Thinking in **data.table**

Arun Srinivasan

@arun_sriniv

Co-developer, data.table

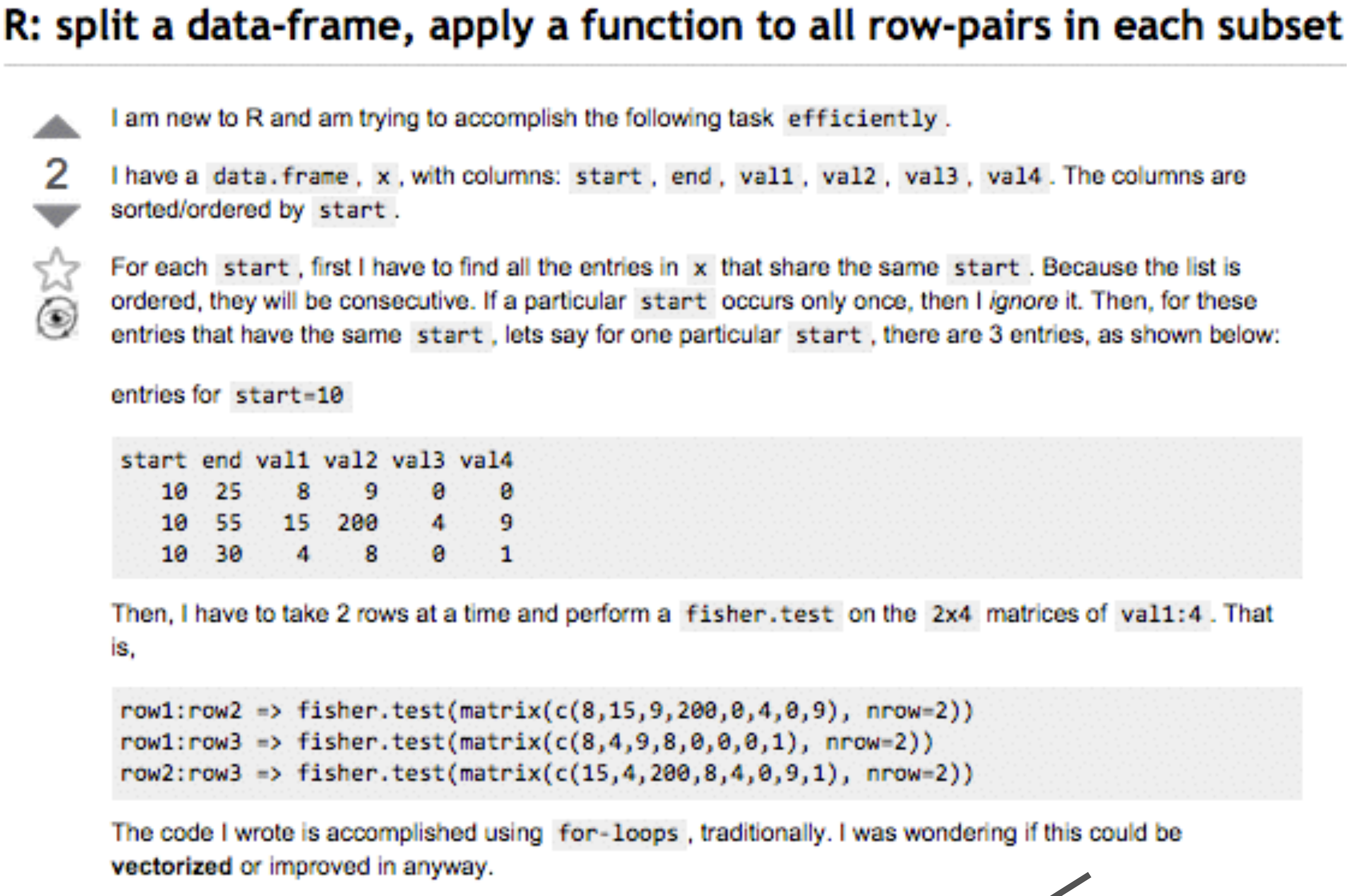
Data analyst, OpenAnalytics

RBelgium - May 26, 2015

About OpenAnalytics

http://www.openanalytics.eu

My first R question



R: split a data-frame, apply a function to all row-pairs in each subset

I am new to R and am trying to accomplish the following task `efficiently`.

I have a `data.frame`, `x`, with columns: `start`, `end`, `val1`, `val2`, `val3`, `val4`. The columns are sorted/ordered by `start`.

For each `start`, first I have to find all the entries in `x` that share the same `start`. Because the list is ordered, they will be consecutive. If a particular `start` occurs only once, then I *ignore* it. Then, for these entries that have the same `start`, lets say for one particular `start`, there are 3 entries, as shown below:

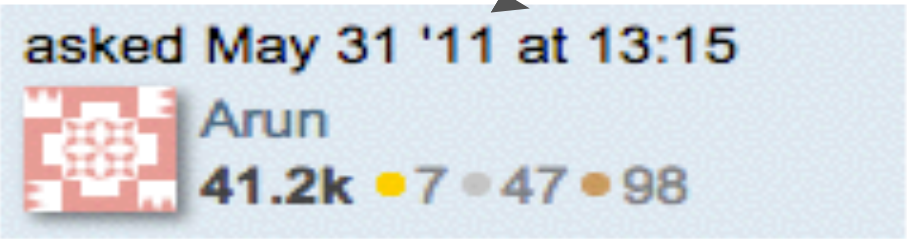
entries for `start=10`

```
start end val1 val2 val3 val4
   10  25    8    9    0    0
   10  55   15  200    4    9
   10  30    4    8    0    1
```

Then, I have to take 2 rows at a time and perform a `fisher.test` on the `2x4` matrices of `val1:4`. That is,

```
row1:row2 => fisher.test(matrix(c(8,15,9,200,0,4,0,9), nrow=2))
row1:row3 => fisher.test(matrix(c(8,4,9,8,0,0,0,1), nrow=2))
row2:row3 => fisher.test(matrix(c(15,4,200,8,4,0,9,1), nrow=2))
```

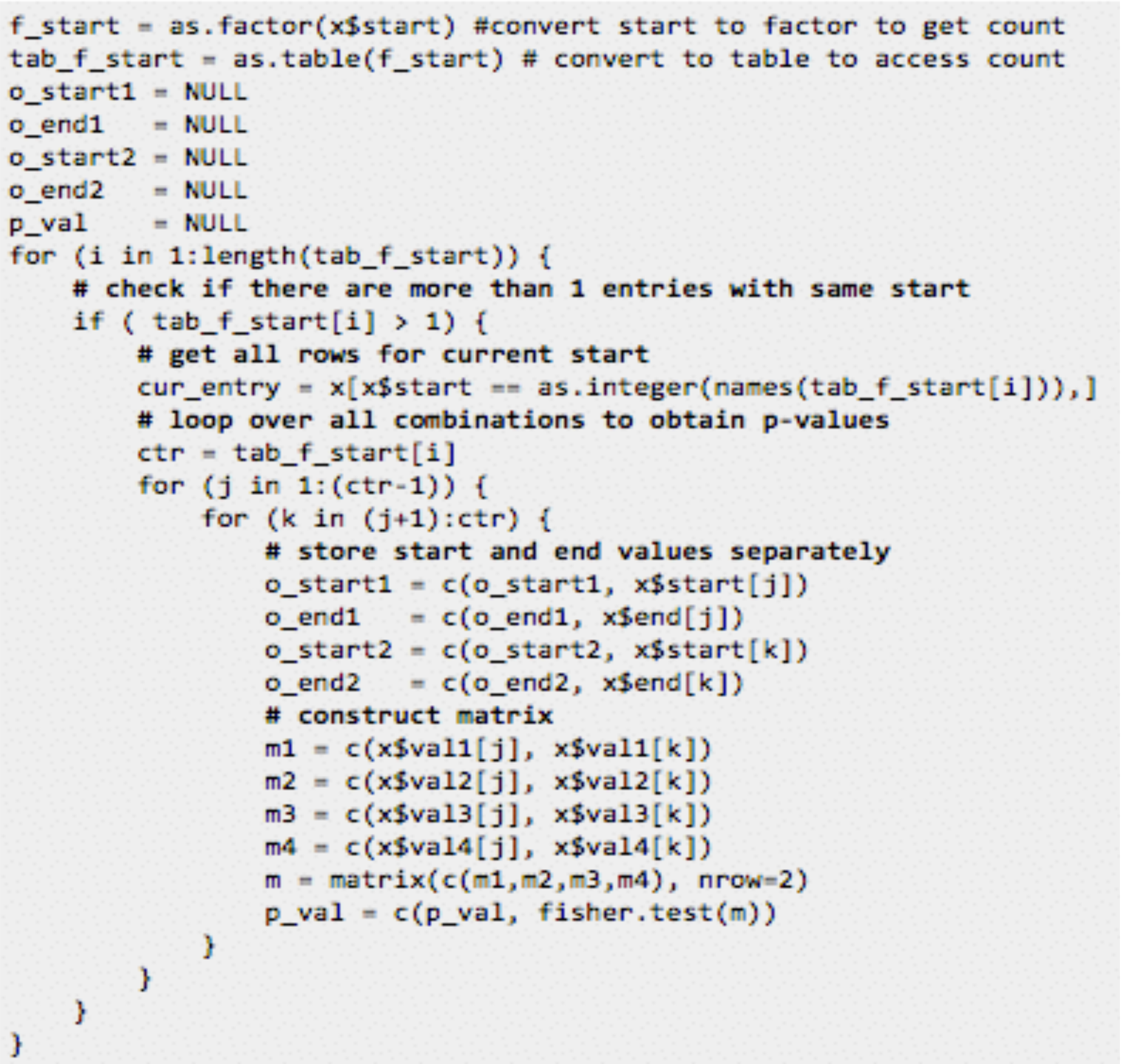
The code I wrote is accomplished using `for-loops`, traditionally. I was wondering if this could be **vectorized** or improved in anyway.



asked May 31 '11 at 13:15

Arun

41.2k ●7 ●47 ●98



```
f_start = as.factor(x$start) #convert start to factor to get count
tab_f_start = as.table(f_start) # convert to table to access count
o_start1 = NULL
o_end1   = NULL
o_start2 = NULL
o_end2   = NULL
p_val    = NULL
for (i in 1:length(tab_f_start)) {
    # check if there are more than 1 entries with same start
    if ( tab_f_start[i] > 1) {
        # get all rows for current start
        cur_entry = x[x$start == as.integer(names(tab_f_start[i])),]
        # loop over all combinations to obtain p-values
        ctr = tab_f_start[i]
        for (j in 1:(ctr-1)) {
            for (k in (j+1):ctr) {
                # store start and end values separately
                o_start1 = c(o_start1, x$start[j])
                o_end1   = c(o_end1, x$end[j])
                o_start2 = c(o_start2, x$start[k])
                o_end2   = c(o_end2, x$end[k])
                # construct matrix
                m1 = c(x$val1[j], x$val1[k])
                m2 = c(x$val2[j], x$val2[k])
                m3 = c(x$val3[j], x$val3[k])
                m4 = c(x$val4[j], x$val4[k])
                m = matrix(c(m1,m2,m3,m4), nrow=2)
                p_val = c(p_val, fisher.test(m))
            }
        }
    }
}
```

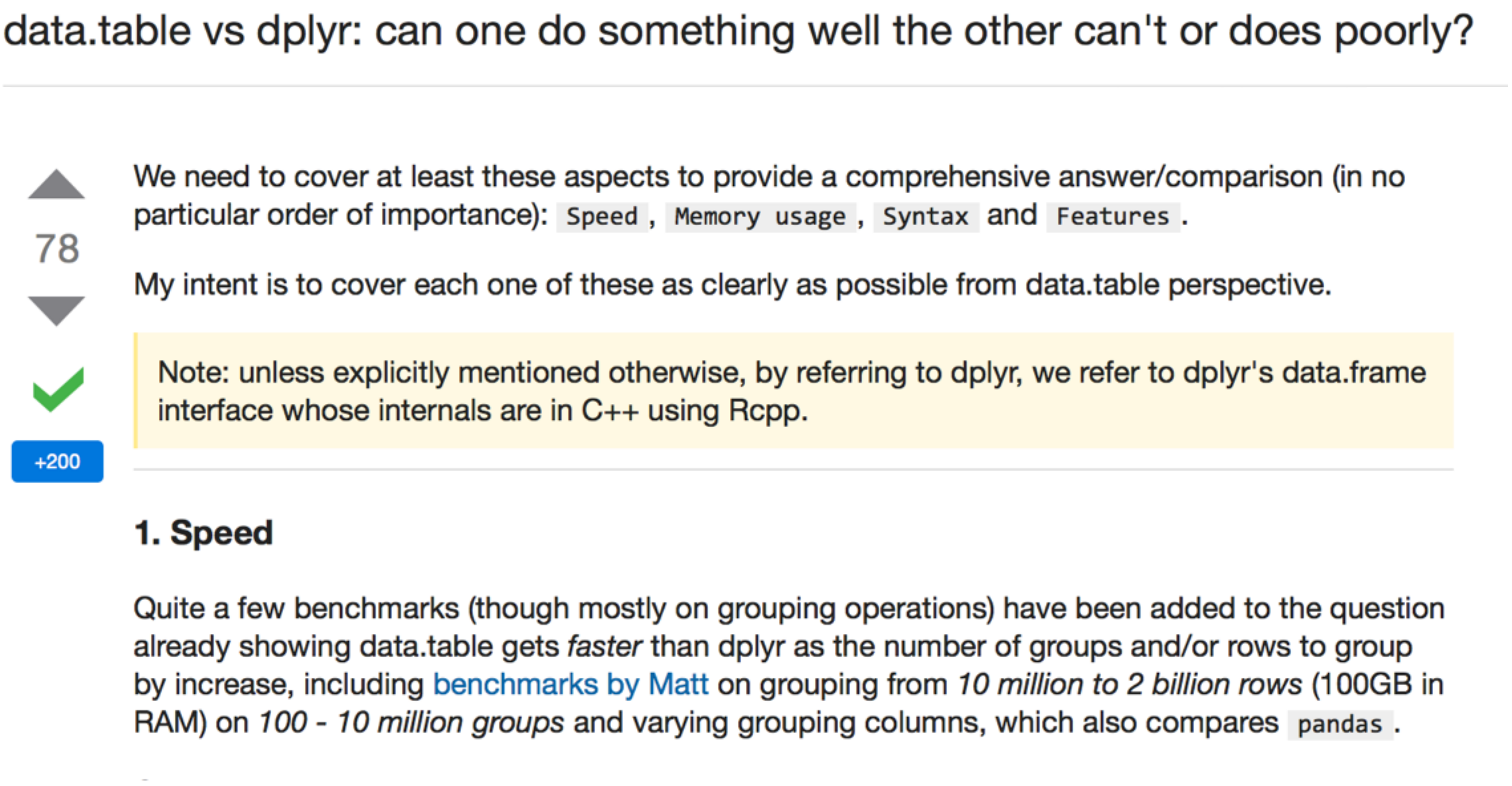
http://stackoverflow.com/q/16152161/559784

Every question is a good question!
Feel free to interrupt.

About data.table

https://github.com/Rdatatable/data.table

http://stackoverflow.com/tags/data.table/topusers



data.table vs dplyr: can one do something well the other can't or does poorly?

We need to cover at least these aspects to provide a comprehensive answer/comparison (in no particular order of importance): `Speed`, `Memory usage`, `Syntax` and `Features`.

My intent is to cover each one of these as clearly as possible from data.table perspective.

> Note: unless explicitly mentioned otherwise, by referring to dplyr, we refer to dplyr's data.frame interface whose internals are in C++ using Rcpp.

1. Speed

Quite a few benchmarks (though mostly on grouping operations) have been added to the question already showing data.table gets *faster* than dplyr as the number of groups and/or rows to group by increase, including benchmarks by Matt on grouping from *10 million to 2 billion rows* (100GB in RAM) on *100 - 10 million groups* and varying grouping columns, which also compares `pandas`.

data.table vs dplyr SO

data.table goals

Goal 1: Reduce programming time

(fewer function calls, less variable name repetition)

Goal 2: Reduced computing time

(fast aggregations, *equi* joins, *rolling* joins, *overlapping range* joins, file reader, data cleaning, update by reference)

fread **melt, dcast**

Data Frames

Looking at `[.data.frame` function

The general form is: `DF[i, j]` + **drop**

Subset rows

```
DF[DF$code == 3L,]
```

code	vA	vB
3	1	6
3	5	10

Select Columns

```
DF[, c(“code”, “vA”)]
```

code	vA
3	1
2	2
1	3
1	4
3	5
2	6

What’s different in a data.table then?

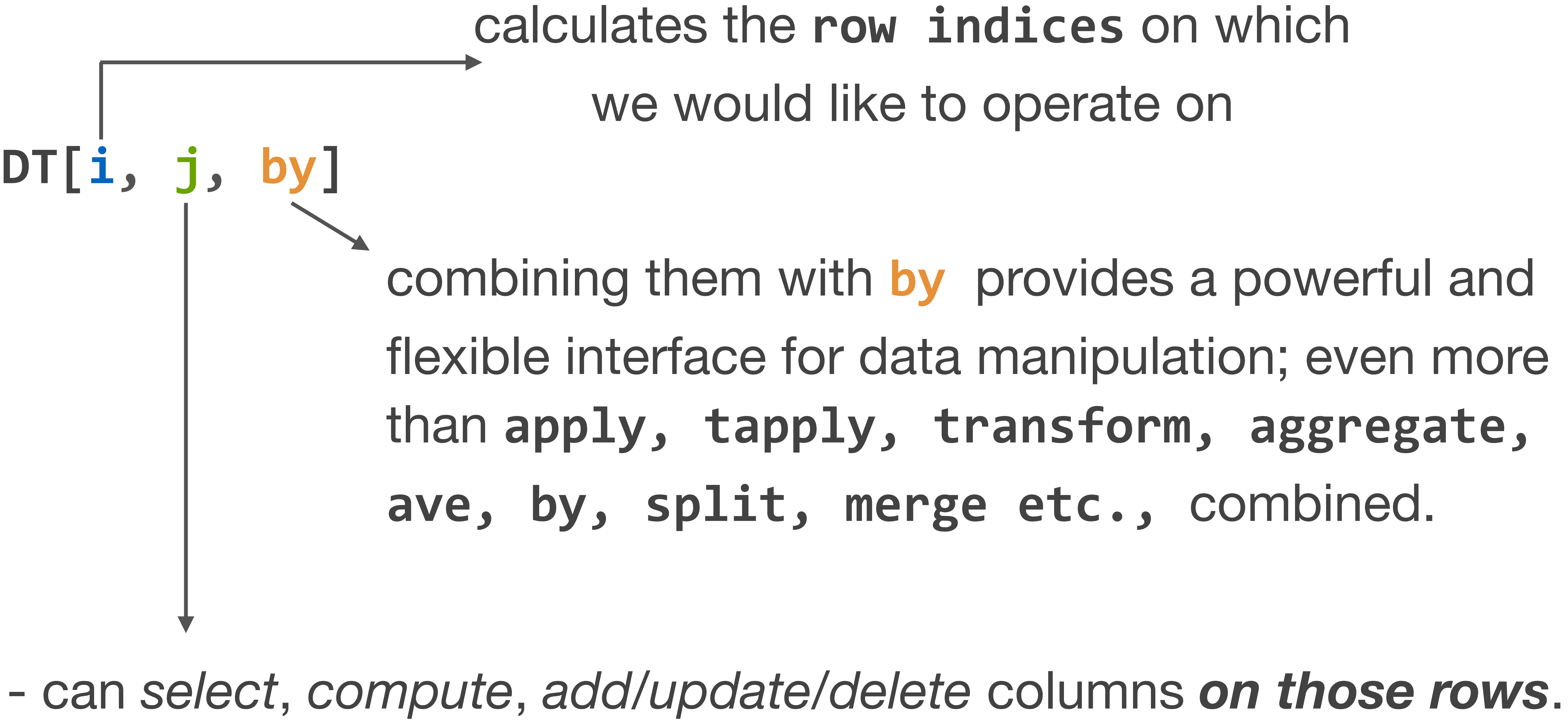
`[.data.table` on the contrary, is quite feature packed

The general form is: **`DT[i, j, by]`** `# + ...`

Take **DT**, subset rows using **i**, then calculate **j**, grouped by **by**

<table>
<tr><td><b>R:</b></td><td><b>i</b></td><td><b>j</b></td><td><b>by</b></td></tr>
<tr><td><b>SQL:</b></td><td><b>WHERE</b></td><td><b>SELECT|COMPUTE</b></td><td><b>GROUP BY</b></td></tr>
</table>

Continued ...



`DT[i, j, by]`

calculates the **`row indices`** on which we would like to operate on

combining them with **`by`** provides a powerful and flexible interface for data manipulation; even more than **`apply, tapply, transform, aggregate, ave, by, split, merge etc.,`** combined.

- can *select*, *compute*, *add/update/delete* columns ***on those rows***.

Overview of today’s talk

Reading

- fread

Cleaning

- melt, dcast

Analysing

- subsets (automatic indexing)
- ordering (fast radix ordering, setorder)
- Aggregations and updates
- Interval joins (foverlaps)

Reading

50 MB CSV file, 1 million rows x 6 columns

Command	Run time
read.csv(“test.csv”)	30-60s
read.csv(“test.csv”, colClasses=, nrows=, …)	10s
fread(“test.csv”)	3s

Reading

20 GB CSV file, 200 million rows x 16 columns

Command	Run time
`read.csv(“big.csv”, colClasses=, nrows=, …)`	hours
`fread(“big.csv”)`	8m

Cleaning

Consider this sample data:

dad	mom	child1_sex	child2_sex	child3_sex	child1_age	child2_age	child3_age
David	Angela	M	F	NA	8	12	NA
Aaron	Anita	F	NA	NA	7	NA	NA
Michael	Katya	F	F	M	5	7	15

Cleaning

How can we clean this data to get to this?

dad	mom	child	sex	age
David	Angela	child1	M	8
Aaron	Anita	child1	F	7
Michael	Katya	child1	F	5
David	Angela	child2	F	12
Aaron	Anita	child2	NA	NA
Michael	Katya	child2	F	7
David	Angela	child3	NA	NA
Aaron	Anita	child3	NA	NA
Michael	Katya	child3	M	15

Cleaning

```
# old (and convoluted) way:
DT.m = melt(DT, id = 1:2)
DT.m[, child := gsub("_.*$", "", variable)]
DT.m[, variable := gsub(".*_", "", variable)]
dcast(DT.m, dad + mom + child ~ variable, value.var = "value")
# WHY ARE WE COMBINING ALL COLUMNS TOGETHER HERE ONLY TO SPLIT THEM AGAIN?
# This is both not straightforward and extremely inefficient!!
# melt should be able to combine multiple columns together
# (v1.9.5 does it right)
vars = lapply(c("sex$", "age$"), grep, names(DT), value=TRUE)
DT.m1 = melt(DT, measure = vars, variable.name = "child",
                value.name = c("sex", "age"))
setattr(DT.m1$child, 'levels', gsub("_.*$", "", vars[[1L]]))
DT.m1

# use na.rm=TRUE directly
```

Illustration

tidyr vs data.table benchmark

Subsets

How can we get all rows where **`child == “child1”`**?

DT.m1[child == “child1”]

dad	mom	child	sex	age
David	Angela	child1	M	8
Aaron	Anita	child1	F	7
Michael	Katya	child1	F	5
David	Angela	child2	F	12
Aaron	Anita	child2	NA	NA
Michael	Katya	child2	F	7
David	Angela	child3	NA	NA
Aaron	Anita	child3	NA	NA
Michael	Katya	child3	M	15

Automatic indexing

Build indices automatically on the first run

Allows for *fast binary search* based subsets on subsequent runs

Possible because of *fast radix ordering* in data.tables

Illustration

Ordering

data.table implements *fast radix ordering* for **integers**, **doubles** and **characters**

`DT[order(…)]` is optimised to use internal fast radix ordering

`setorder()` is even more memory efficient way to reorder data.tables (and **also data.frames since 1.9.5+**)

Illustration

setorder() benchmark

Aggregations

How many kids do each family have?

```
DT.m1[!is.na(sex), .N, by = .(dad, mom)]
```

Note: The entire subset is not materialised here after computing expression in ‘i’

If we already removed `NAs` using `na.rm=TRUE` argument in `melt`, then

```
DT.m2[, .N, by = .(dad, mom)]
```

Aggregations

Get the sex of oldest kid for each family

```
DT.m2[, sex[which.max(age)], by = .(dad, mom)]
```

Name the result column as ‘oldest_sex’

```
DT.m2[, .(oldest_sex = sex[which.max(age)]), by = .(dad, mom)]
```

Add / update column

Add a new column with the sex of oldest child for each family

```
DT.m2[, oldest_sex := sex[which.max(age)], by = .(dad, mom)]
```

`:=` takes a character vector of column names (or indices) on the left and a list of values on the right.

It updates the data.table *by reference* (in-place). We don't need to do **`DT.m <- …`**

When LHS contains only one column the “” is optional (for convenience). Similarly RHS need not be wrapped with **`list()`**.

Overlapping range joins - new feature

Which ranges from **Query** overlap with **Subject**?

start	end
12	15
41	50
7	9
33	34

Query

start	end
10	16
20	35
30	45

Subject

→

start	end
10	16
NA	NA
NA	NA
20	35
30	45

```
foverlaps(query, subject, type=”within”)
```