dplyr : Grammar of data manipulation

What is dplyr?

- dplyr is mainly authored by Hadley Wickham and Romain Francois. It is designed to be intuitive and easy to learn, thereby making "doing things" in R more user-friendly.
- dplyr is a new package which provides a set of tools for efficiently manipulating datasets in R.
- dplyr is the next iteration of plyr, focussing on only data frames.
- dplyr is faster, has a more consistent API and should be easier to use.

dplyr : abstract by Hadley Wickham

Hadley Wickham's Abstract

There are three key ideas that underlie dplyr:

1. Your time is important, so Romain Francois has written the key pieces in **Rcpp** to provide blazing fast performance.

   Performance will only get better over time, especially once we figure out the best way to make the most of multiple processors.

dplyr : abstract by Hadley Wickham

2 Tabular data is tabular data regardless of where it lives, so you should use the same functions to work with it.

With dplyr, anything you can do to a local data frame you can also do to a remote database table.

PostgreSQL, MySQL, SQLite and Google bigquery support is built-in; adding a new backend is a matter of implementing a handful of S3 methods.

dplyr : abstract by Hadley Wickham

3 The bottleneck in most data analyses is the time it takes for you to figure out what to do with your data

dplyr makes this easier by having individual functions that correspond to the most common operations (`group_by`, `summarise`,`mutate`, `filter`, `select` and `arrange`).

Each function does one only thing, but does it well.

Working with dplyr

dplyr focussed on tools for working with data frames (hence the **d** in the name).
dplyr has three main goals:

- Identify the most important data manipulation tools needed for data analysis and make them easy to use from `R`.
- Provide very fast performance for in-memory data by writing key pieces in C++.
- Use the same interface to work with data no matter where it's stored, whether in a data frame, a data table or database.

Tidy Data

- To make the most of dplyr, Hadley Wickham recommends that you familiarise yourself with the **principles of tidy data**.
- This will help you get your data into a form that works well with **dplyr**, **ggplot2** and R's many modelling functions.

Three Principles from Hadley Wickham's paper

1. Each variable forms a column,
2. Each observation forms a row,
3. Each table/file stores data about one kind of observation.

Remark:
The paper "**Tidy data**" by Hadley Wickham (RStudio) can be downloaded from

`http://vita.had.co.nz/papers/tidy-data.pdf`

Key data structures

The key object in **dplyr** is a `tbl`, a representation of a tabular data structure. Currently dplyr supports:

- data frames - the most commonly encountered R data structure.
- data tables - a data structure that is designed for intensive data analysis.

Advances Database Users

For advanced users, **dplyr** also supports the following databases: *SQLite*, *PostgreSQL*, *Redshift*, *MySQL/MariaDB*, *Bigquery*, *MonetDB* and data cubes with arrays (partial implementation).

We will not cover those topics in this workshop.

CRAN tutorial

Introduction to dplyr

2015-01-13

When working with data you must:

- Figure out what you want to do.
- Precisely describe what you want in the form of a computer program.
- Execute the code.

The dplyr package makes each of these steps as fast and easy as possible by:

- Elucidating the most common data manipulation operations, so that your options are helpfully constrained when thinking about how to tackle a problem.
- Providing simple functions that correspond to the most common data manipulation verbs, so that you can easily translate your thoughts into code.
- Using efficient data storage backends, so that you spend as little time waiting for the computer as possible.

Installing dplyr

Straightforward R package installation.

```
install.packages("dplyr")
library(dplyr)

# Data Set Examples
# 1. iris
# 2. mtcars
```

iris data set

```
> names(iris)
[1] "Sepal.Length"
[2] "Sepal.Width"
[3] "Petal.Length"
[4] "Petal.Width"
[5] "Species"
```

mtcars data set

```
> names(mtcars)
 [1] "mpg"  "cyl"  "disp" "hp"
 [5] "drat" "wt"   "qsec" "vs"
 [9] "am"   "gear" "carb"
```

Example Data Sets

```
dim(iris)
[1] 150   5
class(iris)
[1] "data.frame"
mode(iris)
[1] "list"

dim(mtcars)
[1] 32 11
class(mtcars)
[1] "data.frame"
mode(mtcars)
[1] "list"
```

Data frame can be problematic

```
> mtcars
                     mpg cyl  disp  hp drat    wt  qsec v
Mazda RX4           21.0   6 160.0 110 3.90 2.620 16.46
Mazda RX4 Wag       21.0   6 160.0 110 3.90 2.875 17.02
Datsun 710          22.8   4 108.0  93 3.85 2.320 18.61
Hornet 4 Drive      21.4   6 258.0 110 3.08 3.215 19.44
Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02
Valiant             18.1   6 225.0 105 2.76 3.460 20.22
Duster 360          14.3   8 360.0 245 3.21 3.570 15.84
Merc 240D           24.4   4 146.7  62 3.69 3.190 20.00
Merc 230            22.8   4 140.8  95 3.92 3.150 22.90
Merc 280            19.2   6 167.6 123 3.92 3.440 18.30
Merc 280C           17.8   6 167.6 123 3.92 3.440 18.90
Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40
Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60
Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00
Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98
Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82
```

tbl can be much better

```
> mtcars <- as.tbl(mtcars)
> mtcars
Source: local data frame [32 x 11]
    mpg cyl  disp  hp drat    wt  qsec vs am
1  21.0   6 160.0 110 3.90 2.620 16.46  0  1
2  21.0   6 160.0 110 3.90 2.875 17.02  0  1
3  22.8   4 108.0  93 3.85 2.320 18.61  1  1
4  21.4   6 258.0 110 3.08 3.215 19.44  1  0
5  18.7   8 360.0 175 3.15 3.440 17.02  0  0
6  18.1   6 225.0 105 2.76 3.460 20.22  1  0
7  14.3   8 360.0 245 3.21 3.570 15.84  0  0
8  24.4   4 146.7  62 3.69 3.190 20.00  1  0
9  22.8   4 140.8  95 3.92 3.150 22.90  1  0
10 19.2   6 167.6 123 3.92 3.440 18.30  1  0
..  ... ...   ... ...  ...   ...   ... .. ..
Variables not shown: gear (dbl), carb (dbl)
```

dplyr; Single Table Verbs

dplyr aims to provide a function for each basic verb of data manipulating:

- `filter()` (and `slice()`)
- `arrange()`
- `select()` (and `rename()`)
- `distinct()`
- `mutate()` (and `transmute()`)
- `summarise()`
- `sample_n()` and `sample_frac()`

Grouped operations

- The verbs are useful, but they become really powerful when you combine them with the idea of group by, repeating the operation individually on groups of observations within the dataset.
- In **dplyr**, you use the `group_by()` function to describe how to break a dataset down into groups of rows.
- You can use the resulting object in the same functions as above; they'll automatically work 'by group' when the input is a grouped.

group_by

Group a tbl by one or more variables.

Description

Most data operations are useful done on groups defined by variables in the the dataset. The **group_by** function takes an existing tbl and converts it into a grouped tbl where operations are performed "by group".

Summary Statistics

You use `summarise()` with aggregate functions, which take a vector of values, and return a single number.

There are many useful functions in base R like `min()`, `max()`, `mean()`, `sum()`, `sd()`, `median()`, and `IQR()`.

dplyr provides a handful of others:

- `n()`: number of observations in the current group
- `n_distinct(x)`: count the number of unique values in `x`.

Grouping with the group_by command

```
> iris.sp <- group_by(iris,Species)
> class(iris.sp)
[1] "grouped_df" "tbl_df"   "tbl"    "data.frame"

> summarise(iris.sp,mean(Sepal.Length),
            sd(Petal.Length))

Source: local data frame [3 x 3]

     Species mean(Sepal.Length) sd(Petal.Length)
1     setosa                5.006         0.1736640
2 versicolor                5.936         0.4699110
3  virginica                6.588         0.5518947
```

```
> summarise(mtcars2,mean(mpg),sd(mpg))
Source: local data frame [6 x 4]
Groups: cyl

  cyl  am mean(mpg)    sd(mpg)
1   4   0  22.90000 1.4525839
2   4   1  28.07500 4.4838599
3   6   0  19.12500 1.6317169
4   6   1  20.56667 0.7505553
5   8   0  15.05000 2.7743959
6   8   1  15.40000 0.5656854
>
```

Filter rows with `filter()`

- `filter()` allows you to select a subset of the rows of a data frame.
- The first argument is the name of the data frame, and the second and subsequent are filtering expressions evaluated in the context of that data frame.

```
# Species is Virginica
> iris.vir1 <- filter(iris,Species=="virginica")

# Species is Virginica OR Petal.length > 3.2
> iris.vir2 <- filter(iris,
     Species=="virginica" | Petal.Length > 3.2)

# Species is Virginica AND Petal.length > 3.9
> iris.vir3 <- filter(iris,
     Species=="virginica" & Petal.Length > 3.9)
```

Ordering Data Sets with `arrange()`

- `arrange()` works similarly to `filter()` except that instead of filtering or selecting rows, it reorders them.
- It takes a data frame, and a set of column names (or more complicated expressions) to order by.
- If you provide more than one column name, each additional column will be used to break ties in the values of preceding columns.
- Use `desc()` (or `rev()`) to order a column in descending order.

```
> arrange(iris,Petal.Length,Petal.Width)
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	4.6	3.6	1.0	0.2	setosa
2	4.3	3.0	1.1	0.1	setosa
3	5.8	4.0	1.2	0.2	setosa
4	5.0	3.2	1.2	0.2	setosa
5	4.7	3.2	1.3	0.2	setosa
6	5.5	3.5	1.3	0.2	setosa
7	4.4	3.0	1.3	0.2	setosa
8	4.4	3.2	1.3	0.2	setosa
9	5.0	3.5	1.3	0.3	setosa
10	4.5	2.3	1.3	0.3	setosa
11	5.4	3.9	1.3	0.4	setosa
12	4.8	3.0	1.4	0.1	setosa
13	4.9	3.6	1.4	0.1	setosa
14	5.1	3.5	1.4	0.2	setosa
15	4.9	3.0	1.4	0.2	setosa
16	5.0	3.6	1.4	0.2	setosa
17	4.4	2.9	1.4	0.2	setosa
18	5.2	3.4	1.4	0.2	setosa
19	5.5	4.2	1.4	0.2	setosa

```
> arrange(iris,Petal.Length,rev(Petal.Width))
    Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
1            4.6         3.6          1.0         0.2     setosa
2            4.3         3.0          1.1         0.1     setosa
3            5.8         4.0          1.2         0.2     setosa
4            5.0         3.2          1.2         0.2     setosa
5            5.4         3.9          1.3         0.4     setosa
6            4.5         2.3          1.3         0.3     setosa
7            4.4         3.2          1.3         0.2     setosa
8            4.4         3.0          1.3         0.2     setosa
9            4.7         3.2          1.3         0.2     setosa
10           5.5         3.5          1.3         0.2     setosa
11           5.0         3.5          1.3         0.3     setosa
12           5.1         3.5          1.4         0.2     setosa
13           4.8         3.0          1.4         0.1     setosa
```

Select columns with `select()`

- Often you have a dataset with many columns of which only a few are of interest to you.
- `select()` allows you to rapidly zoom in on a useful subset using operations that usually only work on numeric variable positions.

Selection Options with `select()`

- `starts_with(x, ignore.case = TRUE)`: names starts with x
- `ends_with(x, ignore.case = TRUE)`: names ends in x
- `contains(x, ignore.case = TRUE)`: selects all variables whose name contains x
- `matches(x, ignore.case = TRUE)`: selects all variables whose name matches the regular expression x
- `num_range("x", 1:5, width = 2)`: selects all variables (numerically) from x01 to x05.
- `one_of("x", "y", "z")`: selects variables provided in a character vector.
- `everything()`: selects all variables.

Selection Options with `select()`

```
> select(iris,ends_with("idth"))
    Sepal.Width Petal.Width
1           3.5         0.2
2           3.0         0.2
3           3.2         0.2
4           3.1         0.2
5           3.6         0.2
6           3.9         0.4
7           3.4         0.3
8           3.4         0.2
9           2.9         0.2
10          3.1         0.1
11          3.7         0.2
12          3.4         0.2
13          3.0         0.1
```

Selection Options with `select()`

```
> select(iris,contains("etal"))
    Petal.Length Petal.Width
1            1.4         0.2
2            1.4         0.2
3            1.3         0.2
4            1.5         0.2
5            1.4         0.2
6            1.7         0.4
7            1.4         0.3
8            1.5         0.2
9            1.4         0.2
```

Sampling rows with `sample_n()` and `sample_frac()`

- Use `sample_n()` and `sample_frac()` to take a random sample of rows
  - Fixed number for `sample_n()`
  - Fixed fraction for `sample_frac()`.

Sampling rows with sample_n()

```
# Sample 2 row from iris
> sample_n(iris, 2)
Source: local data frame [2 x 5]

  Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
1          6.3         2.7          4.9         1.8 virginica
2          5.1         3.5          1.4         0.3    setosa
```

Sampling rows with sample_n() on a grouped object

```
# Sample 2 row per species
# iris.sp is a grouped object
> class(iris.sp)
[1] "grouped_df" "tbl_df"     "tbl"        "data.frame"

> sample_n(iris.sp, 2)
Source: local data frame [6 x 5]
Groups: Species

  Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
1          4.6         3.6          1.0         0.2     setosa
2          4.9         3.0          1.4         0.2     setosa
3          5.4         3.0          4.5         1.5 versicolor
4          6.4         3.2          4.5         1.5 versicolor
5          6.7         3.3          5.7         2.1  virginica
6          6.3         2.5          5.0         1.9  virginica
```

Sampling rows with `sample_frac()` on a grouped object

```
> sample_frac(iris.sp, 0.02)
Source: local data frame [3 x 5]
Groups: Species

  Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
1          4.8         3.0          1.4         0.3     setosa
2          4.9         2.4          3.3         1.0 versicolor
3          6.4         3.1          5.5         1.8  virginica
```

Add new columns with `mutate()`

It's often useful to add new columns that are functions of existing columns. This is the job of `mutate()`:

```
> mutate(iris,
       PW2 = log(Petal.Width),
       PL2=sqrt(Petal.Length))

Source: local data frame [150 x 7]

   Sepal.Length Sepal.Width Petal.Length Petal.Width Species        PW2      PL2
1           5.1         3.5          1.4         0.2  setosa -1.6094379 1.183216
2           4.9         3.0          1.4         0.2  setosa -1.6094379 1.183216
3           4.7         3.2          1.3         0.2  setosa -1.6094379 1.140175
4           4.6         3.1          1.5         0.2  setosa -1.6094379 1.224745
5           5.0         3.6          1.4         0.2  setosa -1.6094379 1.183216
6           5.4         3.9          1.7         0.4  setosa -0.9162907 1.303840
7           4.6         3.4          1.4         0.3  setosa -1.2039728 1.183216
8           5.0         3.4          1.5         0.2  setosa -1.6094379 1.224745
9           4.4         2.9          1.4         0.2  setosa -1.6094379 1.183216
10          4.9         3.1          1.5         0.1  setosa -2.3025851 1.224745
..          ...         ...          ...         ...     ...        ...      ...
```

Add new columns with `mutate()`

`mutate()` allows you to refer to columns that you just created:

```
> mutate(iris,
       PW2 = log(Petal.Width),
       PL2=sqrt(Petal.Length),
       Ratio=PL2/PW2 )

Source: local data frame [150 x 8]

   Sepal.Length Sepal.Width Petal.Length Petal.Width Species        PW2      PL2      Ratio
1           5.1         3.5          1.4         0.2  setosa -1.6094379 1.183216 -0.7351734
2           4.9         3.0          1.4         0.2  setosa -1.6094379 1.183216 -0.7351734
3           4.7         3.2          1.3         0.2  setosa -1.6094379 1.140175 -0.7084308
4           4.6         3.1          1.5         0.2  setosa -1.6094379 1.224745 -0.7609768
5           5.0         3.6          1.4         0.2  setosa -1.6094379 1.183216 -0.7351734
6           5.4         3.9          1.7         0.4  setosa -0.9162907 1.303840 -1.4229550
7           4.6         3.4          1.4         0.3  setosa -1.2039728 1.183216 -0.9827597
8           5.0         3.4          1.5         0.2  setosa -1.6094379 1.224745 -0.7609768
9           4.4         2.9          1.4         0.2  setosa -1.6094379 1.183216 -0.7351734
10          4.9         3.1          1.5         0.1  setosa -2.3025851 1.224745 -0.5318999
..          ...         ...          ...         ...     ...        ...      ...        ...
```

Multiple table verbs

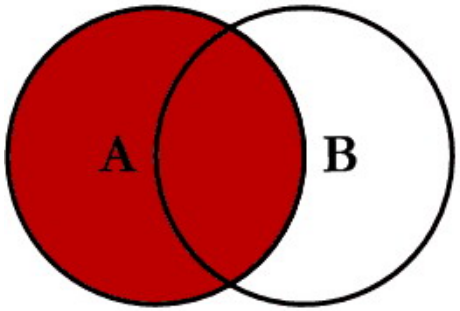
As well as verbs that work on a single `tbl`, there are also a set of useful verbs that work with two `tbls` at a time: joins and set operations.

Joins

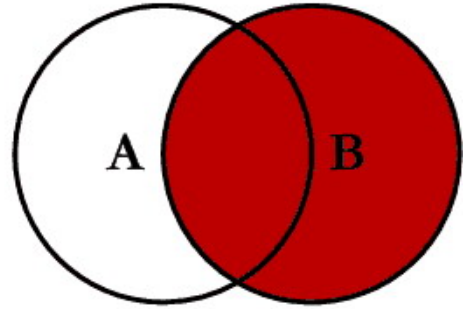
dplyr implements the four most useful joins from SQL:

- `inner_join(x, y)`: matching x + y
- `left_join(x, y)`: all x + matching y
- `semi_join(x, y)`: all x with match in y
- `anti_join(x, y)`: all x without match in y

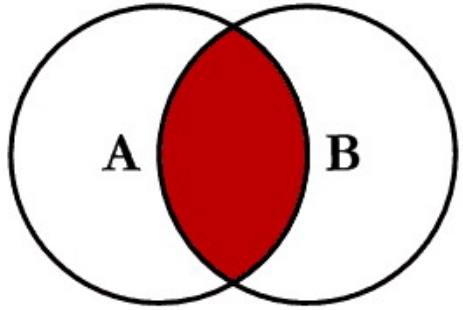
SQL JOINS



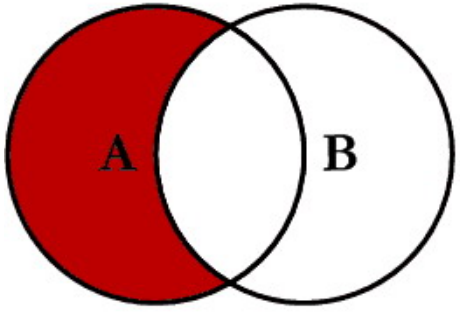

```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```



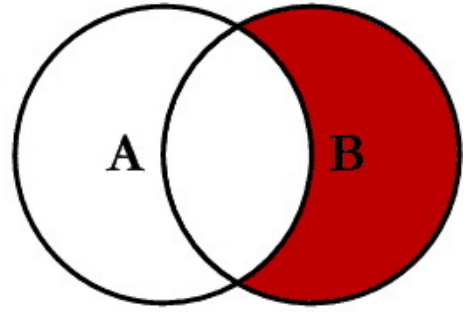

```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
```



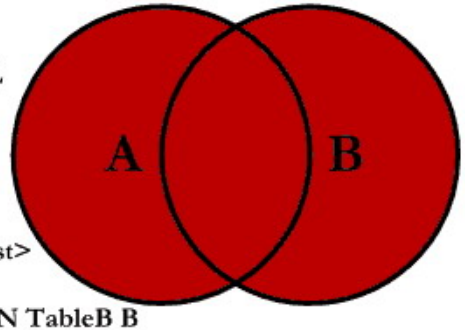

```
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```



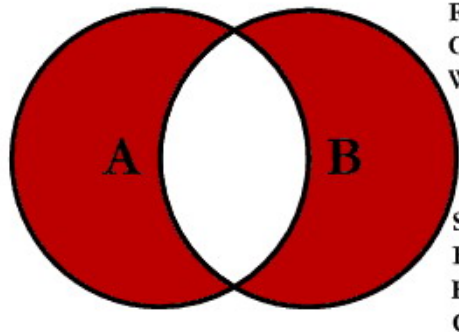

```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```




```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
```




```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
```




```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
OR B.Key IS NULL
```

Joins

Pretend data set listing country of origin for each species. The variables "Species" is common to both data frames.

```
> irisnewdata
     Species   origin
1  virginica  Ireland
2     setosa Scotland
3 versicolor    Wales
>
```

Figure: Second Data Frame

Joins

```
>
>
> irisnewdata = data.frame(Species=c("virginica","setosa","versicolor"),origin=c(
>
>
> left_join(iris,irisnewdata)
Joining by: "Species"
       Species Sepal.Length Sepal.Width Petal.Length Petal.Width   origin
1       setosa          5.1         3.5          1.4         0.2 Scotland
2       setosa          4.9         3.0          1.4         0.2 Scotland
3       setosa          4.7         3.2          1.3         0.2 Scotland
4       setosa          4.6         3.1          1.5         0.2 Scotland
5       setosa          5.0         3.6          1.4         0.2 Scotland
6       setosa          5.4         3.9          1.7         0.4 Scotland
7       setosa          4.6         3.4          1.4         0.3 Scotland
8       setosa          5.0         3.4          1.5         0.2 Scotland
```

Set Theory Operations

dplyr implements the methods for set theory operations

- `intersect(x, y)`: all rows in both x and y
- `union(x, y)`: rows in either x or y
- `setdiff(x, y)`: rows in x, but not y