Overview

- data.table in a nutshell (10 mins)
- Q & A. Our backgrounds (10 mins)
- Main features in more detail (30 mins)
- Q & A (10 mins)

**Every question is a good question!**

- Please complete feedback form at the end of the conference
What is data.table?

- Think `data.frame`, inherits from it
- `data.table()` and `?data.table`

Goals:

- Reduce programming time
  fewer function calls, less variable name repetition
- Reduce compute time
  fast aggregation, update by reference
- In-memory only, 64bit and 8GB+ routine
- Useful in finance but wider use in mind, too
  e.g. genomics
Reducing programming time

```r
trades[
  filledShares < orderedShares,
  sum( (orderedShares - filledShares) * orderPrice / fx ),
  by = "date,region,algo"
]
```

R :  i  j  by
SQL :  WHERE  SELECT  GROUP BY
Reducing compute time

e.g. 10 million rows x 3 columns x,y,v 230MB

DF[DF$x=="R" & DF$y==123,]  # 8 s
DT[.("R",123)]  # 0.008s

tapply(DF$v,DF$x,sum)  # 22 s
DT[,sum(v),by=x]  # 0.83s

See above in timings vignette (copy and paste)
Fast and friendly file reading

e.g. 50MB .csv, 1 million rows x 6 columns

```
read.csv("test.csv")  # 30-60s
read.csv("test.csv", colClasses=, nrows=, etc...)  # 10s
fread("test.csv")  # 3s
```

e.g. 20GB .csv, 200 million rows x 16 columns

```
read.csv("big.csv", ...)  # hours
fread("big.csv")  # 8m
```
Update by reference using :=

Add new column "sectorMCAP" by group:

```
DT[, sectorMCAP := sum(MCAP), by = Sector]
```

Delete a column (0.00s even on 20GB table):

```
DT[, colToDelete := NULL]
```

Be explicit to really copy entire 20GB:

```
DT2 = copy(DT)
```
Why R?

1) R's lazy evaluation enables the syntax:
   - `DT[ filledShares < orderedShares ]`
   - query optimization before evaluation

2) Pass `DT` to any package taking `DF`. It works.
   ```r
   is.data.frame(DT) == TRUE
   ```

3) CRAN (cross platform release, quality control)

4) Thousands of statistical packages to use with data.table
Q & A

- My background

- Your background; e.g.
  - Bank, asset management, other?
  - Research, trading, risk, all, other?
  - Equity, futures, other?
  - Low frequency, high frequency?
  - How long using R, SQL, data.table?
  - Question?
Given a 10,000 x 10,000 matrix in any language
Sum the rows
Sum the columns
Is one way faster, and why?
setkey(DT, colA, colB)

- Sorts the table by colA then colB. That's all.
- Like a telephone number directory: last name then first name
- X[Y] is just binary search to X's key
- You **DO** need a key for joins X[Y]
- You **DO NOT** need a key for by= (but many examples online include it)
Joins: X[Y]

- Vector search vs binary search
- One column == is ok, but not 2+ (see example above)
- J(), .(), list(), data.table()
- CJ()
- SJ()
- nomatch
- mult
"Cold" by (i.e. without setkey)

Consecutive calls unrelated to key are fine and common practice:

```r
> DT[, sum(v), by="x,y"]
> DT[, sum(v), by="z"]
> DT[, sum(v), by=colA%%5]
```

Also known as "ad hoc by"
DT[i, j, by]

- Out loud: "Take DT, subset rows using i, then calculate j grouped by by"

- Once you grok the above reading, you don't need to memorize any other functions as all operations follow the same intuition as base.
I have a data frame that is some 35,000 rows, by 7 columns. It looks like:

```
head(nuc)
```

<table>
<thead>
<tr>
<th>chr</th>
<th>feature</th>
<th>start</th>
<th>end</th>
<th>gene_id</th>
<th>pctAT</th>
<th>pctGC</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CDS</td>
<td>67000042</td>
<td>67000051</td>
<td>NM_032291</td>
<td>0.60000</td>
<td>0.40000</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>CDS</td>
<td>67091530</td>
<td>67091593</td>
<td>NM_032291</td>
<td>0.60937</td>
<td>0.39062</td>
<td>64</td>
</tr>
<tr>
<td>3</td>
<td>CDS</td>
<td>67098753</td>
<td>67098777</td>
<td>NM_032291</td>
<td>0.60000</td>
<td>0.40000</td>
<td>25</td>
</tr>
<tr>
<td>4</td>
<td>CDS</td>
<td>67101627</td>
<td>67101698</td>
<td>NM_032291</td>
<td>0.47222</td>
<td>0.52778</td>
<td>72</td>
</tr>
<tr>
<td>5</td>
<td>CDS</td>
<td>67105460</td>
<td>67105516</td>
<td>NM_032291</td>
<td>0.63157</td>
<td>0.36842</td>
<td>57</td>
</tr>
<tr>
<td>6</td>
<td>CDS</td>
<td>67108493</td>
<td>67108547</td>
<td>NM_032291</td>
<td>0.43636</td>
<td>0.56363</td>
<td>55</td>
</tr>
</tbody>
</table>

gene_id is a factor, that has about 3,500 unique levels. I want to, for each level of gene_id get the
```
min(start), max(end), mean(pctAT), mean(pctGC), and sum(length).
```

I tried using `lapply` and `do.call` for this, but it's taking forever +30 minutes to run. The code I'm using is:

```
nuc_prof = lapply(levels(nuc$gene_id), function(gene){
  t = nuc[nuc$gene_id==gene, ]
  return(list(gene_id=gene, start=min(t$start), end=max(t$end), pctGC = mean(t$pctGC), pct = mean(t$pctAT), cdslength = sum(t$length)))
})
nuc_prof = do.call(rbind, nuc_prof)
```

I'm certain I'm doing something wrong to slow this down. I haven't waited for it to finish as I'm sure it can be faster. Any ideas?
data.table answer

Since I'm in an evangelizing mood ... here's what the fast data.table solution would look like:

```r
library(data.table)
dt <- data.table(nuc, key="gene_id")

dt[,list(A=min(start),
        B=max(end),
        C=mean(pctAT),
        D=mean(pctGC),
        E=sum(length)), by=key(dt)]
```

```
# gene_id   A    B      C        D      E
# 1: NM_032291 67000042 67108547 0.5582567 0.4417433 283
# 2: ZZZ 67000042 67108547 0.5582567 0.4417433 283
```

NB: It isn't just the speed, but the simplicity. It's easy to write and easy to read.
"Holy fudge buckets!!! data.table is awesome! That took about 3 seconds for the whole thing!!!"

"I think that congratulations are well in order for the frankly amazingly well written quick start guide and FAQ. Seriously."

Davy Kavanagh, 15 Jun 2012
but ...

- Example had `by=key(dt)`?

- Yes, but it didn't need to.

- If the data is very large (1GB+) and the groups are big too then getting the groups together in memory can speed up a bit (cache efficiency).
DT[,,by=] -vs- DT[,,keyby=]

- **by** preserves order of groups (by order of first appearance)

- Both preserve order of rows within groups (important!) and unlike SQL

- **keyby** is a **by** as usual, followed by **setkeyv(DT,by)**
Prevailing join (roll=TRUE)

- One reason for setkey's design.
- Last Observation (the prevailing one) Carried Forward (LOCF), efficiently
- Roll forwards or backward
- Roll the last observation forwards, or not
- Roll the first observation backwards, or not
- Limit the roll; e.g. 30 days (roll = 30)
- Join to nearest value (roll = ”nearest”)
- i.e. ordered joins
Variable name repetition

- The 3rd highest voted [R] question (of 43k)
  How to sort a dataframe by column(s) in R (*)

- DF[with(DF, order(-z, b)), ]
  - VS -
  DT[ order(-z, b) ]

- quarterlyreport[with(lastquarterlyreport,order(-z,b))), ]
  - VS -
  quarterlyreport[ order(-z, b) ]

(*) Click link for more information
but ...

- Yes order() is slow when used in i because that's base R's order().
- That's where "optimization before evaluation" comes in. We now auto convert order() to the internal forder() so you don't have to know.
- Available in v1.9.3 on R-Forge, soon on CRAN
Why "split" 10GB into many small groups???

Since 2010, data.table:
- Allocates memory for largest group
- Reuses that same memory for all groups
- Allocates result data.table up front
- Implemented in C
- `eval()` of `j` within each group
Recent innovations

- Instead of the eval(j) from C, dplyr converts to an Rcpp function and calls that from C. Skipping the R eval step.

- In response, data.table now has **GForce**: one function call that computes the aggregate across groups. Called once only so no need to speed up many calls!

- Both approaches limited to simple aggregates: sum, mean, sd, etc. But often that's all that's needed.
data.table over-allocates

data.frame  |  data.table

cbind       |  DT[,newcol:=1]
\(:= \text{ and } `:=` () \)

\[
\text{DT[} \text{col1}==\text{something}, \text{ col2}:=\text{col3}+1]\]

\[
\text{DT[}, \ `:=` `(\text{newCol1}=\text{mean}(\text{colA}),\\n\hspace{1em}\hspace{1em}\hspace{1em}\hspace{1em}\text{newCol2}=\text{sd}(\text{colA}))$,\\n\hspace{1em}\hspace{1em}\hspace{1em}\hspace{1em}\text{by}=\text{sector}]\\n\]
set* functions

- `set()`
- `setattr()`
- `setnames()`
- `setcolorder()`
- `setkey()`
- `setkeyv()`
All options

datatable.verbose                   FALSE

datatable.nomatch             NA_integer_

datatable.optimize                    Inf

datatable.print.nrows                100L

datatable.print.topn                   5L

datatable.allow.cartesian           FALSE

datatable.alloccol
quote(max(100L,ncol(DT)+64L))

datatable.integer64  "integer64"
All symbols

- `.N`
- `.SD`
- `.I`
- `.BY`
- `.GRP`
stocks[, head(.SD,2), by=sector]

stocks[, lapply(.SD, sum), by=sector]

stocks[, lapply(.SD, sum), by=sector, .SDcols=c("mcap",paste0(revenueFQ",1:8)))]
if (length(err <- allocation[, ,
    if (length(unique(Price)) > 1) .I,
    by = stock ]$V1 )) {

    warning("Fills allocated to different accounts at different prices! Investigate.")

    print(allocation[err])

} else {

    cat("Ok All fills allocated to each account at same price\n")

}
Analogous to SQL

```
DT[ where,
    select | update,
    group by ]
[ having ]
[ order by ]
[ i, j, by ] ... [ i, j, by ]
```
37 new features and 43 bug fixes
- set() can now add columns just like :=
- .SDcols “de-select” columns by name or position; e.g.,
  \[DT[, lapply(.SD, mean), by=colA, .SDcols=-c(3,4)]\]
- fread() a subset of columns
- fread() commands; e.g.,
  fread("grep blah file.txt")
- Speed gains
Radix sort for integer

- R's method="radix" is not actually a radix sort ... it's a counting sort. See ?setkey/Notes.
- data.table liked and used it, though.
- A true radix sort caters for range > 100,000
- ( Negatives was a one line change to R we suggested and was accepted in R 3.1 )
- Adapted to integer from Terdiman and Herf's code for float ...
Radix sort for numeric

- R reminder: numeric == floating point numbers
- Radix Sort Revisited, Pierre Terdiman, 2000
  http://codercorner.com/RadixSortRevisited.htm
- Radix Tricks, Michael Herf, 2001
  http://stereopsis.com/radix.html
- Their C code now in data.table with minor changes; e.g., NA/NaN and 6-pass for double
Faster for those cases

20 million rows x 4 columns, 539MB
a & b (numeric), c (integer), d (character)

<table>
<thead>
<tr>
<th></th>
<th>v1.8.10</th>
<th>v1.8.11</th>
</tr>
</thead>
<tbody>
<tr>
<td>setkey(DT, a)</td>
<td>54.9s</td>
<td>7.2s</td>
</tr>
<tr>
<td>setkey(DT, c)</td>
<td>48.0s</td>
<td>7.0s</td>
</tr>
<tr>
<td>setkey(DT, a, b)</td>
<td>102.3s</td>
<td>16.9s</td>
</tr>
</tbody>
</table>

"Cold" grouping (no setkey first):

DT[, mean(b), by=c] 47.0s 8.7s

https://gist.github.com/arunsrinivasan/7997273
New feature: melt/cast

i.e. reshape2 for data.table

20 million rows x 6 columns (a:f)  768MB

\[
\text{melt(DF, id="d", measure=1:2)} \quad 191 \text{ sec}
\]

\[
\text{melt(DT, id="d", measure=1:2)} \quad 3 \text{ sec}
\]

\[
\text{dcast(DF, d~e, ..., fun=sum)} \quad 184 \text{ sec}
\]

\[
\text{dcast(DT, d~e, ..., fun=sum)} \quad 28 \text{ sec}
\]

https://gist.github.com/arunsrinivasan/7839891

Similar to melt_ in Kmisc by Kevin Ushey
Q: Why not submit a pull request to reshape2?

A: This C implementation calls data.table internals at C-level (e.g. fastorder, grouping, and joins). It makes sense for this code to be together.
DT[, (myvar):=NULL]

Space and specials; e.g., by="a, b, c"

DT[4:7, newCol:=8][[]
  ▪ extra [] to print at prompt
  ▪ auto fills rows 1:3 with NA
53 examples in:

example(data.table)
Thank you

http://datatable.r-forge.r-project.org/
http://stackoverflow.com/questions/tagged/data.table

> install.packages("data.table")
> require(data.table)
> ?data.table
> ?fread

Learn by example:
> example(data.table)