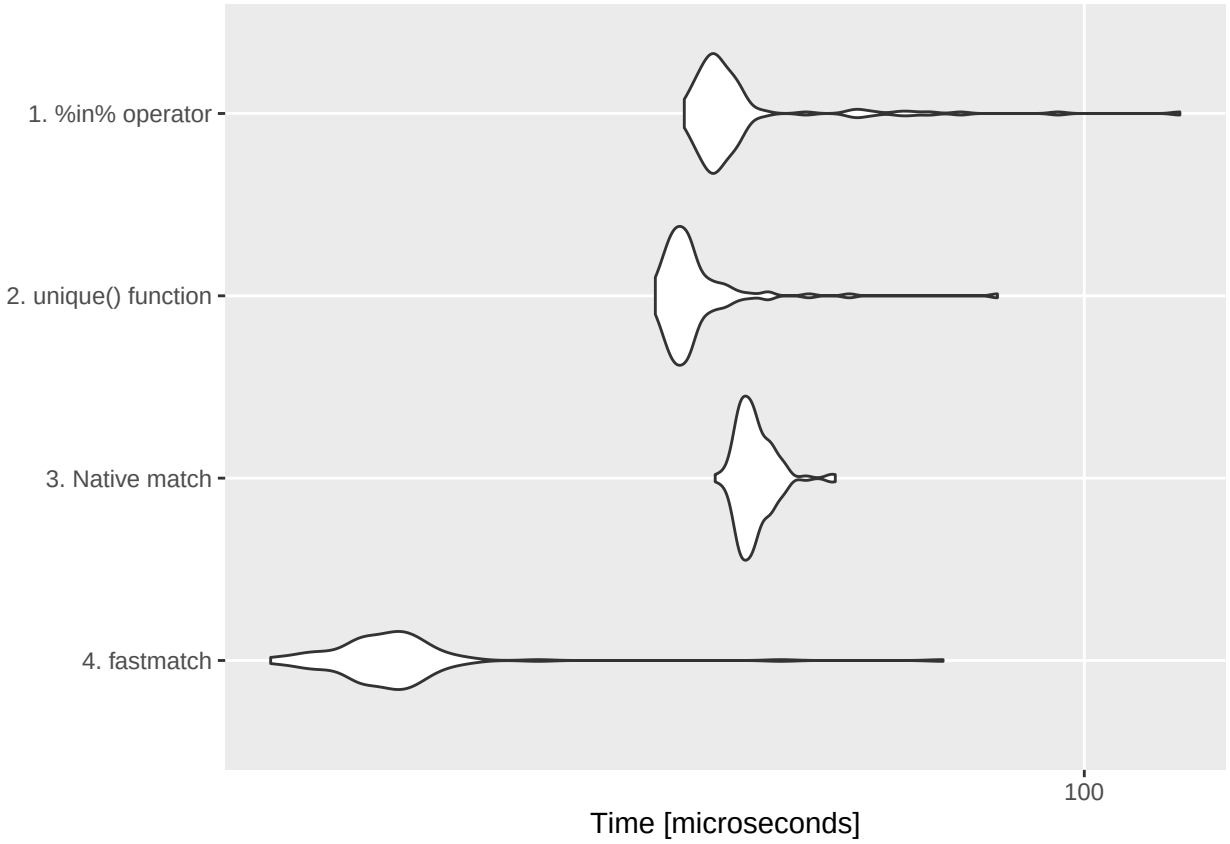

```
## Unit: microseconds
##  expr    min      lq     mean median     uq     max neval cld
##     1 37.753 38.1200 39.87545 38.853 39.953  73.673   100  b
##     2 45.450 47.2830 53.47378 48.566 50.216 135.251   100   c
##     3 37.020 37.7535 39.34397 38.486 39.220  70.375   100  b
##     4 13.562 15.3950 17.72235 16.494 17.228  70.008   100 a
```

The conclusion is that the **fastest intersection is provided by fastmatch**.

Union

1. Union using the union() function
2. Union using the unique() function and concatenation
3. Union using native match
4. Union using fastmatch

```
res <- microbenchmark(
    "1" = union(first_set, second_set),
    "2" = unique(c(first_set, second_set)),
    "3" = c(first_set, second_set[is.na(match(second_set, first_set))]),
    "4" = c(first_set, second_set[is.na(fmatch(second_set, first_set))])
)
```



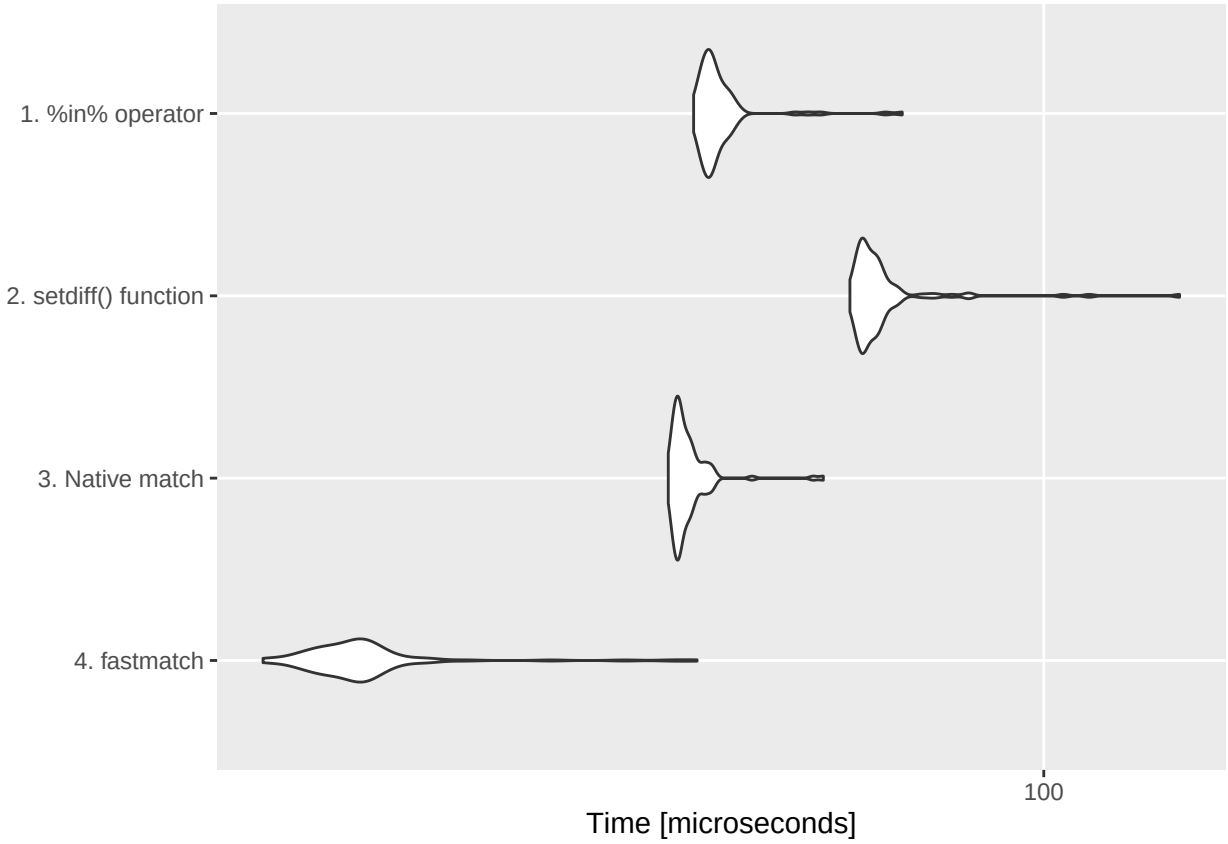

```
## Unit: microseconds
##  expr    min     lq     mean  median      uq     max neval cld
##     1 40.319 42.335 46.55003 43.6175 45.4500 124.255   100   c
##     2 37.753 39.220 41.37467 40.3190 41.4185  82.104   100  b
##     3 43.251 45.817 47.30499 46.9160 48.1990  56.813   100   c
##     4 15.761 19.060 21.19705 20.5260 21.6260  72.574   100 a
```

The conclusion is that the **fastest union is provided by fastmatch**.

Difference

1. Difference using the %in% operator
2. Difference using the setdiff() function
3. Difference using native match
4. Difference using fastmatch

```
res <- microbenchmark(
    "1" = first_set [! first_set %in% second_set],
    "2" = setdiff(first_set, second_set),
    "3" = first_set [is.na(match(first_set, second_set))],
    "4" = first_set [is.na(fmatch(first_set, second_set))]
)
```



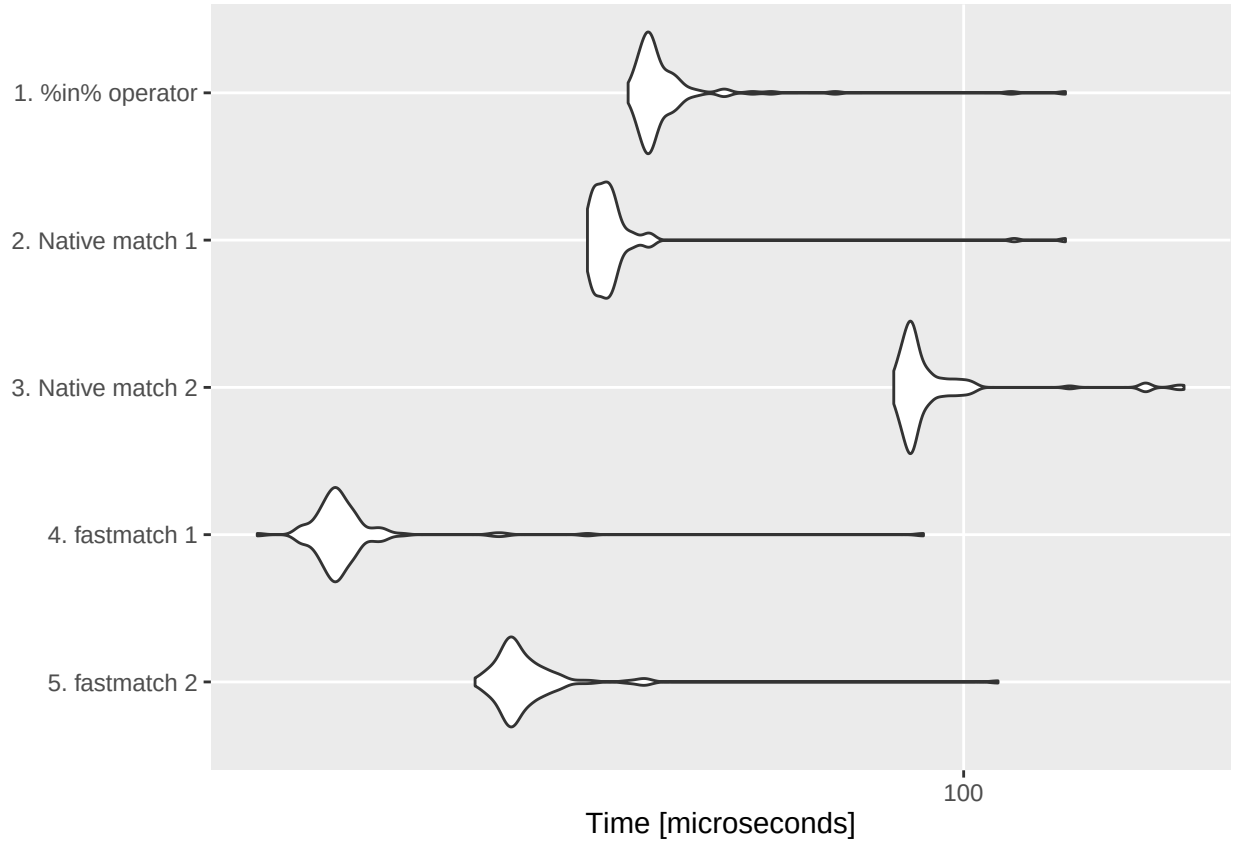

```
## Unit: microseconds
##  expr    min     lq     mean median      uq     max neval  cld
##     1 40.319 41.419 43.22554 42.152 43.6175  69.275   100   c
##     2 60.478 62.311 66.25465 63.777 65.6090 142.215   100    d
##     3 37.753 38.486 39.79852 38.853 39.9530  56.446   100  b
##     4 13.195 15.395 17.31913 16.861 17.5940  40.685   100 a
```

The conclusion is that the **fastest difference is provided by fastmatch**.

XOR

1. XOR using the %in% operator (assuming intersection and union is already computed)
2. XOR using native match (assuming intersection and union is already computed)
3. XOR using native match alternative implementation
4. XOR using fastmatch (assuming intersection and union is already computed)
5. XOR using fastmatch alternative implementation

```
union_set = c(first_set, second_set[is.na(fmatch(second_set, first_set))])
intersection_set = first_set[!is.na(fmatch(first_set, second_set))]
res <- microbenchmark(
    "1" = union_set[!union_set %in% intersection_set],
    "2" = union_set[is.na(match(union_set, intersection_set))],
    "3" = c(first_set[is.na(match(first_set, second_set))],
                        second_set[is.na(match(second_set, first_set))]),
    "4" = union_set[is.na(fmatch(union_set, intersection_set))],
    "5" = c(first_set[is.na(fmatch(first_set, second_set))],
                        second_set[is.na(fmatch(second_set, first_set))])
)
```




```
## Unit: microseconds
##  expr    min     lq     mean  median     uq     max neval   cld
##     1 49.849 51.681 54.62833 52.4145 54.247 123.521   100    d
##     2 45.816 46.550 49.11938 47.6490 48.383 123.521   100   c
##     3 86.502 89.068 94.44462 90.1670 93.099 157.975   100     e
##     4 23.092 26.757 28.33704 27.1240 28.223  92.000   100 a
##     5 36.287 38.487 40.88707 39.5855 40.869 107.394   100  b
```

The conclusion is that the **fastest union is provided by fastmatch**. If intersection and union are already computed use the first version, otherwise use the alternative implementation.

Compiling function to byte code

The just-in-time compilation allows that the code of our R code is translated into byte code and save some interpretation time at execution time. Our implementation is based on the compiler package included in base R.

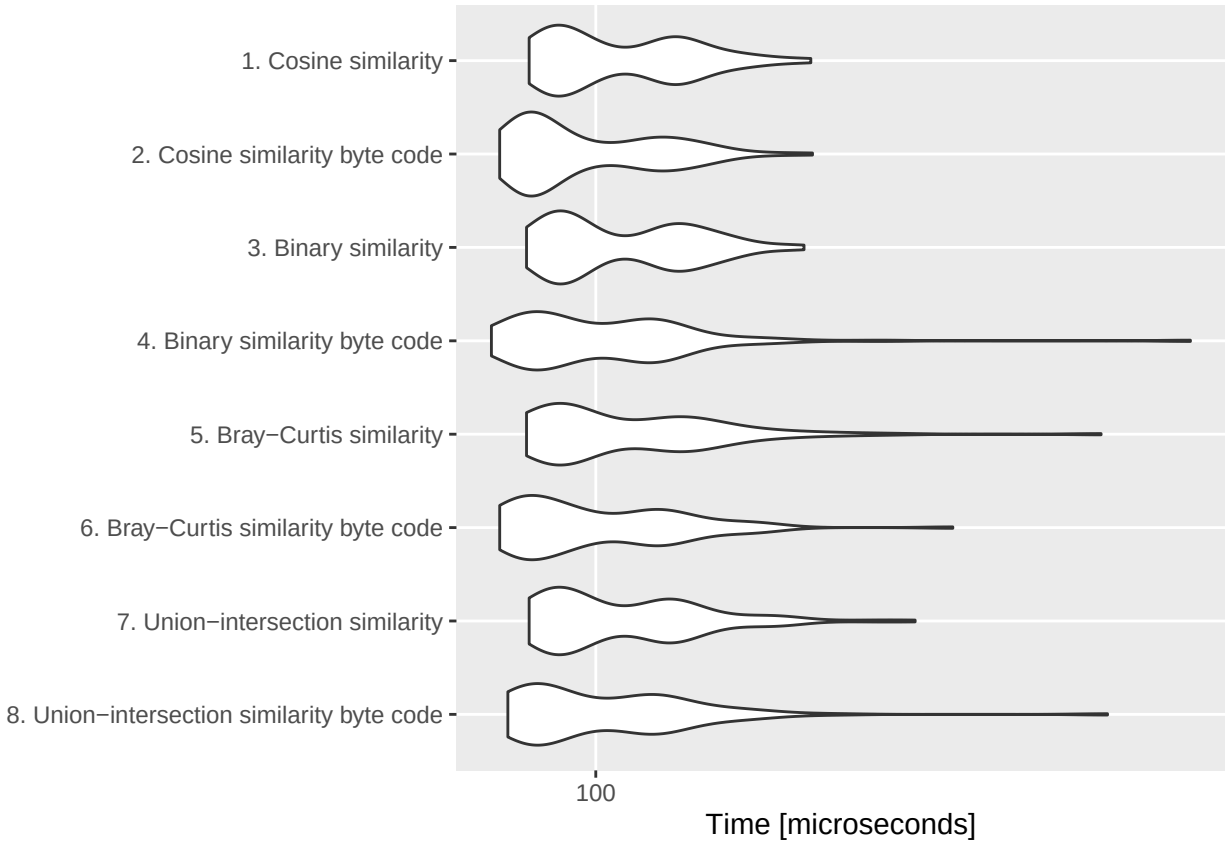
```
get_ui_similarity_bc <- compiler::cmpfun(get_ui_similarity)
```

When benchmarking the execution time of several JIT compiled functions with their plain R counterparts we observe a slight increase in performance.

```
random_term_1 <- get_random_term(raw_annotations)
random_term_2 <- get_random_term(raw_annotations)
kegg <- TCGAome::load_kegg()
```

```
## INFO [2016-09-22 19:59:40] Loading human KEGG...
## INFO [2016-09-22 19:59:41] Loaded 229 KEGG pathways and 5869 genes
```

```
res <- microbenchmark(
    "1" = get_cosine_similarity(kegg, random_term_1, random_term_2),
    "2" = get_cosine_similarity_bc(kegg, random_term_1, random_term_2),
    "3" = get_binary_similarity(kegg, random_term_1, random_term_2),
    "4" = get_binary_similarity_bc(kegg, random_term_1, random_term_2),
    "5" = get_bc_similarity(kegg, random_term_1, random_term_2),
    "6" = get_bc_similarity_bc(kegg, random_term_1, random_term_2),
    "7" = get_ui_similarity(kegg, random_term_1, random_term_2),
    "8" = get_ui_similarity_bc(kegg, random_term_1, random_term_2)
)
```




```
## Unit: microseconds
##  expr    min      lq      mean   median       uq     max neval cld
##     1 90.168 94.1990 104.58655 100.0635 113.8090 139.649   100  ab
##     2 86.135 89.8010  99.08493  92.9160 108.4940 140.015   100  a
##     3 89.801 94.1990 104.82480  99.1470 113.4420 138.183   100  ab
##     4 85.036 90.3505 102.43499  96.7645 109.5930 251.808   100  ab
##     5 89.801 93.6495 106.34217  98.5970 114.3580 219.186   100   b
##     6 86.136 89.6175 100.93594  96.0315 109.7770 174.103   100  ab
##     7 90.167 94.1990 105.16197  99.8800 112.1595 164.207   100  ab
##     8 87.235 90.9000 102.47526  97.6810 109.5935 221.385   100  ab
```