Introduction to dplyr and magrittr

Denver R Users Group
www.meetup.com/DenverRUG

Peter DeWitt
peter.dewitt@ucdenver.edu

1 July 2014

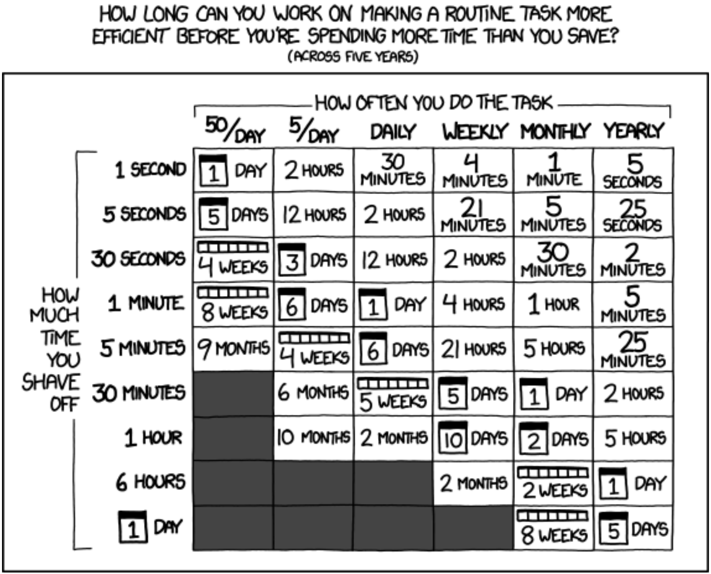
Goals:

- Showcase `dplyr`, compare the ease of use and speed to base R.
- Introduce the data manipulation grammar and philosophy behind `dplyr`
- Illustrate the usefulness of the forward-piping operator which is part of `dplyr` and extended further in `magrittr`.

- Convey: `dplyr` will save time in initial coding, debugging, code maintenance, . . .

Is it Worth the Time?

http://xkcd.com/1205/



`dplyr`: a grammar of data manipulation

- Authored by Hadley Wickham and Romain Francois
- Current CRAN version 0.2

dplyr: a grammar of data manipulation

- Authored by Hadley Wickham and Romain Francois
- Current CRAN version 0.2
- Paraphrasing from a post on the RStudio blog `http://blog.rstudio.org/2014/01/17/introducing-dplyr`
  - `dplyr` is the next iteration of `plyr`
  - focuses only on `data.frames`
  - faster, thanks in part to Francois work in `Rcpp`, some use of multiple processors.
  - improved API.
  - interface with remote database (PostgreSQL, MySQL, SQLite, and Google bigquery) tables using the same verbs for interacting with `data.frames`. (Extendible to other backends)
  - Common operations:
    - `group_by`, `summarize`, `mutate`, `filter`, `select`, and `arrange`.

Data Import

`dplyr` does not have special tools for reading in data, but, if you need to `rbind` sets together...

```
# FAAs wildlife strikes on aircraft since 1990.  The data
# can be downloaded, in a Microsoft Access DB,  from
# http://www.faa.gov/airports/airport_safety/wildlife/database/
# Tables in the DB were exported to csv files.
# A data dictionary, in an Excel file, was also
# included in the download from faa.gov

# column classes are set (in R code not shown) to ensure
# that each column of the imported data is of the same class
wls.90.99 <- read.csv("../data/STRIKE_REPORTS (1990-1999).csv",
                      colClasses = clclss)
wls.00.09 <- read.csv("../data/STRIKE_REPORTS (2000-2009).csv",
                      colClasses = clclss)
wls.10.14 <- read.csv("../data/STRIKE_REPORTS (2010-Current).csv",
                      colClasses = clclss)
```

Data Import

```
dim(wls.90.99)

## [1] 30150    94

nrow(wls.90.99) + nrow(wls.00.09) + nrow(wls.10.14)

## [1] 142911

bnchmrk <-
  benchmark(base  = rbind(wls.90.99, wls.00.09, wls.10.14),
            dplyr = rbind_list(wls.90.99, wls.00.09, wls.10.14),
            replications = 100)
bnchmrk[, c("test", "replications", "elapsed", "relative")]

##    test replications elapsed relative
## 1  base          100  113.09    4.625
## 2 dplyr          100   24.45    1.000
```

Data Import

```
wls_df <- rbind(wls.90.99, wls.00.09, wls.10.14)
class(wls_df)

## [1] "data.frame"

wls <- rbind_list(wls.90.99, wls.00.09, wls.10.14)
class(wls)

## [1] "data.frame"

# A data frame tbl wraps a local data frame. The main
# advantage to using a tbl_df over a regular data frame is
# the printing: tbl objects only print a few rows and all
# the columns that fit on one screen, providing describing
# the rest of it as text. [source: R help doc]
wls_tbl_df <- tbl_df(wls)
class(wls_tbl_df)

## [1] "tbl_df"     "tbl"         "data.frame"
```

Data Printing

```
# print(wls_df)  # takes a long time, not helpful
# head(wls_df)   # two many columns to be useful
print(wls_tbl_df, n = 3)

## Source: local data frame [142,911 x 94]
##
##    INDEX_NR OPID          OPERATOR     ATYPE AMA AMO EMA EMO
## 1    100000  AAL AMERICAN AIRLINES     B-727 148  10  34  10
## 2    100001  UAL   UNITED AIRLINES B-737-300 148  24  10  01
## 3    100002  UAL   UNITED AIRLINES B-737-300 148  24  10  01
## ..      ...  ...               ...       ... ... ... ... ...
## Variables not shown: AC_CLASS (chr), AC_MASS (int), NUM_ENGS
##   (chr), TYPE_ENG (chr), ENG_1_POS (chr), ENG_2_POS (int),
##   ENG_3_POS (chr), ENG_4_POS (int), REG (chr), FLT (chr),
##   REMAINS_COLLECTED (lgl), REMAINS_SENT (lgl), INCIDENT_DATE
##   (chr), INCIDENT_MONTH (int), INCIDENT_YEAR (int),
##   TIME_OF_DAY (chr), TIME (int), AIRPORT_ID (chr), AIRPORT
##   (chr), STATE (chr), FAAREGION (chr), ENROUTE (chr), RUNWAY
##   (chr), LOCATION (chr), HEIGHT (int), SPEED (int), DISTANCE
##   (dbl), PHASE_OF_FLT (chr), DAMAGE (chr), STR_RAD (lgl),
##   DAM_RAD (lgl), STR_WINDSHLD (lgl), DAM_WINDSHLD (lgl),
##   STR_NOSE (lgl), DAM_NOSE (lgl), STR_ENG1 (lgl), DAM_ENG1
```

The verbs

- "Variable and function names should be lowercase. Use an underscore (_) to separate words within a name. Generally, variable names should be nouns and function names should be verbs. Strive for names that are concise and meaningful (this is not easy!)." - Hadley Wickham, `http://adv-r.had.co.nz/Style.html`
- Verbs in `dplyr`
  - `select`,
  - `arrange`,
  - `filter`,
  - `mutate`,
  - `summarize`.

select

```
# Select columns of a data.frame, tbl_df.
wls_yr <- select(wls_tbl_df, INCIDENT_YEAR, AIRPORT,
                 ENG_1_POS, ENG_2_POS, DAM_ENG1, DAM_ENG2,
                 HEIGHT, DISTANCE, SPEED)
print(wls_yr, n = 5)

## Source: local data frame [142,911 x 9]
##
##    INCIDENT_YEAR                    AIRPORT ENG_1_POS
## 1           1992 DALLAS/FORT WORTH INTL ARPT         5
## 2           1996            SACRAMENTO INTL         1
## 3           1996        DENVER INTL AIRPORT         1
## 4           1996            EPPLEY AIRFIELD         1
## 5           1996 WASHINGTON DULLES INTL ARPT         1
## ..           ...                         ...       ...
## Variables not shown: ENG_2_POS (int), DAM_ENG1 (lgl),
##   DAM_ENG2 (lgl), HEIGHT (int), DISTANCE (dbl), SPEED (int)
```

select

```
# relative speed betwuen dplyr and base R
bnch <-
  benchmark(base  = wls_tbl_df[, c("INCIDENT_YEAR", "AIRPORT",
                                   "ENG_1_POS", "ENG_2_POS",
                                   "DAM_ENG1", "DAM_ENG2",
                                   "HEIGHT", "DISTANCE", "SPEED")],
            dplyr = select(wls_tbl_df,
                           INCIDENT_YEAR, AIRPORT,
                           ENG_1_POS, ENG_2_POS,
                           DAM_ENG1, DAM_ENG2,
                           HEIGHT, DISTANCE, SPEED),
            replications = 100)
select(bnch, test, replications, elapsed, relative)

##    test replications elapsed relative
## 1  base          100   0.007        1
## 2 dplyr          100   0.028        4
```

Selection of columns might be slower in dplyr, but, there are some tools to help speed up the coding, and maintenance. `select` will be very helpful when chaining together many operations or when using *super cool helper functions*.

select

```
# num_range("x", 1:5, width = 2): selects all variables
# (numerically) from x01 to x05.
select(wls_tbl_df, num_range("DAM_ENG", 1:4))

## Source: local data frame [142,911 x 4]
##
##    DAM_ENG1 DAM_ENG2 DAM_ENG3 DAM_ENG4
## 1     FALSE    FALSE    FALSE    FALSE
## 2     FALSE    FALSE    FALSE    FALSE
## 3     FALSE    FALSE    FALSE    FALSE
## 4     FALSE    FALSE    FALSE    FALSE
## 5     FALSE    FALSE    FALSE    FALSE
## 6     FALSE    FALSE    FALSE    FALSE
## 7     FALSE    FALSE    FALSE    FALSE
## 8     FALSE    FALSE    FALSE    FALSE
## 9     FALSE    FALSE    FALSE    FALSE
## 10    FALSE    FALSE    FALSE    FALSE
## ..      ...      ...      ...      ...
```

select

```
# starts_with(x, ignore.case = FALSE): names starts with x
select(wls_tbl_df, starts_with("DAM"))

## Source: local data frame [142,911 x 15]
##
##    DAMAGE DAM_RAD DAM_WINDSHLD DAM_NOSE DAM_ENG1 DAM_ENG2
## 1       N   FALSE        FALSE    FALSE    FALSE    FALSE
## 2           FALSE        FALSE    FALSE    FALSE    FALSE
## 3           FALSE        FALSE    FALSE    FALSE    FALSE
## 4       N   FALSE        FALSE    FALSE    FALSE    FALSE
## 5       N   FALSE        FALSE    FALSE    FALSE    FALSE
## 6       M   FALSE        FALSE    FALSE    FALSE    FALSE
## 7       N   FALSE        FALSE    FALSE    FALSE    FALSE
## 8      M?   FALSE        FALSE    FALSE    FALSE    FALSE
## 9       N   FALSE        FALSE    FALSE    FALSE    FALSE
## 10          FALSE        FALSE    FALSE    FALSE    FALSE
## ..    ...     ...          ...      ...      ...      ...
## Variables not shown: DAM_ENG3 (lgl), DAM_ENG4 (lgl), DAM_PROP
##   (lgl), DAM_WING_ROT (lgl), DAM_FUSE (lgl), DAM_LG (lgl),
##   DAM_TAIL (lgl), DAM_LGHTS (lgl), DAM_OTHER (lgl)
```

select

```r
# ends_with(x, ignore.case = FALSE): names ends in x
select(wls_tbl_df, ends_with("4"))

## Source: local data frame [142,911 x 2]
##
##    STR_ENG4 DAM_ENG4
## 1     FALSE    FALSE
## 2     FALSE    FALSE
## 3     FALSE    FALSE
## 4     FALSE    FALSE
## 5     FALSE    FALSE
## 6     FALSE    FALSE
## 7     FALSE    FALSE
## 8     FALSE    FALSE
## 9     FALSE    FALSE
## 10    FALSE    FALSE
## ..      ...      ...
```

select

```
# matches(x, ignore.case = FALSE): selects all variables
# whose name matches the regular expression x
select(wls_tbl_df, matches("ENG|DAM"))

## Source: local data frame [142,911 x 26]
##
##    NUM_ENGS TYPE_ENG ENG_1_POS ENG_2_POS ENG_3_POS ENG_4_POS
## 1         3        D         5         6         5        NA
## 2         2        D         1         1                  NA
## 3         2        D         1         1                  NA
## 4         2        D         1         1                  NA
## 5         2        D         1         1                  NA
## 6         2        D         1         1                  NA
## 7         3        D         5         6         5        NA
## 8         2        C         4         4                  NA
## 9         2        D         1         1                  NA
## 10        2        D         5         5                  NA
## ..      ...      ...       ...       ...       ...       ...
## Variables not shown: DAMAGE (chr), DAM_RAD (lgl),
##   DAM_WINDSHLD (lgl), DAM_NOSE (lgl), STR_ENG1 (lgl),
##   DAM_ENG1 (lgl), STR_ENG2 (lgl), DAM_ENG2 (lgl), STR_ENG3
##   (lgl), DAM_ENG3 (lgl), STR_ENG4 (lgl), DAM_ENG4 (lgl),
```

select

```
# contains(x, ignore.case = FALSE): selects all
# variables whose name  contains x
select(wls_tbl_df, contains("ENG"))

## Source: local data frame [142,911 x 14]
##
##    NUM_ENGS TYPE_ENG ENG_1_POS ENG_2_POS ENG_3_POS ENG_4_POS
## 1         3        D         5         6         5        NA
## 2         2        D         1         1                  NA
## 3         2        D         1         1                  NA
## 4         2        D         1         1                  NA
## 5         2        D         1         1                  NA
## 6         2        D         1         1                  NA
## 7         3        D         5         6         5        NA
## 8         2        C         4         4                  NA
## 9         2        D         1         1                  NA
## 10        2        D         5         5                  NA
## ..      ...      ...       ...       ...       ...       ...
## Variables not shown: STR_ENG1 (lgl), DAM_ENG1 (lgl), STR_ENG2
##   (lgl), DAM_ENG2 (lgl), STR_ENG3 (lgl), DAM_ENG3 (lgl),
##   STR_ENG4 (lgl), DAM_ENG4 (lgl)
```

select

What about dropping variables?

```
print(wls_yr, n = 2)

## Source: local data frame [142,911 x 9]
##
##    INCIDENT_YEAR                     AIRPORT ENG_1_POS
## 1           1992 DALLAS/FORT WORTH INTL ARPT         5
## 2           1996             SACRAMENTO INTL         1
## ..           ...                         ...       ...
## Variables not shown: ENG_2_POS (int), DAM_ENG1 (lgl),
##   DAM_ENG2 (lgl), HEIGHT (int), DISTANCE (dbl), SPEED (int)

# omit AIRPORT and any column with a name starting with ENG.
print(select(wls_yr, -AIRPORT, -starts_with("ENG")), n = 3)

## Source: local data frame [142,911 x 6]
##
##    INCIDENT_YEAR DAM_ENG1 DAM_ENG2 HEIGHT DISTANCE SPEED
## 1           1992    FALSE    FALSE    300       NA   142
## 2           1996    FALSE    FALSE      0        0    NA
## 3           1996    FALSE    FALSE      0        0    NA
##
```

arrange

arrange: reorder the rows. Multiple inputs are ordered from left-to-right.

```
dat <- data.frame(var1 = c(3, 8, 2, 1),
                  var2 = c("E", "A", "A", "B"))
dat
```

```
##   var1 var2
## 1    3    E
## 2    8    A
## 3    2    A
## 4    1    B
```

arrange

```
arrange(dat, var2)

##   var1 var2
## 1    8    A
## 2    2    A
## 3    1    B
## 4    3    E

arrange(dat, var2, var1)

##   var1 var2
## 1    2    A
## 2    8    A
## 3    1    B
## 4    3    E

# this would be very helpful for collecting data by a
# subject id, visit number, ...
#
# Very helpful for geepack::geeglm() where the data *must*
# be sorted by cluster and in temporal order as well.
```

filter

filter: return only a subset of the rows. If multiple conditions are supplied they are combined with &.

```
dim(wls_yr)

## [1] 142911      9

filter(wls_yr, INCIDENT_YEAR > 2000, INCIDENT_YEAR <= 2005)

## Source: local data frame [31,947 x 9]
##
##    INCIDENT_YEAR                      AIRPORT ENG_1_POS
## 1           2001         JOHN F KENNEDY INTL
## 2           2001     SAN FRANCISCO INTL ARPT         1
## 3           2001                ORLANDO INTL         1
## 4           2001                MOLOKAI ARPT         4
## 5           2001          LAMBERT-ST LOUIS INTL         5
## 6           2001           KANSAS CITY INTL         1
## 7           2001                     UNKNOWN         1
## 8           2001       AKRON-CANTON REGIONAL         7
## 9           2001 DESTIN-FORT WALTON BEACH ARPT         5
## 10          2001         JOHN F KENNEDY INTL         1
##
```

filter

```r
# How does dplyr::filter compare to base::subset?
bnch <-
  benchmark(base = subset(wls_yr,
                          INCIDENT_YEAR > 2000 & INCIDENT_YEAR <= 2005),
            dplyr = filter(wls_yr,
                           INCIDENT_YEAR > 2000, INCIDENT_YEAR <= 2005),
            replications = 100)
select(bnch, test, replications, elapsed, relative)

##    test replications elapsed relative
## 1  base          100   7.977    5.364
## 2 dplyr          100   1.487    1.000
```

mutate

mutate: add new columns. Multiple inputs create multiple columns.

```
eng.lbls <- c("mounted below the wing", "mounted above the wing",
              "part of the wing root", "nacelle-mounted on the wing",
              "mounted on the aft fuselage")
str(mutate(wls_yr,
       SPEED_MPH = SPEED * 1.15078, # SPEED was in knots
       ENG_1_POS = factor(ENG_1_POS, 19:23, eng.lbls),
       ENG_2_POS = factor(ENG_2_POS, 19:23, eng.lbls)))
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame': 142911 obs. of  10 variables:
##  $ INCIDENT_YEAR: int  1992 1996 1996 1996 1996 1996 1991 1993 1995 1990
##  $ AIRPORT      : chr  "DALLAS/FORT WORTH INTL ARPT" "SACRAMENTO INTL" "D
##  $ ENG_1_POS    : Factor w/ 5 levels "mounted below the wing",..: NA NA N
##  $ ENG_2_POS    : Factor w/ 5 levels "mounted below the wing",..: NA NA N
##  $ DAM_ENG1     : logi  FALSE FALSE FALSE FALSE FALSE FALSE ...
##  $ DAM_ENG2     : logi  FALSE FALSE FALSE FALSE FALSE FALSE ...
##  $ HEIGHT       : int  300 0 0 0 1000 5000 0 1500 0 100 ...
##  $ DISTANCE     : num  NA 0 0 0 NA NA 0 NA 0 NA ...
##  $ SPEED        : int  142 NA NA NA NA NA 100 220 NA 135 ...
##  $ SPEED_MPH    : num  163 NA NA NA NA ...
```

mutate

```
# dplyr::mutate is conceptually similar to base::within
bnch <-
benchmark(base = within(wls_yr, {
                        SPEED_MPH = SPEED * 1.15078
                        ENG_1_POS = factor(ENG_1_POS, 19:23, eng.lbls)
                        ENG_2_POS = factor(ENG_2_POS, 19:23, eng.lbls)}),
          dplyr = mutate(wls_yr,
                         SPEED_MPH = SPEED * 1.15078,
                         ENG_1_POS = factor(ENG_1_POS, 19:23, eng.lbls),
                         ENG_2_POS = factor(ENG_2_POS, 19:23, eng.lbls)),
          replications = 100)
select(bnch, test, replications, elapsed, relative)

##    test replications elapsed relative
## 1  base          100   5.958    1.000
## 2 dplyr          100   5.979    1.004
```

summarize

summarise: reduce each group to a single row. Multiple inputs create multiple output summaries. (Two spellings: summarize and summarise.)

```
summarise(wls_yr,
          "Mean speed" = mean(SPEED, na.rm = TRUE),
          "SD speed"   = sd(SPEED, na.rm = TRUE),
          n            = sum(!is.na(SPEED)),
          records      = n(),
          airports     = n_distinct(AIRPORT))

## Source: local data frame [1 x 5]
##
##   Mean speed SD speed     n records airports
## 1      141.3    46.09 58938  142911     2093
```

group_by

```
summarise(group_by(wls_yr, ENG_1_POS),
          "Mean speed" = mean(SPEED, na.rm = TRUE),
          "SD speed"   = sd(SPEED, na.rm = TRUE),
          n            = sum(!is.na(SPEED)),
          records      = n(),
          airports     = n_distinct(AIRPORT))

## Source: local data frame [11 x 6]
##
##    ENG_1_POS Mean speed SD speed     n records airports
## 1                113.20    40.40  1303   34269      742
## 2          1     154.81    43.14 27634   56104      565
## 3          2      62.00    43.39     4       6        6
## 4          3     108.32    33.48    31      59       45
## 5          4     123.90    41.74  7953   13116     1028
## 6          5     143.80    42.31 17701   33165      896
## 7          6      99.06    36.67   482     620      109
## 8          7      83.91    29.34  3829    5569     1186
## 9          A      90.00       NA     1       1        1
## 10         C        NaN       NA     0       1        1
## 11         T        NaN       NA     0       1        1
```

group_by

```r
bnch <-
  benchmark(base = aggregate(SPEED ~ ENG_1_POS, wls_yr,
                             function(x) c(mean = mean(x, na.rm = TRUE),
                                           sd   = sd(x, na.rm = T),
                                           n    = sum(!is.na(x)))),
            dplyr = summarise(group_by(wls_yr, ENG_1_POS),
                              "Mean speed" = mean(SPEED, na.rm = TRUE),
                              "SD speed"   = sd(SPEED, na.rm = TRUE),
                              n            = sum(!is.na(SPEED))),
            replications = 100)
select(bnch, test, replications, elapsed, relative)
```

```
##    test replications elapsed relative
## 1  base          100  82.651    54.81
## 2 dplyr          100   1.508     1.00
```

dplyr::summarise is much faster than stats::aggregate.

Say we need to filter, group_by, and summarise data

```
# What is the mean distance from the airport, in kilometers,
# where the strike took place, by damage to engine, on twin
# engine aircraft, between 2002 and 2010, inclusive?
summarize(group_by(mutate(filter(wls, INCIDENT_YEAR >= 2002, INCIDENT_YEAR <
    2010, NUM_ENGS == 2), DISTANCE_KM = DISTANCE * 1.60934), DAM_ENG1,
    DAM_ENG2), `mean distance in KM` = mean(DISTANCE_KM, na.rm = TRUE))

## Source: local data frame [4 x 3]
## Groups: DAM_ENG1
##
##   DAM_ENG1 DAM_ENG2 mean distance in KM
## 1    FALSE    FALSE              1.3715
## 2    FALSE     TRUE              1.3228
## 3     TRUE    FALSE              0.8347
## 4     TRUE     TRUE              0.6584

# Without a comment to explain, how long would it take to
# explain the above code?  You need to read from the inside
# out.  THERE IS A BETTER WAY!
```

Chain together multiple operations.

```
wls %>%
filter(INCIDENT_YEAR >= 2002,
       INCIDENT_YEAR <= 2010,
       NUM_ENGS      == 2) %>%
mutate(DISTANCE_KM = DISTANCE * 1.60934) %>%
group_by(DAM_ENG1, DAM_ENG2) %>%
summarise("mean distance in KM" = mean(DISTANCE_KM, na.rm = TRUE))

## Source: local data frame [4 x 3]
## Groups: DAM_ENG1
##
##   DAM_ENG1 DAM_ENG2 mean distance in KM
## 1    FALSE    FALSE              1.3715
## 2    FALSE     TRUE              1.3228
## 3     TRUE    FALSE              0.8347
## 4     TRUE     TRUE              0.6584
```

More detailed examples of the forward-piping operator follow.

joining data sets

- `dplyr` version 0.2 has the following joins:
  - `inner_join`,
  - `left_join`,
  - `semi_join`, and
  - `anti_join`.
- Stated milestone for version 0.3 includes
  - `outer_join`,
  - `right_join`, and
  - `cross_join`.

joining data sets

Data sets for examples:

```
# Baseball data from Lahman
batting_df  <- data("Batting",  package = "Lahman")
pitching_df <- data("Pitching", package = "Lahman")
person_df   <- data("Master",   package = "Lahman")
batting_df  <- Batting  %>% tbl_df()
pitching_df <- Pitching %>% tbl_df()
person_df   <- Master   %>% tbl_df()
```

joining data sets

Data sets for examples:

```
print(batting_df,  n = 6)

## Source: local data frame [96,600 x 24]
##
##     playerID yearID stint teamID lgID  G G_batting AB R H X2B
## 1  aardsda01   2004     1    SFN   NL 11        11  0 0 0   0
## 2  aardsda01   2006     1    CHN   NL 45        43  2 0 0   0
## 3  aardsda01   2007     1    CHA   AL 25         2  0 0 0   0
## 4  aardsda01   2008     1    BOS   AL 47         5  1 0 0   0
## 5  aardsda01   2009     1    SEA   AL 73         3  0 0 0   0
## 6  aardsda01   2010     1    SEA   AL 53         4  0 0 0   0
## ..       ...    ...   ...    ...  ... ..       ... .. . . ...
## Variables not shown: X3B (int), HR (int), RBI (int), SB
##   (int), CS (int), BB (int), SO (int), IBB (int), HBP (int),
##   SH (int), SF (int), GIDP (int), G_old (int)
```

joining data sets

Data sets for examples:

```
print(pitching_df, n = 6)

## Source: local data frame [41,857 x 30]
##
##     playerID yearID stint teamID lgID W L  G GS CG SHO SV
## 1  aardsda01   2004     1    SFN   NL 1 0 11  0  0   0  0
## 2  aardsda01   2006     1    CHN   NL 3 0 45  0  0   0  0
## 3  aardsda01   2007     1    CHA   AL 2 1 25  0  0   0  0
## 4  aardsda01   2008     1    BOS   AL 4 2 47  0  0   0  0
## 5  aardsda01   2009     1    SEA   AL 3 6 73  0  0   0 38
## 6  aardsda01   2010     1    SEA   AL 0 6 53  0  0   0 31
## ..       ...    ...   ...    ...  ... . . .. .. .. ... ..
## Variables not shown: IPouts (int), H (int), ER (int), HR
##   (int), BB (int), SO (int), BAOpp (dbl), ERA (dbl), IBB
##   (int), WP (int), HBP (int), BK (int), BFP (int), GF (int),
##   R (int), SH (lgl), SF (lgl), GIDP (lgl)
```

joining data sets

Data sets for examples:

```
print(person_df,   n = 6)

## Source: local data frame [18,125 x 35]
##
##    lahmanID  playerID managerID      hofID birthYear
## 1         1 aaronha01        NA aaronha01h      1934
## 2         2 aaronto01        NA         NA      1939
## 3         3  aasedo01        NA         NA      1954
## 4         4  abadan01        NA         NA      1972
## 5         5 abadijo01        NA         NA      1854
## 6         6 abbated01        NA         NA      1877
## ..      ...       ...       ...        ...       ...
## Variables not shown: birthMonth (int), birthDay (int),
##   birthCountry (chr), birthState (chr), birthCity (chr),
##   deathYear (int), deathMonth (int), deathDay (int),
##   deathCountry (chr), deathState (chr), deathCity (chr),
##   nameFirst (chr), nameLast (chr), nameNote (chr), nameGiven
##   (chr), nameNick (chr), weight (int), height (int), bats
##   (fctr), throws (fctr), debut (date), finalGame (date),
##   college (chr), lahman40ID (chr), lahman45ID (chr), retroID
##   (chr), holtzID (chr), bbrefID (chr), deathDate (date),
```

joining data sets

inner_join

Return all rows from x where there are matching values in y, and all columns from x and y. If there are multiple matches between x and y, all combination of the matches are returned.

```
# build a data.frame for the pitching stats of players born in Colorado
base_inner <- person_df %>%
              subset(birthState == "CO") %>%
              merge(x = .,
                    y = pitching_df,
                    by = "playerID",
                    all = FALSE) %>%
              tbl_df()

dplyr_inner <- person_df %>%
               filter(birthState == "CO") %>%
               inner_join(x = ., pitching_df, by = "playerID")
```

joining data sets

inner_join

```r
dim(dplyr_inner)
```

```
## [1] 297  64
```

```r
all.equal(base_inner, dplyr_inner)
```

```
## [1] TRUE
```

joining data sets

inner_join

```
bnch <-
  benchmark(base = person_df %>%
                   subset(birthState == "CO") %>%
                   merge(x = .,
                         y = pitching_df,
                         by = "playerID",
                         all = FALSE),
            dplyr = person_df %>%
                    filter(birthState == "CO") %>%
                    inner_join(x = ., pitching_df, by = "playerID"),
            replications = 100)
bnch %>% select(test, replications, elapsed, relative)

##    test replications elapsed relative
## 1  base          100   4.305    8.768
## 2 dplyr          100   0.491    1.000
```

joining data sets

left_join

Return all rows from x, and all columns from x and y. If there are multiple matches between x and y, all combination of the matches are returned.

```
base_left <- person_df %>%
             subset(birthState == "CO") %>%
             merge(x = .,
                   y = pitching_df,
                   by = "playerID",
                   all.x = TRUE) %>%
             tbl_df()

dplyr_left <- person_df %>%
              filter(birthState == "CO") %>%
              left_join(x = ., pitching_df, by = "playerID")

all.equal(base_left, dplyr_left)

## [1] TRUE

dim(dplyr_left)
```

joining data sets

left_join

```
bnch <-
  benchmark(base = person_df %>%
                   subset(birthState == "CO") %>%
                   merge(x = .,
                         y = pitching_df,
                         by = "playerID",
                         all.x = TRUE),
            dplyr = person_df %>%
                    filter(birthState == "CO") %>%
                    left_join(x = ., pitching_df, by = "playerID"),
            replications = 100)
bnch %>% select(test, replications, elapsed, relative)

##    test replications elapsed relative
## 1  base          100   4.659    2.988
## 2 dplyr          100   1.559    1.000
```

joining data sets

semi_join

Return all rows from x where there are matching values in y, keeping just columns from x.
A semi join differs from an inner join because an inner join will return one row of x for each matching row of y, where a semi join will never duplicate rows of x.

```
dplyr_semi <- person_df %>%
              filter(birthState == "CO") %>%
              semi_join(pitching_df,
                        by = "playerID")
dplyr_inner %>% dim()

## [1] 297  64

dplyr_semi %>% dim()

## [1] 54 35

# the semi join returns a subset of the person_df data.frame
# which has at least one match in the pitching_df.
```

joining data sets

anti_join

Return all rows from x where there are not matching values in y, keeping just columns from x

```
dplyr_left %>% summarise(n_distinct(playerID))

## Source: local data frame [1 x 1]
##
##   n_distinct(playerID)
## 1                   83

dplyr_semi %>% summarise(n_distinct(playerID))

## Source: local data frame [1 x 1]
##
##   n_distinct(playerID)
## 1                   54

# there are 83 - 54 = 29 players who have no pitching stats...
# Who are they?
```

joining data sets

anti_join

```
dplyr_anti <- person_df %>%
              filter(birthState == "CO") %>%
              anti_join(pitching_df, by = "playerID")
dplyr_anti

## Source: local data frame [29 x 35]
##
##    lahmanID  playerID  managerID      hofID birthYear
## 1     18178 headlch01         NA         NA      1984
## 2     15005 welshji01         NA         NA      1902
## 3     13711 straijo01         NA         NA      1954
## 4     13595 stenhmi01         NA         NA      1958
## 5     13556 stearjo01         NA stearjo01h      1951
## 6     12360  ryanbu01         NA         NA      1885
## 7     12326 runneto01 runneto01m         NA      1955
## 8     12043 roberda02         NA         NA      1936
## 9     11592 radtkja01         NA         NA      1913
## 10    10411 niehobe01         NA         NA      1884
## 11    10226 myattge01 myattge01m         NA      1914
## 12    10155 murphbu01         NA         NA      1895
## 13    10069 moutoja01         NA         NA      1968
```

joining data sets

outer_join

- Return all rows from x and y, keeping all columns of x and y.
- Not implemented in `dplyr_0.2`, will be implemented in `dplyr_0.3`.

```
base_outer <- merge(batting_df, pitching_df,
                    by = "playerID", all = TRUE,
                    suffixes = c(".batting", ".pitching")) %>%
              tbl_df()

# workaround for dplyr_0.2, outer_join should be part of dplyr_0.3
my_outer_join <- function(dfx, dfy, suffixes = c(".x", ".y"), ...) {
  d1 <- left_join(dfx, dfy, ...)
  d2 <- left_join(dfy, dfx, ...)

  names(d1) <- names(d1) %>%
               gsub("\\.x", suffixes[1], x = .)  %>%
               gsub("\\.y", suffixes[2], x = .)
  names(d2) <- names(d2) %>%
               gsub("\\.y", suffixes[1], x = .)  %>%
               gsub("\\.x", suffixes[2], x = .)

  out <- dplyr::union(d1, d2)
```

joining data sets

outer_join

```
dplyr_outer <- my_outer_join(batting_df, pitching_df,
                             c(".batting", ".pitching"),
                             by = "playerID")

all.equal(base_outer, dplyr_outer)

## [1] TRUE

dim(dplyr_outer)

## [1] 450488     53
```

joining data sets

outer_join

The workaround is still faster than using base::merge!

```
bnch <-
  benchmark(base = {
              base_outer <- merge(batting_df, pitching_df,
                                  by = "playerID", all = TRUE,
                                  suffixes = c(".batting", ".pitching"))
                    },
            dplyr = {
              dplyr_outer <- my_outer_join(batting_df, pitching_df,
                                           c(".batting", ".pitching"),
                                           by = "playerID")
            },
            replications = 10)

bnch %>% select(test, replications, elapsed, relative)

##    test replications elapsed relative
## 1  base           10  138.48    3.953
## 2 dplyr           10   35.03    1.000
```

joining data sets

cross_join and right_join

- right_join(y, x) ≡ left_join(x, y)
- cross_join(x, y): every row of y is matched with every row of x.

```
dfx <- data.frame(id = 1:15, var1 = rnorm(15), var2 = runif(15))
dfy <- data.frame(id = 1:8, var1 = LETTERS[1:8], var2 = letters[1:8])

base_cross <- merge(dfx, dfy, by = NULL) %>% tbl_df()

# my_cross_join, a function for cross joins via dplyr
my_cross_join <- function(dfx, dfy) {
  nrx <- nrow(dfx)

  dfy2 <- replicate(nrx, dfy, simplify = FALSE) %>%
          rbind_all() %>%
          mutate(special.id = rep(1:nrx, each = nrow(dfy)))
  dfx2 <- dfx %>% mutate(special.id = 1:nrx)
  out <- inner_join(dfx2, dfy2, by = "special.id") %>% select(-special.id)
  return(out)
}
```

joining data sets

cross_join and right_join

```
dplyr_cross <- my_cross_join(dfx, dfy)

all.equal(base_cross, dplyr_cross)

## [1] TRUE
```

joining data sets

cross_join

```
dfx <- data.frame(id = 1:60, var1 = rnorm(60), var2 = runif(60))
dfy <- data.frame(id = 1:13, var1 = LETTERS[1:13], var2 = letters[1:13])

bnch <-
  benchmark(base = {
              base_cross <- merge(dfx, dfy, by = NULL) %>% tbl_df()
            },
            dplyr = {
              dplyr_cross <- my_cross_join(dfx, dfy)
            },
            replications = 1000)

bnch %>% select(test, replications, elapsed, relative)

##    test replications elapsed relative
## 1  base         1000   4.908    1.644
## 2 dplyr         1000   2.986    1.000
```

dplyr is fast, fast enough to overcome the additional scripting.

Memory usage

```r
this_df2 <- this_df <- data.frame(var1 = 1:5, var2 = rnorm(5))
changes(this_df, this_df2)

## <identical>

this_df$var1 <- rexp(5, rate = 2)
changes(this_df, this_df2)

## Changed variables:
##           old       new
## var1      0x10d3c660 0x94764a8
##
## Changed attributes:
##           old       new
## row.names 0x548836f0 0x548833d8

location(this_df2)

## <0x10988118>
## Variables:
##  * var1:        <0x94764a8 >
```

Memory usage

```
this_df2 <- this_df <- data.frame(var1 = 1:5, var2 = rnorm(5)) %>% tbl_df()
changes(this_df, this_df2)

## <identical>

this_df <- this_df %>% mutate(var1 = rexp(5))
changes(this_df, this_df2)

## Changed variables:
##           old       new
## var1      0xb77dea8 0x1fa8d620
##
## Changed attributes:
##           old        new
## names     0xa70c070  0x972d510
## class     0x1f3e35e8 0x1fa79298
## row.names 0x9cba5d8  0x9cba8f0
```

`dplyr` "smart enough to create only one new column: all the other columns continue to point at their old locations."

dplyr memory usage

From the `vignette("memory", "dplyr")`

- `tbl_df()` and `group_by()` don't copy columns
- `select()` never copies columns, even when you rename them
- `mutate()` never copies columns, except when you modify an existing column
- `arrange()` must copy because you're changing the order of every column. This is an expensive operation for big data, but you can generally avoid it using the order argument to window functions
- `summarise()` creates new data, but it's usually at least an order of magnitude smaller than the original data.

Window Functions

- See `vignette("window-functions", package = "dplyr")`
- window functions are variations of aggregation functions.
  - Aggregation functions such as `sum()` and `median()` are maps between $\mathbb{R}^n \to \mathbb{R}^1$.
  - Window function are maps between $\mathbb{R}^n \to \mathbb{R}^n$. Examples: `cumsum()`, `rank()`, `lag()`

Window Functions

Examples

The following examples come from `vignette("window-functions", "dplyr")`

```
batting <- Batting %>% tbl_df() %>%
           select(playerID, yearID, teamID, G, AB:H, HR)

# For each player, find the two years with most hits
batting %>%
  group_by(playerID) %>%
  filter(min_rank(desc(H)) <= 2 & H > 0)

## Source: local data frame [24,834 x 8]
## Groups: playerID
##
##     playerID yearID teamID   G  AB   R   H HR
## 1  aaronha01   1959    ML1 154 629 116 223 39
## 2  aaronha01   1963    ML1 161 631 121 201 44
## 3  aaronto01   1962    ML1 141 334  54  77  8
## 4  aaronto01   1968    ATL  98 283  21  69  1
## 5   abadan01   2003    BOS   9  17   1   2  0
## 6   abadfe01   2012    HOU  37   7   0   1  0
## 7  abadijo01   1875    PH3  11  45   3  10  0
```

Window Functions

Examples

```
# Within each player, rank each year by the number of games played
batting %>%
  group_by(playerID) %>%
  mutate(G_rank = min_rank(G))
```

```
## Source: local data frame [96,600 x 9]
## Groups: playerID
##
##     playerID yearID teamID   G  AB   R   H HR G_rank
## 1  aardsda01   2004    SFN  11   0   0   0  0      2
## 2  aardsda01   2006    CHN  45   2   0   0  0      4
## 3  aardsda01   2007    CHA  25   0   0   0  0      3
## 4  aardsda01   2008    BOS  47   1   0   0  0      5
## 5  aardsda01   2009    SEA  73   0   0   0  0      7
## 6  aardsda01   2010    SEA  53   0   0   0  0      6
## 7  aardsda01   2012    NYA   1  NA  NA  NA NA      1
## 8  aaronha01   1954    ML1 122 468  58 131 13      4
## 9  aaronha01   1955    ML1 153 602 105 189 27     13
## 10 aaronha01   1956    ML1 153 609 106 200 26     13
## ..       ...    ...    ... ... ... ... ... ..    ...
```

Window Functions

Examples

```
# For each player, the proportion of years with more home runs than the prio
#DRUG}DRUG} year
batting %>%
  group_by(playerID) %>%
  mutate(more_hrs = HR > lag(HR)) %>%                  # Window function
  summarize(more_hrs = mean(more_hrs, na.rm = TRUE))  # aggregate function

## Source: local data frame [17,908 x 2]
##
##     playerID more_hrs
## 1  aardsda01   0.0000
## 2  aaronha01   0.4545
## 3  aaronto01   0.3333
## 4   aasedo01   0.0000
## 5   abadan01   0.0000
## 6   abadfe01   0.0000
## 7  abadijo01   0.0000
## 8  abbated01   0.2222
## 9  abbeybe01   0.0000
## 10 abbeych01   0.7500
## ..       ...      ...
```

Window Functions

Examples

```
# For each player, compute avg change in games played per year
batting %>%
  group_by(playerID) %>%
  mutate(G_change = (G - lag(G)) / (yearID - lag(yearID)))

## Source: local data frame [96,600 x 9]
## Groups: playerID
##
##     playerID yearID teamID   G  AB   R   H HR G_change
## 1  aardsda01   2004    SFN  11   0   0   0  0       NA
## 2  aardsda01   2006    CHN  45   2   0   0  0       17
## 3  aardsda01   2007    CHA  25   0   0   0  0      -20
## 4  aardsda01   2008    BOS  47   1   0   0  0       22
## 5  aardsda01   2009    SEA  73   0   0   0  0       26
## 6  aardsda01   2010    SEA  53   0   0   0  0      -20
## 7  aardsda01   2012    NYA   1  NA  NA  NA NA      -26
## 8  aaronha01   1954    ML1 122 468  58 131 13       NA
## 9  aaronha01   1955    ML1 153 602 105 189 27       31
## 10 aaronha01   1956    ML1 153 609 106 200 26        0
## ..       ...    ...    ... ... ... ... ... ..      ...
```

Window Functions

Examples

```
# For each player, find all when they played more games than average
batting %>%
  group_by(playerID) %>%
  filter(G > mean(G)) %>%
  select(playerID, yearID)

## Source: local data frame [47,769 x 2]
## Groups: playerID
##
##     playerID yearID
## 1  aardsda01   2006
## 2  aardsda01   2008
## 3  aardsda01   2009
## 4  aardsda01   2010
## 5  aaronha01   1955
## 6  aaronha01   1956
## 7  aaronha01   1957
## 8  aaronha01   1958
## 9  aaronha01   1959
## 10 aaronha01   1960
## ..       ...    ...
```

Other Data Sources

- `dplyr` works for
  - `data.frames`,
  - `data.tables`, databases, and multidimensional arrays.
  - Same verbs used for all data sources.
  - See `vignette("databases", package = "dplyr")` for more details.

`data.table` vs `dplyr`

From the
dplyr introduction vignette:

- For multiple operations, data.table can be faster because you usually use it with multiple verbs at the same time. For example, with data table you can do a mutate and a select in a single step, and it's smart enough to know that there's no point in computing the new variable for the rows you're about to throw away.
- The advantages of using `dplyr` with data tables are:
  - For common data manipulation tasks, it insulates you from reference semantics of data.tables, and protects you from accidentally modifying your data.
  - Instead of one complex method built on the subscripting operator ([), it provides many simple methods.

magrittr: a forward-pipe operator for R

ceci n'est pas un pipe (this is not a pipe)

- dplyr functionality is made more powerful via the %>%, or equivalently, \%.%$, operator.
- Additional functionally provided by the magrittr package authored by Stefan Bache and Hadley Wickham.
- These operators are similar to
  - F#'s | >, or
  - Linux's |.
- Use of these operators will drastically change your R syntax.
- Helpful to writing complex, nested, operations.
- "Read from left to right instead of inside out."

magrittr: a foward-pipe operator for R

Examples

```
mu <- 1; sigma <- 4; N <- 5; y <- rnorm(N, mu, sigma)

# -2 log likelihood, standard nested operations, i.e, infix notation
-2 * log((1/sqrt(2 * pi * sigma^2))^(N) * exp(-1/(2 * sigma^2) * sum((y - mu

## [1] 29.03

# -2 log likelihood, using forward-piping, somewhat like postfix notation
y %>%
subtract(mu) %>%
raise_to_power(2) %>%
sum %>%
divide_by(-2 * sigma^2) %>%
exp %>%
multiply_by((2 * pi * sigma^2)^(-N/2)) %>%
log %>%
multiply_by(-2)

## [1] 29.03
```

What do you think? Pros and Cons?

Other resources

- Wickham's presentation at useR!2014: Data manipulation with dplyr, `http://blog.revolutionanalytics.com/2014/06/user-2014-is-underway.html`
- R Meetup - Hadley Wickham and Joe Cheng of RStudio return to BARUG. `http://www.youtube.com/watch?v=qRSfxSRdL5Y`

Reproducibility

The data, code, sides, etc. all at `github.com/dewittpe/dplyr-demo`

```
print(sessionInfo(), locale = FALSE)

## R version 3.1.0 (2014-04-10)
## Platform: x86_64-pc-linux-gnu (64-bit)
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets
## [6] methods   base
##
## other attached packages:
## [1] rbenchmark_1.0.0 dplyr_0.2        magrittr_1.0.1
## [4] knitr_1.6        vimcom_0.9-93    setwidth_1.0-3
## [7] colorout_1.0-3
##
## loaded via a namespace (and not attached):
##  [1] assertthat_0.1  codetools_0.2-8 digest_0.6.4
##  [4] evaluate_0.5.5  formatR_0.10    highr_0.3
##  [7] parallel_3.1.0  Rcpp_0.11.2     stringr_0.6.2
## [10] tools_3.1.0
```

Denver R User Group Notes

- Future MeeetUp Topics:
  - (Possible) iPython / R speaker for later in July
  - We need others speakers!
- MeetUp locations/times