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 21:50:29] Loading human GOA...
## INFO [2016-09-22 21:51:01] Loaded 14291 GO terms and 16536 genes
```

```r
# 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.

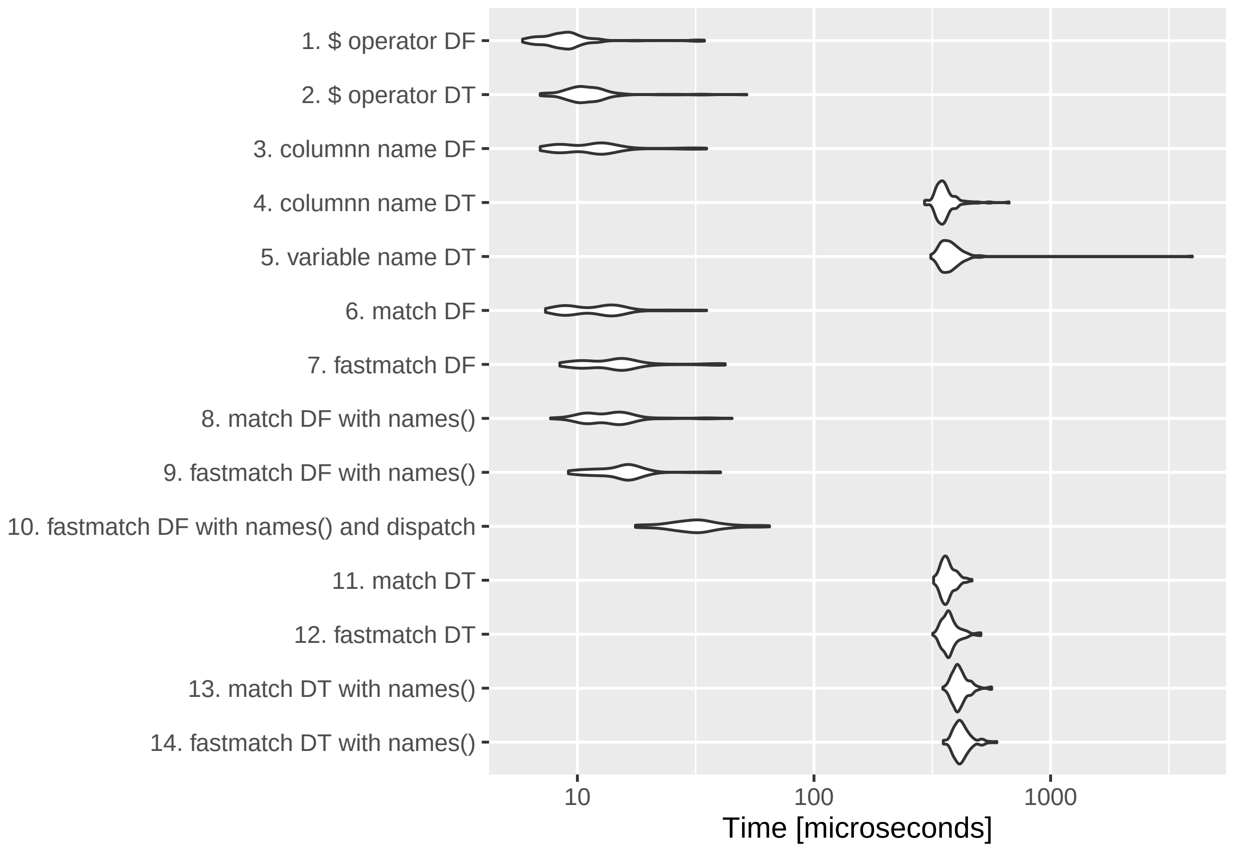
```r
first_set = sample(10000, size = 1000, replace = FALSE)
second_set = sample(10000, size = 1000, replace = FALSE)
```

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())

```r
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   7.6975   9.48644   8.7970   9.5310   34.455   100 a
##     2   6.965   9.5305  11.87991  10.6300  12.0965   52.048   100 a
##     3   6.964   8.6140  12.29036  12.0960  13.1955   35.187   100 a
##     4 293.225 335.5600 360.66355 350.9545 368.7310  667.820   100  b
##     5 311.552 347.6550 411.89364 370.9305 398.0535 3974.665   100   cd
##     6   7.332   8.9805  12.57254  12.8290  14.2955   35.187   100 a
##     7   8.431  10.9960  15.51575  14.6620  16.1280   42.151   100 a
##     8   7.698  10.9970  14.46019  13.7460  15.5780   45.083   100 a
##     9   9.164  12.4630  15.77243  15.7620  17.2270   40.319   100 a
##    10  17.594  26.5740  32.55221  31.5225  36.6535   64.877   100 a
##    11 320.349 348.2050 368.48160 361.7670 381.5590  465.495   100  bc
##    12 317.783 355.5360 377.54600 371.6630 389.8060  507.646   100  b d
##    13 350.404 393.2885 415.34634 405.3840 429.2090  563.725   100   cd
##    14 351.870 398.6030 423.85714 415.0975 439.1050  592.681   100    d
```

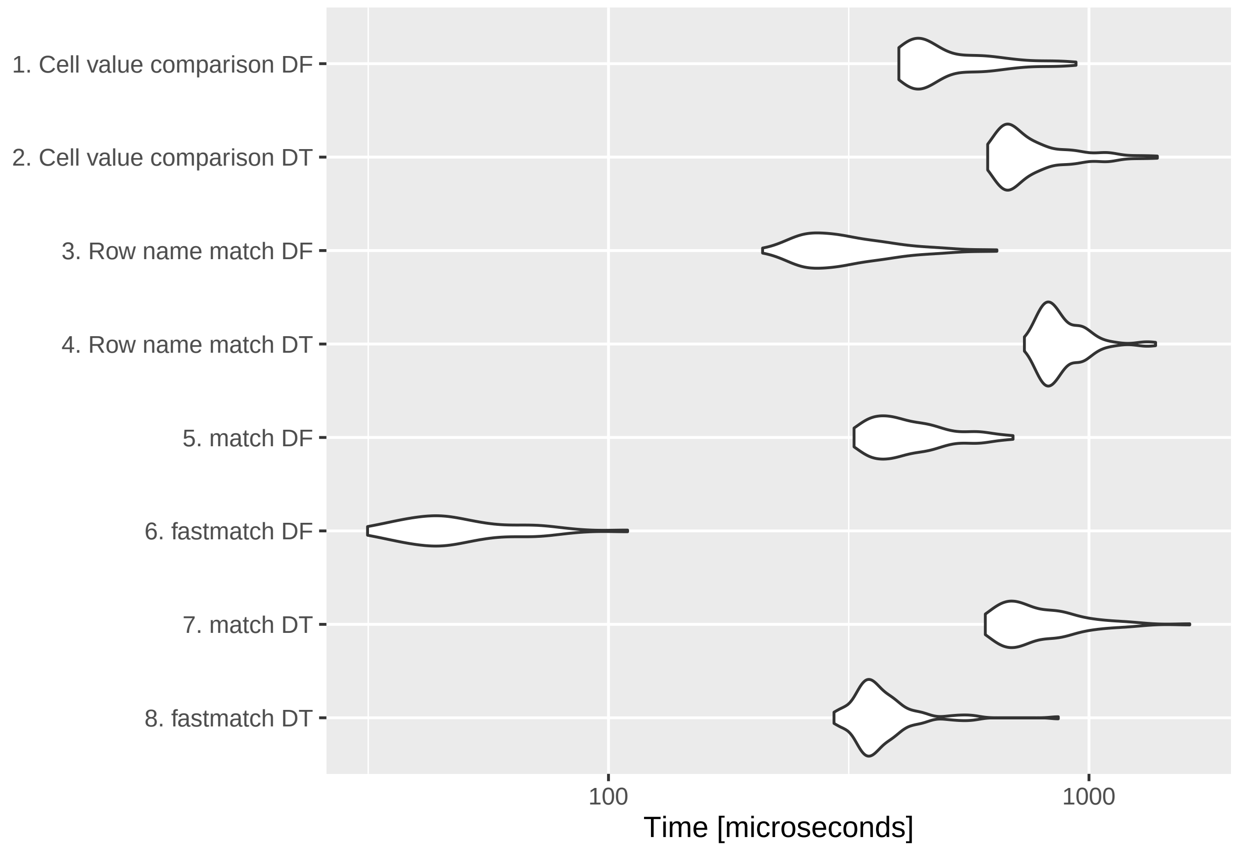
The conclusion is that when using a data.frame or a data.table the **fastest method is the $ operator**. Beware that package resolution is a big burden on performance.

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

```r
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
##     1 402.085 430.4915 516.75077 461.8295 583.8845  939.786   100     e
##     2 615.406 671.6685 771.77586 711.6205 812.7840 1387.687   100      f
##     3 209.290 260.9710 318.32184 297.6235 356.4520  643.263   100  b
##     4 733.796 808.2025 881.07177 846.6875 946.2005 1375.959   100       g
##     5 324.381 365.7985 429.37717 401.9020 465.4955  694.944   100    d
##     6  31.522  39.5860  50.17134  45.6340  56.4455  109.594   100 a
##     7 608.442 671.4855 796.02189 736.9110 871.0615 1620.801   100      f
##     8 294.692 339.0415 375.31739 357.1850 387.7905  863.182   100   c
```

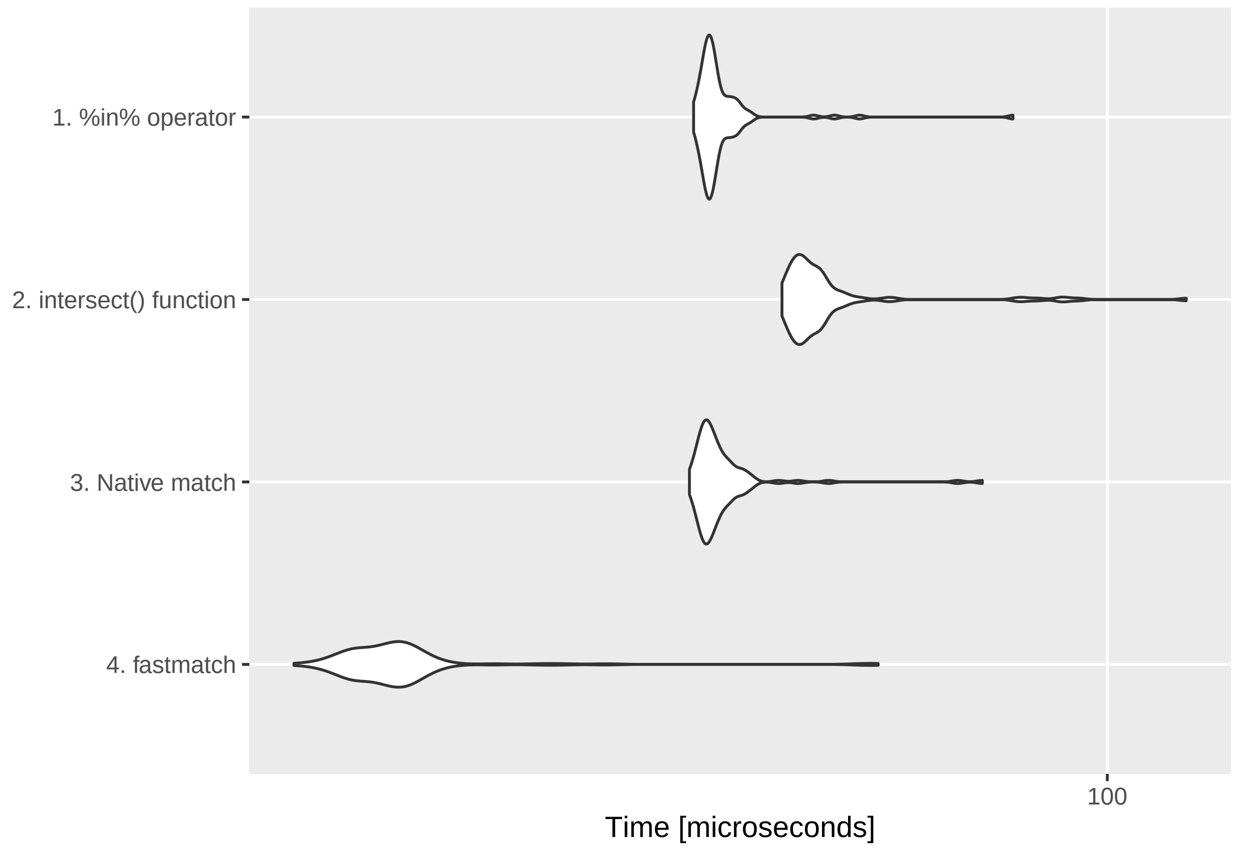
There are two alternatives with very close times, only when looking at the median we can conclude that the **fastest row selection is by row name matching on a data.frame**. The implementation using **fastmatch on a data.frame** gets almost the same average execution time.

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

```r
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))]
)
```



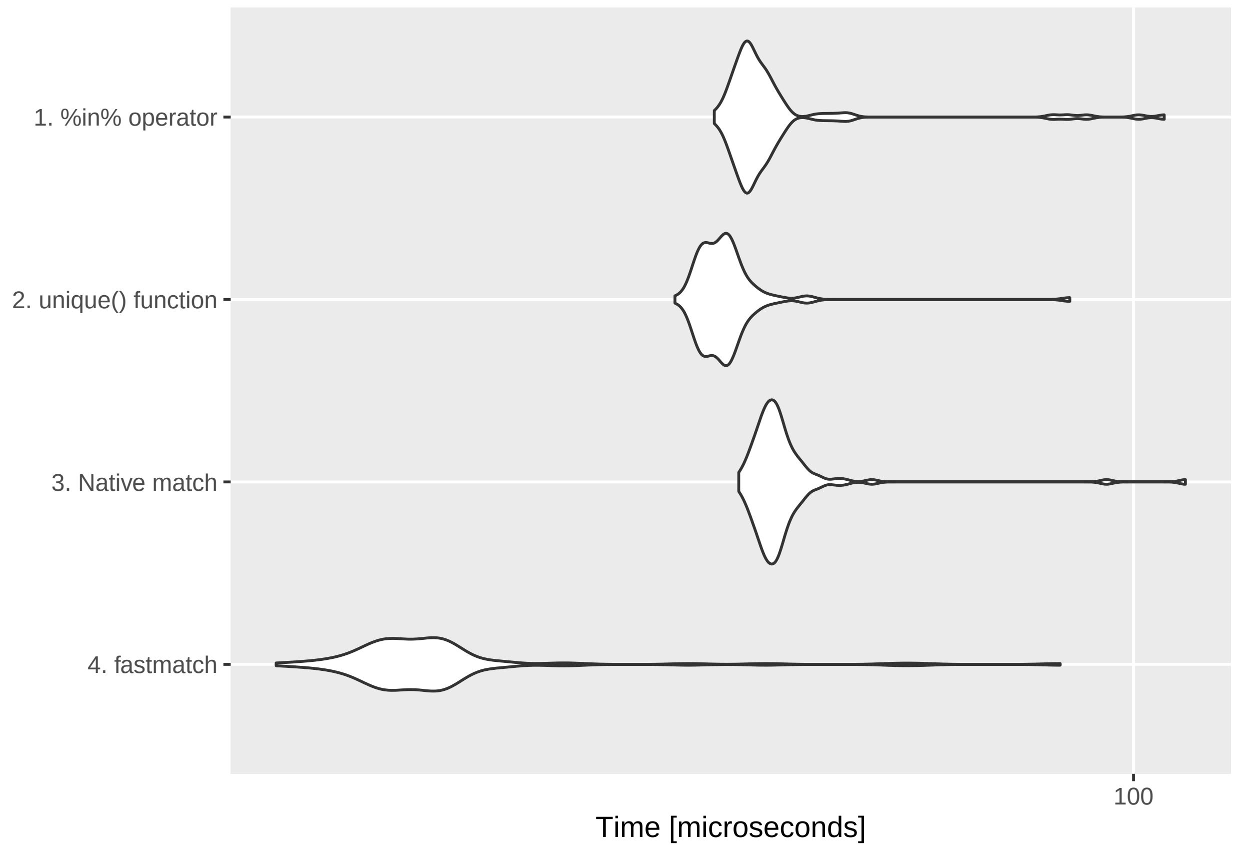
```
## Unit: microseconds
##  expr    min      lq     mean median      uq     max neval cld
##     1 35.188 36.2870 37.90354 36.654 37.9365  78.805   100  b 
##     2 43.984 45.6335 50.35088 46.916 48.7490 122.056   100   c
##     3 34.821 36.1040 38.00986 36.654 38.1200  72.940   100  b 
##     4 12.830 15.0290 17.21661 16.128 17.2275  56.080   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))])
)
```



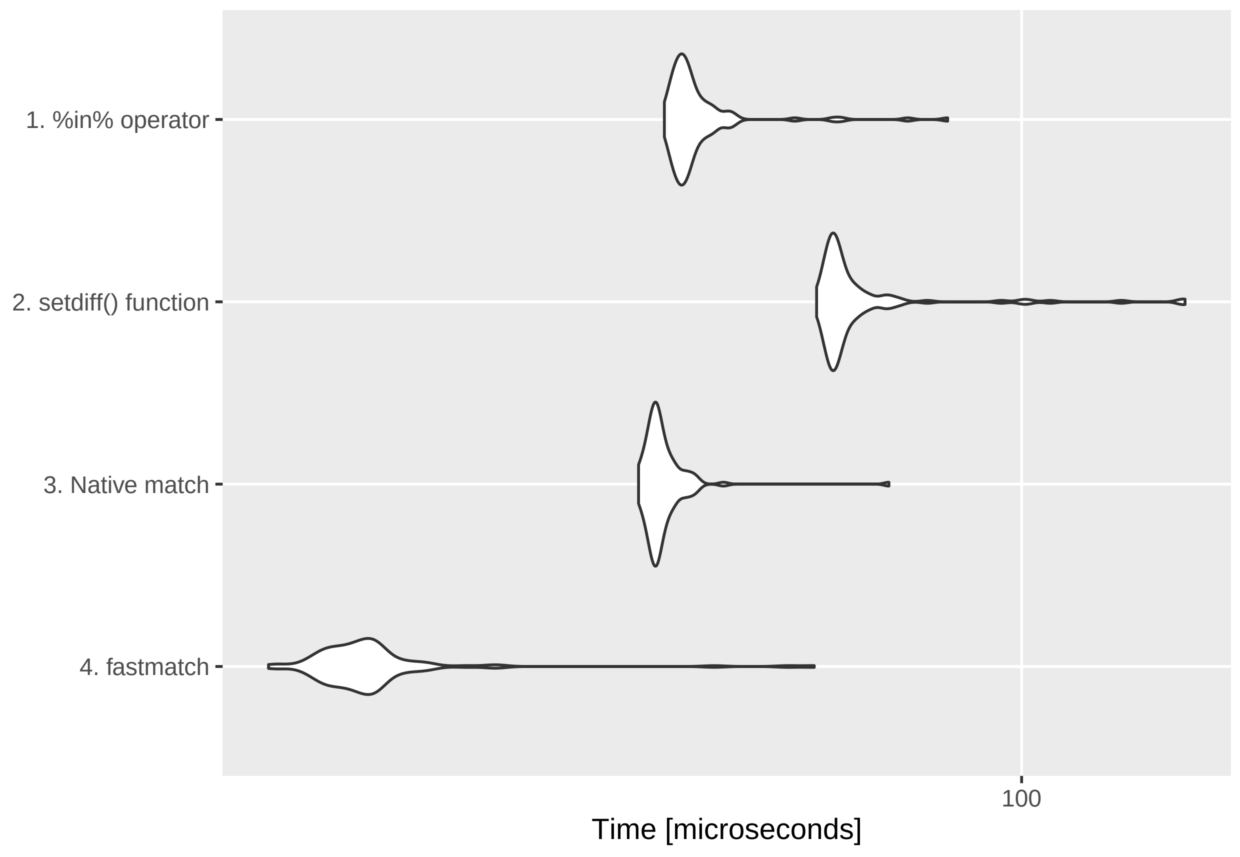
```
## Unit: microseconds
##  expr    min      lq     mean median     uq     max neval cld
##     1 39.586 42.1515 45.95625 42.885 44.534 107.027   100   c
##     2 36.287 38.8525 40.77717 40.319 41.418  86.868   100  b
##     3 41.785 43.9845 46.57565 45.084 46.367 112.159   100   c
##     4 15.029 19.0600 22.17568 20.343 21.992  85.035   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))]
)
```



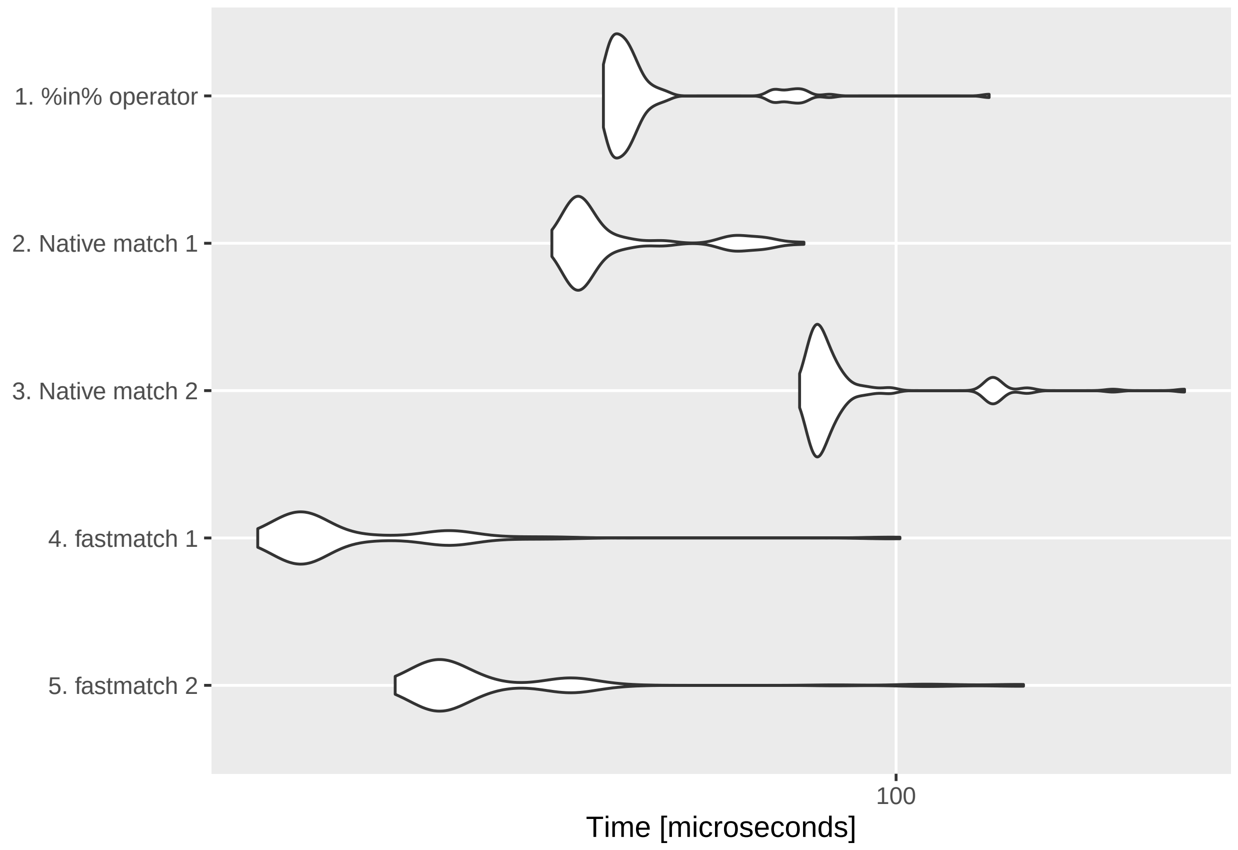
```
## Unit: microseconds
##   expr    min     lq     mean median     uq     max neval  cld
##      1 37.753 39.219 41.63845 39.953 41.602  81.737   100   c
##      2 57.179 59.379 65.62065 60.478 63.410 156.143   100    d
##      3 35.187 36.288 37.72754 37.020 38.120  69.641   100  b
##      4 12.829 15.395 17.59406 16.495 17.228  56.813   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 54.980 56.080 60.82642 57.5455 59.0120 120.956   100   c
##     2 49.482 51.681 56.84955 52.7810 56.9965  82.836   100   c
##     3 82.104 84.669 92.91615 86.1350 89.8005 180.334   100    d
##     4 27.124 29.323 33.86073 30.0560 35.9205 100.797   100 a
##     5 35.921 38.853 46.15412 39.9530 47.2830 129.753   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.

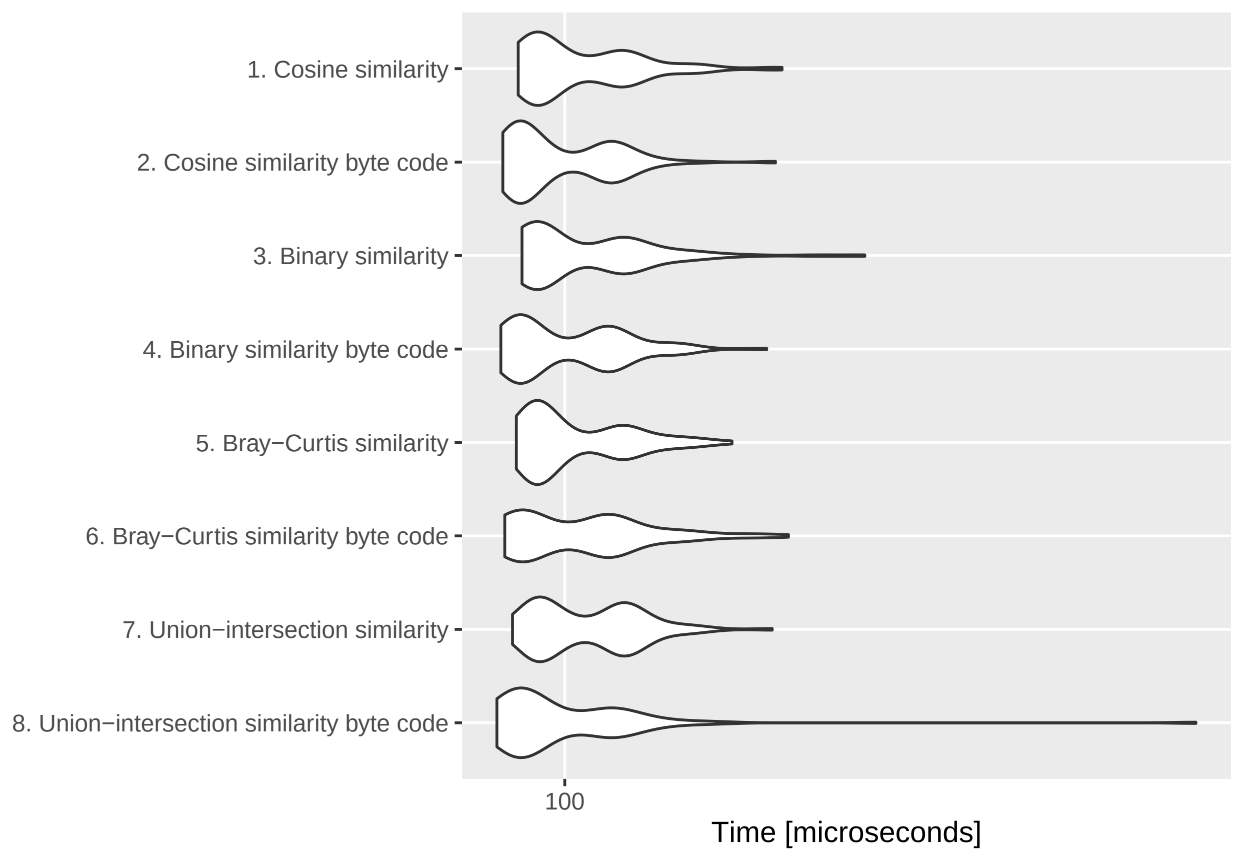
```
get_ui_similarity_bc <- compiler::cmpfun(get_ui_similarity)
```

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 21:51:36] Loading human KEGG...
## INFO [2016-09-22 21:51:37] 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 90.533 93.8320 104.12101  97.6810 112.5260 159.075   100   a
##     2 87.601 90.3500  99.50278  93.6495 109.7770 156.875   100   a
##     3 91.266 93.8330 105.56151  96.2155 113.2590 189.864   100   a
##     4 87.234 90.5340 101.65794  95.2985 110.3270 153.944   100   a
##     5 90.168 93.4660 102.98112  96.0315 112.7095 142.948   100   a
##     6 87.968 91.2665 104.85045 105.5615 111.0595 161.274   100   a
##     7 89.434 94.5655 105.80702 103.7290 113.8080 155.776   100   a
##     8 86.502 90.1670 101.64328  93.4660 109.5935 385.225   100   a
```

Session info

```
sessionInfo()
```

```
## R version 3.2.4 Revised (2016-03-16 r70336)
## Platform: x86_64-w64-mingw32/x64 (64-bit)
## Running under: Windows 7 x64 (build 7601) Service Pack 1
##
## locale:
## [1] LC_COLLATE=English_United States.1252
## [2] LC_CTYPE=English_United States.1252
## [3] LC_MONETARY=English_United States.1252
## [4] LC_NUMERIC=C
## [5] LC_TIME=English_United States.1252
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
## [1] TCGAome_0.0.0.9000     RSQLite_1.0.0          DBI_0.5-1
## [4] ggplot2_2.1.0          fastmatch_1.0-4        microbenchmark_1.4-2.1
## [7] data.table_1.9.6
##
## loaded via a namespace (and not attached):
##   [1] TH.data_1.0-7        colorspace_1.2-6     ggbiplot_0.55
##   [4] qvalue_2.2.2         corpcor_1.6.8        futile.logger_1.4.3
##   [7] roxygen2_5.0.1.9000  topGO_2.22.0         mvtnorm_1.0-5
##  [10] AnnotationDbi_1.32.3 codetools_0.2-14     splines_3.2.4
##  [13] doParallel_1.0.10    GOSemSim_1.28.2      knitr_1.14
##  [16] mixOmics_6.1.0       ade4_1.7-4           jsonlite_1.1
##  [19] Cairo_1.5-9          rJava_0.9-8          gridBase_0.4-7
##  [22] cluster_2.0.4        GO.db_3.2.2          graph_1.48.0
##  [25] shiny_0.14           graphite_1.16.0      compiler_3.2.4
##  [28] assertthat_0.1       Matrix_1.2-6         limma_3.26.9
##  [31] formatR_1.4          htmltools_0.3.5      tools_3.2.4
##  [34] igraph_1.0.1         gtable_0.2.0         reshape2_1.4.1
##  [37] DO.db_2.9            dplyr_0.5.0          rappdirs_0.3.1
##  [40] Rcpp_0.12.7          Biobase_2.30.0       RJSONIO_1.3-0
##  [43] gdata_2.17.0         iterators_1.0.8      made4_1.44.0
##  [46] stringr_1.1.0        RTCGAToolbox_2.0.0   testthat_1.0.2
##  [49] ontoCAT_1.22.0       mime_0.5             gtools_3.5.0
##  [52] devtools_1.11.1      XML_3.98-1.4         DOSE_2.8.3
##  [55] org.Hs.eg.db_3.2.3   zoo_1.7-13           MASS_7.3-45
##  [58] RCircos_1.1.3        scales_0.4.0         treemap_2.4-1
##  [61] reactome.db_1.54.1   doSNOW_1.0.14        sandwich_2.3-4
##  [64] parallel_3.2.4       SparseM_1.7          lambda.r_1.1.9
##  [67] RColorBrewer_1.1-2   yaml_2.1.13          memoise_1.0.0
##  [70] gridExtra_2.2.1      biomaRt_2.26.1       reshape_0.8.5
##  [73] stringi_1.1.1        S4Vectors_0.8.11     foreach_1.4.3
##  [76] ReactomePA_1.14.4    caTools_1.17.1       BiocGenerics_0.16.1
##  [79] chron_2.3-47         bitops_1.0-6         rgl_0.96.0
##  [82] evaluate_0.9         lattice_0.20-33      htmlwidgets_0.7
##  [85] omicade4_1.10.0      cowplot_0.6.2        plyr_1.8.4
##  [88] magrittr_1.5         R6_2.1.3             IRanges_2.4.8
##  [91] snow_0.4-1           gplots_3.0.1         multcomp_1.4-5
##  [94] withr_1.0.2          survival_2.39-4      RCurl_1.95-4.8
##  [97] tibble_1.2           crayon_1.3.2         futile.options_1.0.0
## [100] KernSmooth_2.23-15   ellipse_0.3-8        RGCCA_2.0
## [103] rmarkdown_0.9.6      grid_3.2.4           digest_0.6.10
## [106] xtable_1.8-2         VennDiagram_1.6.17   tidyr_0.6.0
## [109] httpuv_1.3.3         stats4_3.2.4         munsell_0.4.3
```