Gestión de Información para el Periodismo Digital

Máster en Periodismo Digital: Innovación e Investigación

Universidad de Valladolid

Curso 2023/24

Jesús Vegas (jvegas@uva.es)

Índice

Presentación

En este documento contiene el material docente de la asignatura de Gestión de Información para el Periodismo Digital que se imparte en el Máster en Periodismo Digital de la Universidad de Valladolid.

La primera parte de este documento va a servir de guía para que los estudiantes puedan adquirir los objetivos de aprendizaje marcados en la fase de docencia no presencial con la que se inicia la asignatura.

En él se van a presentar conceptos que el estudiante deberá trabajar de forma autónoma y que están agrupados alrededor de las siguientes secciones:

- Conceptos básicos de programación
- Formatos de intercambio de información

En cada una de las secciones que se han considerado se irán identificando los conceptos a trabajar y se sugerirá al estudiante un conjunto de documentos de referencia accesibles en la red, sin perjuicio de que éste pueda consultar otras fuentes que considere oportunas.

Una vez en la fase presencial, se trabajarán los siguientes temas centrados en el uso del lenguaje de programación R para la gestión de información:

- Introducción a R y RStudio
- Acceso a información en datos estructurados mediante SQLite
- Importación y acceso a información en datos públicos con R

En las distintas secciones se irán intercalando enlaces a la documentación a utilizar como referencia.

Cualquier duda sobre los conceptos incluidos en este documento será atendida en tutoría previa petición de cita al profesor en la dirección jvegas@uva.es.

Motivación

Hoy en día, los periodistas operan en un entorno digital en el que la tecnología desempeña un papel fundamental en la difusión de noticias e información. Comprender los principios de la programación es cada vez más esencial para los periodistas por varias razones.

Los periodistas trabajan a menudo con grandes cantidades de datos. Los conocimientos de programación les permiten recopilar, **analizar e interpretar eficazmente conjuntos de datos**. Esta capacidad es crucial para el periodismo de investigación, la comprobación de hechos y la presentación de información compleja de forma comprensible.

Disponer de unos conocimientos básicos de programación permite a los periodistas **automatizar tareas repetitivas**, como la recopilación, el formateo o el análisis de datos. Esto libera tiempo para realizar reportajes más en profundidad y les permite cumplir plazos estrictos con mayor eficacia.

Con conocimientos de programación, los periodistas pueden crear **contenidos interactivos** y atractivos. Esto podría incluir visualizaciones de datos, mapas interactivos o presentaciones multimedia que ofrezcan a los lectores una experiencia más inmersiva e informativa.

La difusión y el consumo de noticias está determinado de forma significativa por la tecnología. Los periodistas que entienden los principios de programación pueden comprender mejor cómo los algoritmos, la inteligencia artificial y las plataformas digitales influyen en la difusión y recepción de la información. Este conocimiento ayuda a **evaluar críticamente las implicaciones éticas y sociales de los avances tecnológicos** en el periodismo.

Estamos inmersos en una era en la que la **colaboración interdisciplinar** es cada vez más común, los periodistas equipados con conocimientos de programación pueden colaborar eficazmente con científicos de datos, desarrolladores y tecnólogos. Esta colaboración da lugar a métodos narrativos innovadores y a una comprensión más profunda de cuestiones complejas.

El periodismo evoluciona rápidamente con los avances tecnológicos. Los periodistas familiarizados con los principios de la programación están mejor equipados para **adaptarse a las nuevas herramientas y tecnologías**, manteniéndose relevantes y competitivos en una industria en constante cambio.

Por todo ello, la integración de los conocimientos de programación en el periodismo capacita a los periodistas para desenvolverse con mayor eficacia en el ámbito digital, mejora la calidad de los reportajes gracias a la información basada en datos y permite crear relatos atractivos e interactivos que tengan eco entre el público actual.

🤔 Para Reflexionar

En la siguiente entrada de LinkedIn se comenta la relación entre periodismo y la programación en un contexto cambiante por el impulso de la IA.

- Programación y periodismo, dos actividades no tan diferentes

Aquí otra entrada, un poco más vieja, de 2018, que trata el tema y aporta recursos por donde empezar. Estos recursos mencionados son un poco generalistas, pero sirven para comenzar a entender este mundo.

- Programación para periodistas: ¿Por dónde empezar?

Conceptos Básicos de Programación

En un principio, no se busca realizar un curso de programación al uso, sino de presentar los conceptos básicos y de familiarizarnos con las ideas que están detrás del hecho de programar un ordenador para conseguir resolver un problema, que en nuestro contexto, estará asociado a la obtención de información a partir de un conjunto de datos. Es por ello que las secciones siguientes se van a centrar en los conceptos básicos de la

programación, presentando los objetivos de aprendizaje a conseguir a partir de la lectura de los documentos de referencia siguientes.

Documentación de Referencia

En esta sección nos vamos a centrar en los aspectos básicos de la programación y para trabajar este tema tomaremos como referencia el documento Introducción a la Programación: Guía de estudio, aunque el estudiante podría utilizar cualquier otro material que así se considere.

Como apoyo al recurso anterior, resulta muy oportuno seguir el curso Introducción a R: aprendiendo R sin morir en el intento ya que está centrado en el lenguaje de programación R y en el manejo de la aplicación R Studio, que es el lenguaje de programación que vamos a utilizar en este curso. En ese documento se ilustra de modo sencillo los pasos a seguir para instalar todo lo necesario en nuestro ordenador y poder realizar algunos ejercicios.

Cuestiones Generales de Programación

Lea las secciones 1.1 a 1.4 de la guía.

Objetivos de Aprendizaje

- Entender la relación entre la programación de computadoras y los lenguajes de programación.
- Comprender la relación entre software y hardware y como con la colaboración de éstos elementos es posible la computación.
- Definir un algoritmo como una secuencia finita de instrucciones para resolver un problema o alcanzar un objetivo.
- Comprender que un programa es sólo una codificación de un algoritmo usando un lenguaje de programación concreto sobre una computadora determinada.
- Comprender el concepto de nivel de abstracción referido a los lenguajes de programación.
- Diferenciar entre un lenguaje de programación compilado y otro interpretado.
- Conocer que existen distintos tipos de error que pueden darse durante el proceso de depuración asociado a la programación y comprender en qué fase se producen cada uno de ellos.

Más contexto

Unos vídeos muy ilustrativos de estos conceptos son los siguientes 👇, donde se explica cómo se representa la información en un ordenador, a partir del sistema binario y qué es un algoritmo, respectivamente.

- ¿Cómo funciona una computadora?
- ¿Qué es un algoritmo?

Objetos y Tipos de datos

Lea las secciones 2.1 a 2.3 de la guía.

Objetivos de Aprendizaje

- Considerar los objetos como almacenes de información en memoria y las funciones que la pueden manipular.
- Considerar las variables como almacenes de información en memoria cuyo valor puede cambiar con el tiempo.

- Considerar las constantes como almacenes de información en memoria cuyo valor puede no cambia con el tiempo.
- Considerar el identificador de un objeto, variable o constante como una etiqueta alfanumérica que sirve para referirse a él y conocer las restricciones para su creación.
- Conocer los distintos tipos de datos primitivos que se pueden almacenar en un objeto, una variable y una constante y diferenciar el tipo de información que puede almacenar cada uno.
- Diferenciar entre las operación de declaración y asignación.
- Conocer y distinguir los distintos operadores que se pueden aplicar a cada tipo de dato.
- Conocer los mecanismos básicos de entrada y salida de información de un programa.

Más contexto

Un vídeo muy ilustrativo de estos conceptos es el siguiente: 👇

- ¿Cómo se representa la información?

Estructuras de Control

Lea las secciones 3.1 a 3.3 de la guía.

Objetivos de Aprendizaje

- Considerar las estructuras de control de un programa como las reglas que controlan el flujo de acciones a ejecutar.
- Conocer y distinguir entre los distintos tipos de estructuras de control que existen.
- Conocer las estructuras de control condicionales y el concepto de anidación.
- Conocer las estructuras de control iterativas y distinguir las situaciones en las que está indicado utilizar cada una de ellas.

Más contexto

Aquí un vídeo que presenta las diferentes estructuras de control de manera sencilla pero eficaz. 👇

- Estructuras de programación

Descomposición Algorítmica

Lea las secciones 4.1 y 4.2 de la guía.

Objetivos de Aprendizaje

- Comprender que la descomposición algorítmica busca descomponer un problema en partes más pequeñas más sencillas de resolver mediante subalgoritmos llamados funciones o procedimientos.
- Distinguir entre funciones y procedimientos dependiendo de su comportamiento y modo de ser invocado.
- Comprender que las funciones pueden recibir datos al ser invocadas mediante el paso de argumentos.
- Distinguir entre el paso de argumentos por valor y por referencia.

Más contexto

Aquí un vídeo interesante sobre el pensamiento computacional. 👇

- ¿Qué es el pensamiento computacional?

Estructuras de Datos

📚 Lea las sección 5.1 de la guía.

🎯 Objetivos de Aprendizaje

- Conocer que llamamos estructura de datos a un conjunto de datos que cuentan con un sistema de organización.
- Saber que unas de las estructuras de datos más sencillas y utilizadas son los arreglos.
- Comprender las propiedades de ordenamiento y homogeneidad de los arreglos.
- Distinguir entre distintos tipos de arreglos en función de su dimensión: vectores (unidimensional) y matrices (bidimensional).

🔬 Más contexto

Aquí un video, que aunque incluye información sobre estructuras que ahora mismo no necesitamos conocer, hace una buena introducción al tema. 👇

- Estructuras de datos - Introducción

Datos Abiertos y Formatos de Intercambio de Información

En un principio, no se busca realizar un curso de programación al uso, sino de presentar los conceptos básicos y de familiarizarnos con las ideas que están detrás del hecho de programar un ordenador para conseguir resolver un problema, que en nuestro contexto, estará asociado a la obtención de información a partir de un conjunto de datos. Es por ello que las secciones siguientes se van a centrar en los conceptos básicos de la programación, presentando los objetivos de aprendizaje a conseguir a partir de la lectura de los documentos de referencia siguientes.

Documentación de Referencia

Esta sección está construida a partir del contenido de El manual de Open Data. Este manual se refiere a los aspectos legales, sociales y técnicos de la apertura de datos, incluyendo un anexo dedicado a los distintos formatos de los archivos de intercambio de datos más utilizados.

Datos Abiertos

📚 Lea las 4 primeras secciones del Manual de Open Data.

🎯 Objetivos de Aprendizaje

- Conocer la definición de datos abiertos.
- Ser consciente de la importancia y las posibilidades que aportan los datos abiertos.
- Comprender la importancia de los archivos de datos abiertos para el acceso a la información.
- Conocer las reglas que rigen en la apertura de datos.

🔬 Más contexto

En los siguientes enlaces, se pueden encontrar opiniones sobre los datos abiertos en relación con el periodismo.

- La Reutilización de Datos Abiertos en el Periodismo: Un Recurso Inagotable para la Información de Calidad
- La reutilización de datos abiertos en el periodismo: una ventana abierta a nuevos modelos de negocio y comunidades

Aquí el enlace al portal de datos abiertos del Gobierno de España. 👇

- https://datos.gob.es/es

Formatos de Intercambio

📑 Lea el anexo formatos de archivos [🟤 Atención, el enlace al anexo formato de datos desde el índice del manual está roto, usar este.🤖]

🎯 Objetivos de Aprendizaje

- Conocer los formatos de intercambio más habituales para cada tipo de información.
- Distinguir entre formatos de datos abiertos y propietarios.
- Entender la conveniencia de que los datos sean fácilmente legibles tanto para humanos como para máquinas.

🔬 Más contexto

En los siguientes enlaces, se aporta más información sobre los formatos de intercambio más usados.

- XML
  - https://es.wikipedia.org/wiki/Extensible_Markup_Language
- JSON
  - https://es.wikipedia.org/wiki/JSON
  - https://www.ibm.com/docs/es/baw/23.x?topic=formats-javascript-object-notation-json-format
- CSV
  - https://es.wikipedia.org/wiki/Valores_separados_por_comas

Introducción a R y RStudio

R es un lenguaje orientado a la realización de procesos estadísticos y gráficos, y para poder construir programas en R y ejecutarlos utilizaremos la aplicación **RStudio**, que es el entorno de programación integrado (o IDE, *Integrated Development Environment*) más popular para trabajar con R. Un IDE es un programa que hace que la codificación sea más sencilla porque permite manejar varios archivos de código, visualizar el *ambiente* de trabajo, utilizar resaltado con colores para distintas partes del código, emplear auto-completado para escribir más rápido, explorar páginas de ayuda, implementar estrategias de depuración e incluso intercalar la ejecución de instrucciones con la visualización de los resultados mientras avanzamos en el análisis o solución del problema.

En esta sección, se comenzará por la instalación de R y de RStudio en nuestro ordenador, si es que aún no se ha hecho.

Para instalar tanto R como RStudio se pueden seguir las indicaciones dadas en el capítulo 2 de curso de Introducción a R.

Introducción a RStudio

Para comprender el modo de trabajar con RStudio resultará de utilidad seguir las explicaciones contenidas al final de la sección primera de la guía Introducción a la Programación: Guía de estudio usada como base en la fase no presencial .

Sobre Cuadernos Markdown

Para contener tanto las explicaciones como la demostración de las ejecuciones de código se ha escrito este cuaderno R Markdown. Un cuaderno Markdown permite combinar celdas con texto y celdas código incluido el resultado de su ejecución, lo que resulta muy conveniente para nuestro caso. Dentro de cada celda de código aparece un símbolo de *play* que si se pica desencadena la ejecución del código que contiene la celda. Cuando se ejecute código de una celda los resultados aparecerán a continuación. También aparece un símbolo de un *play* en vertical sobre una barra de color verde. Cuando se pica esto permite ejecutar todas las celdas de código que preceden a ésta, lo que equivale a ejecutar todo el código desde el principio pero sin ejecutar esta celda en concreto.

Para ejecutar el siguiente fragmento de código hacer clic en el botón *Ejecutar* dentro del fragmento o colocando el cursor dentro de él y pulsando *Cmd+Shift+Enter*. Como ejemplo, a continuación está una celda de código cuya ejecución producirá una representación de puntos según la velocidad y la distancia contenidas en el conjunto de datos `cars`.

```
plot(cars)
```

Se puede añadir una nueva celda de código haciendo clic en el botón *Insertar trozo* de la barra de herramientas o pulsando *Cmd+Opción+I*.

```
# Insertar aquí el comando en R
```

Cuando se guarda el cuaderno, se guardará junto a él un archivo HTML que contiene el código y el resultado (haz clic en el botón *Previsualizar* o pulsa *Cmd+Mayús+K* para previsualizar el archivo HTML).

La vista previa muestra una copia HTML renderizada del contenido del editor. En consecuencia, a diferencia de *Knit*, *Preview* no ejecuta ningún trozo de código R. En su lugar, se muestra la salida del trozo cuando se ejecutó por última vez en el editor.

Bibliografía sobre R

Para completar la información presentada en este documento se recomienda acudir a los siguientes cursos en los que está basado:

- Introducción a R: aprendiendo R sin morir en el intento, de Javiér Álvarez Liébana.
- R for Journalists, de Andrew Ba Tran.

Del primer curso, resulta interesante la sección de ejercicios, del segundo, además de estar específicamente orientado a periodistas, los vídeos explicativos son realmente interesantes.

También se puede encontrar otros tutoriales muy útiles que servirán para profundizar en el tema.

Sintaxis R

A la hora de escribir las instrucciones en R se deben seguir ciertas reglas de sintaxis de modo que R sepa exactamente qué tiene que procesar.

- R distingue entre mayúsculas y minúsculas, a diferencia de SQL, y es un lenguaje interpretado, a diferencia de C. Esto quiere decir que no genera una forma intermedia de código ejecutable, sino que ejecuta las instrucciones una a una.
- Los comandos se pueden introducir en consola escribiéndolos uno a uno justo después del **prompt** `>` o bien en forma de programa guardado en un archivo `.R`.
- Los comentarios van precedidos de `#`. Es importante intercalar comentarios con el código de forma que éste sea comprensible y facilitar su mantenimiento en el tiempo.
- Para separar sentencias entre sí se pueden usar `;` o `ENTER`.

Directorio de trabajo

Su *directorio de trabajo* es la carpeta de su ordenador en la que está trabajando actualmente. Cuando le pida a R que abra un determinado fichero, buscará este fichero en el directorio de trabajo, y cuando le diga a R que guarde un fichero de datos o una figura, lo guardará en el directorio de trabajo. Por eso es muy importante establecer claramente cuál es el directorio de trabajo.

Antes de empezar a trabajar, por favor establezca su directorio de trabajo donde todos sus datos y archivos de script están o deberían estar almacenados.

A continuación se muestra cómo se puede configurar el directorio de trabajo a través de comando `setwd()`.

```
# En un mac, esto podría ser como sigue
setwd("~/GIPD/R_Datos_No_Estructurados")

# En un PC con MS Windows, podría ser así
setwd("C:/Documents/GIPD/R_Datos_No_Estructurados")
```

Usando los menús de RStudio, se puede fijar el directorio de trabajo a través del menú *`Session > Set Working Directory`*.

Esto es todo lo que se necesita saber inicialmente respecto al directorio de trabajo, pero si se necesita profundizar en este aspecto, que está ligado con el de proyecto R, conviene repasar la sección correspondiente del curso Introducción a R de *Javier Álvarez Liébana* que ya hemos visitado en otras ocasiones.

Bibliotecas y Paquetes

Aunque R tiene muchas capacidades de realizar tareas estadísticos y de datos por sí mismo, R base, éstas se pueden ampliar mediante la incorporación de *paquetes* o *bibliotecas* (mal traducidas por *librerías*).

Estas son ampliaciones creadas por otros usuarios con funciones que resuelven ciertos problemas que resultan comunes.

La lista de todos los paquetes instalados se puede obtener, bien en la ventada de paquetes, `Packages`, o escribiendo la instrucción `library()` en la ventana de la consola. Si la casilla situada delante del nombre del paquete está marcada en la ventana de paquetes, el paquete está cargado y las funciones que contiene están listas para ser ejecutadas.

Hay muchos más paquetes disponibles en el sitio web de R. Si desea instalar y utilizar un paquete (por ejemplo, el paquete llamado `dplyr`) se debe:

- **Instalar** el paquete: haciendo clic en `install packages` en la ventana de paquetes y escribiendo `dplyr` o escribiendo `install.packages("dplyr")` directamente en la ventana de la consola.
- **Cargar** el paquete: Marcando la casilla delante de `dplyr` o escribiendo `library("dplyr")` en la ventana de la consola.

Calculadora

Aunque pueda resultar chocante, R puede utilizarse como simple calculadora.

Sólo hay que escribir la expresión matemática en la ventana de la consola después del prompt `>`.

```
2+3
2/3
2*3
2^3 # Para calcular 2 elevado a la 3
```

Variables

En muchas ocasiones conviene dar un nombre a esos números para poder referirse a ellos más tarde. A continuación creamos 2 variables numéricas, `a` y `b`, y operamos con ellas.

```
a <- 2
b <- 3
a
b
a+b
```

A partir de ese momento puede observar el valor que tienen esas 2 variables creadas, `a` y `b`, en la ventada `Environmet`.

Estas variables numéricas son en realidad de tipo `double` y pueden almacenar números en general, con o sin decimales. Aunque existen otros tipos de variables numéricas más adaptadas para almacenar y manipular números enteros, de tipo `integer`, en general se usará el tipo de variable numérica `double` por simplicidad.

```
typeof(a)
class(a)
```

```
etiqueta = "hola"
class(etiqueta)
typeof(etiqueta)
```

Para almacenar información textual, hay que utilizar variables de otro tipo, `character`. Esto se determina al delimitar entre comillas dobles `" "` la cadena de texto que debe almacenar la variable.

```
nombre <- "Ana"
altura <- 1.78
nombre
altura
typeof(nombre)
class(nombre)
typeof(altura)
class(altura)
```

Este detalle es muy importante, ya que todo lo que vaya entre comillas será interpretado como texto.

```
uno <- 1
dos <- "2"
uno + dos
```

En el ejemplo anterior, la variable `dos` es una cadena de texto y no se puede sumar con la variable `uno`, que es de tipo numérico, por lo que la operación da un error.

Una de las operaciones que se pueden hacer con las cadenas de texto es concatenarlas. Esto requiere invocar a funciones, como por ejemplo `paste()`, que concatena 2 o más cadenas manteniendo un espacio en blanco entre ellas como separador. Como curiosidad, existe otra función llamada `paste0()` que concatena cadenas de caracteres sin añadir ese espacio en blanco.

```
apellido1 <- "Pérez"
paste(nombre, apellido1)
apellido2 <- "Campos"
paste0(nombre, apellido1, apellido2)
```

Otra curiosidad es que se puede elegir el carácter que se usa como separador, y en lugar del espacio en blanco, "", se podría utilizar el punto`"."`, por ejemplo.

```
paste(nombre, apellido1, sep=".")
```

La función `paste()` también permite mostrar un mensaje compuesto a partir de variables textuales, numéricas y de textos fijos, como por ejemplo;

```
paste(nombre, "mide", altura, "metros" )
```

Para manejar cadenas de texto de forma más avanzada existen otros paquetes, como por ejemplo, `glue` y `string`.

El contenido de las variables es dinámico y se puede cambiar, por lo que una variable puede recibir distintos valores durante su vida. Un ejemplo curioso de esto es que una variable puede alterar su valor a partir del sí misma.

```
a <- a + 1
a
```

Variables lógicas

En R, como en los demás lenguajes de programación, se pueden manejar los valores asociados a los conceptos cierto y falso. Para representar estos valores se usan los términos `TRUE` y `FALSE`, respectivamente y en R esto se guarda internamente como un `1` o un `0`.

```
llueve <- FALSE
llueve
hace_sol <- TRUE
hace_sol
typeof(llueve)
valor <- a < b
valor
```

Fechas

El manejo de fechas en cualquier lenguaje de programación resulta más complicado de el manejo de números, por ejemplo, ya que es fácil saber cuál es el número siguiente al 3, 3+1, pero no es tan sencillo calcular cuál es el día siguiente del 28 de febrero de 2024.

Aunque podríamos almacenar fechas como texto, no se podría operar con ellas de esa manera.

```
hoy <- "20240228"
hoy + 1

# Error in hoy + 1 : non-numeric argument to binary operator
```

Para ayudar con el manejo de fechas conviene utilizar el paquete `lubridate`. De este modo, a partir de las funciones `as_date()` y `ymd_hms()` podremos convertir texto a formado fecha.

```
library(lubridate)

dia <- as_date("28-02-2024", format="%d-%m-%Y") #  %y en caso de 2 dígitos para
año
dia
dia + 1
otro_dia <- ymd_hms("2024-02-28 00:00:00")
otro_dia
```

Otras funciones muy útiles son `now()` para obtener la fecha y hora actual (en un objeto de tipo POSIXct que es guardado como el número de segundos transcurridos desde 1970-01-01 00:00:00), `today()` para obtener la fecha del día corriente en un formato fecha y las funciones para extraer información sobre fechas, como por ejemplo `year()`, `month()`, `day()`, `week()`, `hour()`.

```
ahora <- now()
ahora
hoy <- today()
hoy
year(hoy)
month(hoy)
week(hoy)
day(hoy)
hour(hoy)
minute(hoy)
second(hoy)
```

Ejercicios

Realizar los siguientes ejercicios ayudará a reforzar la comprensión del modo en que R maneja las variables numéricas y las fechas:

- https://aprendiendo-r-intro.netlify.app/var-num#ejercicios-2
- https://aprendiendo-r-intro.netlify.app/fechas#ejercicios-5

Vectores

Como en muchos otros lenguajes de programación, R organiza los números en escalares (un único número 0-dimensional), vectores (una fila de números, también llamados arrays uni-dimensionales) y matrices (arrays n-dimensionales), entre otras posibilidades.

Todas las variables que hemos definido hasta ahora son escalares, permiten almacenar un valor. Los vestores o arrays no son más que una concatenación de **items del mismo tipo**. De hecho, los números escalares pueden ser vistos como un array o vector de longitud 1.

Para definir un vector con los números 3, 4 y 5, se usará la función `c()`. Para recordar el nombre de esta función se puede ver como una abreviatura de concatenar (o pegar).

```
un_vector_de_números <- c(3, 4, 5)
un_vector_de_números
un_vector_de_texto <- c("a", "b", "c")
un_vector_de_texto
un_vector_mixto <- c(1, "uno")
un_vector_mixto
```

Como los vectores sólo pueden contener un tipo de dato, en el último caso, al crear el vector mixto ha convertido el primer elemento a texto.

La longitud de los vectores se puede conocer con la función `length()`.

```
length(un_vector_de_números)
```

Existen algunas formas interesantes y potentes de crear arrays numéricos con secuencias, repeticiones y demás variaciones.

```
n_del_1_al_10 <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
n_del_1_al_10
n1 <- seq(1, 10) # del 1 al 10
n1
n2 <- seq(1, 10, by=0.5) # del 1 al 10 con incrementos de 0.5
n2
n3 <- seq(1, 7, l = 10) # 10 números del 1 al 7
n3
n4 <- rep(0, 10) # 10 ceros repetidos
n4
```

Los elementos de los vectores pueden ser accedidos mediante índices, usando la notación `[i]`, donde `i` se refiere a la posición del vector deseada, comenzando a contar desde `1` hasta `length()`. Esta notación se puede usar tanto para leer una posición del vector como para escribir en ella.

```
n1[1]
n4[5] <- 44
n4
```

También podemos acceder a los elementos de un vector usando secuencias.

```
n1
n1[1:5]
n1[-6] # devuelve todos los elementos menos el que ocupa la posición 6
n1[n1>7] # devuelve todos los elementos que son mayores que 7
```

Existe una función que nos permite averiguar el lugar que ocupa un elemento dentro de un vector. Esta función es `which()`.

```
x <- c(56,78,34,54,34,56,22,18,95,22)
which(x==22)
which.min(x)
which.max(x)
median(x)
which(x<median(x))
```

Ejercicios de vectores

Realice los ejercicios de

- https://aprendiendo-r-intro.netlify.app/vectores#ejercicios-3

Dado el vector de 4 elementos mensaje, y el vector clave también de 3 elementos, se puede cifrar y descifrar un mensaje.

```
mensaje <- c("Hola","me","llamo","Ana")
clave <- c(2,1,4,3)
mensaje_cifrado <- mensaje[clave]
mensaje_cifrado
mensaje_cifrado[clave]
```

Matrices

Las matrices son vectores de 2 dimensiones, que se organizan en filas y columnas. Por ejemplo, se puede obtener una matriz a partir de un vector, si le indicamos el número de filas y columnas a considerar. Por ejemplo, el siguiente vector de 6 elementos puede ser transformado como una matriz de 2 filas y 3 columnas cada una, rellenando la matriz por columas.

```
mat1 <- matrix(data=c(1,2,3,4,5,6), ncol=3)
mat1
mat2 <- matrix(data=c(1,2,3,4,5,6), nrow=3)
mat2
```

El argumento `data` especifica qué números deben aparecer en la matriz.

Utilice `ncol` para especificar el número de columnas o `nrow` para especificar el número de filas. Fijado uno de ellos, el otro se determina automáticamente.

Para acceder a una matriz, se deben indicar los 2 índices de la posición deseada `[i,j]`, indicando primero la fila `i` y después la columna `j` dentro de esa fila. Al igual que sucede con los vectores, la primera posición tiene el índice 1, tanto para las filas como para las columnas.

```
mat1[1,2]
mat1[2,1]
```

Si se desea obtener toda una fila o columna, vale con dejar vacío el índice de las columnas, o de las filas, respectivamente.

```
mat1[1,]
mat1[ ,1]
```

También se puede crear una matriz como un vector de vectores, todos del mismo tipo de datos y longitud. En este caso, `rbind()` permite construir una matriz por filas a partir de vectores, siendo en este caso cada uno una de las filas que forma la nueva matriz. La función `cbind()` hace lo mismo, pero en este caso los vectores son las columnas de la nueva matriz.

```
mat_por_filas <- rbind(c(9,2,3),c(4,5,6))
mat_por_filas
mat_por_columnas <- cbind(c(9,2,3),c(4,5,6))
mat_por_columnas
```

Para visualizar una matriz se puede usar la función `view()`.

```
View(mat1)
```

Esto produce que la matriz sea mostrada en una etiqueta a parte, con ciertas opciones de ordenación.

Para conocer la longitud de las filas y columnas de una matriz se puede usa la función `dim()`. También se pueden obtener el número de filas y de columnas por separado, usando las funciones `nrow()` y `ncol()`, respectivamente.

```
dim(mat_por_filas)
dim(mat_por_columnas)
nrow(mat_por_filas)
ncol(mat_por_filas)
```

Las matrices suelen manipularse para intercambiar filas por columnas y viceversa. Esa nueva matriz se llama matriz transpuesta y se puede conseguir con la función `t()`.

```
mat_por_filas
t(mat_por_filas)
```

Operaciones sobre Vectores y Matrices

Para comprender cómo operan los operadores aritméticos cuando los aplicamos sobre un vector o matriz, ayuda tener presente que un escalar es en realidad un vector de un elemento.

Así, cuando aplicamos el operador, digamos, suma + al escalar x estaríamos sumando una cierta cantidad al único elemento de ese vector de longitud y dimensión 1. Generalizando esto a un vector de longitud mayor de 1, aplicar una operación de suma sobre él equivale a sumar esa cantidad a **todos sus elementos**.

```
escalar <- 5
escalar + 1 # suma 1
vector <- c(1, 2 ,3)
vector + 1 # suma 1 a todos y cada uno de los elementos del vector
matriz <- matrix(data=c(1,2,3,4,5,6), nrow=2)
matriz + 1 # suma 1 a todos y cada uno de los elementos de la matriz
```

Del mismo modo, se pueden sumar 2 o más vectores, de forma que cada uno de los elementos del vector resultado es la suma de los elementos i-esimos de los vectores de partida.

```
vector_a = c(1,2,3)
vector_b = c(4,5,6)
vector_a + vector_b
```

Pero ¿qué sucederá si sumamos dos vectores de distinta longitud?

```
vector_d = c(7,8,9,10)
vector_a + vector_d
```

El caso es que, aunque R se queja dando un mensaje de aviso, intenta resolver el problema tomando tantos valores repetidos del vector más corto somo sean necesarios para igualar la longitud del vector más largo. En este caso, se toma de nuevo la primera posición del `vector_a` que vale `1`, para sumar a la última posición del `vector_d`, que es `10`, con el resultado de `11`.

Como hemos comentado anteriormente, los valores lógicos `TRUE/FALSE` son guardados internamente como `0/1` por lo que podemos usar operaciones aritméticas con ellos. Una forma muy común de aprovechar esta característica es averiguar el número de elementos de un vector que cumplen una condición lógica (por ejemplo, `<= 3`), lo que producirá un vector en el que las posición que los que lo hagan tendrán asignado un `1` y los que no un `0`, por lo que bastará con sumar dicho vector lógico para obtener el número de elementos que cumplen dicha condición.

```
vector_valores <- c(1,2,3,4,5,6)
vector_pos_3_o_menos <-(vector_valores <= 3)
vector_pos_3_o_menos
sum(vector_pos_3_o_menos)
```

```
vector_valores * vector_pos_3_o_menos # vector_pos_3_o_menos se ha comportado como
una máscara
sum(vector_valores * vector_pos_3_o_menos)
```

En el ejemplo anterior se ha utilizado la función `sum()` para sumar todos los elementos del vector. Del mismo modo, existen funciones estadísticas para calcular la media, la mediana o la suma acumulada.

```
v <- c(1, 2, 3, 4, 5, 5, 6, 6, 7, 3, 4, 8, 9, 1)
length(v)
sum(v)
mean(v) # media aritmética, suma de los valores dividida entre el número de
elementos
median(v) # mediana, el valor medio de los valores ordenados
cumsum(v) # suma acumulada
```

Por supuesto, también se pueden ordenar los elementos de un vector.

```
sort(v)
```

Ahora que tenemos los datos ordenados en el vector, podríamos calcular los percentiles con la función `quantile()`.

```
quantile(v)
```

Datos ausentes

Si se almacenara en un vector las lecturas de temperaturas, pero algunos días el termómetro no funcionó, resultará que en esos casos tendremos un valor ausente, que en R se representa como `NA`.

```
temp <- c(21, NA, 13, NA, NA, 25, 36, 17, 19, 5)
temp
median(temp)
```

Dado que hay días para los que no hay lectura de la temperatura, no será posile conocer la media de la temperatura de todos los días (salvo que obviemos los días donde no tenemos dato). Para evitar que un dato ausente en nuestros datos nos impida hacer ciertas operaciones, en muchas funciones de R podemos añadir el argumento `na.rm = TRUE` que **elimina los datos ausentes** y luego ejecuta la función.

```
median(temp, na.rm=TRUE)
```

Otro enfoque exigirá limpiar de datos ausentes el vector de lecturas. Esto se puede conseguir combinando la función `is.na()`, que localiza el lugar que ocupan los ausentes, con el operador `!`, que niega el valor lógico que venga detrás.

```
temp
is.na(temp)
!is.na(temp)
temp[!is.na(temp)]
```

Más ejercicios de vectores

Realice los ejercicios de

- https://aprendiendo-r-intro.netlify.app/operacionesvectores#ejercicios-4

Data Frames

Hasta ahora se han presentado los vectores y las matrices, pero R tiene una estructura de datos mucho más adecuada para trabajar con grandes volúmenes de datos, similar a una hoja de cálculo, denominada **data frame**. Cada fila de la hoja de datos corresponde a una observación o valor de una instancia, mientras que cada columna corresponde a un vector que contiene los datos de una variable.

La estructura de un `data.frame`es muy similar a la de una matriz. La diferencia es que las filas de un `data.frame` pueden contener valores de diferentes tipos de datos. La función `data.frame()` permite crear un marco de datos a partir de vectores.

```
ID_paciente <- c(111, 208, 113, 408)
edad <- c(25, 34, 28, 52)
sexo <- c(1,2,1,1) # 1 femenino, 2 masculino
diabetes <- c("Tipo1", "Tipo2", "Tipo1", "Tipo1")
estatus <- c(1,2,3,1)

datos_pacientes <- data.frame(ID_paciente, edad, sexo, diabetes, estatus)
datos_pacientes
```

```
ID_paciente <- c(111, 208, 113, 408)
edad <- c(25, 34, 28, 52)
sexo <- c("femenino","masculino","femenino","femenino") # 1 femenino, 2 masculino
diabetes <- c("Tipo1", "Tipo2", "Tipo1", "Tipo1")
estatus <- c("bien", "mal", "grave", "bien")

datos_f <- data.frame(ID_paciente, edad, sexo, diabetes, estatus, stringsAsFactors
= TRUE)
datos_f

str(datos_f)
```

Los data frame requieren que las variables sean de la misma longitud. Por este motivo, hay que asegurase de que el número de argumentos pasados a la función `c()` sea el mismo. Además, las cadenas de caracteres deben están entre `""`.

Ahora, podemos hacer algunas operaciones sobre ese data frame. Hay que tener en cuenta que los índices de un data frame son `[fila,columna]`.

```
datos_pacientes[1:2] # seleccionar columnas 1 a 2
datos_pacientes[2:3,1:2] # seleccionar de las filas 2 a 3 las columnas 1 a 2
datos_pacientes[c("diabetes", "estatus")] # seleccionar las columnas por su nombre
datos_pacientes$edad # seleccionar una columna por su nombre
mean(datos_pacientes$edad) # calcula la media de edad de los pacientes
str(datos_pacientes) # muestra la estructura y contenido del data frame
summary(datos_pacientes) # muestra un resumen del data frame
```

Como se ha podido comprobar, hay mucha similitud en el modo de acceso y las operaciones que se pueden realizar en vectores, matrices y data frames.

Factores

Los **factores** son la forma en la que R guarda toda la información categórica, es decir, aquella de define una categoría en sí misma. Un ejemplo de factor puede ser el genero (masculino/femenino), tienen sus propios valores y orden.

Para tener un factor, tenemos que transformar nuestro vector atómico en la función factor. R transformará los datos en el vector como `integers` y los guardará en el mismo. Ademas, añadirá atributos de niveles o `levels` que son el conjunto de etiquetas que se usan para mostrar los valores del factor.

```
genero <- c("masculino","femenino","masculino","femenino", "otro")
genero
typeof(genero)

# ahora, componemos el mismo vector pero esta vez como un factor
genero <- factor(c("masculino","femenino","masculino","femenino", "otro"))
genero
typeof(genero)
```

Los factores facilitan trabajar con variables categóricas en un modelo estadístico porque las variables se codifican como números. Sin embargo, pueden llevar a confusión porque parecen `string` y se comportan como `integer`.

Usando la función `unclass()` se puede observar el modo en que se encapsulan los valores de un factor. En este caso, vemos cómo en efecto, R ha convertido en `integer` las cadenas de caracteres que se ha encontrado.

```
unclass(genero) # permite abrir el array de factores para analizarlo
```

Listas

Aún existe una estructura de datos más, las listas, que son colecciones de vectores y tienen la ventaja sobre vectores, matrices y data frames de que no necesitan que las columnas sean de la misma longitud. Aunque no daremos más detalles sobre ellas, se nombran aquí para completar la presentación de las estructuras de datos presentes en R.

Ejercicios sobre listas

Realice los ejercicios de

- https://aprendiendo-r-intro.netlify.app/datos#ejercicios-6

Borrado

Por último, conviene saber que se pueden borrar todas las variables del espacio de trabajo picando en `Session > Clear Workspace` o en el icono de un cepillo de la ventana `Environment`. Hay que tener en cuenta que esta operación no se puede deshacer y todas las variables se borrarán sin remedio.

SQLite

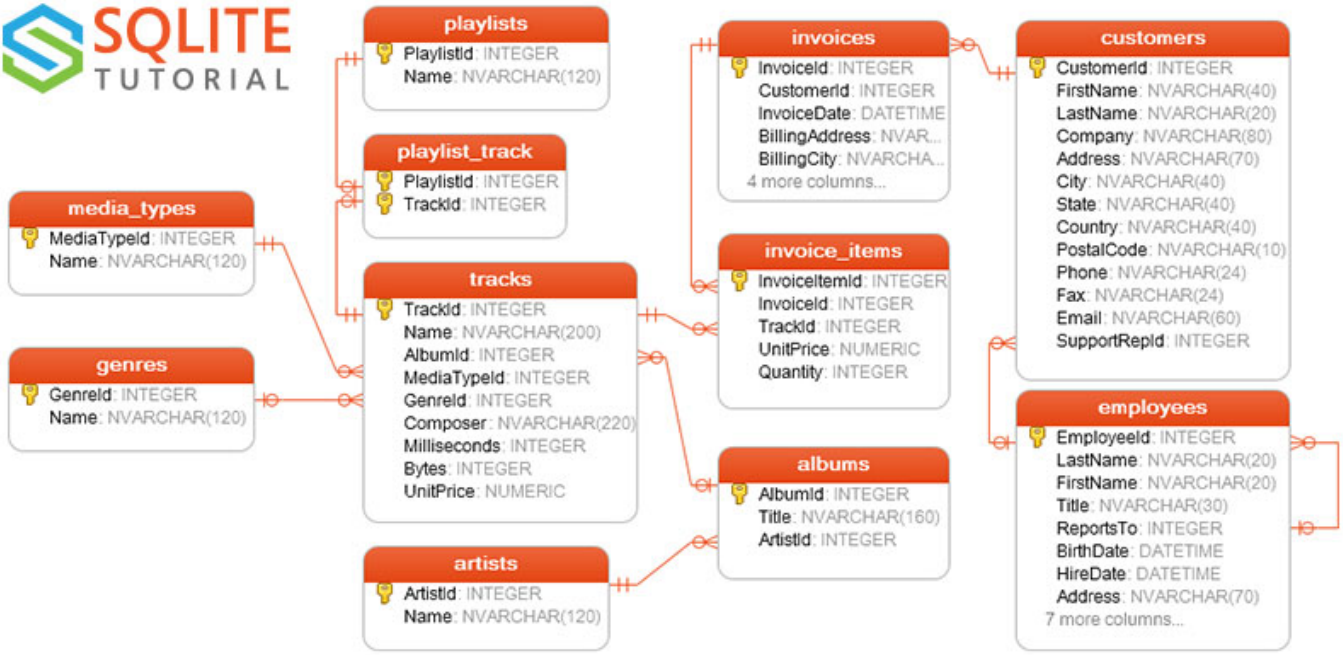
En esta sección vamos a centrarnos en el trabajo con bases de datos, BBDD, usando el lenguaje SQL a través de R. Usaremos SQLite que es un motor de base de datos relacional de código abierto, sin configuración, autónomo, independiente y transaccional, diseñado para ser integrado fácilmente en cualquier aplicación, incluyendo aplicaciones de escritorio, web y móviles.

Podrá encontrar más información sobre SQLite en el siguiente tutorial, esta sección está basada en gran medida en él.

Introducción a la base de datos de ejemplo SQLite de chinook

En este curso se va a trabajar sobre una base de datos SQLite de ejemplo llamada **chinook**. Esta base de datos de ejemplo es una buena base de datos para practicar con SQL, especialmente con SQLite.

El siguiente diagrama de base de datos ilustra las tablas de la base de datos chinook y sus relaciones.



Tablas de la base de datos de muestras de Chinook

En la BD **chinook** hay 11 tablas:

- La tabla `employees` almacena datos de los empleados como el identificador del empleado, el apellido, el nombre, etc. También tiene un campo llamado `ReportsTo` para especificar quién informa a quién. Los identificadores sirven para distinguir de una forma inequívoca los datos que se almacenan en las tablas y suelen ser de tipo numérico. Si usáramos el nombre del empleado para identificarlos podría resultar que 2 o más empleados se llamaran igual, lo que no permitiría su identificación por el nombre.
- La tabla `customers` almacena los datos de los clientes.
- Las tablas `invoices` e `invoice_items` almacenan los datos de las facturas. La tabla `invoices` almacena los datos de la cabecera de la factura y la tabla `invoice_items` almacena los datos de las partidas de la factura.
- La tabla `artists` almacena los datos de los artistas. Es una tabla sencilla que sólo contiene el identificador y el nombre del artista.
- La tabla `albums` almacena datos sobre una lista de canciones. Cada álbum pertenece a un artista. Sin embargo, un artista puede tener varios álbumes.
- La tabla `media_types` almacena tipos de archivos multimedia como archivos de audio MPEG y AAC asignando también un identificador numérico a cada uno de ellos.
- La tabla `genres` almacena tipos de música como rock, jazz, metal, etc. En este caso, se asocia un identificador numérico a cada tipo de música.
- La tabla `tracks` almacena los datos de las canciones. Cada pista pertenece a un álbum.
- Las tablas `playlists` y `playlist_track` almacenan datos sobre listas de reproducción. Cada lista de reproducción contiene una lista de pistas. Cada pista puede pertenecer a varias listas de reproducción.

> La relación entre la tabla `playlists` y la tabla `tracks` es muchos-a-muchos. La tabla `playlist_track` se utiliza para reflejar esta relación.

Descarga de la base de datos de ejemplo SQLite

Puede descargar la base de datos de muestra SQLite utilizando el siguiente enlace:

Descargar la BD SQLite de ejemplo

Una vez descargada, déjela en la carpeta de trabajo, ya que así no necesitamos indicar toda la ruta donde está el archivo, así bastará poner sólo `"./chinook.bd"` (que inicia la ruta en la carpeta de nuestro proyecto) para indicar el archivo a cargar.

Para cambiar la carpeta de trabajo se puede usar la opción `Session > Set Working Directory` y elegir la opción que corresponda entre las que se ofrecen. También resulta conveniente comenzar abriendo un nuevo proyecto en `File > New Project` para cada proyecto de programación que se inicie.

Primeros pasos con SQLite

Este script muestra cómo disponerlo todo para trabajar con SQLite en R. Inicialmente se cargan las librerías necesarias.

```
# Este script ayuda a aquellos que intentan trabajar con SQLite en R
# A continuación, encontrará consultas sencillas

## Importación de los paquetes principales

library(tidyverse) # metapaquete con muchas funciones útiles
library('RSQLite') # paquete SQLite para R
library(DBI) # Interfaz de base de datos R. Más información: https://dbi.r-
dbi.org3
library(jsonlite) # para leer archivos json
```

Después, se crea la conexión con la BD cargando el archivo que almacena la BD.

```
# Crear una conexión de base de datos efímera en memoria RSQLite:
conn <- dbConnect(RSQLite::SQLite(), "./chinook.db")
# Ver las tablas que tiene la BD
dbListTables(conn)
# Ver la estructura de la tabla tracks
```

A partir de ahora, cada vez que se desee consultar la BD usaremos la conexión `conn` para ello.

Como tener una conexión establecida consume recursos, especialmente memoria, las conexiones con BBDD se deberían desconectar una vez que ya no vayan a usarse. Para ello se usa la siguiente sentencia. Se recordará esto al final del documento, ya que si ahora se ejecutara la sentencia de desconexión, no funcionarían los ejemplos que están a continuación.

```
> dbDisconnect(conn)
```

Durante este ejercicio y para reducir las posibilidades de error, se mantendrá la conexión con la BBDD abierta hasta el final del mismo.

Introducción a los tipos de datos SQLite

A diferencia de otros sistemas de bases de datos, SQLite utiliza un sistema de tipado dinámico. En otras palabras, un valor almacenado en una columna determina su tipo de datos, no el tipo de datos de la columna.

SQLite proporciona cinco tipos de datos primitivos que se denominan clases de almacenamiento.

Las clases de almacenamiento describen los formatos que SQLite utiliza para almacenar datos en memoria.

La siguiente tabla ilustra 5 clases de almacenamiento en SQLite:

Clase de almacenamiento	**Significado**
NULL	Los valores NULL representan información ausente o desconocida
INTEGER	Número enteros, bien positivos o negativos como +15 o -43
REAL	Números reales con decimales
TEXT	Para almacenar datos de tipo texto de longitud ilimitada
BLOB	Objetos binarios grandes que pueden almacenar cualquier tipo de datos, de tamaño ilimitado

SQLite no soporta clases de almacenamiento de fecha y/o hora. En su lugar, aprovecha clases de almacenamiento como TEXT, REAL o INTEGER para almacenar los valores de fecha y hora.

Por ejemplo, cuando se utiliza la clase de almacenamiento TEXT para almacenar valores de fecha y hora, se utiliza el formato de cadena ISO8601 que se indica a continuación: `AAAA-MM-DD HH:MM:SS.SSS`

Por ejemplo, la fecha `2016-03-01 10:20:05.123` se refiere al 01 de marzo de 2016.

Consultar datos con SQL

La sentencia SELECT es una de las más utilizadas en SQL ya que se utiliza para consultar datos de una o varias tablas. La sintaxis general de la sentencia SELECT es la siguiente:

```
SELECT DISTINCT column_list
FROM table_list
  JOIN table ON join_condition
WHERE row_filter
ORDER BY column
LIMIT count OFFSET offset
GROUP BY column
HAVING group_filter;
```

La sentencia SELECT es la más compleja de SQL ya que puede tener múltiples argumentos:

- ORDER BY para ordenar el conjunto de resultados
- DISTINCT para consultar filas únicas en una tabla
- WHERE para filtrar filas en el conjunto de resultados
- LIMIT OFFSET para restringir el número de filas devueltas
- INNER JOIN o LEFT JOIN para consultar datos de múltiples tablas utilizando join.
- GROUP BY para obtener las filas agrupadas en grupos y aplicar la función de agregado para cada grupo.
- HAVING para filtrar grupos

En el resto de este cuaderno se irán presentando cada una de estas cláusulas.

Para poder crear una instrucción en R que incluya la sentencia SELECT hay que crear una consulta en R con la función `dbGetQuery()` a la que se pasa como argumentos la conexión con la base de datos creada con anterioridad y una cadena de texto con la propia sentencia SQL. Esto se puede ver en el siguiente bloque.

```
dbGetQuery(conn, "sentencia_SQL")
```

En la sentencia SQL entre comillas podremos indicar o no `;` al final de la misma.

A modo de ejemplo, a continuación se presentan algunas consultas sencillas precedidas por los comentarios que comienzan por el carácter `#` y que explican qué se trata de hacer.

```
# Algunas sentencias de selección sencillas

# Extraer todas las columnas de la tabla artists
dbGetQuery(conn, "select * from artists")

# Extraer sólo las primeras filas de la tabla track
dbGetQuery(conn, "select * from tracks limit 5")
```

Por ahora vamos a centrarnos en la forma más sencilla de la sentencia SELECT, que permite consultar datos de una única tabla.

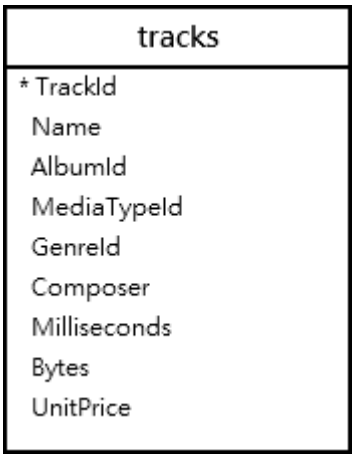
```
SELECT lista_columnas FROM tabla;
```

Aunque la cláusula SELECT aparece antes que la cláusula FROM, SQLite evalúa primero la cláusula FROM y después la cláusula SELECT, por lo tanto:

1. En primer lugar, se especifica la tabla de la que desea obtener datos en la cláusula FROM.
2. En segundo lugar, se determinan en la cláusula SELECT la columna o lista de columnas (separadas por comas) que se quieren conseguir de la tabla especificada anteriormente.

Ejemplos de SQL SELECT

Para empezar con unos ejemplos sencillos de consultas, echemos un vistazo a la tabla `tracks` de la base de datos de ejemplo.



En el caso de que busquemos datos de la cada una de las pistas almacenadas en `tracks` como `trackid`, nombre de la pista, compositor y precio unitario, se puede utilizar la siguiente consulta:

```
SELECT
    trackid,
    name,
    composer,
    unitprice
FROM
    tracks;
```

```
dbGetQuery(conn, "SELECT trackid, name, composer, unitprice FROM tracks")
```

Las tablas contienen columnas y filas. Se podría decir que tienen el aspecto de una hoja de cálculo. Por ejemplo, para ver todas las columnas de la tabla `tracks` podemos `*` en lugar de la lista de columnas. En este caso, el `*` se comporta como un valor comodín que sirve para que la consulta devuelva todas las columnas de todas las filas de la tabla.

```
SELECT * FROM tracks;
```

```
dbGetQuery(conn, "SELECT * FROM tracks")
```

A continuación vamos a explorar cada una de las cláusulas de la sentencia SELECT.

Cláusula ORDER BY

SQLite almacena los datos en las tablas en un orden no especificado. Esto significa que las filas de la tabla pueden estar o no en el orden en que fueron insertadas.

Si utilizas la sentencia SELECT para consultar datos de una tabla, el orden de las filas en el conjunto de resultados no será previsible.

Para ordenar el conjunto de resultados, se añade la cláusula ORDER BY a la sentencia SELECT de la siguiente forma:

```
SELECT
   lista_de_columnas
FROM
   tabla
ORDER BY
    columna_1 ASC,
    columna_2 DESC;
```

La cláusula ORDER BY va después de la cláusula FROM y permite ordenar el conjunto de resultados en función del valor de una o varias columnas en orden ascendente o descendente, según se indique. Para ello se coloca el nombre de la columna por la que se desea ordenar después de la cláusula ORDER BY seguida de la palabra clave ASC o DESC.

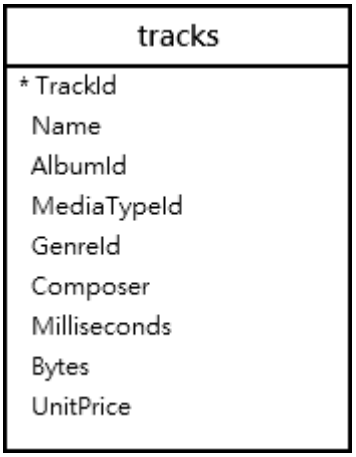
La palabra clave ASC significa ascendente. Y la palabra clave DESC significa descendente. Si no se especifica nada, por defecto se ordena el conjunto de resultados en orden ascendente.

Si desea ordenar el conjunto de resultados por varias columnas, éstas se separan mediante una coma. La cláusula ORDER BY ordena las filas utilizando columnas o expresiones de izquierda a derecha. En otras palabras, la cláusula ORDER BY ordena las filas utilizando la primera columna de la lista. A continuación, ordena las filas utilizando la segunda columna, y así sucesivamente.

Es posible ordenar el conjunto de resultados utilizando una columna que no aparezca en la lista de selección de la cláusula SELECT.

Ejemplo de cláusula ORDER BY

Dada la tabla `tracks` de la base de datos de ejemplo.



Supongamos que se quiere obtener datos de las columnas nombre, milisegundos e id de álbum, para ello se puede utilizar la siguiente sentencia:

```
SELECT
    name,
    milliseconds,
  albumid
FROM
    tracks;
```

```
dbGetQuery(conn,"SELECT name, milliseconds, albumid FROM tracks")
```

La sentencia SELECT que no utiliza la cláusula ORDER BY devuelve un conjunto de resultados que no está en ningún orden concreto.

La siguiente sentencia permite obtener el conjunto de resultados ordenador por la columna AlbumId en orden ascendente:

```
SELECT
    name,
    milliseconds,
    albumid
FROM
    tracks
ORDER BY
    albumid ASC;
```

```
dbGetQuery(conn,"SELECT name, milliseconds, albumid FROM tracks ORDER BY albumid
ASC")
```

En el caso de que ahora se desee obtener el resultado anterior ordenado, a su vez, por la columna `Milliseconds` en orden descendente se debe añadir la columna `Milliseconds` a la cláusula ORDER BY tras la columna `albumid` de la siguiente forma:

```
SELECT
    name,
    milliseconds,
    albumid
FROM
    tracks
ORDER BY
    albumid ASC,
    milliseconds DESC;
```

```
dbGetQuery(conn,"SELECT name, milliseconds, albumid FROM tracks ORDER BY albumid
ASC, milliseconds DESC")
```

Ordenar NULLs

En el mundo de las bases de datos, el valor NULL es especial. Se usa cuando falta información o se quiere indicar que el dato no es aplicable en ese caso concreto.

Esto puede suceder cuando se quiere almacenar el cumpleaños de un artista en una tabla, pero en el momento de guardar el registro del artista, no se dispone de la información de la fecha del cumpleaños.

En ese caso, para representar la información de cumpleaños desconocida en la base de datos, se podría utilizar una fecha especial como `01.01.1900` o una cadena vacía `""`. Sin embargo, estos dos valores no muestran claramente que la fecha de nacimiento es desconocida y pueden ser malinterpretados.

NULL se inventó para resolver este problema. En lugar de utilizar un valor especial para indicar que falta información, se utiliza NULL.

NULL es especial porque no se puede comparar con otro valor. Esto es consecuencia de que si los dos datos son desconocidos, no se pueden comparar.

Por ello, NULL no se puede comparar consigo mismo; NULL no es igual a sí mismo por lo que NULL = NULL siempre resulta en falso. Para poder encontrar los valores nulos se debe utilizar el operador IS NULL, como por ejemplo en la siguiente consulta que pretende encontrar todas pistas cuyo compositor sea desconocido, esto es, tenga el valor NULL.

```
SELECT
    Name,
    Composer
FROM
    tracks
WHERE
    Composer IS NULL
ORDER BY
    Name;
```

```
dbGetQuery(conn,"SELECT Name, Composer FROM tracks WHERE Composer IS NULL")
```

A la hora de ordenar, SQLite considera que NULL es menor que cualquier otro valor.

Esto significa que los NULL aparecerán al principio del conjunto de resultados si se utiliza ASC o al final del conjunto de resultados si se utiliza DESC en la cláusula ORDER BY.

El siguiente ejemplo utiliza la cláusula ORDER BY para ordenar las pistas por compositores.

```
SELECT
    TrackId,
    Name,
    Composer
FROM
   tracks
ORDER BY
   Composer;
```

```
dbGetQuery(conn, "SELECT TrackId, Name, Composer FROM tracks ORDER BY Composer")
```

En la salida se puede observar que los NULL aparecen bajo la forma de `N/A` , ya que es de esta manera cómo R denomina a los valores nulos. Estos valores nulos se encuentran al principio del conjunto de resultados porque SQLite los trata como los valores más bajos. Desplazándose hacia abajo en el resultado, se pueden ver el resto de valores en orden ascendente.

Ejercicio de búsqueda ordenada

Obtenga el listado de empleados ordenados de mayor a menor edad.

```

```

Cláusula SELECT DISTINCT

La cláusula DISTINCT es una cláusula opcional de la sentencia SELECT que permite eliminar las filas duplicadas del conjunto de resultados.

La siguiente sentencia ilustra la sintaxis de la cláusula DISTINCT:

```
SELECT DISTINCT select_list
FROM table;
```

En esta sintaxis:

- La cláusula DISTINCT debe aparecer inmediatamente después de la palabra clave SELECT.
- A continuación se coloca una columna o una lista de columnas después de la palabra clave DISTINCT.

Si utiliza una sola columna, SQLite utiliza los valores de esa columna para evaluar el duplicado. Si utiliza varias columnas, SQLite utiliza la combinación de valores de esas columnas para evaluar el duplicado. SQLite considera los valores NULL como duplicados. Si se utiliza la cláusula DISTINCT con una columna que tiene valores NULL, el resultado contendrá una fila con un valor NULL.

Ejemplos de SELECT DISTINCT

Sea la tabla clientes de la base de datos de ejemplo.

En el caso de querer conocer las ciudades donde se ubican los clientes, se puede utilizar la sentencia SELECT para obtener los datos de la columna ciudad de la tabla clientes de la siguiente manera:

```
SELECT city
FROM customers
ORDER BY city;
```

```
dbGetQuery(conn, "SELECT city FROM customers ORDER BY city")
```

Esta consulta devuelve 59 filas, algunas de las cuales están duplicadas, como Berlín, Londres y Mountain View, por ejemplo. Para eliminar estas filas duplicadas, se debe utilizar la cláusula DISTINCT del siguiente modo:

```
SELECT DISTINCT city
FROM customers
ORDER BY city;
```

```
dbGetQuery(conn, "SELECT DISTINCT city FROM customers ORDER BY city")
```

Ahora la consulta devuelve 53 filas porque la cláusula DISTINCT ha eliminado 6 filas duplicadas.

Se puede usar SELECT DISTINCT en múltiples columnas de modo que la consulta elimine cualquier valor repetido en esa combinación de columnas. Por ejempo, la siguiente sentencia busca las ciudades y países de todos los clientes y las ordena por país.

```
SELECT  DISTINCT
    city,
    country
FROM
    customers
ORDER BY
    country;
```

```
dbGetQuery(conn, "SELECT  DISTINCT city, country FROM customers ORDER BY country")
```

En el caso de aplicar la cláusula DISTINCT a la sentencia cuando el resultado tenga valores NULL la salida mantendrá sólo una fila con valor NULL.

Por ejemplo, la siguiente sentencia devuelve los nombres de las empresas de los clientes de la tabla clientes.

```
SELECT DISTINCT company
FROM customers;
```

```
dbGetQuery(conn, "SELECT DISTINCT Company FROM Customers ORDER BY Company")
```

Como en algunos casos la empresa es desconocida, devolverá una fila con el valor NULL.

Cláusula WHERE

La cláusula WHERE sirve para especificar la condición de búsqueda de las filas devueltas por la consulta y es una cláusula opcional de la sentencia SELECT. Aparece después de la cláusula FROM según la siguiente sintáxis:

```
SELECT
    lista_de_columnas
FROM
    tabla
WHERE
    condition_de_búsqueda;
```

Se añade una cláusula WHERE a la sentencia SELECT para filtrar las filas devueltas por la consulta. Al evaluar una sentencia SELECT con una cláusula WHERE, se siguen los siguientes pasos:

1. Comprobar la tabla en la cláusula FROM.
2. Evalar las condiciones de la cláusula WHERE para obtener las filas que cumplen dichas condiciones.

3. Crear el resultado final basándose en las filas del paso anterior con las columnas de la cláusula SELECT.

La condición de búsqueda en la cláusula WHERE tiene la siguiente forma:

```
expresión_izquierda OPERADOR_DE_COMPARACIÓN expresión_derecha
```

Un operador de comparación comprueba si dos expresiones son iguales. La siguiente tabla muestra los operadores de comparación que se pueden utilizar para construir expresiones:

Operador	Significado
=	Igual a
<> or !=	No igual a
<	Menor que
>	Mayor que
<=	Menor o igual que
>=	Mayor o igual que

Los operadores lógicos permiten comprobar la veracidad de algunas expresiones. Un operador lógico devuelve 1, 0 o un valor NULL.

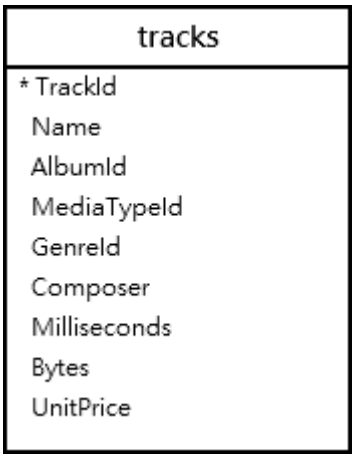
Tenga en cuenta que SQLite no proporciona datos de tipo booleano, por lo que 1 significa TRUE y 0 significa FALSE.

La siguiente tabla muestra los operadores lógicos:

Operador	Significado
ALL	devuelve 1 si todas las expresiones son 1
AND	devuelve 1 si ambas expresiones son 1, y 0 si cualquiera de ellas es 0
ANY	devuelve 1 si cualquiera de un conjunto de comparaciones es 1
BETWEEN	devuelve 1 si un valor está dentro de un rango
EXISTS	devuelve 1 si una subconsulta contiene alguna columna
IN	devuelve 1 si un valor está en una lista de valores
LIKE	devuelve 1 si un valor concuerda con un patrón
NOT	cambia el valor de otros operadores como por ejemplo NOT EXISTS, NOT IN, NOT BETWEEN, etc.
OR	devuelve 1 si cualquier expresión es 1

Ejemplo de cláusula WHERE

Se usará la tabla `tracks` de la base de datos de ejemplo para ilustrar cómo utilizar la cláusula WHERE.



El operador de igualdad = es el más utilizado. Por ejemplo, la siguiente consulta utiliza la cláusula WHERE con el operador de igualdad para encontrar todas las pistas en el álbum cuya id sea 1:

```
SELECT
   name,
   milliseconds,
   bytes,
   albumid
FROM
   tracks
WHERE
   albumid = 1;
```

```
dbGetQuery(conn, "SELECT name, milliseconds, bytes, albumid FROM tracks WHERE
albumid = 1")
```

Cuando se comparan dos valores, se debe asegurarse de que son del mismo tipo de dato. Se deben comparar números con números, cadenas de text con cadenas de texto, etc.

En caso de que se compare valores de tipos de datos diferentes, por ejemplo, una cadena con un número, SQLite realiza conversiones implícitas de tipos de datos, pero en general, esto debe evitarse ya que puede producir conversiones no esperadas.

El operador lógico se utiliza para combinar expresiones. Por ejemplo, para obtener las pistas del álbum 1 que tengan una duración superior a 250.000 ms, utilice la siguiente expresión:

```
SELECT
    name,
    milliseconds,
    bytes,
    albumid
FROM
    tracks
WHERE
```

```
    albumid = 1
AND milliseconds > 250000;
```

```
dbGetQuery(conn, "SELECT name, milliseconds, bytes, albumid FROM tracks WHERE
albumid = 1 AND milliseconds > 250000")
```

La cláusula WHERE se puede combinar con otros operadores LIKE, GLOB e IN para hacer búsquedas más sofisticadas, según se verá más adelante.

Cláusula WHERE con operador LIKE

A veces, es posible que no se conozca exactamente que datos desean buscar o los datos a buscar son variaciones de un patrón base. En este caso, se debe realizar una búsqueda inexacta utilizando el operador LIKE ya que permite buscar valores que casen con un cierto patrón.

SQLite proporciona dos comodines para construir patrones. Son el signo de porcentaje `%` y el guión bajo `_` :

- El comodín `%` coincide con cualquier secuencia de cero o más caracteres.
- El guión bajo `_` coincide con cualquier carácter, pero sólo uno.

Por ejemplo, para encontrar las pistas compuestas por cualquier compositor cuyo nombre contenga la cadena de caracteres `Smith`, se podría formar el patrón `%Smith%`. La consulta se formaría utilizando el operador LIKE de siguiente manera:

```
SELECT
    name,
    albumid,
    composer
FROM
    tracks
WHERE
    composer LIKE '%Smith%'
ORDER BY
    albumid;
```

```
dbGetQuery(conn, "SELECT name, albumid, composer FROM tracks WHERE composer LIKE
'%Smith%' ORDER BY albumid")
```

Hay que tener en cuenta que el operador LIKE, por defecto, no distingue entre mayúsculas y minúsculas, es decir `"A" LIKE "a"` es TRUE.

Cláusula WHERE con operador GLOB

El operador GLOB también se usa para buscar cadenas de caracteres que cumplan un patrón, pero a diferencia de LIKE, el operador el operador GLOB distingue entre mayúsculas y minúsculas y utiliza los comodines usados por los sistemas UNIX.

A continuación se muestran los comodines utilizados con el operador GLOB:

- El asterisco `*` se usa para representar cualquier número de caracteres.
- El signo de interrogación `?` representa exactamente un carácter cualquiera.

Además de estos comodines, puede definir patrones a partir de una lista delimitada por `[` y `]`. Por ejemplo, la lista `[xyz]` coincidiría con los caracteres `x`, `y` o `z`.

El comodín de lista también permite determinar un rango de caracteres mediante el guión `-`, por ejemplo, la lista `[a-z]` coincide con cualquier carácter en minúscula de la 'a' a la 'z'. El patrón `[a-zA-Z0-9]` coincide con cualquier carácter alfanumérico, tanto en minúscula como en mayúscula.

Además, puede utilizar el carácter `^` al principio de la lista para que coincida con cualquier carácter excepto con cualquier carácter de la lista. Por ejemplo, el patrón `[^0-9]` coincide con cualquier carácter excepto con un carácter numérico.

Por ejemplo, para encontrar las pistas cuyo nombre comience por `Man` se podría usar la siguiente consulta.

```
SELECT
    trackid,
    name
FROM
    tracks
WHERE
    name GLOB 'Man*';
```

```
dbGetQuery(conn, "SELECT trackid, name FROM tracks WHERE name GLOB 'Man*'")
```

Cláusula WHERE con el operador IN

El operador IN permite comprobar si un valor se encuentra en una lista de valores separados por comas. Por ejemplo, para buscar pistas que tengan un id de tipo de soporte 2 o 3, se puede utilizar el operador IN como se muestra en la siguiente sentencia:

```
SELECT
    name,
    albumid,
    mediatypeid
FROM
    tracks
WHERE
    mediatypeid IN (2, 3);
```

```
dbGetQuery(conn, "SELECT name, albumid, mediatypeid FROM tracks WHERE mediatypeid
IN (2, 3);")
```

Cláusula LIMIT

La cláusula LIMIT es una parte opcional de la sentencia SELECT que se utiliza para restringir el número de filas que devuelve la consulta.

Por ejemplo, una sentencia SELECT puede devolver un millón de filas. Sin embargo, si sólo necesitan las 10 primeras filas del conjunto de resultados, se puede añadir la cláusula LIMIT a la secuencia SELECT para recuperar únicamente esas 10 primeras filas.

A continuación se ilustra la sintaxis de la cláusula LIMIT.

```
SELECT
    lista_columnas
FROM
    tabla
LIMIT contador_filas;
```

Donde `contador_filas` es un número entero positivo que especifica el número de filas devueltas.

Por ejemplo, para obtener las 10 primeras filas de la tabla `tracks`, se utiliza la siguiente sentencia:

```
SELECT
    trackId,
    name
FROM
    tracks
LIMIT 10;
```

```
dbGetQuery(conn,"SELECT trackId, name FROM tracks LIMIT 10;")
```

Si se desea obtener las 10 primeras filas a partir de la fila 10 del conjunto de resultados (esto es, de la fila 11 a la 20) se debe utilizar la palabra clave OFFSET y a continuación el valor a partir del cual se obtendrán las filas, como se indica a continuación:

```
SELECT
    trackId,
    name
```

```
FROM
    tracks
LIMIT 10 OFFSET 10;
```

```
dbGetQuery(conn,"SELECT trackId, name FROM tracks LIMIT 10 OFFSET 10")
```

Normalmente se utiliza la cláusula LIMIT junto con la cláusula ORDER BY, ya que suele ser conveniente obtener un número determinado de filas en un orden especificado. Por ejemplo, para obtener las 5 pistas más cortas, se ordenan las pistas por la longitud especificada por la columna `milliseconds` utilizando la cláusula ORDER BY y se restringe la salida a las 5 primeras filas utilizando la cláusula LIMIT.

```
SELECT
    trackid,
    name,
    milliseconds
FROM
    tracks
ORDER BY
    milliseconds ASC
LIMIT 5;
```

```
dbGetQuery(conn, "SELECT trackid, name, milliseconds FROM tracks ORDER BY
milliseconds ASC LIMIT 5;")
```

Ejercicio

Realice una consulta para conseguir el nombre y la fecha de nacimiento del segundo empleado de mayor edad.

```
dbGetQuery(conn, "")
```

Operador BETWEEN

El operador BETWEEN es un operador lógico que comprueba si un valor se encuentra dentro de un rango de valores. Si el valor está dentro del rango especificado, el operador BETWEEN devuelve TRUE. El operador BETWEEN puede utilizarse en la cláusula WHERE de las sentencias SELECT y también en otras sentencias.

A continuación se ilustra la sintaxis del operador BETWEEN:

```
expresión_a_comprobar BETWEEN expresión_menor AND expresión_mayor
```

En esta sintaxis

- `expresión_a_comprobar` es una expresión que debe comprobarse en el intervalo definido por `expresión_menor` y `expresión_mayor`.
- `expresión``_menor` y `expresión_mayor` son cualquier expresión válida que especifique los valores bajo y alto del intervalo. El valor de `expresión_menor` debe ser menor o igual que el valor de `expresión_``mayor`, de lo contrario BETWEEN siempre devuelve FALSE. La palabra clave AND es un operador lógico que indica que la `expresión_a_probar` debe estar dentro del intervalo especificado por los valores de `expresión_menor` y `expresión_mayor`. Hay que tener en cuenta que el operador BETWEEN es inclusivo, devuelve TRUE cuando la `expresión_a_comprobar` es menor o igual que `expresión_mayor` y mayor o igual que el valor de `expresión_menor`, como la siguiente expresión:

```
expresión_a_comprobar >= expresión_menor AND expresión_a_comprobar <= expresión_mayor
```

Para especificar un rango exclusivo, utilice los operadores mayor que `>` y menor que `<`.

Tenga en cuenta que si alguna entrada del operador BETWEEN es NULL, el resultado es NULL, o desconocido para ser precisos.

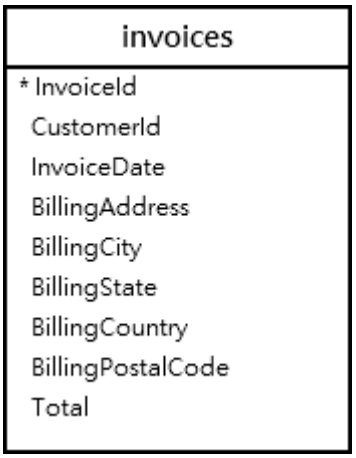
Para negar el resultado del operador BETWEEN, utilice el operador NOT BETWEEN como se indica a continuación:

```
expresión_a_comprobar NOT BETWEEN expresión_menor AND expresión_mayor
```

El operador NOT BETWEEN devuelve TRUE si el valor de `expresión_a_comprobar` es menor que el valor de `expresión_menor` o mayor que el valor de `expresión_mayor`.

Ejemplos del operador BETWEEN

Para la demostración se utilizará la tabla de facturas de la base de datos de ejemplo:



La siguiente consulta encuentra las facturas cuyo total está entre 14,96 y 18,86:

```
SELECT
    InvoiceId,
    BillingAddress,
    Total
FROM
```

```
    invoices
WHERE
    Total BETWEEN 14.91 and 18.86
ORDER BY
    Total;
```

```
dbGetQuery(conn, "SELECT InvoiceId, BillingAddress, Total FROM invoices WHERE
Total BETWEEN 14.91 and 18.86 ORDER BY Total" )
```

El siguiente ejemplo busca facturas cuyas fechas de facturación están entre el 1 de enero de 2010 y el 31 de enero de 2010. Hay que prestar atención al formato de las fechas, ya que están especificadas en formato `'AAAA-MM-DD'`.

```
SELECT
    InvoiceId,
    BillingAddress,
    InvoiceDate,
    Total
FROM
    invoices
WHERE
    InvoiceDate BETWEEN '2010-01-01' AND '2010-01-31'
ORDER BY
    InvoiceDate;
```

```
dbGetQuery(conn,"SELECT InvoiceId, BillingAddress, InvoiceDate, Total FROM
invoices WHERE InvoiceDate BETWEEN '2010-01-01' AND '2010-01-31' ORDER BY
InvoiceDate")
```

Operador IN

El operador IN determina si un valor coincide con cualquier valor de una lista o de una subconsulta. La sintaxis del operador IN es la siguiente:

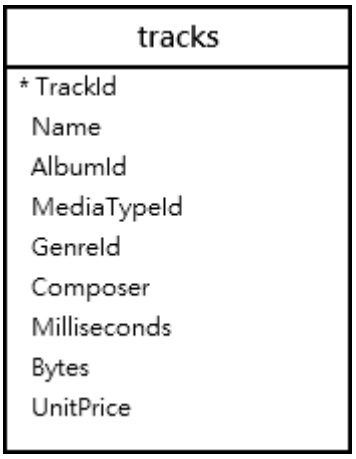
`expresión [NOT] IN (lista_de_valores|subconsulta);`

La `expresión` puede ser cualquier expresión válida o una columna de una tabla.

Una `lista_de_valores` es una lista de valores fijos o un conjunto de resultados de una sola columna devuelto por una `subconsulta`. El tipo de expresión devuelto y los valores de la lista deben ser iguales.

El operador IN devuelve TRUE o FALSE en función de si la expresión coincide o no con algún valor de una lista de valores. Para negar la lista de valores, se utiliza el operador NOT IN.

Para la demostración se utilizará la tabla `Trakcs` de la base de datos de ejemplo.



La siguiente sentencia utiliza el operador IN para consultar las pistas cuyo id de tipo de soporte es 1 o 2.

```
SELECT
    TrackId,
      Name,
      Mediatypeid
FROM
      Tracks
WHERE
      MediaTypeId IN (1, 2)
ORDER BY
      Name ASC;
```

```
dbGetQuery(conn, "SELECT TrackId, Name, Mediatypeid FROM Tracks WHERE MediaTypeId
IN (1, 2) ORDER BY Name ASC;")
```

Esta consulta utiliza el operador OR en lugar del operador IN para devolver el mismo conjunto de resultados que la consulta anterior:

```
SELECT
    TrackId,
      Name,
      Mediatypeid
FROM
      Tracks
WHERE
      MediaTypeId IN 1 OR MediaTypeId = 2
ORDER BY
      Name ASC;
```

Aunque el verdadero potencial del operador IN está asociado a las búsquedas en conjunto de valores devuelto por una subconsulta. Por ejemplo, para obtener las pistas que pertenecen al artista con id 12, se

puede combinar el operador IN con una subconsulta que devuelva los identificadores de los álbunes que son de ese artista de la siguiente manera:

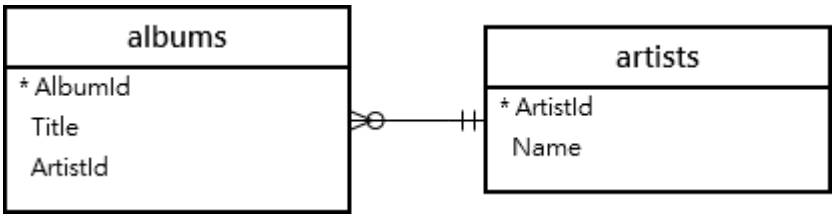
```
SELECT
      TrackId,
      Name,
    AlbumId
FROM
    Tracks
WHERE
      AlbumId IN (
                SELECT
                    AlbumId
                FROM
                    Albums
                WHERE
                    ArtistId = 12
              );
```

```
dbGetQuery(conn, "
            SELECT TrackId, Name, AlbumId FROM Tracks WHERE AlbumId IN (
                  SELECT AlbumId FROM Albums WHERE ArtistId = 12
                )
          ")
```

JOIN

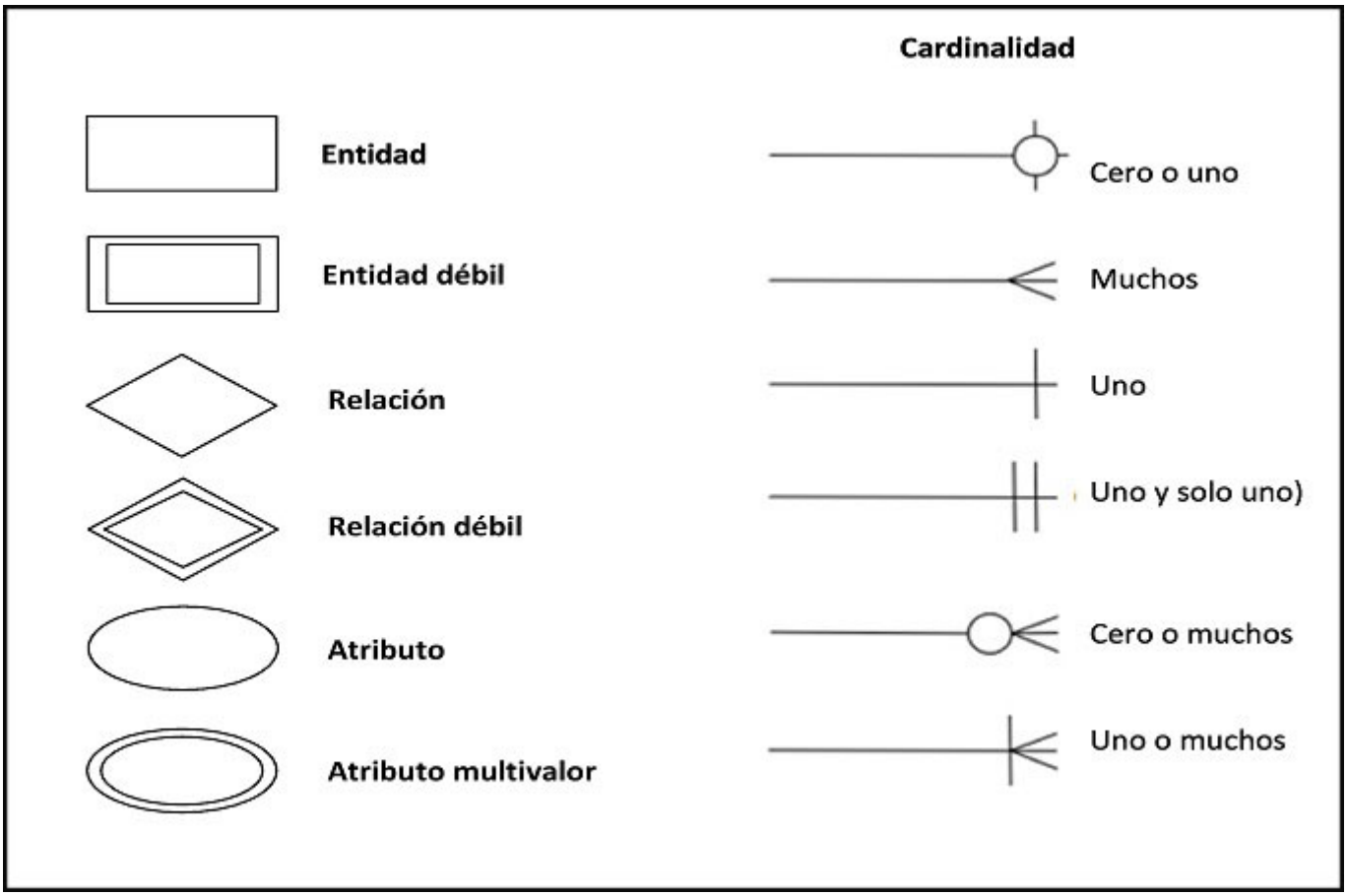
Cuando se han de consultar datos relacionando 2 o más tablas se debe usar alguno de los tipos de uniones, (Join), que existen.

Para la demostración, se utilizarán las tablas de artistas y álbumes de la base de datos de ejemplo.



En esta relación entre las tablas se puede ver que un artista puede tener cero o muchos álbumes, mientras que un álbum pertenece a un artista. Esto es conocido como cardinalidad de la relación entre las 2 tablas.

Para ayudar a comprender la notación, a continuación se muestra una tabla con las cardinalidades de las relaciones.



Para consultar datos de las tablas de artistas y álbumes, puede utilizar una cláusula INNER JOIN, LEFT JOIN o CROSS JOIN. Cada cláusula de unión determina cómo se utilizan los datos de una tabla para hacerlos coincidir con las filas de otra tabla y esto tiene un efecto directo además de en el resulatdo, en el tiempo de cómputo necesario para su cálculo.

INNER JOIN

Este tipo de unión selecciona registros de las tablas que tienen valores coincidentes en ambas tablas. Para ello, se debe determinar los campos de las tablas que van a ser utilizados como referencia.

La siguiente sentencia devuelve los títulos de los álbumes y sus nombres de artista a través del campo `ArtistId` que está presente en ambas tablas:

```
SELECT
    Title,
    Name
FROM
    albums
INNER JOIN artists
    ON artists.ArtistId = albums.ArtistId;
```

```
dbGetQuery(conn,"SELECT Title, Name FROM albums INNER JOIN artists ON
artists.ArtistId = albums.ArtistId;")
```

En este ejemplo, la cláusula INNER JOIN hace coincidir cada fila de la tabla de álbumes con cada fila de la tabla de artistas basándose en la condición de unión (`artists.ArtistId = albums.ArtistId`) especificada después de la palabra clave ON.

Si la condición de unión es TRUE (o 1), se incluyen en el conjunto de resultados las columnas de las filas `title` de las tablas de álbumes y `name` de la tabla de artistas.

Si, como en este caso, los nombres de las columnas de las tablas unidas sean los mismos, como es el caso de `ArtistId`, se puede utilizar la sintaxis USING de la siguiente manera:

```
SELECT
   Title,
   Name
FROM
   albums
INNER JOIN artists USING (ArtistId);
```

```
dbGetQuery(conn,"SELECT Title, Name FROM albums INNER JOIN artists USING
(ArtistId)")
```

LEFT JOIN

Este tipo de unión izquierda incluye en el resultado todos los registros de la primera tabla (izquierda), incluso si no existen valores coincidentes para registros en la segunda tabla (derecha).

Por ejemplo, para seleccionar los nombres de los artistas y los títulos de los álbumes de las tablas `artists` y `albums` se puede hacer utilizando la cláusula LEFT JOIN de la siguiente manera:

```
SELECT
    Name,
    Title
FROM
    artists
LEFT JOIN albums ON
    artists.ArtistId = albums.ArtistId
ORDER BY Name;
```

```
dbGetQuery(conn, "SELECT Name, Title FROM artists LEFT JOIN albums ON
artists.ArtistId = albums.ArtistId ORDER BY Name")
```

La unión izquierda devuelve todas las filas de la tabla de artistas (o tabla izquierda) y las filas coincidentes de la tabla de álbumes (o tabla derecha).

Si una fila de la tabla izquierda no tiene una fila coincidente en la tabla derecha, se incluye las columnas de la fila de la tabla izquierda y NULL para las columnas de la tabla derecha, tal y como se puede ver en la primera fila del resultado.

De forma similar a la cláusula INNER JOIN, se puede utilizar la sintaxis USING para la condición de unión izquierda cuando el nombre de las columnas que hacen la unión sean el mismo.

Ejercicio de uso de LEFT JOIN

Escriba la consulta que encuentre los artistas que no tengan ningún álbum.

```
dbGetQuery(conn, "")
```

CROSS JOIN

La cláusula CROSS JOIN crea un producto cartesiano de filas de las tablas unidas. Esto es, crea una combinación de todas las filas de la tabla de la izquierda con todas las filas de la tabla de la derecha.

A diferencia de las cláusulas INNER JOIN y LEFT JOIN, una CROSS JOIN no tiene una condición de unión. A continuación se muestra la sintaxis básica de la cláusula CROSS JOIN:

```
SELECT
    lista_campos_seleccionados
FROM tabla1
CROSS JOIN tabla2;
```

El CROSS JOIN combina cada fila de la primera tabla (tabla1) con cada fila de la segunda tabla (tabla2) para formar el conjunto de resultados.

Si la primera tabla tiene N filas y la segunda tabla tiene M filas, el resultado final tendrá NxM filas.

El uso práctico de la cláusula CROSS JOIN es combinar dos conjuntos de datos para formar un conjunto de datos inicial para su posterior procesamiento. Por ejemplo, la tercera consulta del siguiente script crea las tablas de géneros y tipos de medio. Como el campo de nombre se repite en ambas tablas, se usa el nombre de tabla para distinguirlos. Las dos consultas primeras se presentan para poder ver el contenido de las tablas que luego forman el CROSS JOIN.

```
dbGetQuery(conn,"Select name FROM media_types")
dbGetQuery(conn,"Select name FROM genres")
dbGetQuery(conn,"Select media_types.name, genres.name from media_types cross join
genres")
```

En este ejemplo se puede observar que el CROSS JOIN devuelve 125 filas, resultado de multiplicar 5 x 25, que son las filas de las tablas de tipos de medio y géneros, respectivamente.

Cláusula GROUP BY

La cláusula GROUP BY es una cláusula opcional de la sentencia SELECT que permite agrupar filas de la salida por el valor de una o más columnas.

La cláusula GROUP BY devuelve una fila por cada grupo. Para cada grupo, puede aplicar una de las funciones de agregación existentes MIN, MAX, SUM, COUNT o AVG para proporcionar más información sobre cada grupo.

La siguiente sentencia ilustra la sintaxis de la cláusula GROUP BY de SQLite.

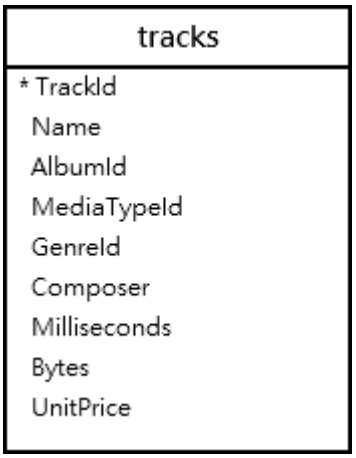
```
SELECT
    columna_1,
    función_agregada(columna_2)
FROM
    tabla
GROUP BY
    columna_1,
    columna_2;
```

La cláusula GROUP BY va después de la cláusula FROM de la sentencia SELECT. Si una sentencia contiene una cláusula WHERE, la cláusula GROUP BY debe ir después de la cláusula WHERE.

Después de la cláusula GROUP BY aparecerán la columna o la lista de columnas separadas por comas que se utilizarán para especificar el grupo.

Ejemplos de GROUP BY

Para la demostración utilizamos la tabla `tracks` de la base de datos de ejemplo.



Cláusula GROUP BY con COUNT

La siguiente sentencia devuelve el identificador del álbum y el número de pistas por álbum utilizando la cláusula GROUP BY para agrupar las pistas por álbum y aplica la función COUNT() a cada grupo.

```
SELECT
    albumid,
    COUNT(trackid)
FROM
    tracks
GROUP BY
    albumid;
```

```
dbGetQuery(conn,"SELECT albumid, COUNT(trackid) FROM tracks GROUP BY albumid")
```

Cláusula GROUP BY con ORDER BY

Si se quisiera obtener esa misma lista de álbumes pero ordenados por el orden decreciente del número de pistas se podría combinar la consulta con la cláusula ORDER BY como sigue:

```
SELECT
    albumid,
    COUNT(trackid)
FROM
    tracks
GROUP BY
    albumid
ORDER BY COUNT(trackid) DESC;
```

```
dbGetQuery(conn,"SELECT albumid, COUNT(trackid) FROM tracks GROUP BY albumid ORDER
BY COUNT(trackid) DESC")
```

Cláusula GROUP BY con INNER JOIN

Si además se quisiera tener ese mismo listado, pero con el nombre del álbum en lugar de su identificador, se podría usar una unión INNER JOIN como sigue:

```
SELECT
    tracks.albumid,
    title,
    COUNT(trackid)
FROM
    tracks
INNER JOIN albums ON albums.albumid = tracks.albumid
GROUP BY
    tracks.albumid;
```

```
dbGetQuery(conn,"SELECT tracks.albumid, title, COUNT(trackid) FROM tracks INNER
JOIN albums ON albums.albumid = tracks.albumid GROUP BY tracks.albumid ")
```

Cláusula GROUP BY con HAVING

Si ahora se quisiera obtener ese mismo listado, pero sólo para aquellos álbumes con más de 15 pistas, se podría lanzar la siguiente consulta:

```
SELECT
    tracks.albumid,
    title,
    COUNT(trackid)
FROM
    tracks
INNER JOIN albums ON albums.albumid = tracks.albumid
GROUP BY
    tracks.albumid
HAVING COUNT(trackid) > 15;
```

```
dbGetQuery(conn,"SELECT tracks.albumid, title, COUNT(trackid) FROM tracks INNER
JOIN albums ON albums.albumid = tracks.albumid GROUP BY tracks.albumid HAVING
COUNT(trackid) > 15;")
```

Cláusula GROUP BY con SUM

Se puede utilizar la función SUM para calcular el total por grupo. Por ejemplo, para obtener la longitud total y los bytes de cada álbum se utiliza la función SUM para calcular el total de milisegundos y bytes. En este caso, a continuación de las columnas calculadas se añade el título que va a adoptar esa columna calculada. En este caso la longitud (`length`) y el tamaño (`size`).

```
SELECT
    albumid,
    SUM(milliseconds) length,
    SUM(bytes) size
FROM
    tracks
GROUP BY
    albumid;
```

```
dbGetQuery(conn, "SELECT albumid, SUM(milliseconds) length, SUM(bytes) size FROM
tracks GROUP BY albumid;")
```

Cláusula GROUP BY con MAX, MIN y AVG

Mediante la siguiente sentencia se pueden obtener el identificador del álbum, el título del álbum, la duración máxima, la duración mínima y la duración media de las pistas de cada uno de los álbumes. Como hay 2 tablas implicadas, habrá que hacer un INNER JOIN para relacionar los datos que buscamos.

```
SELECT
    tracks.albumid,
    title,
    min(milliseconds),
    max(milliseconds),
    round(avg(milliseconds),2)
FROM
    tracks
INNER JOIN albums ON albums.albumid = tracks.albumid
GROUP BY
    tracks.albumid;
```

```
dbGetQuery(conn,"SELECT tracks.albumid, title, min(milliseconds),
max(milliseconds), round(avg(milliseconds),2) FROM tracks INNER JOIN albums ON
albums.albumid = tracks.albumid GROUP BY tracks.albumid;")
```

Se puede observar como el encabezamiento de las columnas calculadas muestra la fórmula usada en cada una de ellas. Esto se puede cambiar usando la técnica vista en el ejemplo anterior donde se indica junto a las columnas calculadas el encabezamiento que debe aparecer.

Desconexión con la BD

Una vez llegados al final del ejercicio, es el momento de desconectar con la BD, ya que de no hacerlo estaremos consumiendo recursos del ordenador.

Hay que tener en cuenta que si se necesitara volver a acceder a información de la BD, se debe volver a conectar con ella con el comando `dbConnect()` que se vio al principio.

El comando para desconectar con la BD es el siguiente.

```
dbDisconnect(conn)
```

Importación y análisis de datos públicos en R

Acceder a los datos es el primer paso para poder luego tratar los datos. En la mayoría de los casos la herramienta elegida para hacer esto es Excel, ya que basta con hacer doble click sobre el archivo `.xsls` elegido. La mayoría de la gente está acostumbrada a hacer doble clic en su archivo de datos y hacer que algún software como Excel lo abra para después trabajar con él.

Pero en nuestro caso pretendemos extraer información utilizando R, así que deberemos importarlo primero. Aunque es posible utilizar la utilidad de importación automática de un conjunto de datos en RStudio, `File > Import Dataset`, vamos a hacerlo de una manera más controlada con comandos.

Hay muchos paquetes que permiten a R importar todo tipo de datos, incluyendo entre ellos archivos **CSV**, **Excel**, **de texto delimitado**, **JSON** y **SPSS**, entre ellos.

En nuestro caso, vamos a centrarnos en la importación de datos desde archivos CSV y de texto delimitado.

Importar y Exportar Datos de Archivos CSV

Los archivos **CSV** (*comma-separated values*) son un tipo de documento sencillo para representar datos en forma de tabla, en las que las columnas se separan por comas (o puntos y coma) y las filas por saltos de línea. Los archivos separados por comas son la forma más común de guardar hojas de cálculo que no requieren de un programa expecífico para abrirse y pueden leerse con un editor de textos sencillo.

Aunque se pueden importar CSV utilizando las funcionalidades básicas de R, resulta recomendable utilizar el paquete `readr`. Hay que recordar para para instalar este paquete, se puede indicar el nombre en el panel `Packages > Install` de RStudio o usar el comando `install.packages("readr")` en la consola. El paquete `readr` lee datos tabulares rápidamente y asume que los caracteres son `string` y no factores por defecto.

En este caso vamos a trabajar con el conjunto de datos que está en el archivo `Titanic.csv` dentro de la carpeta `repositorios` que está accesible en el campus virtual y que te habrás bajado cuando has descargado este cuaderno.

```
library(readr)  # carga la libreria
titanic <- read_csv("repositorios/Titanic.csv") # importar el archivo csv
head(titanic)  # visualizar las primeras filas del data frame en el que se ha
cargado el archivo
```

Los datos se pueden importar directamente si el archivo está en el ordenador. Pero si se conoce la dirección URL del archivo, no será necesario descargarlo primero al ordenador para después importarlo, ya que se puede importar a R directamente desde la web utilizando la URL en lugar del nombre del archivo en la función `read_csv()`.

Este archivo está basado en este otro conjunto de datos del Titanic que contiene información sobre los pasajeros a bordo y que es un ejemplo muy utilizado para hacer análisis de datos y como base para ejercicios de aprendizaje máquina.

Exportar CSV

Una vez que se hayan realizado el análisis o transformación de los datos en el `data frame`, se puede almacenar en un archivo `CSV` con la función `write_csv()` también del paquete `readr`.

```
write_csv(titanic, "repositorios/Titanic_v2.csv")  # salvar, por ejemplo, la
versión 2 del archivo
```

Importar Datos Delimitados por Texto

A veces encontrarás datos con valores delimitados (separados) por caracteres distintos de la coma. Esto puede ser debido a que los propios datos incluyen comas, como el separador de decimales, y entonces hay que elegir otra forma de separar los campos. En otras ocasiones esto se debe a que los datos se exportaron de una base de datos y el usuario eligió esta opción.

Una de las casos más frecuentes es encontrarse con archivos donde se ha utilizado un tabulador como separador. En este caso se puede utilizar la función `read_tsv()`, como en el siguiente ejemplo.

```
titanic2 <- read_tsv("repositorios/Titanic.tsv")
titanic2
```

Para leer archivos de datos delimitados por texto en el que se ha usado cualquier otro carácter como delimitador, se puede usar `read_delim()` de readr en el que se indica el carácter concreto que se ha usado como delimitador.

Por ejemplo, el siguiente fragmento de código permite leer el archivo `Titanic.txt` en el que se ha usado `|` como delimitador indicándolo en el argumento `delim=|`.

```
titanic3 <- read_delim("repositorios/Titanic.txt", delim="|")
```

Analizando los datos del Titanic

Una vez que se han mostrado diferentes modos de cargar datos en R, se van a mostrar los métodos adecuados para transformar y analizar esos datos.

En este caso se utilizará el paquete `dplyr`.

```
library(dplyr)
```

Este paquete aporta funciones con nombres similares a verbos y convierte el análisis de datos en una especie de ejercicio de gramática que simplifica el código que habría que escribir si se usaran las funciones básicas de R. Esto facilita tanto la escritura del código como la posterior lectura y modificación del mismo.

Antes de presentar las posibilidades de analizar y transformar datos que nos ofrecen estas librerías, prestemos un poco de atención a la estructura de datos presente en el archivo `Titanic.csv`.

Vamos a utilizar las funciones verbales básicas de `dplyr` para analizar los datos y ver si hay alguna historia que merezca la pena investigar.

```
View(titanic)
```

Este comando abre una nueva pestaña donde se muestra el contenido del `data frame` donde hemos cargado los datos. Se puede observar claramente que cada columna es una variable, cada fila es una observación (que se corresponde con uno de los pasajeros) y cada celda contiene un único valor.

Las columnas y su descripción son las siguientes

Variable	Definición	Valores
id	Número de identificación del pasajero	1/2/3 ...
sex	Sexo/Género	male/female
age	Edad	
sibsp	El número de hermanos / cónyuges del pasajero a bordo	0/1/2 ...
parch	Número de padres / hijos a bordo	0/1/2 ...
fare	Tarifa pagada	
embarked	Puerto de embarque	Cherbourg, Queenstown o Southampton
class	Clase del billete	First / Second / Third
who	Descripción del pasajero	male, female, child
alone	Viajaba solo en el barco	FALSE/TRUE
survived	Superviviente	0/1

Se puede ver cómo en la columna `age` para algunas filas no existen valores (*NA*).

A continuación vamos a echar un vistazo a los datos.

Número de Pasajeros

Para saber de cuántos pasajeros tenemos datos, podemos contar las filas del data frame.

```
nrow(titanic)
```

Puertos de Embarque

Para ello tenemos que contar cuántas ciudades distintas aparecen en `embarked`. Para ello creamos un array `puertos_embarque` con la salida de la función `unique()` aplicada a la columna `embarked`. Si contamos el número de elementos en el array, conoceremos el número de ciudades desde donde se embarcaron los pasajeros.

```
# Hacer un vector con la lista de ciudades con la función unique()
puertos_embarque <- unique(titanic$embarked)
```

```
# Contar el número de elementos en el vector
length(puertos_embarque)

puertos_embarque
```

En este caso, el array `puertos_embarque` resulta que tiene 4 elementos pues se incluye el valor ausente *NA*, dándonos un resultado incorrecto.

```
puertos_embarque
```

Para evitar que se consideren los valores ausentes, podemos usar la función **`na.omit()`** , ya que elimina las filas que contienen valores *NA* (ausentes) antes de extraer los valores únicos utilizando **`unique()`**.

```
puertos_embarque <- unique(na.omit(titanic$embarked))
length(puertos_embarque)
puertos_embarque
```

Vistazo General

Con la función `glimpse()` podemos dar un vistazo a las variables del conjunto de datos y al tipo de datos.

```
glimpse(titanic)
```

Ahora podemos utilizar las funciones de `dplyr` para hacer preguntas sobre los datos:

- `filter()`
- `select()`
- `arrange()`
- `mutate()`
- `summarize()` junto con `group_by()`

Filtrar Filas

Con `filter()` se pueden extraer algunos registros que cumplan con una condición dada, pasando como argumentos el `data frame` seguido de la condición lógica.

```
filter(titanic, sex=="male", age > 60) # pasajeros masculinos mayores de 60

filter(titanic, sex=="male" & age > 60) # equilamente a la anterior
```

Hay que tener en cuenta que en R un '=' es lo mismo que '<-' y es un operador de asignación. El operador de igualdad en las pruebas lógicas es '==', El operador '&' es el y lógico (AND) y '|' será el lógico (OR). Hay que tener claro cómo funcionan los operadores lógicos para no establecer condiciones equivocadas o erróneas.

Operadores Lógicos

Operador	Descripción
`<`	Menor que
`<=`	Menor o igual a
`>`	Mayor que
`>=`	Mayor o igual a
`==`	Exactamente igual a
`!=`	No igual a
`!x`	No x
`x | y`	x OR y
`x & y`	x AND y
`%in%`	Pertenencia a un grupo
`isTRUE(x)`	Prueba si x es TRUE
`is.na(x)`	Prueba si x es NA
`!is.na(x)`	Prueba si x no es NA

Por ejemplo, para averiguar los datos de los pasajeros de primera o segunda clase se pueden hacer varias consultas, unas más eficientes que otras.

```
filter(titanic, class == "First" | class =="Second") # Correcta, pero

filter(titanic, class %in% c("First", "Second")) # esta es más eficiente
```

La consulta de pasajeros mayores de 20 y menores de 30 servirá como ejemplo de un error común.

```
> filter(titanic, 20 < age < 30 ) # Error
```

```
filter(titanic, age > 20, age < 30 ) # Correcta
filter(titanic, age > 20 & age < 30 ) # Esta también
```

Ejercicio Filtrado

> Pasajeros que, siendo menores de edad, viajaban solos.

```

```

Seleccionar Columnas

Cuando se desea restringir el conjunto de columnas se puede usa la función `selecc()` enumerando los nombres de las columnas deseadas a continuación del marco de datos del que desea extraer los datos.

```
select(titanic, class, fare) # Sólo tarifa y clase para TODOS los pasajeros

select(titanic, id:age) # Columnas entre id y age
```

Eliminación de columnas

Del mismo modo, se podría usar select() para eliminar columnas usando `-` con el nombre de columna o columnas a eliminar.

Por ejemplo, el código siguiente serviría para eliminar la columna de la ciudad de embarque del cojunto de datos del Titanic.

```
titanic <- select(titanic, -embarked) # Se escluyen algunas columnas
```

Ejercicio Selección

> Ciudad de embarque y número de hermanos embarcados para **TODOS** los pasajeros.

```

```

Ordenar Filas

La función `arrange()` permite ordenar un `data frame` por una o más columnas. La primera columnas tendrá prioridad y las variables restantes servirán para desempatar. El orden por defecto es ascendente, pero se puede modificar con la función `desc()`.

```
arrange(titanic, age, desc(fare))
```

Ejercicio Ordenación

> Ordene los pasajeros por la clase y de forma descendente por la tarifa asociada al billete.

```
```

Crear Columnas

Se pueden crear nuevas variables (columnas) con `mutate()`, indicando tras el data frame el nombre de la nueva columna y la fórmula a aplicar.

Por ejemplo, se podría hacer una nueva columna denominada `relatives`, con la suma de herminanos, hijos y cónyuges.

```
mutate(titanic, relatives = siblings_abroad + parents_children_abroad)
```

Para ello se pueden usar los siguientes operadores matemáticos:

Operadores Aritméticos

Operador	Descripción
`+`	Suma
`-`	Subtracción
`*`	Multiplicación
`/`	División
`^`	Exponenciación

Ejercicio Creación Columnas

> Cree una nueva columna `discount` que almacene el descuento a aplicar si se hiciera un 5% de descuento por cada hijo o padre que acompañara al pasajero.

```
```

Ejemplo de traducción columna categórica

> Crear una nueva columna numérica relativa a la clase, que es categórica.

```
# Definir reglas de asignación numérica para cada categoría
categoria_a_numero <- c("First" = 1, "Second" = 2, "Third" = 3) # crear un array
con los números a conseguir accesible por las etiquetas de la categoría
```

```
# Usar mutate para crear una nueva columna numérica basada en las reglas
titanic_clases <- mutate(titanic, num_class = categoria_a_numero[class])

# Mostrar el resultado
titanic_clases
```

Renombrar Columnas

Se puede renombrar variables (columnas) fácilmente con la función `rename()`. Sólo hay que indicar el nuevo nombre como si fuera una asignación `nuevo_nombre = nombre_actual`.

Por ejemplo, podemos renombrar las columnas de hermanos y cónyuges e hijos a bordo por algo más corto.

```
rename(titanic, siblings = siblings_abroad, parchild = parents_children_abroad)
```

La operación de renombrado se puede hacer junto con la de selección, ahorrando así operaciones.

Ejercicio Renombrado

> Extraer las columnas relativas al id, el número de hermanos, los cónyuges e hijos renombrando las columnas como en el ejemplo anterior para todos los pasajeros que no viajaban solos.

```

```

Agregación

Mediante la función `summarize()` se puede obtener una visión simplificada de la tabla mediante la agregación de filas, al estilo de las tablas dinámicas de Excel (pivots).

Funciones de Agregación

Función	Descripción
`mean(x)`	Media aritmética. `mean(c(1,10,100,1000))` devuelve 277.75
`median(x)`	Mediana. `median(c(1,10,100,1000))` devuelve 55
`sd(x)`	Desviación estandar. `sd(c(1,10,100,1000))` devuelve 483.57
`sum(x)`	Suma. `sum(c(1,10,100,1000))`devuelve 1111
`min(x)`	Mínimo. `min(c(1,10,100,1000))` devuelve 1
`max(x)`	Máximo. `max(c(1,10,100,1000))` devuelve 1000
`abs(x)`	Valor absoluto. `abs(-8)` devuelve 8

Función	Descripción
`n()`	Número de valores / filas
`n_distinct()`	Número de valores únicos
`first()`	Primer valor de un grupo
`last()`	Último valor de un grupo
`nth()`	N-ésimo valor de un grupo

Por ejemplo, se pueden extraer los números mínimo, máximo y medio de hermanos para los pasajeros.

```
summarize(titanic,  min_siblings=min(siblings_abroad),
                    max_siblings=max(siblings_abroad),
                    avg_siblings=mean(siblings_abroad),
          sd_siblings=sd(siblings_abroad))
```

Este resumen se puede hacer agrupando las filas por algún valor usando `group_by()`. Por ejemplo, en el caso anterior, podría ser interesante hacer los cálculos respecto a los hermanos agrupando los pasajeros en función de la clase.

```
summarize(group_by(titanic, class),  min_siblings=min(siblings_abroad),
                    max_siblings=max(siblings_abroad),
                    avg_siblings=mean(siblings_abroad))
```

Ejercicio Agregación

> Calcule el número de tarifas distintas, la tarifa máxima, media y mínima para cada clase.

```

```

Tuberías `%>%`

Los análisis de datos suelen implicar más de un paso.

Vamos a repasar algunas opciones sobre cómo enfocar los procesos de varios pasos con el conjunto de datos del Titanic para, a continuación, mostrar la mejor manera de tratarlos.

El proceso teórico a ejecutar en el marco de datos del Titanic para obtener las clases ordenadas por numero decreciente de pasajeros que embarcaron en Southampon es el siguiente:

1. Extraer las filas de los pasajeros embarcados en Southampton,
2. Agrupar los pasajeros por clase y luego
3. Contar el número de pasajeros

4. Ordenar los clases por orden creciente de pasajeros

Este proceso se puede hacer de distintas maneras.

Opción 1

Hacer una secuencia de operaciones guardando el resultado en un nuevo data frame.

```
Southampton_1 <- filter(titanic, embarked=="Southampton")

Southampton_2 <- group_by(Southampton_1, class)

Southampton_3 <- summarize(Southampton_2, total=n())

Southampton_4 <- arrange(Southampton_3, desc(total))

head(Southampton_4)
```

Esto produce que se creen múltiples estructuras de datos que hay que almacenar en memoria. Se puede comprobar esto en el panel `Environment`, donde se pueden encontrar los múltiples data frames creados.

Opción 2

Se puede hacer todo en una línea anidando funciones.

```
Southampton <- arrange(summarize(group_by(filter(titanic,
embarked=="Southampton"), class), total=n()), desc(total))

head(Southampton)
```

Estas segunda opción, aunque no malgasta tanta memoria como la anterior y es eficiente con la codificación, no facilita seguir lo que está pasando con los datos.

Es fácil caer en la cuenta que el primer argumento en las funciones de `dplyr` es un marco de datos y que todas ellas producen como salida otro marco de datos.

Es aquí donde entra en juego el operador de tubería `%>%` (o *pipe*).

En esencia, un `%>%` es el equivalente gramatical *de y a continuación* ... y permite ejecutar varias funciones conectando la entrada de una con la salida de la anterior, a modo de tubería.

Por ejemplo, en el caso anterior donde se extraían las columnas relativas al id, el número de hermanos, los cónyuges e hijos renombrado las columnas como antes para todos los pasajeros que no viajaban solos, esto se puede hacer con pipes así:

```
filter(titanic,!alone) %>% select(id, siblings = siblings_abroad, parchild =
parents_children_abroad)
```

Opción 3

En este caso usaremos pipes `%>%`.

```
filter(titanic, embarked=="Southampton") %>%
  group_by(class) %>%
  summarize(total=n()) %>%
  arrange(desc(total))
```

Ejercicio `%>%`

> Obtenga la lista de ciudades de embarque ordenada por el número de pasajeros que no sobrevivieron.

```

```

Visualizando Datos

Una de las capacidades más útiles relacionada con el análisis de datos es la visualización de los mismos, ya que muestra mucha información que es más difícil observar en una tabla.

Para visualizar los datos se utilizará la librería `ggplot2`.

```
library(ggplot2)
```

Como parte de este curso no se pretende presentar todas las posibilidades que aporta esta librería a la hora de mostrar gráficas con los datos, ya que esto está fuera del objetivo del curso. Sin embargo, vamos a presentar unos cuantos gráficos sencillos que nos pueden dar una idea de las posibilidades de `gpplot2`. Para ello se van a plantear algunas preguntas sobre los datos del Titanic que se intentarán responder de manera gráfica.

Diagrama de Barras

Como primer ejemplo de gráfico, se trata de mostrar la distribución de clases entre los pasajeros. A continuación se muestra el código que hace eso posible.

```
ggplot(titanic) +
   aes(x=class) +
    geom_bar()
```

En él se pueden distinguir 3 partes. Hay que tener en cuenta, que en lugar de una tubería `%>%`, `ggplot2` requiere un `+` entre funciones. Esto es debido a que este paquete fue creado antes de que se implementaran las tuberías `%>%`.

La primera función `ggplot(titanic)` indica el conjunto de datos a considerar, el marco de datos `titanic`. La segunda función `aes(x=class)` define la estética de la gráfica, esto es, los datos a considerar y algunos parámetros más, indicando que en el eje `x` se dispongan los distintos valores de la columna `class`. Como no se indica qué variable se representará en el eje `y`, será el conteno del número de pasajeros para cada una de las clases. La tercera función, `geom_bar()`, determina el tipo de gráfico a crear, un diagrama de barras en este caso.

Esta es sólo una de las formas que puede presentar la invocación a la función `ggplot()`, y en los ejemplos siguientes veremos algunas variantes de ella.

Veámos cuál es el gráfico resultado de ejecutar dicho fragmento de código.

```
ggplot(titanic) +
   aes(x=class) +
    geom_bar()
```

El resultado es un diagrama de barras con los pasajeros de cada clase. Se puede observar cómo la más numerosa es la clase tercera, siendo, curiosamente, la segunda clase la menos frecuente.

Este gráfico es informativo, pero resulta demasiado plano y podría representar más información. Con un pequeño cambio, podríamos representar, además, el puerto de embarque de los pasajeros para cada una de las clases. Esto se consigue añadiendo el argumento `fill=embarked` a la función `aes()`, tal y como se puede ver a continuación.

```
ggplot(titanic) +
   aes(x=class, fill=embarked) +
    geom_bar()
```

Ahora, además sabremos que el puerto donde embarcó más gente fue Shouthampton. Se podría mejorar este gráfico para conocer con más precisión las proporciones de cada puerto de embarque en las clases transformando el gráfico haciendo que las barras mostraran el porcentaje de pasajeros por puerto de embarque. Esto se puede hacer añadiendo el argumento `position="fill"` a la función `geom_bar()`, tal y como se muestra a continuación.

```
ggplot(titanic) +
   aes(x=class, fill=embarked) +
    geom_bar(position="fill")
```

Ahora sabemos, además, que casi el 90% de los pasajeros de segunda clase embarcaron en Southampton.

Otros gráficos interesantes del mismo estilo serían los que muestran la distribución de hombres, mujeres y niños entre los pasajeros.

Los siguientes 2 fragmentos de código generan esos gráficos, añadiendo, además, la distribución entre clases para cada tipo de pasajero.

En el primero considerando el número total de cada uno de los tipos de pasajero.

```
ggplot(titanic,
       aes(x=who, fill=class))+
    geom_bar()
```

En él podemos observar cómo la mayoría de los pasajeros eran hombres.

Veamos ahora cuál es la proporción de cada clase entre hombres, mujeres y niños.

```
ggplot(titanic,
       aes(x=who, fill=class))+
    geom_bar(position="fill")
```

Se puede ver que más de la mitad de los hombres eran de tercera clase. En cambio, en el caso de las mujeres, las de tercera clase eran menos del 50%.

Diagrama de Puntos

Este tipo de diagramas resulta interesante cuando se quieren enfrentar 2 variables representando una de ellas en el eje `X` y la otra en el eje `Y`. El tipo de función a utilizar para hacer este tipo de gráficas es `geom_point()`.

Siguiendo con el análisis de los datos del Titanic, podría ser interesante ver la relación que hay entre la clase y la tarifa del billete. A continuación se muestra el código que consigue esta gráfica. En este caso, la función `aes()` se pasa como un argumento a la función `geom_point()` en lugar de utilizar el operador `+` para ello. El gráfico muestra las clases en el eje `X` (`x=class`) y la tarifa en el eje `Y` (`y=fare`).

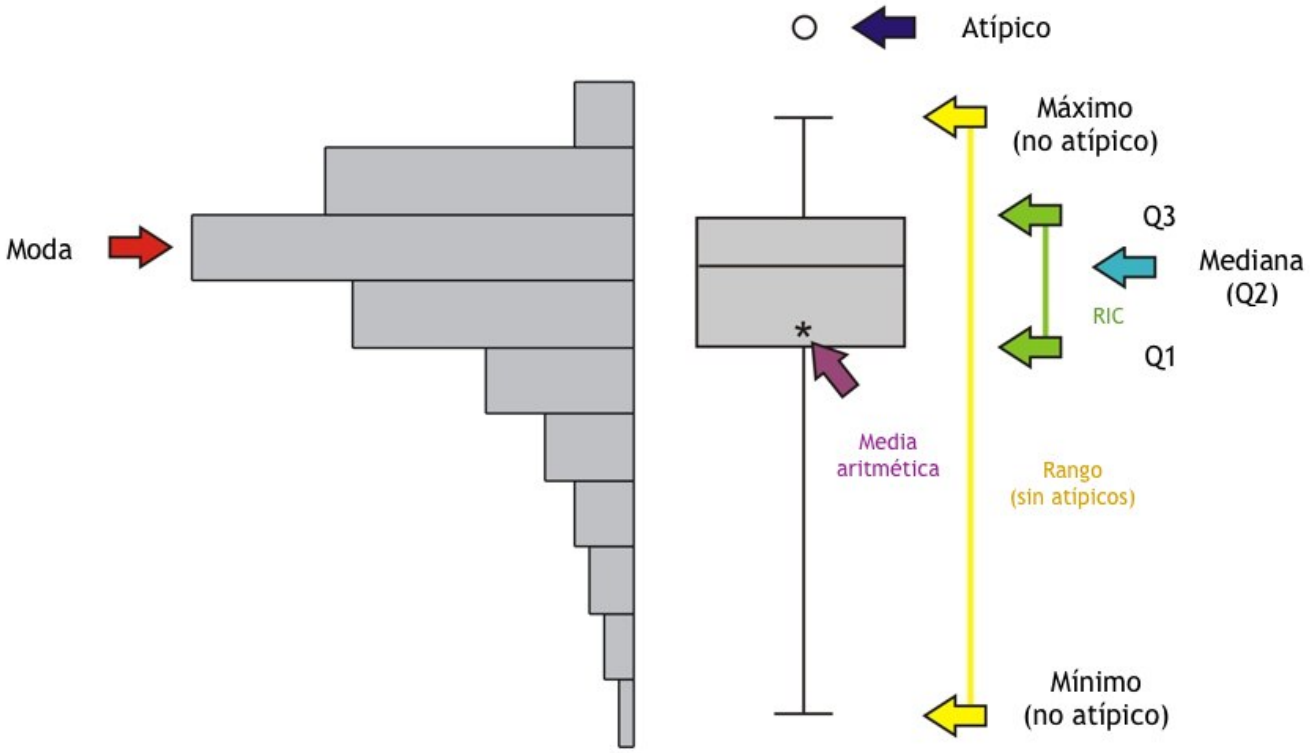
```
ggplot(data=titanic) +
   geom_point(mapping=aes(x=class, y=fare))
```

En él vemos que existe una mayor diferencia entre las tarifas de la primera clase que la que se observa en las otra dos clases. Para poder analizar esa dispersión con más precisión necesitamos el siguiente tipo de diagrama.

Diagramas de Caja

Estos diagramas permiten representar gráficamente una serie de datos numéricos a través de sus cuartiles. De esta manera, se muestran a simple vista la mediana y los cuartiles de los datos, pudiendo también

representarse los valores atípicos.



Los diagrama de caja se obtienen con la función `geom_boxplot()`.

A continuación se presenta el diagrama de cajas de la distribución de tarifas por clase.

```
ggplot(titanic) + aes(x=class, y=fare) +
  geom_boxplot()
```

En él se puede ver que la variabilidad de tarifas (altura de la caja) decrece de primera a tercera clase, y que la distribución de tarifas en la primera clase es más simétrica que en las otras dos (la mediana está más centrada en la caja). Las lineas verticales (a menudo llamadas bigotes) muestran el intervalo entre el máximo y el mínimo de precios, y en este caso permiten observar que en todas las clases se encuentran tarifas de valor 0,0. También se puede ver que existen múltiples casos que resultan atípicos en todas las clases (los puntos que están fuera de las cajas) y que estos siempre son por arriba, significando precios mucho más altos que el resto para cada clase.

Ejercicios Visualización

Realice varias gráficas que ayuden a comprender la distribución de supervivientes/fallecidos entre los pasajeros del Titanic, atendiendo a las diferentes clases y a la condición de hombre, mujer y niño.