

# Prácticas de Tecnologías de Gestión y Manipulación de Datos

*Guillermo López Taboada (guillermo.lopez.taboada@udc.es) y  
Rubén F. Casal (ruben.fcasal@udc.es)*

*2019-11-07*



# Índice general



# Prólogo

Este libro contiene algunas de las prácticas de la asignatura de Tecnologías de Gestión de Datos del Máster interuniversitario en Técnicas Estadísticas).

Este libro ha sido escrito en R-Markdown empleando el paquete **bookdown** y está disponible en el repositorio Github: [gltaboada/tgdbook](https://gltaboada/tgdbook). Se puede acceder a la versión en línea a través del siguiente enlace:

<https://gltaboada.github.io/tgdbook>.

donde puede descargarse en formato pdf.

Para ejecutar los ejemplos mostrados en el libro será necesario tener instalados los siguientes paquetes: **dplyr** (colección **tidyverse**), **tidyr**, **stringr**, **readxl**, **openxlsx**, **RODBC**, **sqldf**, **RSQLite**, **foreign**, **SparkR**, **magrittr**, **knitr**. Por ejemplo mediante los comandos:

```
pkgs <- c('dplyr', 'tidyr', 'stringr', 'readxl', 'openxlsx', 'magrittr',  
          'RODBC', 'sqldf', 'RSQLite', 'foreign', 'SparkR', 'knitr')  
# install.packages(pkgs, dependencies=TRUE)  
install.packages(setdiff(pkgs, installed.packages()[,'Package']), dependencies = TRUE)
```

Para generar el libro (compilar) se recomendaría consultar el libro de “Escritura de libros con bookdown” en castellano.



Este obra está bajo una licencia de Creative Commons Reconocimiento-NoComercial-SinObraDerivada 4.0 Internacional (esperamos poder liberarlo bajo una licencia menos restrictiva más adelante...).



# Capítulo 1

## Introducción a las Tecnologías de Gestión y Manipulación de Datos

La información relevante de la materia está disponible en la guía docente y la ficha de la asignatura

En particular, los resultados de aprendizaje son:

- Manejar de forma autónoma y solvente el software necesario para acceder a conjuntos de datos en entornos profesionales y/o en la nube.
- Saber gestionar conjuntos de datos masivos en un entorno multidisciplinar que permita la participación en proyectos profesionales complejos que requieran el uso de técnicas estadísticas.
- Saber relacionar el software de diseño y gestión de bases de datos con el específicamente implementado para el análisis de datos.

### 1.1 Contenidos

1. Introducción al lenguaje SQL
  - Bases de datos relacionales
  - Sintaxis SQL
  - Conexión con bases de datos desde R
2. Introducción a tecnologías NoSQL
  - Conceptos y tipos de bases de datos NoSQL (documental, columnar, clave/valor y de grafos)
  - Conexión de R a NoSQL
3. Tecnologías para el tratamiento de datos masivos

- Tecnologías Big Data (Hadoop, Spark, Hive, Rspark, Sparklyr)
- Visualización y generación de cuadros de mando
- Introducción al análisis de datos masivos.

## 1.2 Planificación

*Working draft...*

La impartición de los contenidos durante el curso dependerá de los conocimientos de partida y la asimilación de los conceptos. Para completar nuestra visión de los conocimientos previos os requerimos completar este formulario en la primera sesión de clase: <https://forms.gle/9HR5iFHXoLowrCHLA>

- Clase 1 (12/9-R): Seminario R (Manipulación de datos con el paquete base de R)
- Clase 2 (19/9): Tema 1: Conceptos de bases de datos
- Clase 3 (26/9): Tema 1: Introducción a SQL
- Clase 4 (3/10-R): Seminario dplyr (Manipulación de datos con el paquete dplyr)
- Clase 5 (10/10): Tema 1: Ejercicios prácticos de Entidad-relación y SQL
- Clase 6 (17/10): Tema 1: Continuación de ejercicios prácticos SQL
- Clase 7 (24/10): Tema 2: Introducción a NoSQL
- Clase 8 (31/10): Tema 2: Ejercicios prácticos de NoSQL
- Clase 9 (7/11): Tema 3: Ecosistema Big Data (Hadoop, Spark)
- Clase 10 (14/11): Tema 3: Tecnologías Big Data (Rspark/sparklyr)
- Clase 11 (21/11): Seminario visualización con power BI
- Clase 12 (28/11): Seminario machine learning CESGA/localhost
- Clase 13 (5/12): Tema 3: Introducción al análisis de datos masivos
- Clase 14 (12/12-R): Seminario Aprendizaje Estadístico/Automático, Data Mining
- Clase 15 (19/12-opcional): Seminario aplicaciones Big Data en investigación e industria

### 1.2.1 Evaluación

- **Examen** (60%): El examen de la materia evaluará los siguientes aspectos: Conceptos de la materia: Dominio de los conocimientos teóricos y operativos de la materia. Asimilación práctica de materia: Asimilación y comprensión de los conocimientos teóricos y operativos de la materia.



- **Prácticas de laboratorio** (30%): Evaluación de las prácticas de laboratorio desarrolladas por los estudiantes.
- **Trabajos tutelados** (10%): Evaluación de los trabajos tutelados desarrollados por los estudiantes.

#### 1.2.1.1 Observaciones sobre la evaluación:

- Las prácticas de laboratorio se realizarán en grupos de 2 estudiantes, salvo el trabajo tutelado que es individual y opcional. En caso de no realizar el trabajo tutelado los estudiantes tendrán un 67% de nota del examen y un 33% de prácticas de laboratorio. El plazo para realizar las 3 prácticas será de 2 semanas desde la presentación de la práctica. El plazo para la entrega de los trabajos tutelados es el último día de clase de la asignatura.
- El estudiante que quiera realizar un trabajo tutelado ha de hablar (o mediante correo electrónico) con los profesores para validar y confirmar el tema y alcance del trabajo tutelado.
- Para poder aprobar la asignatura en la primera oportunidad será necesario obtener como mínimo el 30% de la nota máxima de la suma de las prácticas de laboratorio y trabajos tutelados e, igualmente, el 30% de la nota máxima final de la Prueba mixta (examen), y tener una nota total (prácticas más trabajos tutelados más prueba mixta) igual o superior al 50% de la nota máxima.
- En la segunda oportunidad solamente se podrá recuperar la nota del examen. Las notas de prácticas y de trabajos tutelados serán las obtenidas durante el curso. Para los alumnos que utilicen la oportunidad adelantada de diciembre se utilizarán las notas de prácticas y trabajos tutelados que obtuvieran en su último curso. En esta oportunidad solo será necesario para aprobar obtener una nota total igual o superior al 50% de la nota máxima.
- Una vez que un estudiante es evaluado en una práctica de laboratorio o en un trabajo tutelado implica que será calificado. Por tanto, la calificación “No Presentado” no es posible una vez que una práctica/trabajo ha sido evaluada.

## 1.3 Fuentes de información:

### 1.3.1 Básica

- Daroczi, G. (2015). Mastering Data Analysis with R. Packt Publishing
- Grolmund, G. y Wickham, H. (2016). R for Data Science. <https://r4ds.had.co.nz/> & O'Reilly

- Silberschatz, A., Korth, H. y Sudarshan, S. (2014). Fundamentos de Bases de Datos. Mc Graw Hill
- Rubén Fernández Casal (R Machinery):
  - Introducción al Análisis de Datos con R (con Javier Roca y Julián Costa)
  - Ayuda y Recursos para el Aprendizaje de R
  - Escritura de libros con el paquete bookdown (con Tomás Cotos)
  - Apéndice introducción a Rmarkdown
  - Presentación análisis de datos con R

### 1.3.2 Complementaria:

- Wes McKinney (2017). Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython. O'Reilly (2ª ed.)
- Tom White (2015). Hadoop: The Definitive Guide. O'Reilly (4ª ed.)
- Alex Holmes (2014). Hadoop in practice. Manning (2ª ed.)
- Centro de Supercomputación de Galicia (2019). Servicio de Big Data del CESGA. <https://bigdata.cesga.es/>

## Capítulo 2

# Manipulación de datos con R

La mayoría de los estudios estadísticos requieren disponer de un conjunto de datos.

### 2.1 Lectura, importación y exportación de datos

Además de la introducción directa, R es capaz de importar datos externos en múltiples formatos:

- bases de datos disponibles en librerías de R
- archivos de texto en formato ASCII
- archivos en otros formatos: Excel, SPSS, ...
- bases de datos relacionales: MySQL, Oracle, ...
- formatos web: HTML, XML, JSON, ...
- ....

#### 2.1.1 Formato de datos de R

El formato de archivo en el que habitualmente se almacena objetos (datos) R es binario y está comprimido (en formato "gzip" por defecto). Para cargar un fichero de datos se emplea normalmente `load()`:

```
res <- load("data/empleados.RData")
res
```

```
## [1] "empleados"
```

```
ls()
```

```
## [1] "citefig" "citefig2" "empleados" "fig.path" "inline"
## [6] "inline2" "is_html" "is_latex" "latexfig" "latexfig2"
## [11] "res"
```

y para guardar `save()`:

```
# Guardar
save(empleados, file = "data/empleados_new.RData")
```

Aunque, como indica este comando en la ayuda (`?save`):

For saving single R objects, `saveRDS()` is mostly preferable to `save()`, notably because of the functional nature of `readRDS()`, as opposed to `load()`.

```
saveRDS(empleados, file = "data/empleados_new.rds")
## restore it under a different name
empleados2 <- readRDS("data/empleados_new.rds")
# identical(empleados, empleados2)
```

El objeto empleado normalmente en R para almacenar datos en memoria es el `data.frame`.

### 2.1.2 Acceso a datos en paquetes

R dispone de múltiples conjuntos de datos en distintos paquetes, especialmente en el paquete `datasets` que se carga por defecto al abrir R. Con el comando `data()` podemos obtener un listado de las bases de datos disponibles.

Para cargar una base de datos concreta se utiliza el comando `data(nombre)` (aunque en algunos casos se cargan automáticamente al emplearlos). Por ejemplo, `data(cars)` carga la base de datos llamada `cars` en el entorno de trabajo (`".GlobalEnv"`) y `?cars` muestra la ayuda correspondiente con la descripción de la base de datos.

### 2.1.3 Lectura de archivos de texto

En R para leer archivos de texto se suele utilizar la función `read.table()`. Supóngase, por ejemplo, que en el directorio actual está el fichero `empleados.txt`. La lectura de este fichero vendría dada por el código:

```
# Session > Set Working Directory > To Source...?
datos <- read.table(file = "data/empleados.txt", header = TRUE)
# head(datos)
str(datos)
```

```
## 'data.frame': 474 obs. of 10 variables:
## $ id : int 1 2 3 4 5 6 7 8 9 10 ...
```

```
## $ sexo      : Factor w/ 2 levels "Hombre","Mujer": 1 1 2 2 1 1 1 2 2 2 ...
## $ fechnac   : Factor w/ 462 levels " ","1/10/1964",...: 166 275 362 228 176 397 240 290 36 143 ...
## $ educ      : int   15 16 12 8 15 15 12 15 12 ...
## $ catlab    : Factor w/ 3 levels "Administrativo",...: 2 1 1 1 1 1 1 1 1 ...
## $ salario   : num   57000 40200 21450 21900 45000 ...
## $ salini     : int   27000 18750 12000 13200 21000 13500 18750 9750 12750 13500 ...
## $ tiempemp  : int    98 98 98 98 98 98 98 98 98 ...
## $ expprev   : int   144 36 381 190 138 67 114 0 115 244 ...
## $ minoria   : Factor w/ 2 levels "No","Sí": 1 1 1 1 1 1 1 1 1 1 ...
```

Si el fichero estuviese en el directorio `c:\datos` bastaría con especificar `file = "c:/datos/empleados.txt"`. Nótese también que para la lectura del fichero anterior se ha establecido el argumento `header=TRUE` para indicar que la primera línea del fichero contiene los nombres de las variables.

Los argumentos utilizados habitualmente para esta función son:

- **header**: indica si el fichero tiene cabecera (`header=TRUE`) o no (`header=FALSE`). Por defecto toma el valor `header=FALSE`.
- **sep**: carácter separador de columnas que por defecto es un espacio en blanco (`sep=" "`). Otras opciones serían: `sep=","` si el separador es un “;”, `sep="*"` si el separador es un “\*”, etc.
- **dec**: carácter utilizado en el fichero para los números decimales. Por defecto se establece `dec = "."`. Si los decimales vienen dados por “,” se utiliza `dec = ","`.

Resumiendo, los (principales) argumentos por defecto de la función `read.table` son los que se muestran en la siguiente línea:

```
read.table(file, header = FALSE, sep = " ", dec = ".")
```

Para más detalles sobre esta función véase `help(read.table)`.

Están disponibles otras funciones con valores por defecto de los parámetros adecuados para otras situaciones. Por ejemplo, para ficheros separados por tabuladores se puede utilizar `read.delim()` o `read.delim2()`:

```
read.delim(file, header = TRUE, sep = "\t", dec = ".")
read.delim2(file, header = TRUE, sep = "\t", dec = ",")
```

### 2.1.4 Alternativa tidyverse

Para leer archivos de texto en distintos formatos también se puede emplear el paquete `readr` (colección `tidyverse`), para lo que se recomienda consultar el Capítulo 11 del libro *R for Data Science*.

### 2.1.5 Importación desde SPSS

El programa R permite lectura de ficheros de datos en formato SPSS (extensión *.sav*) sin necesidad de tener instalado dicho programa en el ordenador. Para ello se necesita:

- cargar la librería `foreign`
- utilizar la función `read.spss`

Por ejemplo:

```
library(foreign)
datos <- read.spss(file = "data/Employee data.sav", to.data.frame = TRUE)
# head(datos)
str(datos)
```

```
## 'data.frame': 474 obs. of 10 variables:
## $ id : num 1 2 3 4 5 6 7 8 9 10 ...
## $ sexo : Factor w/ 2 levels "Hombre","Mujer": 1 1 2 2 1 1 1 2 2 2 ...
## $ fechnac : num 1.17e+10 1.19e+10 1.09e+10 1.15e+10 1.17e+10 ...
## $ educ : Factor w/ 10 levels "8","12","14",...: 4 5 2 1 4 4 4 2 4 2 ...
## $ catlab : Factor w/ 3 levels "Administrativo",...: 3 1 1 1 1 1 1 1 1 1 ...
## $ salario : Factor w/ 221 levels "15750","15900",...: 179 137 28 31 150 101 121 31 ...
## $ salini : Factor w/ 90 levels "9000","9750",...: 60 42 13 21 48 23 42 2 18 23 ...
## $ tiempemp: Factor w/ 36 levels "63","64","65",...: 36 36 36 36 36 36 36 36 36 ...
## $ expprev : Factor w/ 208 levels "Ausente","10",...: 38 131 139 64 34 181 13 1 14 9 ...
## $ minoria : Factor w/ 2 levels "No","Si": 1 1 1 1 1 1 1 1 1 1 ...
## - attr(*, "variable.labels")= Named chr "Código de empleado" "Sexo" "Fecha de nac..."
## ..- attr(*, "names")= chr "id" "sexo" "fechnac" "educ" ...
## - attr(*, "codepage")= int 1252
```

**Nota:** Si hay fechas, puede ser recomendable emplear la función `spss.get()` del paquete `Hmisc`.

### 2.1.6 Importación desde Excel

Se pueden leer ficheros de Excel (con extensión *.xlsx*) utilizando por ejemplo los paquetes `openxlsx`, `readxl` (colección `tidyverse`), `XLConnect` o `RODBC` (este paquete se empleará más adelante para acceder a bases de datos), entre otros.

Por ejemplo el siguiente código implementa una función que permite leer todos los archivos en formato *.xlsx* en un directorio:

```
library(openxlsx)

read_xlsx <- function(path = '.') {
  files <- dir(path, pattern = '*.xlsx') # list.files
  # file.list <- lapply(files, readWorkbook)
  file.list <- vector(length(files), mode = 'list')
```

```
for (i in seq_along(files))
  file.list[[i]] <- readWorkbook(files[i])
file.names <- sub('\\.xlsx$', '', basename(files))
names(file.list) <- file.names
file.list
}
```

Para combinar los archivos (suponiendo que tienen las mismas columnas), podríamos ejecutar una llamada a `rbind()` o emplear la función `bind_rows()` del paquete `dplyr`:

```
df <- do.call('rbind', file.list)

df <- dplyr::bind_rows(file.list)
```

Sin embargo, un procedimiento sencillo consiste en exportar los datos desde Excel a un archivo de texto separado por tabuladores (extensión `.csv`). Por ejemplo, supongamos que queremos leer el fichero `coches.xls`:

- Desde Excel se selecciona el menú **Archivo -> Guardar como -> Guardar como** y en **Tipo** se escoge la opción de archivo CSV. De esta forma se guardarán los datos en el archivo `coches.csv`.
- El fichero `coches.csv` es un fichero de texto plano (se puede editar con Notepad), con cabecera, las columnas separadas por “;”, y siendo “,” el carácter decimal.
- Por lo tanto, la lectura de este fichero se puede hacer con:

```
datos <- read.table("coches.csv", header = TRUE, sep = ";", dec = ",")
```

Otra posibilidad es utilizar la función `read.csv2`, que es una adaptación de la función general `read.table` con las siguientes opciones:

```
read.csv2(file, header = TRUE, sep = ";", dec = ",")
```

Por lo tanto, la lectura del fichero `coches.csv` se puede hacer de modo más directo con:

```
datos <- read.csv2("coches.csv")
```

### 2.1.7 Exportación de datos

Puede ser de interés la exportación de datos para que puedan leídos con otros programas. Para ello, se puede emplear la función `write.table()`. Esta función es similar, pero funcionando en sentido inverso, a `read.table()` (Sección ??).

Veamos un ejemplo:

```
tipo <- c("A", "B", "C")
longitud <- c(120.34, 99.45, 115.67)
datos <- data.frame(tipo, longitud)
datos
```

```
##   tipo longitud
## 1    A   120.34
## 2    B    99.45
## 3    C   115.67
```

Para guardar el data.frame `datos` en un fichero de texto se puede utilizar:

```
write.table(datos, file = "datos.txt")
```

Otra posibilidad es utilizar la función:

```
write.csv2(datos, file = "datos.csv")
```

que dará lugar al fichero `datos.csv` importable directamente desde Excel.

## 2.2 Manipulación de datos

Una vez cargada una (o varias) bases de datos hay una series de operaciones que serán de interés para el tratamiento de datos:

- Operaciones con variables:
  - crear
  - recodificar (e.g. categorizar)
  - ...
- Operaciones con casos:
  - ordenar
  - filtrar
  - ...
- Operaciones con tablas de datos:
  - unir
  - combinar
  - consultar
  - ...

A continuación se tratan algunas operaciones *básicas*.

### 2.2.1 Operaciones con variables

#### 2.2.1.1 Creación (y eliminación) de variables

Consideremos de nuevo la base de datos `cars` incluida en el paquete `datasets`:



```
data(cars)
# str(cars)
head(cars)
```

```
##   speed dist
## 1     4    2
## 2     4   10
## 3     7    4
## 4     7   22
## 5     8   16
## 6     9   10
```

Utilizando el comando `help(cars)` se obtiene que `cars` es un `data.frame` con 50 observaciones y dos variables:

- `speed`: Velocidad (millas por hora)
- `dist`: tiempo hasta detenerse (pies)

Recordemos que, para acceder a la variable `speed` se puede hacer directamente con su nombre o bien utilizando notación “matricial”.

```
cars$speed
```

```
## [1]  4  4  7  7  8  9 10 10 10 10 11 11 12 12 12 12 13 13 13 13 14 14 14 14
## [24] 15 15 15 16 16 17 17 17 18 18 18 18 19 19 19 20 20 20 20 20 22 23 24
## [47] 24 24 24 25
```

```
cars[, 1] # Equivalente
```

```
## [1]  4  4  7  7  8  9 10 10 10 10 11 11 12 12 12 12 13 13 13 13 14 14 14 14
## [24] 15 15 15 16 16 17 17 17 18 18 18 18 19 19 19 20 20 20 20 20 22 23 24
## [47] 24 24 24 25
```

Supongamos ahora que queremos transformar la variable original `speed` (millas por hora) en una nueva variable `velocidad` (kilómetros por hora) y añadir esta nueva variable al `data.frame` `cars`. La transformación que permite pasar millas a kilómetros es  $\text{kilómetros} = \text{millas} / 0.62137$  que en R se hace directamente con:

```
cars$speed/0.62137
```

Finalmente, incluimos la nueva variable que llamaremos `velocidad` en `cars`:

```
cars$velocidad <- cars$speed / 0.62137
head(cars)
```

```
##   speed dist velocidad
## 1     4    2  6.437388
## 2     4   10  6.437388
## 3     7    4 11.265430
```

```
## 4      7    22 11.265430
## 5      8    16 12.874777
## 6      9    10 14.484124
```

También transformaremos la variable `dist` (en pies) en una nueva variable `distancia` (en metros). Ahora la transformación deseada es `metros=pies/3.2808`:

```
cars$distancia <- cars$dis / 3.2808
head(cars)
```

```
##   speed dist velocidad distancia
## 1     4     2  6.437388 0.6096074
## 2     4    10  6.437388 3.0480371
## 3     7     4 11.265430 1.2192148
## 4     7    22 11.265430 6.7056815
## 5     8    16 12.874777 4.8768593
## 6     9    10 14.484124 3.0480371
```

Ahora, eliminaremos las variables originales `speed` y `dist`, y guardaremos el `data.frame` resultante con el nombre `coches`. En primer lugar, veamos varias formas de acceder a las variables de interés:

```
cars[, c(3, 4)]
cars[, c("velocidad", "distancia")]
cars[, -c(1, 2)]
```

Utilizando alguna de las opciones anteriores se obtiene el `data.frame` deseado:

```
coches <- cars[, c("velocidad", "distancia")]
# head(coches)
str(coches)
```

```
## 'data.frame':   50 obs. of  2 variables:
## $ velocidad: num  6.44 6.44 11.27 11.27 12.87 ...
## $ distancia: num  0.61 3.05 1.22 6.71 4.88 ...
```

Finalmente los datos anteriores podrían ser guardados en un fichero exportable a Excel con el siguiente comando:

```
write.csv2(coches, file = "coches.csv")
```

## 2.2.2 Operaciones con casos

### 2.2.2.1 Ordenación

Continuemos con el `data.frame` `cars`. Se puede comprobar que los datos disponibles están ordenados por los valores de `speed`. A continuación haremos la ordenación utilizando los valores de `dist`. Para ello utilizaremos el conocido como vector de índices de ordenación. Este vector establece el orden en que tienen

que ser elegidos los elementos para obtener la ordenación deseada. Veamos un ejemplo sencillo:

```
x <- c(2.5, 4.3, 1.2, 3.1, 5.0) # valores originales
ii <- order(x)
ii    # vector de ordenación
```

```
## [1] 3 1 4 2 5
```

```
x[ii] # valores ordenados
```

```
## [1] 1.2 2.5 3.1 4.3 5.0
```

En el caso de vectores, el procedimiento anterior se podría hacer directamente con:

```
sort(x)
```

Sin embargo, para ordenar data.frames será necesario la utilización del vector de índices de ordenación. A continuación, los datos de `cars` ordenados por `dist`:

```
ii <- order(cars$dist) # Vector de índices de ordenación
cars2 <- cars[ii, ]    # Datos ordenados por dist
head(cars2)
```

```
##   speed dist velocidad distancia
## 1     4    2  6.437388  0.6096074
## 3     7    4 11.265430  1.2192148
## 2     4   10  6.437388  3.0480371
## 6     9   10 14.484124  3.0480371
## 12    12   14 19.312165  4.2672519
## 5     8   16 12.874777  4.8768593
```

### 2.2.2.2 Filtrado

El filtrado de datos consiste en elegir una submuestra que cumpla determinadas condiciones. Para ello se puede utilizar la función `subset()` (que además permite seleccionar variables).

A continuación se muestran un par de ejemplos:

```
subset(cars, dist > 85) # datos con dist>85
```

```
##   speed dist velocidad distancia
## 47    24   92  38.62433  28.04194
## 48    24   93  38.62433  28.34674
## 49    24  120  38.62433  36.57644
```

```
subset(cars, speed > 10 & speed < 15 & dist > 45) # speed en (10,15) y dist>45
```

```
##   speed dist velocidad distancia
## 19    13   46  20.92151  14.02097
```

```
## 22    14    60  22.53086  18.28822
## 23    14    80  22.53086  24.38430
```

También se pueden hacer el filtrado empleando directamente los correspondientes vectores de índices:

```
ii <- cars$dist > 85
cars[ii, ] # dis>85
```

```
##      speed dist velocidad distancia
## 47      24   92  38.62433  28.04194
## 48      24   93  38.62433  28.34674
## 49      24  120  38.62433  36.57644
```

```
ii <- cars$speed > 10 & cars$speed < 15 & cars$dist > 45
cars[ii, ] # speed en (10,15) y dist>45
```

```
##      speed dist velocidad distancia
## 19      13   46  20.92151  14.02097
## 22      14   60  22.53086  18.28822
## 23      14   80  22.53086  24.38430
```

En este caso puede ser de utilidad la función `which()`:

```
it <- which(ii)
str(it)
```

```
## int [1:3] 19 22 23
cars[it, 1:2]
```

```
##      speed dist
## 19      13   46
## 22      14   60
## 23      14   80
# rownames(cars[it, 1:2])
```

```
id <- which(!ii)
str(cars[id, 1:2])
```

```
## 'data.frame':   47 obs. of  2 variables:
## $ speed: num  4 4 7 7 8 9 10 10 10 11 ...
## $ dist : num  2 10 4 22 16 10 18 26 34 17 ...
```

```
# Se podría p.e. emplear cars[id, ] para predecir cars[it, ]$speed
# ?which.min
```

## 2.2.3 Funciones `apply`

### 2.2.3.1 La función `apply`

Una forma de evitar la utilización de bucles es utilizando la sentencia `apply` que permite evaluar una misma función en todas las filas, columnas, etc. de un array de forma simultánea.

La sintaxis de esta función es:

```
apply(X, MARGIN, FUN, ...)
```

- `X`: matriz (o array)
- `MARGIN`: Un vector indicando las dimensiones donde se aplicará la función. 1 indica filas, 2 indica columnas, y `c(1,2)` indica filas y columnas.
- `FUN`: función que será aplicada.
- `...`: argumentos opcionales que serán usados por `FUN`.

Veamos la utilización de la función `apply` con un ejemplo:

```
x <- matrix(1:9, nrow = 3)
x
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

```
apply(x, 1, sum)      # Suma por filas
```

```
## [1] 12 15 18
```

```
apply(x, 2, sum)      # Suma por columnas
```

```
## [1]  6 15 24
```

```
apply(x, 2, min)      # Mínimo de las columnas
```

```
## [1]  1  4  7
```

```
apply(x, 2, range)    # Rango (mínimo y máximo) de las columnas
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    3    6    9
```

### 2.2.3.2 Variantes de la función `apply`

`lapply()`:

```
# lista con las medianas de las variables
list <- lapply(cars, median)
str(list)
```

```
## List of 4
## $ speed      : num 15
## $ dist       : num 36
## $ velocidad: num 24.1
## $ distancia: num 11

sapply():
# matriz con las medias, medianas y desv. de las variables
res <- sapply(cars,
              function(x) c(mean = mean(x), median = median(x), sd = sd(x)))
# str(res)
res

##           speed      dist velocidad distancia
## mean   15.400000 42.98000 24.783945 13.100463
## median 15.000000 36.00000 24.140206 10.972933
## sd      5.287644 25.76938  8.509655  7.854602
knitr::kable(t(res), digits = 1)
```

	mean	median	sd
speed	15.4	15.0	5.3
dist	43.0	36.0	25.8
velocidad	24.8	24.1	8.5
distancia	13.1	11.0	7.9

```
cfuns <- function(x, funs = c(mean, median, sd))
  sapply(funs, function(f) f(x))
x <- 1:10
cfuns(x)

## [1] 5.50000 5.50000 3.02765
sapply(cars, cfuns)

##           speed      dist velocidad distancia
## [1,] 15.400000 42.98000 24.783945 13.100463
## [2,] 15.000000 36.00000 24.140206 10.972933
## [3,]  5.287644 25.76938  8.509655  7.854602
nfuns <- c("mean", "median", "sd")
sapply(nfuns, function(f) eval(parse(text = paste0(f, "(x)"))))

##      mean median      sd
## 5.50000 5.50000 3.02765
```

### 2.2.3.3 La función tapply

La función `tapply()` es similar a la función `apply()` y permite aplicar una función a los datos desagregados, utilizando como criterio los distintos niveles

de una variable factor. La sintaxis de esta función es como sigue:

```
tapply(X, INDEX, FUN, ...)
```

- X: matriz (o array).
- INDEX: factor indicando los grupos (niveles).
- FUN: función que será aplicada.
- ...: argumentos opcionales .

Consideremos, por ejemplo, el data.frame `ChickWeight` con datos de un experimento relacionado con la repercusión de varias dietas en el peso de pollos.

```
data(ChickWeight)
# str(ChickWeight)
head(ChickWeight)
```

```
##   weight Time Chick Diet
## 1     42    0     1    1
## 2     51    2     1    1
## 3     59    4     1    1
## 4     64    6     1    1
## 5     76    8     1    1
## 6     93   10     1    1
```

```
peso <- ChickWeight$weight
dieta <- ChickWeight$Diet
levels(dieta) <- c("Dieta 1", "Dieta 2", "Dieta 3", "Dieta 4")
tapply(peso, dieta, mean) # Peso medio por dieta
```

```
## Dieta 1 Dieta 2 Dieta 3 Dieta 4
## 102.6455 122.6167 142.9500 135.2627
```

```
tapply(peso, dieta, summary)
```

```
## $`Dieta 1`
##   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   35.00   57.75   88.00  102.65  136.50  305.00
##
## $`Dieta 2`
##   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   39.0   65.5   104.5   122.6   163.0   331.0
##
## $`Dieta 3`
##   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   39.0   67.5   125.5   142.9   198.8   373.0
##
## $`Dieta 4`
##   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   39.00   71.25  129.50  135.26  184.75  322.00
```

Otro ejemplo:

```
provincia <- as.factor(c(1, 3, 4, 2, 4, 3, 2, 1, 4, 3, 2))
levels(provincia) = c("A Coruña", "Lugo", "Orense", "Pontevedra")
hijos <- c(1, 2, 0, 3, 4, 1, 0, 0, 2, 3, 1)
data.frame(provincia, hijos)

##      provincia hijos
## 1     A Coruña     1
## 2       Orense     2
## 3 Pontevedra     0
## 4         Lugo     3
## 5 Pontevedra     4
## 6       Orense     1
## 7         Lugo     0
## 8     A Coruña     0
## 9 Pontevedra     2
## 10      Orense     3
## 11        Lugo     1

tapply(hijos, provincia, mean) # Número medio de hijos por provincia

##      A Coruña      Lugo      Orense Pontevedra
##      0.500000      1.333333      2.000000      2.000000
```

### 2.2.4 Operaciones con tablas de datos

Ver ejemplo *wosdata.R*

**Unir tablas:**

`rbind()`

`cbind()`

**Combinar tablas:**

`match()`

`match(x, table)` devuelve un vector (de la misma longitud que `x`) con las (primeras) posiciones de coincidencia de `x` en `table` (o NA, por defecto, si no hay coincidencia). Para combinar tablas puede ser más cómodo el operador `%in%` (`? '%in%'`).

`pmatch()`



## Capítulo 3

# Introducción al lenguaje SQL

Working draft...

Los sistemas de información gestionan repositorios de información en múltiples formatos, siendo el más popular las bases de datos relacionales a las que se accede mediante SQL (Structured Query Language)

### 3.1 Bases de Datos Relacionales

#### 3.1.1 Definiciones

- **Dominio:** contexto (organización, empresa, evento...) objeto de gestión de la información.
- **Dato:** hecho con significado implícito, registable, relevante en un determinado dominio.
- **Base de datos:** colección de datos de un determinado dominio relacionados entre sí, organizados de forma que sea posible manipularlos y recuperarlos de forma eficiente.
- Sistema de Gestión de Bases de Datos (**SGBD**) (en inglés **RDBMS**, Relational Database Management System): software que permite a los usuarios crear y manipular bases de datos mediante operaciones CRUD:
  - Crear / Insertar Datos (Create)
  - Consultar / Leer (Read)
  - Actualizar / Modificar (Update)
  - Eliminar (Delete)

- **Modelo de datos:** abstracción conceptual que propone una manera de organizar y manipular los datos. Definido mediante:
  - Estructura: elementos para organizar datos
  - Integridad: reglas para relaciones los elementos
  - Manipulación: operaciones sobre los datos adaptadas a la estructura y reglas
- Modelo de datos conceptual **Entidad Relación** (entidades, relaciones, atributos)
- Modelo de datos lógico o de representación (**modelo relacional** de Codd)
  - Datos en relaciones (tablas)
  - Base matemática formal
  - Flexible
- Modelo de datos físico (tal y como se almacenan los datos)

Una fila de la tabla (relación) es una tupla y una columna un atributo (ver Figura ??).

(ver Figura ??)

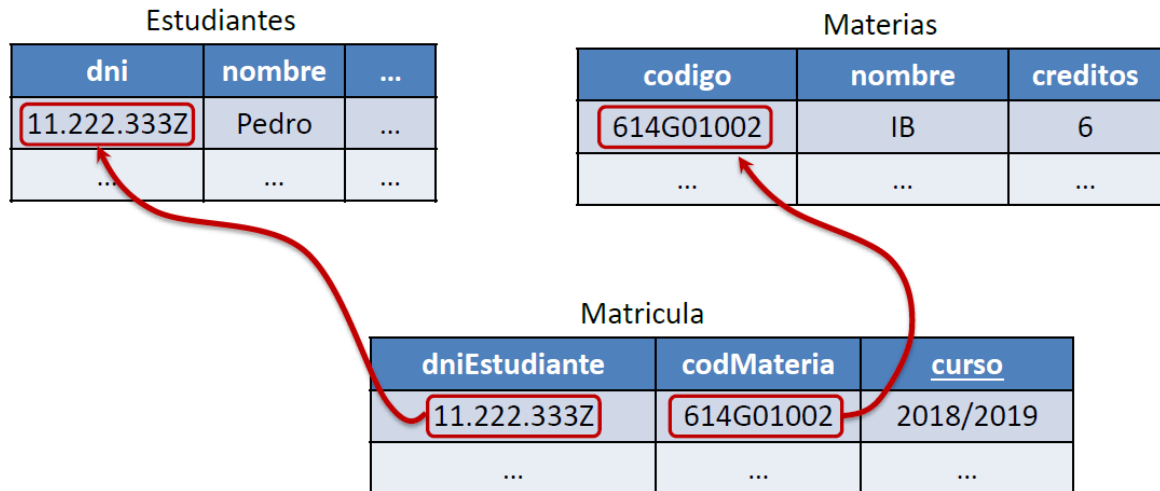
(ver Figura ??)

El diagrama muestra una tabla con el título 'Estudiantes' en un encabezado azul. Las columnas son 'dni', 'nombre', 'apellidos' y 'email'. Las filas de datos son: (11.222.333Z, Pedro, Pérez Pérez, infppp@udc.es), (22.333.444Z, Ana, López Pérez, infalp@udc.es) y (11.888.999Z, Alberto, López López, infall@udc.es). Las flechas indican: 'Atributos' para las columnas, 'Nombre de la relación' para el título, y 'Tuplas' para las filas de datos.

Estudiantes			
<u>dni</u>	nombre	apellidos	email
11.222.333Z	Pedro	Pérez Pérez	infppp@udc.es
22.333.444Z	Ana	López Pérez	infalp@udc.es
11.888.999Z	Alberto	López López	infall@udc.es

Figura 3.1: Esquema de una relación.

Una base de datos es un conjunto de tablas (al menos una).



La tabla no es una relación porque la relación es un conjunto sin orden y una tabla puede tener filas repetidas y tiene orden.

- 
- **Esquema:** estructura de la base de datos
  - **Estado:** contenido de la base de datos
  - Restricción de **integridad:** regla que debe cumplir la información registrada en la base de datos para garantizar la integridad de la información.

Cualquier Base de Datos basada en el modelo relacional ha de cumplir como mínimo estas restricciones (además de las propias del dominio):

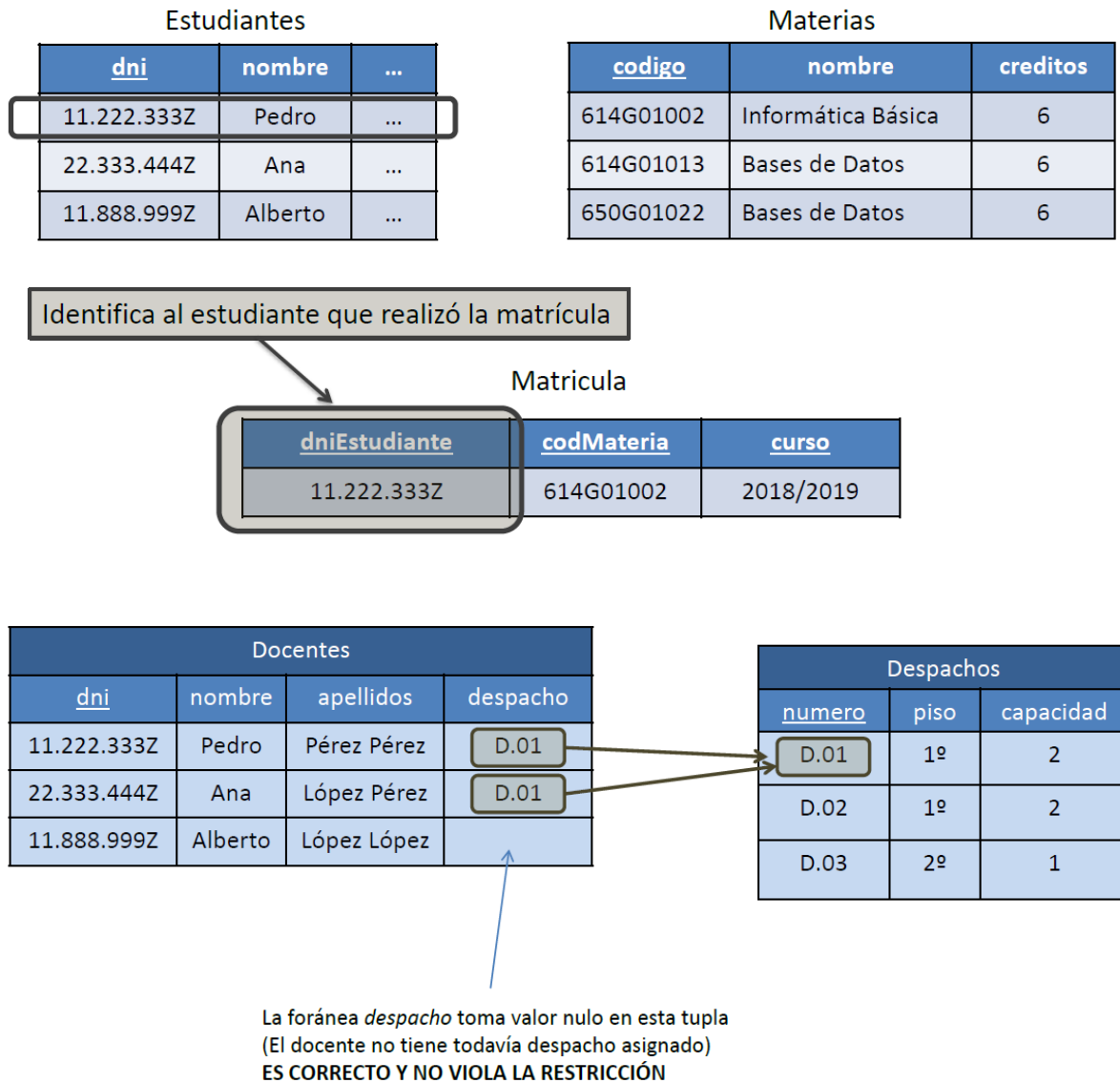
- Restricción de dominio: el valor de cada atributo debe de ser único (teléfono, no valor único), no descomponible (nombre completo descomponible en nombre y apellidos, domicilio en calle, CP, localidad, etc...)
- Una relación es un conjunto de tuplas, por tanto todas las tuplas son distintas.
- Una **superclave** es un subconjunto de atributos tal que no existen dos tuplas con la misma superclave.

Ejercicio. En la relación Empleado(dni, nombre, apellidos, email)  
¿cuántas superclaves existen?

- Una **clave candidata** es una superclave mínima (superclave mínima es la clave a la que no se le puede eliminar un atributo).

¿Cuántas claves candidatas hay en el ejemplo anterior?

- **Clave primaria** es la clave candidata que elegimos que identificar de forma unívoca las tuplas de una relación. Restricción de integridad de entidad: Ningún valor de la clave primaria puede ser un valor nulo.
- **Clave foránea** es un conjunto de atributos de una relación R\_1 que, para cada tupla, identifican a otra tupla de una relación R\_2 con la que está relacionada. La Restricción de integridad referencial nos dice que la clave foránea ha de corresponderse con la clave primaria de R\_2, y si la clave foránea no es nula ha de referir a una tupla en R\_2.



Si borramos/actualizamos un valor de clave foránea podemos: (a) prohibir el cambio, o (b) poner a nulo la clave foránea (borrado) o propagar el cambio (modificación).

- 
- Ventajas de SGBD:
    - Administración centralizada de los datos (por un administrador en un servidor/plataforma central que evita la información en silos - redundante/inconsistente)
    - Desacoplado del almacenamiento físico de los datos (no es necesario conocerlo)
    - Simplicidad de acceso (ODBC + SQL, lenguaje declarativo)
    - Control de integridad (restricciones genéricas, integridad de entidad y referencial, de dominio, y las del dominio en software)
    - Control de acceso concurrente (evita inconsistencia)
    - Seguridad (autenticación, roles de acceso)
    - Recuperación ante fallos (backup, logs y transacciones -rollback-)

## 3.2 Sintaxis SQL

A continuación 27 cláusulas SQL básicas

### 3.2.1 Extracción SQL (11 statements)

```
SELECT column1, column2....columnN
FROM table_name;

SELECT DISTINCT column1, column2....columnN
FROM table_name;

SELECT column1, column2....columnN
FROM table_name
WHERE CONDITION;

SELECT column1, column2....columnN
FROM table_name
WHERE CONDITION-1 {AND|OR} CONDITION-2;

SELECT column1, column2....columnN
FROM table_name
WHERE column_name IN (val-1, val-2,...val-N);

SELECT column1, column2....columnN
FROM table_name
WHERE column_name BETWEEN val-1 AND val-2;
```

```

SELECT column1, column2....columnN
FROM   table_name
WHERE  column_name LIKE { PATTERN };

SELECT column1, column2....columnN
FROM   table_name
WHERE  CONDITION
ORDER BY column_name {ASC|DESC};

SELECT SUM(column_name)
FROM   table_name
WHERE  CONDITION
GROUP BY column_name;

SELECT COUNT(column_name)
FROM   table_name
WHERE  CONDITION;

SELECT SUM(column_name)
FROM   table_name
WHERE  CONDITION
GROUP BY column_name
HAVING (arithmetic function condition);

```

### 3.2.2 Crear/Actualizar/Borrar tablas SQL (8 statements)

```

CREATE TABLE table_name(
column1 datatype,
column2 datatype,
column3 datatype,
.....
columnN datatype,
PRIMARY KEY( one or more columns )
);

DROP TABLE table_name;

CREATE UNIQUE INDEX index_name
ON table_name ( column1, column2,...columnN);

ALTER TABLE table_name
DROP INDEX index_name;

DESC table_name;

```

```
TRUNCATE TABLE table_name;

ALTER TABLE table_name {ADD|DROP|MODIFY} column_name {data_type};

ALTER TABLE table_name RENAME TO new_table_name;
```

### 3.2.3 Añadir/Actualizar/Borrar tuplas en SQL (3 statements)

```
INSERT INTO table_name( column1, column2....columnN)
VALUES ( value1, value2....valueN);

UPDATE table_name
SET column1 = value1, column2 = value2....columnN=valueN
[ WHERE CONDITION ];

DELETE FROM table_name
WHERE {CONDITION};
```

### 3.2.4 Gestión Bases de Datos (5 statements)

```
CREATE DATABASE database_name;

DROP DATABASE database_name;

USE database_name;

COMMIT;

ROLLBACK;
```

### 3.2.5 Ejemplos de consultas SQL

```
SELECT Nombre, Apellido1, Apellido2, Municipio, Provincia
FROM Cliente
WHERE Municipio = 'Lugo'
ORDER BY Apellido1

INSERT Proveedor(Nombre, PersonaContacto, Ciudad, País)
VALUES ('Café Candelas', 'Ivana Candelas', 'Lugo', 'España')

UPDATE Pedidos
SET Cantidad = 2
```

```
WHERE IdProducto = 963

DELETE Cliente
WHERE Email = 'alexandregb@gmail.com'
```

### 3.3 Conexión con bases de datos desde R

#### 3.3.1 Introducción a SQL en R

SQL se usa para manipular datos dentro de una base de datos. Si la base de datos no es muy grande se puede cargar toda en un `data.frame`. No obstante, por escalabilidad y offloading de la carga de trabajo al servidor SGBD utilizaremos SQL.

Existen varios SGBD (SQLite, Microsoft SQL Server, PostgreSQL, etc) los cuales comparten el soporte de SQL (en concreto ANSI SQL) aunque cada gestor extiende SQL de formas sutiles buscando minar cierta portabilidad de código (*vendor-locking*). En efecto, un código SQL desarrollado para SQLite es probable que falle con MySQL aunque tras aplicar ligeras modificaciones ya funcionará. Asimismo el mecanismo de conexión, configuración, rendimiento y operación suele diferir entre SGBD.

A continuación se lista una serie de paquetes utilizados en el acceso a los datos, lo que suele ser el principal esfuerzo a realizar cuando se trabaja con SGBD:

- DBI
- RODBC
- dbConnect
- RSQLite
- RMySQL
- RPostgreSQL

#### 3.3.2 El paquete sqldf

A continuación se presenta una serie de ejercicios con la sintaxis de SQL operando sobre un `data.frame` con el paquete `sqldf`. Esto inicialmente no incluye los detalles de conectarse a un SGBD, ni modificar los datos, solamente el uso de SQL para extraer datos con el objetivo de ser analizados en R.

```
library(sqldf)
```

```
sqldf('SELECT age, circumference FROM Orange WHERE Tree = 1 ORDER BY circumference ASC')
```

```
##      age circumference
## 1  118             30
## 2  484             58
## 3  664             87
```



```
## 4 1004      115
## 5 1231      120
## 6 1372      142
## 7 1582      145
```

### 3.3.3 SQL Queries

El comando inicial es SELECT. SQL no es case-sensitive, por lo que esto va a funcionar:

```
sqldf("SELECT * FROM iris")
sqldf("select * from iris")
```

pero lo siguiente no va a funcionar (a menos que tengamos un objeto IRIS:

```
sqldf("SELECT * FROM IRIS")
```

La sintaxis básica de SELECT es:

```
SELECT variable1, variable2 FROM data
```

#### 3.3.3.1 Asterisco/Wildcard

Lo extrae todo

```
bod2 <- sqldf('SELECT * FROM BOD')
```

#### 3.3.3.2 Limit

Limita el número de resultados

```
sqldf('SELECT * FROM iris LIMIT 5')
```

```
## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1          3.5          1.4          0.2 setosa
## 2          4.9          3.0          1.4          0.2 setosa
## 3          4.7          3.2          1.3          0.2 setosa
## 4          4.6          3.1          1.5          0.2 setosa
## 5          5.0          3.6          1.4          0.2 setosa
```

#### 3.3.3.3 Order By

Ordena las variables

```
ORDER BY var1 {ASC/DESC}, var2 {ASC/DESC}
```

```
sqldf("SELECT * FROM Orange ORDER BY age ASC, circumference DESC LIMIT 5")
```

```
## Tree age circumference
## 1    2 118           33
```

```
## 2    4 118          32
## 3    1 118          30
## 4    3 118          30
## 5    5 118          30
```

### 3.3.3.4 Where

Sentencias condicionales, donde se puede incorporar operadores lógicos AND y OR, expresando el orden de evaluación con paréntesis en caso de ser necesario.

```
sqldf('SELECT demand FROM BOD WHERE Time < 3')
```

```
##    demand
## 1      8.3
## 2     10.3
```

```
sqldf('SELECT * FROM rock WHERE (peri > 5000 AND shape < .05) OR perm > 1000')
```

```
##    area    peri    shape perm
## 1 5048  941.543 0.328641 1300
## 2 1016  308.642 0.230081 1300
## 3 5605 1145.690 0.464125 1300
## 4 8793 2280.490 0.420477 1300
```

Y extendiendo su uso con IN o LIKE (es último sólo con %), pudiendo aplicárseles el NOT:

```
sqldf('SELECT * FROM BOD WHERE Time IN (1,7)')
```

```
##    Time demand
## 1     1      8.3
## 2     7     19.8
```

```
sqldf('SELECT * FROM BOD WHERE Time NOT IN (1,7)')
```

```
##    Time demand
## 1     2     10.3
## 2     3     19.0
## 3     4     16.0
## 4     5     15.6
```

```
sqldf('SELECT * FROM chickwts WHERE feed LIKE "%bean" LIMIT 5')
```

```
##    weight    feed
## 1    179 horsebean
## 2    160 horsebean
## 3    136 horsebean
## 4    227 horsebean
## 5    217 horsebean
```

```
sqldf('SELECT * FROM chickwts WHERE feed NOT LIKE "%bean" LIMIT 5')
```

```
##  weight    feed
## 1    309 linseed
## 2    229 linseed
## 3    181 linseed
## 4    141 linseed
## 5    260 linseed
```

## 3.4 Ejemplo Scopus data

Ver ejemplo *citan.zip* y Apéndice ??.

“If your data fits in memory there is no advantage to putting it in a database: it will only be slower and more frustrating”

— Hadley Wickham – <https://dbplyr.tidyverse.org/articles/dbplyr.html>

## 3.5 Ejercicios SQL con RSQLite

### 3.5.1 Setup de RSQLite

Vamos a utilizar RSQLite. La información para su instalación está en el siguiente enlace.

```
library(DBI)

# Create an ephemeral in-memory RSQLite database
con <- dbConnect(RSQLite::SQLite(), ":memory:")
dbListTables(con)

## character(0)

dbWriteTable(con, "mtcars", mtcars)
dbListTables(con)

## [1] "mtcars"

dbListFields(con, "mtcars")

## [1] "mpg" "cyl" "disp" "hp" "drat" "wt" "qsec" "vs" "am" "gear"
## [11] "carb"

dbReadTable(con, "mtcars")

##      mpg  cyl  disp  hp drat   wt  qsec vs am gear carb
## 1  21.0    6 160.0 110  3.90 2.620 16.46  0  1    4    4
## 2  21.0    6 160.0 110  3.90 2.875 17.02  0  1    4    4
```

```
## 3 22.8 4 108.0 93 3.85 2.320 18.61 1 1 4 1
## 4 21.4 6 258.0 110 3.08 3.215 19.44 1 0 3 1
## 5 18.7 8 360.0 175 3.15 3.440 17.02 0 0 3 2
## 6 18.1 6 225.0 105 2.76 3.460 20.22 1 0 3 1
## 7 14.3 8 360.0 245 3.21 3.570 15.84 0 0 3 4
## 8 24.4 4 146.7 62 3.69 3.190 20.00 1 0 4 2
## 9 22.8 4 140.8 95 3.92 3.150 22.90 1 0 4 2
## 10 19.2 6 167.6 123 3.92 3.440 18.30 1 0 4 4
## 11 17.8 6 167.6 123 3.92 3.440 18.90 1 0 4 4
## 12 16.4 8 275.8 180 3.07 4.070 17.40 0 0 3 3
## 13 17.3 8 275.8 180 3.07 3.730 17.60 0 0 3 3
## 14 15.2 8 275.8 180 3.07 3.780 18.00 0 0 3 3
## 15 10.4 8 472.0 205 2.93 5.250 17.98 0 0 3 4
## 16 10.4 8 460.0 215 3.00 5.424 17.82 0 0 3 4
## 17 14.7 8 440.0 230 3.23 5.345 17.42 0 0 3 4
## 18 32.4 4 78.7 66 4.08 2.200 19.47 1 1 4 1
## 19 30.4 4 75.7 52 4.93 1.615 18.52 1 1 4 2
## 20 33.9 4 71.1 65 4.22 1.835 19.90 1 1 4 1
## 21 21.5 4 120.1 97 3.70 2.465 20.01 1 0 3 1
## 22 15.5 8 318.0 150 2.76 3.520 16.87 0 0 3 2
## 23 15.2 8 304.0 150 3.15 3.435 17.30 0 0 3 2
## 24 13.3 8 350.0 245 3.73 3.840 15.41 0 0 3 4
## 25 19.2 8 400.0 175 3.08 3.845 17.05 0 0 3 2
## 26 27.3 4 79.0 66 4.08 1.935 18.90 1 1 4 1
## 27 26.0 4 120.3 91 4.43 2.140 16.70 0 1 5 2
## 28 30.4 4 95.1 113 3.77 1.513 16.90 1 1 5 2
## 29 15.8 8 351.0 264 4.22 3.170 14.50 0 1 5 4
## 30 19.7 6 145.0 175 3.62 2.770 15.50 0 1 5 6
## 31 15.0 8 301.0 335 3.54 3.570 14.60 0 1 5 8
## 32 21.4 4 121.0 109 4.11 2.780 18.60 1 1 4 2
```

*# You can fetch all results:*

```
res <- dbSendQuery(con, "SELECT * FROM mtcars WHERE cyl = 4")
dbFetch(res)
```

```
##      mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## 1  22.8   4  108.0  93 3.85 2.320 18.61 1  1   4    1
## 2  24.4   4  146.7  62 3.69 3.190 20.00 1  0   4    2
## 3  22.8   4  140.8  95 3.92 3.150 22.90 1  0   4    2
## 4  32.4   4   78.7  66 4.08 2.200 19.47 1  1   4    1
## 5  30.4   4   75.7  52 4.93 1.615 18.52 1  1   4    2
## 6  33.9   4   71.1  65 4.22 1.835 19.90 1  1   4    1
## 7  21.5   4  120.1  97 3.70 2.465 20.01 1  0   3    1
## 8  27.3   4   79.0  66 4.08 1.935 18.90 1  1   4    1
## 9  26.0   4  120.3  91 4.43 2.140 16.70 0  1   5    2
## 10 30.4   4   95.1 113 3.77 1.513 16.90 1  1   5    2
```

```
## 11 21.4    4 121.0 109 4.11 2.780 18.60    1 1    4    2
```

```
dbClearResult(res)
```

```
# Or a chunk at a time
```

```
res <- dbSendQuery(con, "SELECT * FROM mtcars WHERE cyl = 4")
```

```
while(!dbHasCompleted(res)){
```

```
  chunk <- dbFetch(res, n = 5)
```

```
  print(nrow(chunk))
```

```
}
```

```
## [1] 5
```

```
## [1] 5
```

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

```
# Clear the result
```

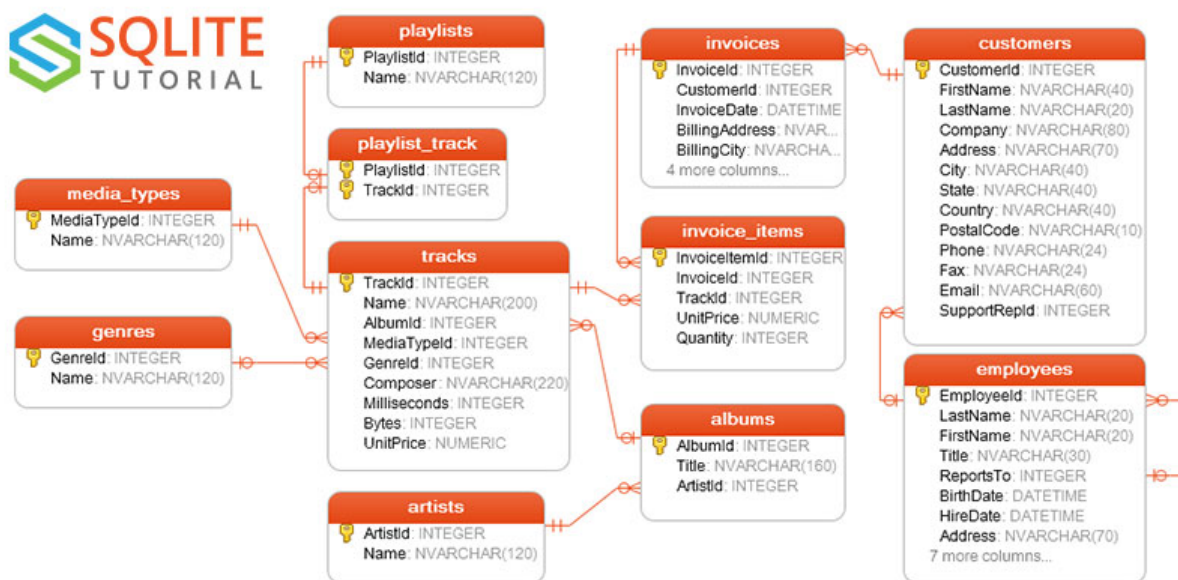
```
dbClearResult(res)
```

```
# Disconnect from the database
```

```
dbDisconnect(con)
```

## 3.6 Práctica 1: SQL

Vamos a utilizar la base de datos Chinook del tutorial de SQLite



Los ejercicios se entregarán por correo electrónico a guillermo.lopez.taboada@udc.es en formato R Markdown con el nombre de archivo P1-Nombre-

Apellidos.Rmd **antes** del 6 de Noviembre.

### 3.6.1 Ejercicios de Análisis Exploratorio

La puntuación de esta práctica será -3 más el número de respuestas correctas (puntuación máxima 10). Se valorará especialmente encontrar la solución más sencilla en una única query SQL.

1. Conocer el importe mínimo, máximo y la media de las facturas.
2. Conocer el total de las facturas de cada uno de los países.
3. Obtener el listado de países junto con su facturación media, ordenado (a) alfabéticamente por país y (b) decrecientemente por importe de facturación media
4. Obtener un listado con Nombre y Apellidos de cliente y el importe de cada una de sus facturas (Hint: WHERE customer.CustomerID=invoices.CustomerID)
5. ¿Qué porcentaje de las canciones son video?
6. Listar los 10 mejores clientes (aquellos a los que se les ha facturado más cantidad) indicando Nombre, Apellidos, Pais y el importe total de su facturación.
7. Listar los géneros musicales por orden decreciente de popularidad (definida la popularidad como el número de canciones de ese género), indicando el porcentaje de las canciones de ese género.
8. Listar los 10 artistas con mayor número de canciones de forma descendente según el número de canciones.
9. Listar Nombre y Apellidos de los “Sales Support Agent” así como la facturación de los clientes que tienen asignados, además de mostrar el porcentaje de la facturación total y del número total de empleados.
10. Listar los géneros musicales que más importe facturan y el porcentaje de su facturación.
11. Listas los géneros musicales ordenados de forma decreciente según el número de canciones por las que no están facturando.
12. Conocer la facturación de los clientes agrupados por su servidor de correo electrónico (e.g., la facturación de los clientes de gmail.com, los de hotmail.com, por orden decreciente de facturación).
13. Ordenar las playlists por la facturación obtenida por sus canciones.

## Capítulo 4

# Manipulación de datos con dplyr

Working draft...

En este capítulo se realiza una breve introducción al paquete `dplyr`. Para mas información, ver por ejemplo la ‘vignette’ del paquete:

Introduction to dplyr,

o el Capítulo 5 Data transformation del libro:

R for Data Science.

### 4.1 El paquete dplyr

```
library(dplyr)
```

**dplyr** Permite sustituir funciones base de R (como `split()`, `subset()`, `apply()`, `sapply()`, `lapply()`, `tapply()` y `aggregate()`) mediante una “gramática” más sencilla para la manipulación de datos:

- `select()` seleccionar variables/columnas (también `rename()`).
- `mutate()` crear variables/columnas (también `transmute()`).
- `filter()` seleccionar casos/filas (también `slice()`).
- `arrange()` ordenar o organizar casos/filas.
- `summarise()` resumir valores.
- `group_by()` permite operaciones por grupo empleando el concepto “dividir-aplicar-combinar” (`ungroup()` elimina el agrupamiento).

Puede trabajar con conjuntos de datos en distintos formatos:

- `data.frame`, `data.table`, `tibble`, ...
- bases de datos relacionales (lenguaje SQL), ...
- bases de datos *Hadoop* (paquete `plyr`).

En lugar de operar sobre vectores como las funciones base, opera sobre objetos de este tipo (solo nos centraremos en `data.frame`).

### 4.1.1 Datos de ejemplo

El fichero `empleados.RData` contiene datos de empleados de un banco. Supongamos por ejemplo que estamos interesados en estudiar si hay discriminación por cuestión de sexo o raza.

## 4.2 Operaciones con variables (columnas)

### 4.2.1 Seleccionar variables con `select()`

```
emplea2 <- select(empleados, id, sexo, minoria, tiempemp, salini, salario)
head(emplea2)
```

```
##   id  sexo minoria tiempemp salini salario
## 1  1 Hombre      No       98  27000  57000
## 2  2 Hombre      No       98  18750  40200
## 3  3 Mujer      No       98  12000  21450
## 4  4 Mujer      No       98  13200  21900
## 5  5 Hombre      No       98  21000  45000
## 6  6 Hombre      No       98  13500  32100
```

Se puede cambiar el nombre (ver también `?rename()`)

```
head(select(empleados, sexo, noblanca = minoria, salario))
```

```
##      sexo noblanca salario
## 1 Hombre      No  57000
## 2 Hombre      No  40200
## 3 Mujer      No  21450
## 4 Mujer      No  21900
## 5 Hombre      No  45000
## 6 Hombre      No  32100
```

Se pueden emplear los nombres de variables como índices:

```
head(select(empleados, sexo:salario))
```

```
##      sexo  fechnac educ      catlab salario
## 1 Hombre 1952-02-03  15    Directivo  57000
## 2 Hombre 1958-05-23  16 Administrativo 40200
## 3 Mujer  1929-07-26  12 Administrativo 21450
```



```
## 4 Mujer 1947-04-15      8 Administrativo  21900
## 5 Hombre 1955-02-09    15 Administrativo  45000
## 6 Hombre 1958-08-22    15 Administrativo  32100
```

```
head(select(empleados, -(sexo:salario)))
```

```
##   id salini tiempemp expprev minoria    sexoraza
## 1  1  27000      98    144      No Blanca varón
## 2  2  18750      98     36      No Blanca varón
## 3  3  12000      98    381      No Blanca mujer
## 4  4  13200      98    190      No Blanca mujer
## 5  5  21000      98    138      No Blanca varón
## 6  6  13500      98     67      No Blanca varón
```

Hay opciones para considerar distintos criterios: `starts_with()`, `ends_with()`, `contains()`, `matches()`, `one_of()` (ver `?select`).

```
head(select(empleados, starts_with("s")))

```

```
##      sexo salario salini    sexoraza
## 1 Hombre   57000 27000 Blanca varón
## 2 Hombre   40200 18750 Blanca varón
## 3 Mujer    21450 12000 Blanca mujer
## 4 Mujer    21900 13200 Blanca mujer
## 5 Hombre   45000 21000 Blanca varón
## 6 Hombre   32100 13500 Blanca varón
```

## 4.2.2 Generar nuevas variables con `mutate()`

```
head(mutate(emplea2, incsal = salario - salini, tsal = incsal/tiempemp ))
```

```
##   id  sexo minoria tiempemp salini salario incsal    tsal
## 1  1 Hombre      No      98 27000  57000 30000 306.12245
## 2  2 Hombre      No      98 18750  40200 21450 218.87755
## 3  3 Mujer      No      98 12000  21450  9450  96.42857
## 4  4 Mujer      No      98 13200  21900  8700  88.77551
## 5  5 Hombre      No      98 21000  45000 24000 244.89796
## 6  6 Hombre      No      98 13500  32100 18600 189.79592
```

## 4.3 Operaciones con casos (filas)

### 4.3.1 Seleccionar casos con `filter()`

```
head(filter(emplea2, sexo == "Mujer", minoria == "Sí"))
```

```
##   id sexo minoria tiempemp salini salario
## 1 14 Mujer      Sí      98 16800  35100
```

```
## 2 23 Mujer      Sí      97 11100 24000
## 3 24 Mujer      Sí      97  9000 16950
## 4 25 Mujer      Sí      97  9000 21150
## 5 40 Mujer      Sí      96  9000 19200
## 6 41 Mujer      Sí      96 11550 23550
```

### 4.3.2 Organizar casos con `arrange()`

```
head(arrange(emplea2, salario))
```

```
##   id  sexo minoria tiempemp salini salario
## 1 378 Mujer      No      70 10200 15750
## 2 338 Mujer      No      74 10200 15900
## 3  90 Mujer      No      92  9750 16200
## 4 224 Mujer      No      82 10200 16200
## 5 411 Mujer      No      68 10200 16200
## 6 448 Mujer      Sí      66 10200 16350
```

```
head(arrange(emplea2, desc(salini), salario))
```

```
##   id  sexo minoria tiempemp salini salario
## 1  29 Hombre      No      96 79980 135000
## 2 343 Hombre      No      73 60000 103500
## 3 205 Hombre      No      83 52500  66750
## 4 160 Hombre      No      86 47490  66000
## 5 431 Hombre      No      66 45000  86250
## 6  32 Hombre      No      96 45000 110625
```

## 4.4 Resumir valores con `summarise()`

```
summarise(empleados, sal.med = mean(salario), n = n())
```

```
##   sal.med  n
## 1 34419.57 474
```

## 4.5 Agrupar casos con `group_by()`

```
summarise(group_by(empleados, sexo, minoria), sal.med = mean(salario), n = n())
```

```
## # A tibble: 4 x 4
## # Groups:   sexo [2]
##   sexo  minoria sal.med    n
##   <fct> <fct>    <dbl> <int>
## 1 Hombre No      44475.  194
## 2 Hombre Sí      32246.   64
```

```
## 3 Mujer No      26707.    176
## 4 Mujer Sí      23062.     40
```

## 4.6 Operador *pipe* %>% (tubería, redirección)

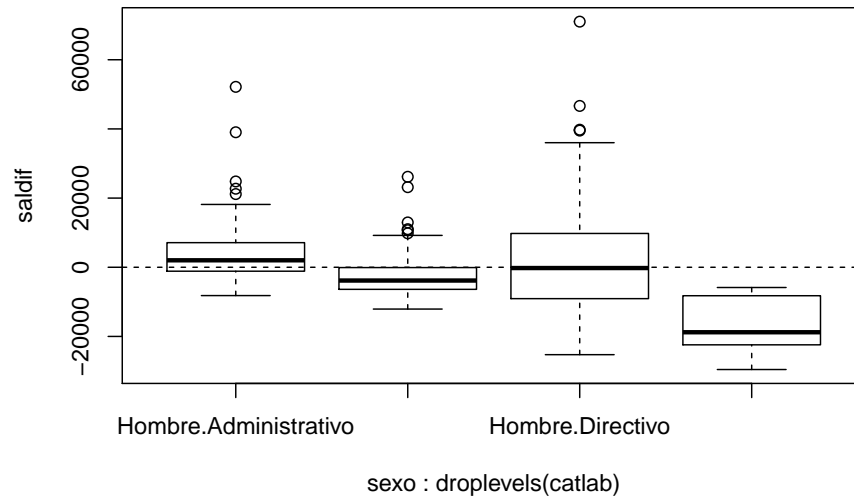
Este operador le permite canalizar la salida de una función a la entrada de otra función. `segundo(primeros(datos))` se traduce en `datos %>% primero %>% segundo` (lectura de funciones de izquierda a derecha).

Ejemplos:

```
empleados %>% filter(catlab == "Directivo") %>%
  group_by(sexo, minoria) %>%
  summarise(sal.med = mean(salario), n = n())
```

```
## # A tibble: 3 x 4
## # Groups:   sexo [2]
##   sexo  minoria sal.med    n
##   <fct> <fct>    <dbl> <int>
## 1 Hombre No      65684.    70
## 2 Hombre Sí      76038.     4
## 3 Mujer  No      47214.    10
```

```
empleados %>% select(sexo, catlab, salario) %>%
  filter(catlab != "Seguridad") %>%
  group_by(catlab) %>%
  mutate(saldif = salario - mean(salario)) %>%
  ungroup() %>%
  boxplot(saldif ~ sexo*droplevels(catlab), data = .)
abline(h = 0, lty = 2)
```



## 4.7 Operaciones con tablas de datos

Join two tbls together

Two-table verbs

## 4.8 Bases de datos con dplyr

Databases using R

dplyr as a database interface

Databases using dplyr

Introduction to dbplyr

SQL databases and R, Data Carpentry

R and Data – When Should we Use Relational Databases?

---

Una alternativa (más rápida) es emplear `data.table`.

## Capítulo 5

# Introducción a Tecnologías NoSQL

Working draft...

### 5.1 Conceptos y tipos de bases de datos NoSQL (documental, columnar, clave/valor y de grafos)

NoSQL - “Not Only SQL” - es una nueva categoría de bases de datos no-relacionales y altamente distribuidas.

Las bases de datos NoSQL nacen de la necesidad de:

- Simplicidad en los diseños
- Escalado horizontal
- Mayor control en la disponibilidad

Pero cuidado, en muchos escenarios las BBDD relacionales siguen siendo la mejor opción.

#### 5.1.1 Características de las bases de datos NoSQL

- Libre de esquemas – no se diseñan las tablas y relaciones por adelantado, además de permitir la migración del esquema.
- Proporcionan replicación a través de escalado horizontal.
- Este escalado horizontal se traduce en arquitectura distribuida
- Generalmente ofrecen consistencia débil

- Hacen uso de estructuras de datos sencillas, normalmente pares clave/valor a bajo nivel
- Suelen tener un sistema de consultas propio (o SQL-like)
- Siguen el modelo BASE (*Basic Availability, Soft state, Eventual consistency*) en lugar de ACID (*Atomicity, Consistency, Isolation, Durability*)

El modelo BASE consiste en:

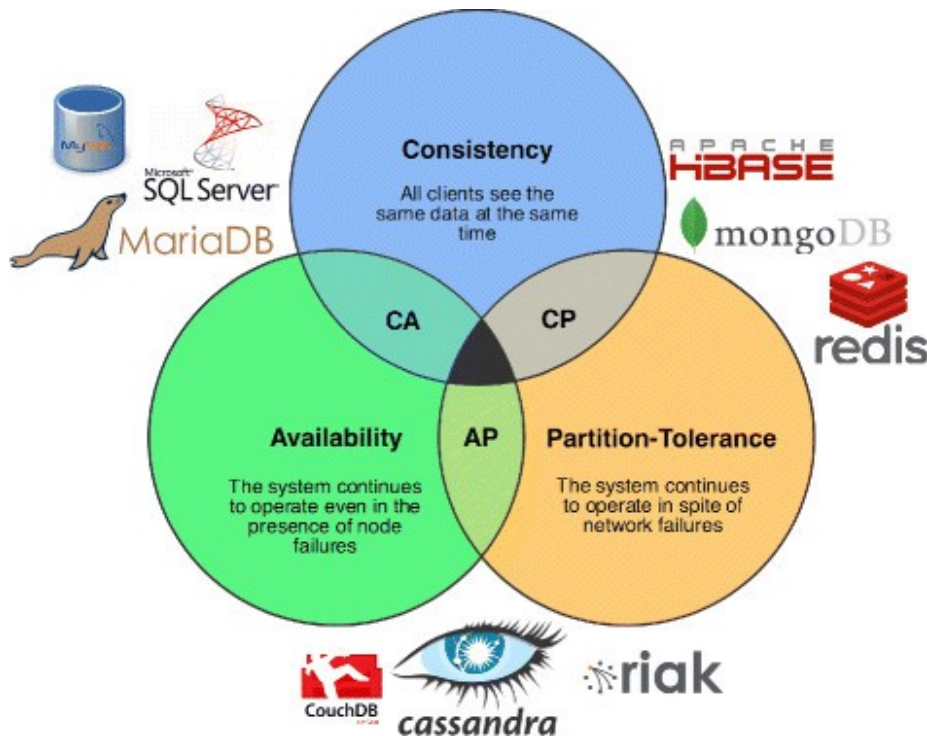
- Basic Availability – el sistema garantiza disponibilidad, en términos del teorema CAP.
- Soft state – el estado del sistema puede cambiar a lo largo del tiempo, incluso sin entrada. Esto es provocado por el modelo de consistencia eventual.
- Eventual consistency – el sistema alcanzará un estado consistente con el tiempo, siempre y cuando no reciba entrada durante ese tiempo.

#### 5.1.1.1 Teorema CAP

Es imposible para un sistema de cómputo distribuido garantizar simultáneamente:

- Consistency – Todos los nodos ven los mismos datos al mismo tiempo
- Availability – Toda petición obtiene una respuesta en caso tanto de éxito como fallo
- Partition Tolerance – El sistema seguirá funcionando ante pérdidas arbitrarias de información o fallos parciales

### 5.1. CONCEPTOS Y TIPOS DE BASES DE DATOS NOSQL (DOCUMENTAL, COLUMNAR, CLAVE/VALOR Y



Las razones para escoger NoSQL son:

- Analítica
- Gran cantidad de escrituras, análisis en bloque
- Escalabilidad
- Tan fácil como añadir un nuevo nodo a la red, bajo coste.
- Redundancia
- Están diseñadas teniendo en cuenta la redundancia
- Rápido desarrollo
- Al ser schema-less o schema on-read son más flexibles que schema on-write
- Flexibilidad en el almacenamiento de datos
- Almacenan todo tipo de datos: texto, imágenes, BLOBs
- Gran rendimiento en consultas sobre datos que no implican relaciones jerárquicas
- Gran rendimiento sobre BBDD desnormalizadas
- Tamaño
- El tamaño del esquema de datos es demasiado grande
- Muchos datos temporales fuera de almacén principal

Razones para NO escoger NoSQL: \* Consistencia y Disponibilidad de los datos son críticas \* Relaciones entre datos son importantes + E.g. joins numerosos y/o importantes \* En general, cuando el modelo ACID encaja mejor

## 5.1.2 Tipos de Bases de Datos NoSQL

Columnar:

1	Things	A	foo	B	bar	C	baz
2	Things	C	bam	E	coh	People	A Emmanuel
3	Languages	A	C	B	java	C	Ceylon

Documental:

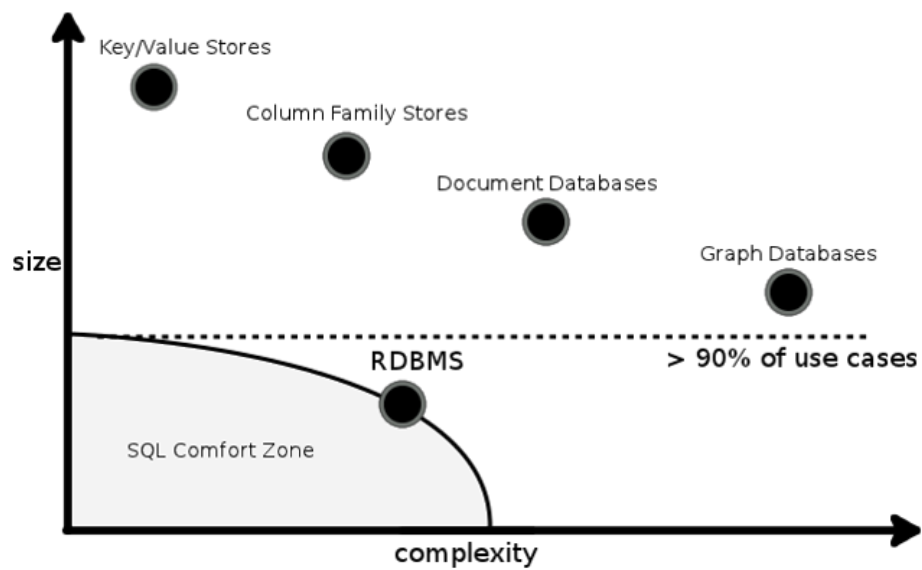
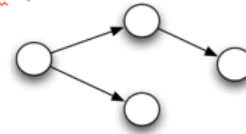
```

{ "user" : {
  "id": "124",
  "name": "Emmanuel",
  "addresses" : [
    { "city": "Paris", "country": "France" },
    { "city": "Atlanta", "country": "USA" }
  ]
}

```

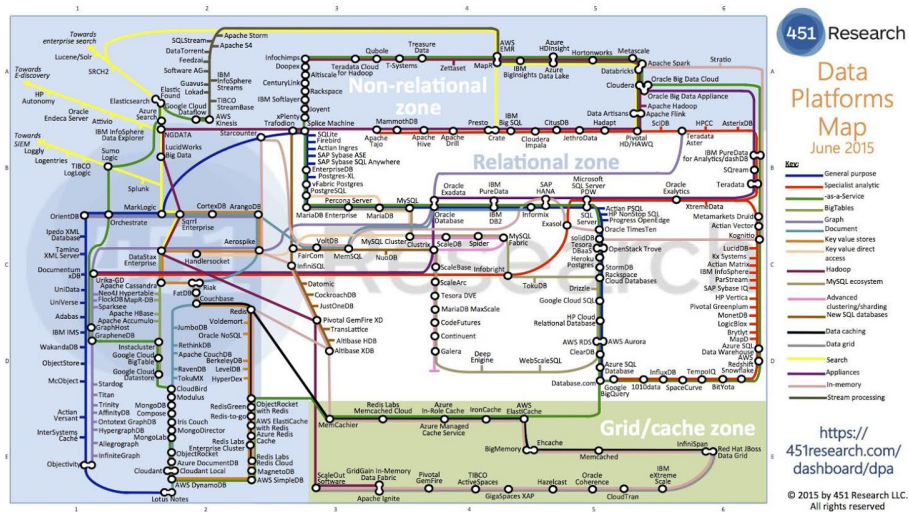
Key-value:

key	value
123	Address@23
126	"Booya"

Graph:

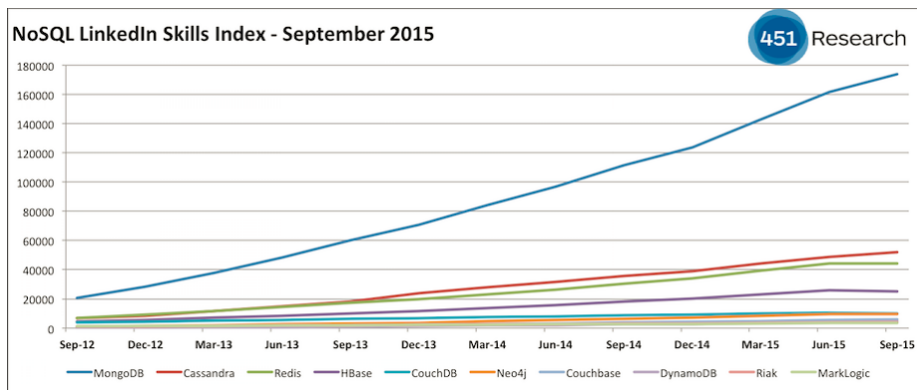


## 5.1. CONCEPTOS Y TIPOS DE BASES DE DATOS NOSQL (DOCUMENTAL, COLUMNAR, CLAVE/VALOR Y Y



352 systems in ranking, September 2019

Rank			DBMS	Database Model	Score		
Sep 2019	Aug 2019	Sep 2018			Sep 2019	Aug 2019	Sep 2018
1.	1.	1.	Oracle +	Relational, Multi-model	1346.66	+7.18	+37.54
2.	2.	2.	MySQL +	Relational, Multi-model	1279.07	+25.39	+98.60
3.	3.	3.	Microsoft SQL Server +	Relational, Multi-model	1085.06	-8.12	+33.78
4.	4.	4.	PostgreSQL +	Relational, Multi-model	482.25	+0.91	+75.82
5.	5.	5.	MongoDB +	Document	410.06	+5.50	+51.27
6.	6.	5.	IBM Db2 +	Relational, Multi-model	171.56	-1.39	-9.50
7.	7.	7.	Elasticsearch +	Search engine, Multi-model	149.27	+0.19	+6.67
8.	8.	8.	Redis +	Key-value, Multi-model	141.90	-2.18	+0.96
9.	9.	9.	Microsoft Access	Relational	132.71	-2.63	-0.69
10.	10.	10.	Cassandra +	Wide column	123.40	-1.81	+3.85



### 5.1.3 MongoDB: NoSQL documental


<b>Rich Queries</b>	<ul style="list-style-type: none"> <li>Find Paul's cars</li> <li>Find everybody in London with a car built between 1970 and 1980</li> </ul>
<b>Geospatial</b>	<ul style="list-style-type: none"> <li>Find all of the car owners within 5km of Trafalgar Sq.</li> </ul>
<b>Text Search</b>	<ul style="list-style-type: none"> <li>Find all the cars described as having leather seats</li> </ul>
<b>Aggregation</b>	<ul style="list-style-type: none"> <li>Calculate the average value of Paul's car collection</li> </ul>
<b>Map Reduce</b>	<ul style="list-style-type: none"> <li>What is the ownership pattern of colors by geography over time? (is purple trending up in China?)</li> </ul>

## MongoDB

```

{
  first_name: 'Paul',
  surname: 'Miller',
  city: 'London',
  location:
[45.123,47.232],
  cars: [
    { model: 'Bentley',
      year: 1973,
      value: 100000, ... },
    { model: 'Rolls Royce',
      year: 1965,
      value: 330000, ... }
  ]
}

```

 **mongoDB**

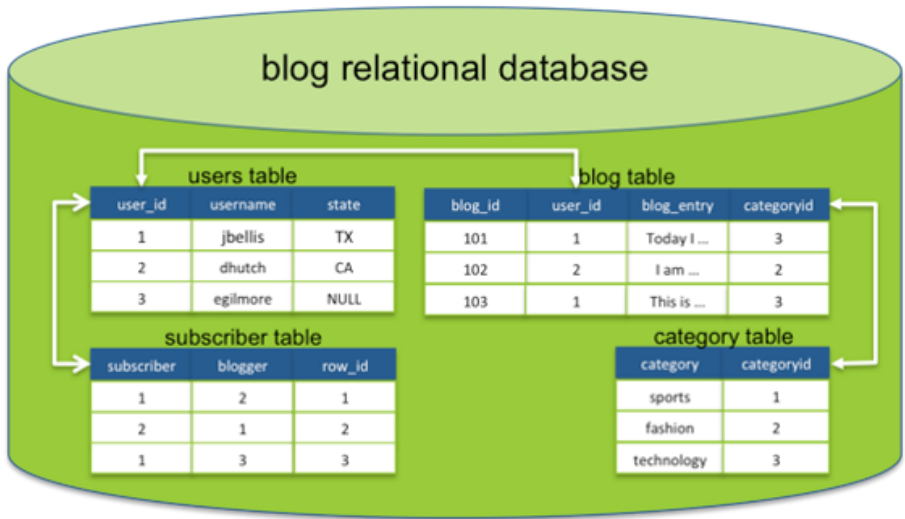
### 5.1.4 Redis: NoSQL key-value

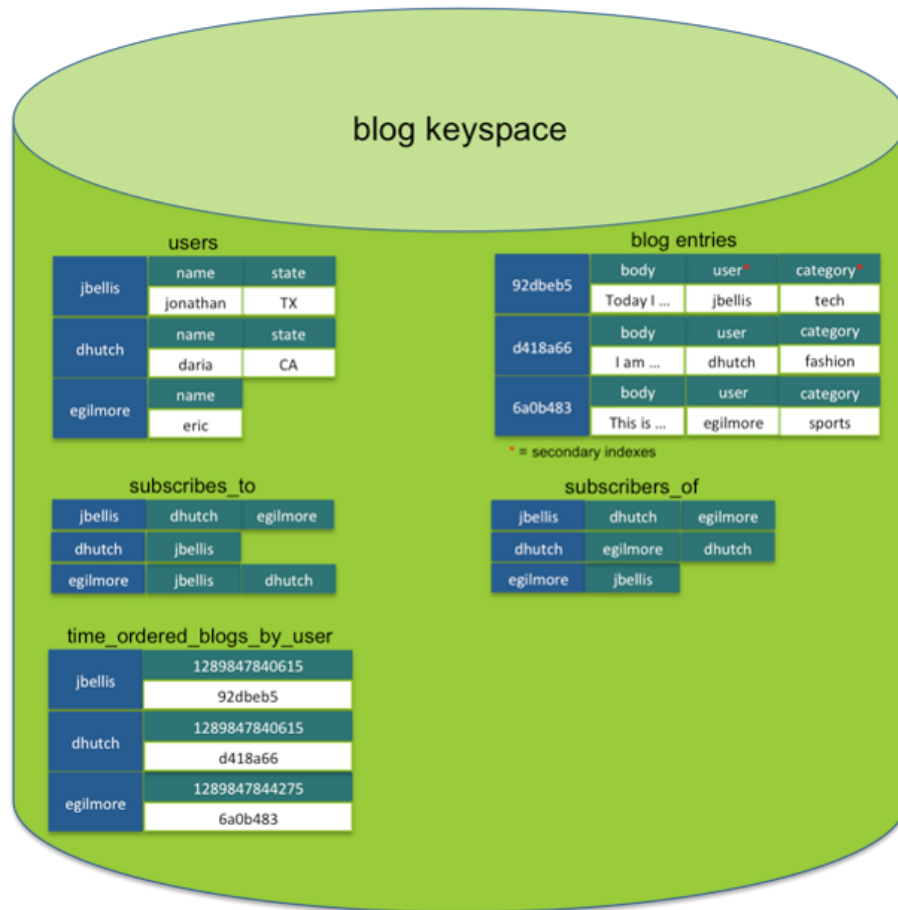
In-memory data structure store, útil para base de datos de login-password, sensor-valor, URL-respuesta, con una sintaxis muy sencilla:

- El comando SET almacena valores
- SET server:name "luna"
- Recuperamos esos valores con GET
- GET server:name
- INCR incrementa atómicamente un valor
- INCR clients
- DEL elimina claves y sus valores asociados
- DEL clients
- TTL (Time To Live) útil para cachés
- EXPIRE promocion 60

5.1. CONCEPTOS Y TIPOS DE BASES DE DATOS NOSQL (DOCUMENTAL, COLUMNAR, CLAVE/VALOR Y

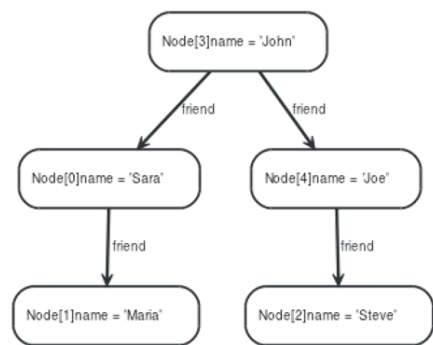
5.1.5 Cassandra: NoSQL columnar



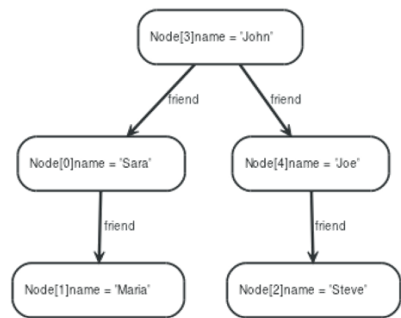


### 5.1.6 Neo4j: NoSQL grafos





```
MATCH (john {name: 'John'})-[:friend]->()-[:friend]->(fof)
RETURN john, fof
```



john	fof
Node[3]{name:"John"}	Node[1]{name:"Maria"}
Node[3]{name:"John"}	Node[2]{name:"Steve"}
2 rows	

5.1.7 Otros: search engines

Son sistemas especializados en búsquedas, procesamiento de lenguaje natural como ElasticSearch, Solr, Splunk (logs de aplicaciones), etc...

5.2 Conexión de R a MongoDB

A través del paquete mongolite, aquí tenéis un Tutorial

```
install.packages("mongolite")

library(mongolite)
```