Programación para Ciencia de Datos

Maestría en Ciencia de Datos

2019

Adolfo De Unánue

Estas notas son para **acompañar** la clase de *Programación para Ciencia de datos*.
Notas creadas usando `GNU Emacs` y `Org-mode`.

Programación para Ciencia de Datos
Maestría en Ciencia de Datos, ITAM
Ciudad de México

Índice general

1 Introducción

Lorem ipsum …

2 Línea de comandos

2.1 La computadora

Las computadoras sólo hacen cuatro cosas:

- Ejecutan programas
- Almacenan datos
- Se comunican entre sí para hacer las cosas recién mencionadas.
- Interactúan con nosotros.
  - La interacción puede ser gráfica (como están acostumbrados) conocida también como **GUI** (*Graphical User Interface*) vía el ratón u otro periférico, o desde la línea de comandos, llamada como **CLI** (*Command Line Interface*).

2.2 Introducción

El `shell` de Unix (en su caso particular es un `shell` de `GNU/Linux`), es más viejo que todos nosotros. Y el hecho de que siga activo, y en uso, se debe a que es una de las invenciones humanas más exitosas para usar la computadora de manera eficiente.
De una manera muy rápida el `shell` puede hacer lo siguiente:

- Un intérprete interactivo: lee comandos, encuentra los programas correspondientes, los ejecuta y despliega la salida.
  - Esto se conoce como **REPL**: *Read, Evaluate, Print, Loop*
- La salida puede ser redireccionada a otro lugar además de la pantalla. (Usando > y <).

- Una cosa muy poderosa (y en la que está basada –como casi todo lo actual–) es combinar comandos que son muy básicos (sólo hacen una sola cosa) con otros para hacer cosas más complicadas (esto es con un **pipe** `|`).
- Mantiene un histórico que permite rejecutar cosas del pasado.
- La información es guardada jerárquicamente en carpetas o directorios.
- Existen comandos para hacer búsquedas dentro de archivos (`grep`) o para buscar archivos (`find`) que combinados pueden ser muy poderosos.
  - Uno puede hacer **data analysis** solamente con estos comandos, así de poderosos son.
- Las ejecuciones pueden ser pausadas, ejecutadas en el **fondo** o en máquinas remotas.
- Además es posible definir variables para usarse por otros programas.
- El `shell` cuenta con todo un lenguaje de programación, lo que permite ejecutar cosas en **bucles**, **condicionales**, y hasta cosas en paralelo.

2.3 ¿Por qué?

En muchas ocasiones, se verán en la necesidad de responder muy rápido y en una etapa muy temprana del proceso de **big data** . Las peticiones regularmente serán cosas muy sencillas, como estadística univariable y es aquí donde es posible responder con las herramientas mágicas de UNIX.

2.4 Línea de comandos

La línea de comandos es lo que estará entre nosotros y la computadora casi todo el tiempo en este curso. De hecho, una

lectura obligada[1] es In the beginning…was de command line de **Neal Stephenson**, el escritor de **Criptonomicon**[2].

La **CLI** es otro programa más de la computadora y su función es ejecutar otros comandos. El más popular es `bash`, que es un acrónimo de **Bourne again shell**. Aunque en esta clase también usaremos `zsh`[3].

[1] No es de tarea, pero debería de serlo

[2] Otra lectura recomendada

[3] `zsh` significa *Z shell*. Lo sé, es algo decepcionante.

2.5 ¡Ten cuidado!

La primera regla de la línea de comandos es: *ten cuidado con lo que deseas, por que se te va a cumplir*. La computadora hará exactamente lo que le digas que haga, pero recuerda que los humanos tienen dificultadas para expresarse en *lenguaje de computadoras*.

Esta dificultad puede ser muy peligrosa, sobre todo si ejecutas programas como `rm` (*borrar*) o `mv` (*mover*).

2.6 Creando archivos de prueba

Puedes crear archivos *dummy* para este curso usando el comando `touch`:

```
touch space\ bars\ .txt
```

Nota que usamos el caracter `\` para indicar que queremos un espacio en el nombre de nuestro archivo. Si no lo incluyes

```
touch space bars .txt
```

…la computadora creará tres archivos separados: `space`, `bars`, and `.txt`[4].

[4] Ve la advertencia en la sección anterior

2.7 Archivos y directorios

La computadora guarda la información de una manera ordenada. El sistema encargado de esto es el `file system`, el cual es básicamente un árbol de información[5] que guarda los datos en una abstracción que llamamos **archivos** y ordena los archivos en **carpetas** o **directorios**, los cuales a su vez pueden contener otros **directorios**.

[5] Aunque hay varios tipos de `file systems` (`ext3`, `ext4`, `xfs`, `bfs`, etc) que pueden utilizar modificaciones a esta estructura de datos, lo que voy a decir aplica desde el punto de vista de usuario del *file system* no su especificación técnica.

TODO en los sistemas operativos `*nix` (como Unix, GNU/Linux, FreeBSD, MacOS, etc) es un *archivo*.

Muchos de los comandos del **CLI** o `shell` tienen que ver con la manipulación del `file system`.

2.8 Filosofía UNIX

Creada originalmente en 1978 por **Doug McIlroy**, la filosofía de UNIX es un acercamiento al diseño de software que enaltece el software modular y minimalista.

Han existido varias adaptaciones[6], pero la que más me gusta es la de **Peter H. Salus**,

Es importante tener estos principios en mente, ya que ayuda a enmarcar los conceptos que siguen.

[6] Una discusión larga y detallada se encuentra en The Art of Unix Programming de **Eric Steven Raymond**.

- Escribe programas que hagan una cosa y que la hagan bien
- Escribe programas que puedan trabajar en conjunto
- Escribe programas que puedan manipular *streams* de texto, ya que el texto es la interfaz universal.

2.9 Conocer los alrededores

2.9.1 Navegación en la terminal

Moverse rápidamente en la **CLI** es de vital importancia. Teclea en tu *terminal*

```
Anita lava la tina
```

Y ahora intenta lo siguiente:

`Ctrl + a` Inicio de la línea
`Ctrl + e` Fin de la línea
`Ctrl + r` Buscar hacia atrás[7]
`Ctrl + b` Mueve el cursor hacia atrás una letra a la vez
`Alt + b` Mueve el cursor hacia atrás una palabra a la vez
`Ctrl + f` Mueve el cursor hacia adelante una letra a la vez
`Alt + f` Mueve el cursor hacia adelante una palabra a la vez
`Ctrl + k` Elimina el resto de la línea (en realidad corta y pone en el búfer circular)
`Ctrl + y` Pega la último del búfer.
`Alt + y` Recorre el búfer circular.
`Ctrl + d` Cierra la terminal
`Ctrl + z` Manda a *background* el programa que se está ejecutando
`Ctrl + c` Intenta cancelar
`Ctrl + l` Limpia la pantalla

[7] Elimina el nefasto y tardado *flechita arriba*

Atención: Estas combinaciones de teclas (*keybindings*) son universales. Te recomiendo que las practiques y configures tus otras herramientas con estos mismas combinaciones, por ejemplo RStudio ó JupyterLab .

Pregunta 1

¿Qué hacen las siguientes combinaciones?

- `Alt + t`
- `Alt + d`
- `Ctrl + j`
- `Alt + 2 Alt + b`

Para tener más información sobre los *bindings* consulta aquí.

2.9.2 ¿Quién soy?

```
whoami
```

2.9.3 ¿Quién está conmigo?

```
who
```

2.9.4 ¿Dónde estoy?

Imprime el nombre del *directorio* actual

```
pwd
```

Cambia el directorio un nivel arriba (a el directorio *padre* [8])

```
cd ..
```

[8] En inglés es *parent directory*, no se me ocurrió otra traducción ¿Alguna sugerencia?

Si quieres regresar al directorio anterior

```
cd -
```

2.9.5 Hogar dulce, hogar

Cambia el directorio `$HOME` (tu directorio) utilizando ~

```
cd ~
```

o bien, no pasando ningún argumento

```
cd
```

2.9.6 ¿Qué hay en mi directorio (*folder*) ?

`ls` Lista los contenidos (archivos y directorios) en el directorio actual, pero no los archivos *ocultos*.

```
ls
```

Lista los contenidos en formato *largo* (`-l`), muestra el tamaño de los archivos, fecha de último cambio y permisos

```
ls -l
```

Lista los contendios en el directorio actual y todos los sub-directorios en una estructura de *árbol*

```
tree
```

Límita la expansión del *árbol* a dos niveles

```
tree -L 2
```

Muestra los archivosshows file sizes (`-s`) in human-readable format (`-h`)

```
tree -hs
```

2.9.7 ¿Qué hay en mi archivo?

Muestra el principio (*head*) del archivo, `-n` especifica el número de líneas (`10`).

```
head -n10 $f
```

Muestra la final (*tail*) del archivo.

```
tail -n10 $f
```

Muestra la parte final del archivo cada segundo (usando `watch`)

```
tail -n10 $f | watch -n1
```

"seguir" (*follows*) (`-f`) la parte final del archivo, cada vez que hay cambios

```
tail -f -n10 $f
```

Seguir archivos es útil cuando estás ejecutando un programa que guarda información a un archivo, por ejemplo un *log*

Cuenta las palabras, caracteres y líneas de un archivo

```
wc $f
```

2.9.8 ¿Dónde está mi archivo?

Encuentra el archivo por nombre

```
find -name "<lost_file_name>" -type f
```

Encuentra directorios por nombre

```
find -name "<lost_dir_name>" -type d
```

2.9.9 *Caveats* con `git`

Mover archivos puede confundir a `git`. Si estás trabajando con archivos en `git` usa lo siguiente:

```
# Para mover o renombrar
git mv /source/path/$move_me /destination/path/$move_me

# Para eliminar
git rm $remove_me
```

2.9.10 Practiquemos un poco

Enciende la máquina virtual

```
vagrant up
```

Conéctate a la máquina virtual con `vagrant`

```
vagrant ssh
```

Teclea `whoami` y luego presiona **enter**. Este comando te dice que usuario eres.
Teclea `cd /`
Para saber donde estamos en el `file system` usamos `pwd` (de *print working directory*).

Estamos posicionados en la raíz del árbol del sistema, el cual es simbolizada como `/`.

Para ver el listado de un directorio usamos `ls`

Ahora estás observando la estructura de directorios de `/`.

Los comandos (como `ls`) pueden tener modificadores o **banderas** (*flags*), las cuales modifican (vaya sorpresa) el comportamiento por omisión del comando. Intenta lo siguiente: `ls -l`, `ls -a`, `ls -la`, `ls -lh`, `ls -lha`. Discute con tu compañero junto a ti las diferencias entre las banderas.

Para obtener ayuda puedes utilizar `man` (de *manual*) y el nombre del comando.

Pregunta 2

¿Cómo puedes aprender que hace `ls`?

Puedes buscar dentro de `man page` para `ls` (o de cualquier otro manual) si tecleas `/` y escribiendo la palabra que buscas, luego presiona `enter` para iniciar la búsqueda. Esto te mostrará la primera palabra que satisfaga el criterio de búsqueda. `n` te mostrará la siguiente palabra. `q` te saca del `man page`.

Busca la bandera para ordenar (*sort*) el listado de directorios por tamaño.

Muestra el listado de archivos de manera ordenada por archivo.

Otro comando muy útil (aunque no lo parecerá ahorita) es `echo`.

Las variables de sistema (es decir globales en tu sesión) se pueden obtener con el comando `env`. En particular presta atención a `HOME`, `USER` y `PWD`.

Para evaluar la variable podemos usar el signo de moneda `$`,

Imprime las variables con `echo`, e.g.

```
echo $USER
```

Pregunta 3

¿Qué son las otras variables `HOME`, `PWD`?

El comando `cd` permite cambiar de directorios (¿Adivinas de donde viene el nombre del comando?) La sintáxis es `cd nombre_directorio`.

Pregunta 4

¿Cuál es la diferencia si ejecutas `ls -la` en ese directorio?

Las dos líneas de hasta arriba son `.` y `..` las cuales significan **este directorio** (`.`) y el directorio padre (`..`) respectivamente. Los puedes usar para navegar (i.e. moverte con `cd`)

Pregunta 5

¿Puedes regresar a raíz?

Pregunta 6

En raíz (`/`) ¿Qué pasa si ejecutas `cd $HOME`?

Otras maneras de llegar a tu `$HOME` son `cd ~` y `cd` (sin argumento).

Verifica que estés en tu directorio (¿Qué comando usarias?) Si no estás ahí, ve a él.

Para crear un directorio existe el comando `mkdir` que recibe como parámetro un nombre de archivo.

Crea la carpeta `test`. Entra a ella. ¿Qué hay dentro de ella?

Vamos a crear un archivo de texto, para esto usaremos `nano`[9]. Por el momento teclea

```
nano hola.txt
```

y presiona enter.

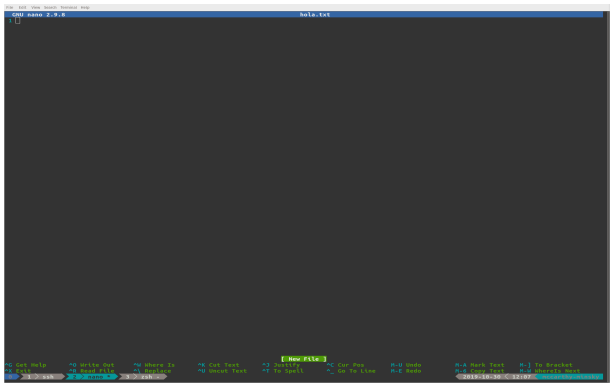

Figura 2.1
Editor `nano`, mostrando el archivo recién creado `hola.txt` .

Teclea en `nano`

```
¡Hola Mundo!
```

y luego presiona la siguiente combinación de teclas: `Ctrl+O` [10]para guardar el archivo (Te va a preguntar *dónde* guardarlo).

`Ctrl+X` para salir de `nano`. Esto te devolverá a la línea de comandos.

Verifica que esté el archivo.

Para ver el contenido de un archivo tienes varias opciones (además de abrir `nano`): `cat`, `more`, `less`

[9] Otras opciones son: `GNU Emacs` (un editor de textos muy poderoso. Es el que estoy usando) o `vi`. No importa cual escojas, aprende a usarlo muy bien. Recuerda, queremos disminuir el dolor.

[10] Las combinaciones de las teclas están desplegadas en la parte inferior de la pantalla

```
cat hola.txt
```

Para borrar usamos el comando `rm` (de *remove*).
Borra el archivo `hola.txt`.

Pregunta 7

¿Cómo crees que se borraría un directorio?
¿Puedes borrar el directorio `test`? ¿Qué falla? ¿De dónde puedes obtener ayuda?

Crea otra carpeta llamada `tmp`, crea un archivo `copiame.txt` con `nano`, escribe en él:

```
Por favor cópiame
```

Averigua que hacen los comandos `cp` y `mv`.
Copia el archivo a uno nuevo que se llame `copiado.txt`.
Borra `copiame.txt`.
Modifica `copiado.txt`, en la última línea pon

```
¡Listo!
```

Renombra `copiado.txt` a `copiame.txt`.
Por último borra toda la carpeta `tmp`.
Desconéctate de la máquina virtual con `Ctrl+D` y luego "apaga" la máquina virtual

```
vagrant halt
```

2.9.11 *Wildcards*: Globbing

La línea de comandos permite usar comodines (*wildcards*)[11] para encontrar archivos:

[11] También permite expresiones regulares (*regular expressions*, a.k.a. *regexp*), pero es importante saber que **NO** son iguales. *Globs* y *regexps* son usadas en diferentes contextos y significan diferentes cosas. Por ejemplo el símbolo `*`, es un modificador de cantidad en regexp, pero expande cuando es

El primer *glob* que veremos es `*`:

```
echo *
```

El `shell` expandió `*` para identificarlo con todos los archivos del directorio actual.
Obvio lo puedes usar con otros comandos, como `ls`: Listar todos los archivos que tienen una extensión `txt`

```
ls *.txt
```

Listar todos los archivos que contienen a en el nombre y extensión `txt`

```
ls *a*.txt
```

`*` no es el único carácter especial. `?` hace *match* con cualquier carácter individual.
Listar todos los archivos que tienen 5 caracteres en el nombre:

```
ls ?????,txt
```

2.10 Datos para jugar

- Para los siguientes ejemplos trabajaremos con los archivos encontrados en **The National UFO Reporting Center Online Database**[12]
- Estos datos representan los *avistamientos* de OVNIS en EUA.
- Usaremos como ejemplo la descarga el mes de Noviembre y Diciembre de 2014
- Se encuentran en la carpeta `data/ufo` en la máquina virtual.

[12] Por cierto, **no** creo que haya vida extraterreste, principalmente debido a la paradoja de Fermi (también ver aquí). Relacionado con esto, creo que la mejor solución a esta paradoja es la teoría del Gran Filtro (más información). Si prefieres vídeos este playlist tiene todo lo que quieres saber.

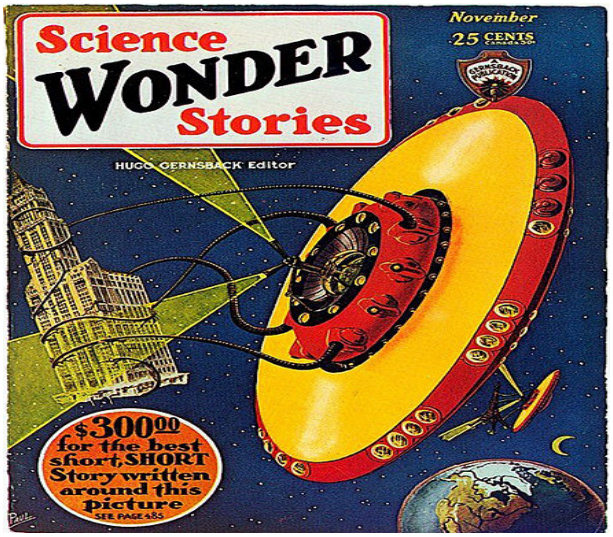

Figura 2.2
Portada de la revista *Science Wonder Stories* (1929). Imagen tomada de **Wikipedia**

2.11 Conectando comandos

2.11.1 Entubando

El símbolo | (*pipe*) "entuba" la salida de un comando al siguiente comando. Por ejemplo:

```
ls -la | wc -l
```

La salida de `ls -la` en lugar de ser impresa en pantalla[13] es enviada a `wc -l`

El siguiente ejemplo utiliza `grep` para buscar y seleccionar cadenas o patrones

```
seq 50 | grep 3
```

Veremos más sobre `seq` y `grep` más adelante.

[13] ¿Recuerdas el **REPL**?

2.11.2 `stdin`, `stdout` y `stderr`

`stdin` (entrada estándar), `stdout` (salida estándar) y `stderr` (error estándar) son *canales* de interacción de la terminal. En tu terminal,

todos apuntan a la pantalla, pero es posible redireccionarlos hacia otros lados.

Los tres canales están asignados a los siguientes *file descriptors*[14]

0 `stdin`, el teclado
1 `stdout`, la pantalla
2 `stderr`, la pantalla

Es posible redireccionar por ejemplo, todos los mensajes de error a un archivo

```
rm este_archivo_no_existe 2> error.txt
```

Si quieres ignorar los errores, puedes mandarlos a un agujero negro[15] estándar, i.e. `/dev/null`:

```
algun_comando 2> /dev/null
```

o apuntar `stdout` y `stderr` al mismo lado (configuración por *default*)

```
algun_comando 2>&1
```

2.11.3 Redireccionando *hacia*

Los símbolos >, » Redireccionan la salida de los comandos a un sumidero (*sink*), e.g. un archivo, o la pantalla o la impresora.
La salida de `ls` se guarda en el archivo `prueba.bat`.

```
ls >> prueba.dat
```

Similar al ejemplo anterior

```
seq 10 > numeros.txt
```

[14] Recuerda, en `GNU/Linux` todo es un archivo, incluído el *hardware*

[15] El sistema operativo está lleno de construcciones similares, por ejemplo `/dev/random`

Pregunta 8

¿Cuál es la diferencia?

TIP: Ejecuta varias veces los ejemplos anteriores

2.11.4 Redireccionando *desde*

< Redirecciona desde el archivo

```
sort < prueba.dat # A la línea de comandos acomoda con sort,
sort < prueba.dat > prueba_sort.dat # Guardar el sort a un archivo.
```

Incluso puedes hacer

```
< prueba.dat wc -l
```

Esto suena muy similar a

```
cat prueba.dat | wc -l
```

`< prueba.dat wc -l` es más eficiente, ya que no está generando un subproceso (lo cual puede ser muy importante en procesamiento intensivo.

2.11.5 Substitución de comandos

En muchas ocasiones quieres la salida estándar (`stdoout`) de un comando para reusarla en algún *script* u otro comando.

```
echo "Hoy es $(date)"
```

Puedes guardar el resultado del comando en una variable:

```
NUMBER_OF_LINES=$(wc -l data/*.txt |  tail -1 | cut -d' ' -f 2)
```

El operador `$( )` se le conoce como *command substitution*

2.11.6 Substitución de procesos

Existe otro operador, `<( )`, llamado *process substitution*. A diferencia del operador `$( )` que sustituye la **salida** del proceso ejecutado dentro del `$( )`, el operador `<( )` sustituye un **archivo que contiene la salida** del proceso que se ejecutó dentro de `<( )`

El ejemplo paradigmático del operador `<( )` ocurre cuando tienes que pasar varios *outputs* a un solo programa, por ejemplo `diff`[16],

[16] `diff` muestra la diferencia entre dos archivos

```
diff <(ls /data) <(ls /vagrant)
```

o si estás usando muchos archivos temporales, por ejemplo, en lugar de hacer

```
curl http://www.nuforc.org/webreports/ndxe201908.html > 201908.txt
curl http://www.nuforc.org/webreports/ndxe201907.html > 201907.txt
cat 201908.txt 201907.txt > 2019.txt
rm 20190?.txt
```

puedes hacer

```
cat <(curl http://www.nuforc.org/webreports/ndxe201908.html) \
    <(curl http://www.nuforc.org/webreports/ndxe201907.html) > 201
```

Una ventaja de este último ejemplo, es que el `shell` ejecutará los procesos en **paralelo** (!!).

2.11.7 Ejecución condicional

`&&` es un AND, sólo ejecuta el comando que sigue a `&&` si el primero es exitoso.

```
ls && echo "Hola"
```

```
lss && echo "Hola"
```

Si sólo quieres ejecutar varios comandos, sin preocuparte por que todos sean exitosos, cambia el `&&` por `;` (punto y coma).

2.12 Algunos comandos útiles

2.12.1 seq

Genera secuencias de números

```
seq 5
```

La sintaxis es: `seq inicio step final`. Por ejemplo

```
seq 1 2 10
```

genera la secuencia de 1 al 10 de dos en dos.
Usando otro separador (`-s`) que no sea es caracter de espacio

```
seq -s '|' 10
```

Agregando *padding*

```
seq -w 1 10
```

2.12.2 tr

Cambia, reemplaza o borra caracteres del `stdin` al `stdout`

```
echo "Hola mi nombre es Adolfo De Unánue"  | tr '[:upper]' '[:lower]'
```

```
echo "Hola mi nombre es Adolfo De Unánue"  | tr -d ' '
```

```
echo "Hola mi nombre es Adolfo De Unánue"  | tr -s ' ' '_'
```

Ejercicio 1

Transforma el archivo de `data/ufo` de tabuladores a `|`, cambia el nombre con terminación `.psv`.

2.12.3 wc

`wc` significa *word count*. Este comando cuenta las palabras,renglones, bytes en el archivo.

En nuestro caso nos interesa la bandera `-l` la cual sirve para contar líneas.

```
seq 30 | grep 3 | wc -l
```

Ejercicio 2

- ¿Cuantos avistamientos existen Noviembre 2014? ¿Y en Diciembre 2014?
- ¿En total?

2.12.4 head, tail

`head` y `tail` sirven para explorar visualmente las primeras diez (*default*) o las últimas diez (*default*) renglones del archivo, respectivamente.

```
head UFO-Dic-2014.tsv

tail -3 UFO-Dic-2014.tsv
```

2.12.5 cat

`cat` concatena archivos y/o imprime al `stdout`

```
echo 'Hola mundo' >> test

echo 'Adios mundo cruel' >> test

cat test

rm test
```

Podrías hacer lo mismo, sin utilizar `echo`

```
cat >> test

# Teclea Hola mundo
# Teclea Adios mundo cruel

# Ctrl-C
```

También podemos concatenar archivos

```
cat UFO-Nov-2014.tsv UFO-Dic-2014.tsv  > UFO-Nov-Dic-2014.tsv
wc -l UFO-Nov-Dic-2014.tsv
```

En el siguiente ejemplo redireccionamos al `stdin` el archivo como entrada del `wc -l` sin generar un nuevo proceso

```
< numeros.txt wc -l
```

2.12.6 split

`split` hace la función contraria de `cat`, divide archivos. Puede hacerlo por tamaño (bytes, `-b`) o por líneas (`-l`).

```
split -l 500 UFO-Nov-Dic-2014.tsv
wc -l UFO-Nov-Dic-2014.tsv
```

2.12.7 cut

Con `cut` podemos dividir el archivo pero por columnas. Las columnas puede estar definidas como campo (`-f`, `-d`), carácter (`-c`) o bytes (`-b`).

Creemos unos datos de prueba

```
echo "Adolfo|1978|Físico" >> prueba.psv
echo "Paty|1984|Abogada" >> prueba.psv
```

Ejecuta los siguientes ejemplos, ¿Cuál es la diferencia?

```
cut -d'|' -f1 prueba.psv
cut -d'|' -f1,3 prueba.psv
cut -d'|' -f1-3 prueba.psv
```

Ejercicio 3

- ¿Qué pasa con los datos de avistamiento? Quisiera

las columnas 2, 4, 6 ó si quiero las columnas Fecha, Posted, Duración y Tipo (en ese orden).

- ¿Notaste el problema? Para solucionarlo requeriremos comandos más poderosos…
- Lee la documentación (`man cut`), ¿Puedes ver la razón del problema?

2.12.8 uniq

- `uniq` Identifica aquellos renglones consecutivos que son iguales.
- `uniq` puede contar (`-c`), eliminar (`-u`), imprimir sólo las duplicadas (`-d`), etc.

2.12.9 sort

- `sort` Ordena el archivo, es muy poderoso, puede ordenar por

columnas (`-k`), usar ordenamiento numérico (`-g`, `-h`, `-n`), mes (`-M`), random (`-r`) etc.

```
sort -t "," -k 2  UFO-Nov-Dic-2014.tsv
```

2.12.10 uniq y sort

- Combinados podemos tener un `group by`:

```
# Group by por estado y fecha
cat UFO-Dic-2014.tsv |\
    cut -d$'\t' -f1,3 |\
    tr '\t' ' ' |\
    cut-d' ' -f1,3 |\
    sort -k 2 -k 1 |\
    uniq -c |\
    sort -n -r -k 1 |\
    head
```

Ejercicio 4

- ¿Cuál es el top 5 de estados por avistamientos?
- ¿Cuál es el top 3 de meses por avistamientos?

2.12.11 Hacer varias cosas a la vez

```
# Ejecuta un servidor web en el puerto 8888
python -m http.server 8888
```

Ve a la dirección mostrada en la terminal en tu navegador
Si presionas `Ctrl+z` *suspenderás* la ejecución de ese programa, pero no será **cancelado**. Observa que el `shell` te devolvió el control

de la terminal. Trata de ver de nuevo la lista de archivos en el navegador … y el navegador se quedará en *Waiting for 0.0.0.0 ...*
Regresa a la terminal y ejecuta

```
jobs
```

`jobs` muestra que programas están ejecutándose y su estatus. En particular, el servidor `http` está detenido.
Para activarlo usa

```
fg
```

`fg` significa *foreground*. Interrumpe la ejecución con `Ctrl+c`.
Si quieres ejecutar el servidor y **no** bloquear la terminal agrega `&` al final.

```
python -m http.server 8888 &
```

De esta manera el servidor está ejecutándose en el *background*.

Ejercicio 5

En la misma terminal ejecuta el editor `nano` y escribe algo en él

- ¿Cómo lo mandas al *background*?
- ¿Cómo lo traes de nuevo en ejecución?
- ¿Cuál es el estatus de los *jobs*?

Atención: Es importante saber que estos procesos o *jobs* estarán ejecutándose mientras tu sesión esté activa. Si te desconectas los procesos serán terminados.
Más adelante veremos como solucionar esto.

2.12.12 Otros comandos útiles

`file -i` Provee información sobre el archivo en cuestion

```
file -i UFO-Dic-2014.tsv
```

`iconv` Convierte entre encodings, charsets etc.

```
iconv -f iso-8859-1 -t utf-8 UFO-Dic-2014.tsv  > UFO-Dic_utf8.tsv
```

`od` Muestra el archivo en octal y otros formatos, en particular la bandera `-bc` lo muestra en octal seguido con su representación ascii. Esto sirve para identificar separadores raros.

```
od -bc UFO-Nov-2014.tsv    | head -4
```

2.13 Descargar datos

Es posible hacer peticiones de HTTP (*http requests*)[17] desde la línea de comandos.

El comando para hacerlo es `curl`.

```
curl http://www.gutenberg.org
```

Si hay redirecciones puedes usar `-L` para seguir la redirección (por ejemplo en los casos donde hay un *url shortener*, como `bit.ly`.

La respuesta (*response*) incluye el *body* (lo que ves en el navegador) y el *header* (metainformación sobre la petición y respuesta). Si sólo quieres ver el *header*

```
curl -I https://duckduckgo.com
```

[17] `HTTP` es el protocolo de comunicación usado por el navegador para solicitar y recibir documentos guardados en otras computadoras o servidores (*páginas web*, les dicen). Más adelante en el curso lo discutiremos en profundidad.

El *header* contiene un pedazo de información crucial: el status code. El *status code* te sirve para ver el estado de la página:

- ¿La página respondió correctamente? `200 OK`
- ¿El recurso solicitado no existe? `404 File Not Found`
- etc

```
curl -I https://duckduckgo.com 2>/dev/null | head -n 1 | cut -d$'
```

donde:

- `2>/dev/null`, redirecciona `stderr` a un agujero negro
- `head -n 1`, lee la primera línea únicamente
- `cut -d$' ' -f2`, separa la línea usando espacios (`' '`) y toma el segundo campo, que contiene el código de estatus `HTTP`.

2.14 Expresiones regulares

> In computing, regular expressions provide a concise and flexible means for identifying strings of text of interest, such as particular characters, words, or patterns of characters. Regular expressions (abbreviated as regex or regexp, with plural forms regexes, regexps, or regexen) are written in a formal language that can be interpreted by a regular expression processor, a program that either serves as a parser generator or examines text and identifies parts that match the provided specification.
>
> Wikipedia: Regular Expressions

2.14.1 Regexp: Básicos

- Hay varios tipos POSIX, Perl, PHP, GNU/Emacs, etc. Nosotros nos enfocaremos en `POSIX`.

Atención: Conocer que tipo de expresiones regulares estás utilizando te quitará muchos dolores de cabeza.

Lee la documentación de tu lenguaje favorito.

- Pensar en patrones (*patterns*).

- Operadores básicos
  - **OR** `gato|gata` hará match con gato o gata.
  - *Agrupamiento o precedencia de operadores* `gat(a|o)` tiene el mismo significado que `gato|gata`.
- Cuantificadores

  ? o ó 1

  + uno o más

  ***** cero o más[18]
- Expresiones básicas

  . Cualquier carácter.

  [ ] Cualquier carácter incluido en los corchetes, e.g. `[xyz]`, `[a-zA-Z0-9-]`.

  [^] Cualquier caracter individual que `n` esté en los corchetes, e.g. `[^abc]`. También puede indicar inicio de líinea (fuera de los corchetes.).

\(\) ó **( )** crea una subexpresión que luego puede ser invocada con `\n` donde `n` es el número de la subexpresión.

{m,n} Repite lo anterior un número de al menos `m` veces pero no mayor a `n` veces.

\b representa el límite de palabra.

[18] Como discutimos anteriormente el *glob* `*` es diferente al *operador* `*`

2.14.2 Regexp: Ejemplos

- username: `[a-z0-9 -]{3,16}`
- contraseña: `[a-z0-9 -]{6,18}`
- IP address:

```
(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?\.){3}(25[0-5]|2[0-4][0-9]|[0
```

Ejercicio 6

- fecha (dd/mm/yyyy): ???
- email (adolfo@itam.edu) : ???
- URL (`http://gmail.com`): ???

2.14.3 Regexp: Expresiones de caracteres

- `[:digit:]` Dígitos del 0 al 9.
- `[:alnum:]` Cualquier caracter alfanumérico 0 al 9 OR A a la Z OR a a la z.
- `[:alpha:]` Caracter alfabético A a la Z OR a a la z.
- `[:blank:]` Espacio o TAB únicamente.

2.14.4 Regexp: ¿Quieres saber más?

- Learn Regular Expressions in 20 minutes
- The 30 minute regex tutorial

2.15 Analizando datos: Comandos avanzados

2.15.1 grep

`grep` nos permite buscar líneas que tengan un patrón específico. Es el equivalente a un filtro.
La sintáxis es sencilla:

```
grep [banderas] patron [archivo]
```

Por ejemplo, todos los avistamientos en California en Noviembre y Diciembre de 2014:

```
grep "CA" UFO-Nov-Dic-2014.tsv
```

En una vuelta irónica de esta historia, ellos clasifican posibles avistamientos como "fraudes" (*hoax*)

```
grep "HOAX" UFO-Nov-Dic-2014.tsv
```

Dos banderas importantes de `grep` son `-v` (negación/complemento):

```
grep "HOAX" UFO-Nov-Dic-2014.tsv
grep -v "18:" UFO-Nov-Dic-2014.tsv
```

y `-E` (interpretar el patrón como una regex).

```
grep -E "18:|19:|20:" UFO-Nov-Dic-2014.tsv
grep -E "[B|b]lue|[O|o]range" UFO-Nov-Dic-2014.tsv
```

Otras banderas importantes son las siguientes

Cuadro 2.1: *Banderas útiles de* `grep`

bandera	significado
`-i`	Ignora mayúsculas / minúsculas
`-o`	Regresa **todos** los *match* en lugar de la línea que contiene el *match*
`-c`	Cuenta el resultado

Usando estas banderas tenemos una versión más simple que el filtrado anterior

```
grep -i -E "[blue|orange]" UFO-Dic-2014.tsv
```

¿Por qué son diferentes los siguientes comandos?

```
grep -c -o -E "[B|b]lue|[O|o]range" UFO-Nov-Dic-2014.tsv  # Ejecut
```

y

```
grep -o -E "[B|b]lue|[O|o]range" UFO-Nov-Dic-2014.tsv | sort | uni
```

Más diversión con expresiones regulares

```
grep "\/[0-9]\{1,2\}\/" UFO-Dic-2014.tsv
grep -v "\/[0-9]\{4\}" UFO-Dic-2014.tsv
grep -E "([aeiou]).*\1" names.txt
echo "Hola grupo ¿Cómo están?" | grep -oE '\w+'
```

Ejercicio 7

Usando los archivos de la carpeta `grep`

- Selecciona las líneas que tienen exactamente cinco dígitos.
- Selecciona las que tienen más de 6 dígitos.
- Cuenta cuántos *javier*, *romina* o *andrea* hay.

2.15.2 awk

awk es un lenguaje de programación muy completo, orientado a archivos de texto que vengan en columnas[19]

Un *programa* de awk consiste en una secuencia de enunciados del tipo patrón-acción:

[19] i.e. nuestros viejos amigos los *dataframes*

```
awk  'search pattern { action statement }' [archivo]
```

awk define algunas variables especiales:

`$1,$2, $3, ...` Valores de las columnas
`$0` toda la línea
`FS` separador de entrada
`OFS` separador de salida
`NR` número de la línea actual
`NF` número de campos en la línea[20]

[20] En awk *lingo* una línea es un *record*

Imprime la Ciudad (`City`) del avistamiento

```
awk -F"[\t]" '{ print $2 }' UFO-Nov-2014.tsv
```

Otros ejemplos:

Verificar el número de columnas en todo el archivo

```
awk -F"[\t]" '{ print NF }' UFO-Nov-Dic-2014.tsv \
     | sort -n | uniq
```

Imprimir el número de columnas de cada renglón

```
awk -F"[\t]" '{ print NF ":" $0 }' UFO-Nov-Dic-2014.tsv
```

Es posible usar *tests* para imprimir o hacer otra operación:

```
awk -F"[\t]" '{ $2 = ""; print }' UFO-Nov-Dic-2014.tsv
```

En este ejemplo, sólo se imprimen las líneas que no contienen Ciudad (`City`)

Pregunta 9

¿Cómo verificas que estas líneas tengan 7 columnas, a pesar de no tener ciudad?

Además de *tests* es posible usar bloques `if-then-else`:

```
awk 'BEGIN{ FS = "\t" }; { if(NF =≠ 7){ print >> "UFO_fixme.tsv"}
 else { print >> "UFO_OK.tsv" } }' UFO-Nov-Dic-2014.tsv
```

Este es un truco que uso **todo** el tiempo para mover las líneas con columnas de más o de menos a otro archivo.

`awk` tiene otros dos modificadores: `BEGIN` y `END` que indican *cuando* debe de ejecutarse las instrucciones, si antes de leer el archivo (`BEGIN`) o al terminar de leer el archivo (`END`)

Como un ejemplo sencillo, lo siguiente es equivalente a `wc -l`

```
awk 'END { print NR }'  UFO-Nov-Dic-2014.tsv
```

La manera de interpretarlo es: *luego de leer todo el archivo* (END) *imprime el número de línea*.

Podemos hacer operaciones con los datos en las columnas:

```
awk 'BEGIN{ FS = "," }; {sum += $1} END {print sum}' data.txt
awk -F, '{sum1+=$1; sum2+=$2;mul+=$2*$3} END {print sum1/NR,sum2/NR,mul/NR}' numbers.dat
awk -F, '$1 > max { max=$1; maxline=$0 }; END { print max, maxline }' numbers.dat
```

También podemos usar *regexs* como condicionales, para substituir o para

```
awk '/CA/ { n++ }; END { print n+0 }' UFO-Nov-Dic-2014.tsv
awk '{ sub(/FL/,"Florida"); print }' UFO-Nov-Dic-2014.tsv
awk '{ gsub(/foo/,"bar"); print }' UFO-Nov-Dic-2014.tsv
awk '/baz/ { gsub(/foo/, "bar") }; { print }' UFO-Nov-Dic-2014.tsv
awk '!/baz/ { gsub(/foo/, "bar") }; { print }' UFO-Nov-Dic-2014.tsv
awk 'a != $0; { a = $0 }' # Como uniq
awk '!a[$0]++' # Remueve duplicados que no sean consecutivos
awk -F"[\t]" '$4 ~/Circle/' UFO-Nov-Dic-2014.tsv
awk -F"[\t]" '
    BEGIN { conteo=0;}
        $4 ~/Circle/ { conteo++; }
    END {
        print "Número de avistamientos circulares en el dataset =",conteo;
    }' UFO-Nov-Dic-2014.tsv
```

Atención:

Existen (al menos) 3 implementaciones de awk:

- **POSIX** awk. El estándar
- gawk, **GNU awk** lleno de funcionalidad, más potente y soporta archivos gigantes.
- mawk, **Minimal awk** tiene el mínimo de funcionalidad pero es más rápido.

Si estás teniendo problemas para ejecutar algo (como quedarte sin memoria o que haya funciones que no existen) probablemente no estás usando gawk. Verifica que tengas esa implementación instalada. Te ahorrará muchos dolores de cabeza.

Para saber más consulta el manual **GNU awk** *Effective AWK Programming*

2.15.3 sed

A veces queremos *editar* o cambiar el contenido de nuestros *datasets*, por ejemplo, quizá queramos deshacernos de ciertos renglones (e.g. *headers* repetidos) o queremos cambiar el valor de una columna (e.g. cambiar el código `1` a `rojo`), `sed` es la herramienta que nos ayudará a hacer esto. `sed` significa **stream editor** . Permite editar archivos de manera automática.

`sed` lee el **flujo de entrada** hasta que encuentra `\n`. Lo copia al **espacio patrón**, y es ahí donde se realizan las operaciones con los datos. `sed` contiene un **búfer** que puede ser utilizado para mantener una memoria, pero es opcional, finalmente copia al **flujo de salida**.

Hay mucho, mucho poder en esta herramienta. La sintaxis es

```
sed [banderas] comando/patrón/[reemplazo]/[modificador] [archivo]
```

Iniciemos con el comando para sustituir: `s`.

```
sed 's/foo/bar/' data3.txt
```

Si queremos guardar la salida a un archivo, no olvides que hay redirecciones

```
sed 's/foo/bar/' < data3.txt > data4.txt
# O también < data3.txt sed 's/foo/bar/'  > data4.txt
```

Nota lo que sucede en el siguiente ejemplo:

```
sed 's/uno/UNO/' < texto.txt
```

este es el funcionamiento por omisión de `sed`.

Estamos usando / como separador ya que es el que usa `vim` o `man` (¿Recuerdas el primer ejercicio?), pero en realidad puedes usar cualquier otro caracter:
Guión bajo (*underscore*)

```
sed 's_uno_UNO_' < texto.txt
```

o también dos puntos (`:`)

```
sed 's:uno:UNO:' < texto.txt
```

son opciones relativamente populares.

Para hacer la sustitución global (i.e. todas las ocurrencias del patrón en la línea), usamos el modificador g

```
sed 's/uno/UNO/g' < texto.txt
```

También es posible hacerlo en *algunas* partes del archivo, especificando la líneas. Por ejemplo en los siguientes ejemplo

```
sed '3s/foo/bar/'  data3.txt # Sólo la tercera linea
sed '3!s/foo/bar/'  data3.txt # Excluye la tercera línea
sed '2,3s/foo/bar/' data3.txt # Con rango
```

Si observas bien, el número de línea funciona como un *filtro*. Es posible extender la idea y usar *patrones*: Sustituir globalmente *foo* por *bar* en las líneas que tengan `123`.

```
sed '/123/s/foo/bar/g'  data3.txt
```

Podemos mezclar ambas ideas y seleccionar partes del archivo usando rangos y patrones

```
sed '/abc/,/456/s/foo/BAR/g'  data3.txt
```

Otro modificadores importantes son `d` (*delete*) y `p` (*print*)[21]

[21] La bandera `-n` elimina la impresión a pantalla.

```
sed -n '2,3p' data3.txt  # Imprime sólo las líneas de la 2 a la 3
sed -n '$p' # Imprime la última línea
sed '/abc/,/-foo-/d' data3.txt # Elimina todas las líneas entre "a
sed 1d data2.txt # Elimina la primera línea del archivo
```

En todos los ejemplos anteriores, `sed` leía la fuente y emitía el resultado modificado a `stdout`. De esta manera, la fuente original no es modificada. La manera de hacerlo *in place* con la bandera `-i`.

```
sed -i 1d data2.txt  # Elimina la primera línea del archivo de mar
```

Ejercicio 8

Elimina los `headers` repetidos con `sed` en los archivos `UFO`.

2.16 *Bash programming*

La mayor parte del tiempo usaremos el `shell`, para hacer pequeños *scripts*, pero existen ocasiones en las cuales es necesario tratar al `shell` como un lenguaje de programación[22]

[22] En este punto es bueno preguntarse si no deberías hacerlo mejor en otro lenguaje de programación, como `python`.

2.16.1 Estructuras de datos

Las variables son declaradas

```
nombre="Adolfo"
```

Nota que no hay espacios alrededor del signo de igual. El valor de la variable se obtiene con el signo de dólares

```
echo $nombre
```

Es posible definir variables que no sean escalares, llamadas arreglos (`arrays`).

```
array=(abc 123 def "programming for data science")
```

Para acceder a elementos en el arreglo usa la posición e.g. 3:

```
echo ${array[3]}
```

Tambien es posible usar *globs*

```
echo ${array[*]}
```

Para conocer el número de elementos del arreglo

```
echo ${#array}
```

Atención: Los arreglos es un punto donde los diferentes *shells* ejecutan diferente. `bash` tiene índices basados en `0`, `zsh` en `1`.

2.16.2 Bucles de ejecución, (*Loops*)

Los `for-loops` en `bash` tienen una estructura muy similar a los de `python`:

```
for var in iterable; do
instrucción
instrucción
...
done
```

Donde `iterable` puede construirse con *globs*

```
for i in *; do
  echo $i;
done
```

listas,

```
for i in hola adios mundo cruel; do
  echo $i;
done
```

arreglos,

```
for i in $array; do
  echo $i;
done
```

E inclusive el horrible formato de C:

```
for (( i = 0; i < 10; i++ )) do
    echo $i;
done
```

También puedes escribirlo en una sola línea: `for i in {a..z}; do echo $i; done`

2.16.3 *Tricks*

También es posible hacer lo siguiente:

```
for i in {a..z}; do
  echo $i;
done
```

En el ejemplo anterior utilizamos *brace expansion*:

```
echo {0..9}
echo {0..10..2}
echo data.{txt,csv,json,parquet}
```

2.16.4 Condicionales

Existen dos operadores para hacer pruebas en bash: `[` y `[[`.

De preferencia usa `[[`

```
[[ 1 = 1 ]]
echo $?    # Imprime el resultado del comando previo (true → 0)
```

Existen algunos operadores para hacer comparaciones los más comunes son `-a`, `-d`, `-z` (unarios) y `-lt`, `-gt`, `-eq`, `-neq` (binarios).

Teniendo los *tests* es posible crear estructuras `if-then-else`

```
if TEST-COMMANDS; then CONSEQUENT-COMMANDS; fi
```

2.16.5 Funciones

Las funciones son símbolo de buena programación, ya que encapsulan comportamiento.

La sintaxis es muy simple:

```
function function_name {
    # cuerpo de la función
}
```

A diferencia de otros lenguajes de programación, no se definen los argumentos de la función.

Para ejecutar la función

```
function_name "Hola mundo" 123
```

En este caso `"Hola mundo"` y `123` son pasados como argumentos a la función. Estos pueden ser usados en el cuerpo de la función usando la posición de los mismos e.g. `$1` es `"Hola mundo"`, `$2` es `123`, etc.

2.16.6 Ejecutar *scripts*

Para cualquier archivo *script* es importante que la primera línea del archivo le diga al `shell` que comando usar para ejecutarlo.
A la primera línea se conoce como **shebang** y se representa por `#!` seguido de la ruta al ejecutable, e.g.:

- `#!/usr/bin/python`,
- `#!/bin/bash`,
- `#!/usr/bin/env Rscript`,
- etc.

Sin el `shebang`, para ejecutar el archivo `ejemplo.py` debes de hacer:

```
python ejemplo.py
```

pero, si agregamos el `shebang` [23], puedes ejecutar el archivo de la siguiente manera:

```
./ejemplo.py
```

[23] Para que el ejemplo funcione es necesario dar permisos de ejecución al archivo `chmod u+x ejemplo.py`. `chmod` proviene de *change modifier*

Te preguntarás ¿Cómo conecto estos *scripts* con los demás usando |, >, etc?
Sencillo: Hay que modificar nuestros *scripts* de `python`, `R` y `bash` para leer del `stdin`.

2.16.7 Python, leyendo de `stdin`

Un ejemplo mínimo de `python` es el siguiente:

```
#!/usr/bin/env python

import sys

def process(linea):
    linea = int(linea.strip())
    print(f"El triple de {linea} es {linea*3}")

for linea in sys.stdin:
    process(linea)
```

Abre nano, copia este código y guarda el archivo como `script.py`. Un ejemplo de uso es el que sigue:

```
seq 1 1000 | script.py
```

2.16.8 R, leyendo de `stdin`

El ejemplo en R se ve así:

```
#!/usr/bin/env Rscript

f ← file("stdin")

x ← c()

open(f)

while(length(line ← readLines(f, n = 1)) > 0) {

  x ← c(x, as,numeric(line))
  print(summary(x))
}


close(f)

print("Final summary:")
summary(x)
```

Ejemplo de uso:

```
< /data/numbers.txt script.R
```

2.17 *Terminal multiplexers*

Al conectarte a la máquina virtual usas el protocolo SSH (*secure shell*). Este protocolo es el que utilizaremos para conectarnos a servidores en la nube, pero por ahora nos conformaremos con usar la máquina virtual como nuestra "nube".

2.17.1 Trabajando remotamente

Al trabajar remotamente vía SSH, una interrupción de la conexión (ya sea intencional o no) detendrá la ejecución de los *scripts* que estes ejecutando (incluido el ambiente). Para evitar esto hay varias opciones: (1) Utilizar un servidor `mosh`, el cual "envuelve" a SSH, pero mantiene las conexiones abiertas por ti o (2) Utilizar un *terminal multiplexer*[24]. Esta segunda opción es la que utilizaremos en el curso.

[24] Ni siquiera intenté traducirlo

Un *multiplexor*[25] permite ejecutar sesiones de la terminal en el servidor remoto usando tu computadora vía SSH. De esta manera si te desconectas, la sesión remota sigue ejecutándose. Los más populares son `screen` y `tmux`.

[25] De verdad, no sé si esto sea una palabra

2.17.2 tmux

La mejor manera de explicar que es `tmux` es mostrarlo, a continuación están los comandos que voy a utilizar en el demo.

```
# ssh
vagrant ssh
# Crea una sesión de tmux
tmux
# Lista las sesiones existentes
tmux ls
# Attach (a) to a target session (-t #)
tmux a -t 1
# Renombrar la ventana
Ctrl+b+,
# Crear un nuevo panel
Ctrl+b+c
# Divide la ventana en dos paneles horizontales
Ctrl+b+"
# Divide la ventana en dos paneles verticales
Ctrl+b+%
# Zoom al panel
Ctrl+z
```

`Tmux` es muy configurable, por ejemplo consulta The Tao of tmux de Tony Narlock.

`tmuxp` es una librería de `python` que permite adminsitrar sesiones de `tmux`. Esto será muy útil más adelante, pero es totalmente opcional.

3 Controlador de versiones git

3.1 Introducción

Un controlador de versiones es una herramienta que gestiona los cambios de un conjunto de archivos. Cada conjunto de cambios genera una nueva versión de los archivos. El controlador de versiones permite, además de la gestión de cambios, recuperar una versión vieja de los archivos o un archivo, así como resolver conflictos entre versiones.

Aunque su principal uso es para agilizar la colaboración en el desarrollo de proyectos de **software**, también puede utilizarse para otros fines (como estas notas) e inclusive para trabajar solo (como en una tesis o proyecto).

Especificamente, un controlador de versiones ofrece lo siguiente:

1. Nada de lo que es "**comiteado**" (*commited*, ahorita vemos que es eso) se perderá.
2. Lleva un registro de **quién** hizo **qué** cambios y **cuándo** los hizo.
3. Es **casi** imposible[27] sobreescribir los cambios de tu colaborador. El controlador de versiones notificará que hay un **conflicto** y pedirá que lo resuelvas antes de continuar.

[27] Nota el casi …

En esta clase usaremos `git`, aunque debemos de notar que no es el único controlador de versiones que existe, entre los más populares se encuentran `bazaar`, `mercurial` y `subversion` (Aunque este último pertenece a una diferente clase de controladores de versiones).

3.2 Configurando git

Ahora personalizaremos `git`. Esto es importante, ya que el acceso a los repositorios está ligado a tu usuario. Además, si tienes configurado `git`, los *commits* quedarán registrados a tu nombre.

El siguiente comando te permite ver la configuración *actual*:

```
git config --list
```

Para configurar `git` ejecuta lo siguiente:

```
git config --global user.name "Tu nombre"
git config --global user.email "username@some.email.server.com"
git config --global color.ui  "auto"
git config --global core.editor "nano"
```

Por último, configura `git` para que haga *push* al *branch* remoto con el mismo nombre que el *branch* local[28]

[28] Todo esto tendrá mas sentido más adelante.

```
git config --global push.default current
```

3.3 *"Solo" Workflow*

3.3.1 Crear un repositorio

El **repositorio** es la carpeta donde `git` guarda y gestiona todas las versiones de los archivos.

Crea una carpeta en tu `$HOME` llamada `ds-test`, ingresa a ella e inicializa el repositorio con `git init`. ¿Notas algún cambio? ¿Qué comando usarías? Hay una carpeta ahí ¿no la ves? ¿Cómo puedes ver una carpeta oculta?

La carpeta `.git` es la carpeta donde se guarda todo el historial, si la borras, toda la historia del repositorio se perderá.

Podemos verificar que todo esté bien, con el comando `status`.

```
git status
```

3.3.2 Llevando registro de los cambios a archivo

Crea un archivo llamado `hola.txt` en la carpeta `ds-test`

```
touch hola.txt
echo "¡hola mundo!" > hola.txt
```

Ahora ejecuta el siguiente comando

```
git status
```

El mensaje de `untracked files` significa que hay archivos en el repositorio de los cuales `git` no está llevando registro, i.e. el repositorio está *sucio*.
`Git` sigue este flujo:

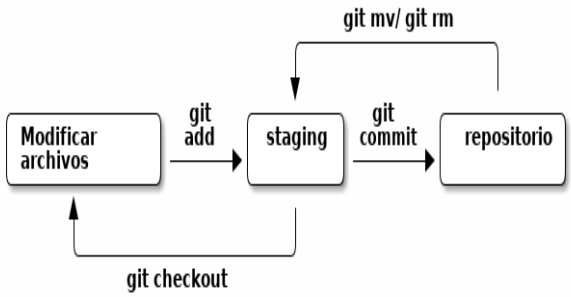

Figura 3.1
Git "solo workflow"

Para que `git` lleve el registro (*tracking*) del archivo, debes de **agregarlo** (`add`):

```
git add hola.txt
git status
```

Ahora `git` sabe que debe de llevar registro de los cambios de `hola.txt`, pero aún se **comprometen** los cambios al repositorio (`Changes to be commited: ...`). Para *comitearlos*[29]:

[29] Lo siento…

```
git commit -m "Commit inicial"
```

Usamos la bandera `-m` para agregar un mensaje que nos ayude a recordar más tarde que se hizo y por qué.
Si ejecutamos

```
git status
```

nos indica que todo está actualizado (`up to date`). Podemos ver la historia de cambio con `git log`.
Edita `hola.txt` luego, ejecuta `git status`. ¿Qué observas ahora?
La parte clave es `no changes added to commit`. Hemos cambiado el archivo, pero aún no están "comprometidas" o guardadas en el repositorio. Para ver que ha cambiado usamos lo siguiente

```
git diff
```

Hagamos `commit` de estos cambios.

```
git commit -m 'actualizamos hola.txt'
```

Pero `git` no nos dejará hacer el *commit*, ya que no lo agregamos antes al índice del repositorio. Agrégalo y repite el `commit`
Modifica de nuevo `hola.txt`. Observa los cambios y agrégalo. ¿Qué sucede si vuelves a ejecutar `git diff`?
`Git` dice que no hay nada, ya que para `git` no hay diferencia entre el área de **staging** y el último **commit** (llamado `HEAD`). Para ver los cambios, ejecuta

```
git diff --staged
```

esto muestra las diferencias entre los últimos cambios **comiteados** y lo que está en el área de **staging**. Ahora realiza el `commit`, verifica el estatus y revisa la historia.

3.3.3 Explorando el pasado

Podemos ver los cambios entre diferentes **revisiones**, podemos usar la siguiente notación: `HEAD~1`, `HEAD~2`, etc. como sigue:

```
git diff HEAD~1 hola.txt
git diff HEAD~2 hola.txt
```

También podemos utilizar el identificador único (el número enorme que aparece en el `git log`), inténtalo.
Modifiquemos de nuevo el archivo `hola.txt`. ¿Qué tal si nos equivocamos y queremos regresar los cambios? Podemos ejecutar el comando

```
git checkout HEAD hola.txt
```

Nota que `git` recomienda un **shortcut** para esta operación: `(use "git checkout -- <file>..." to discard changes in working directory))`

Obviamente aquí podemos regresarnos las versiones que queramos, por lo que podemos utilizar el identificador único o `HEAD~2` por ejemplo.

Ejercicio 9

Recorre el log con `checkout`, observa como cambia el archivo, usando `cat`.

Por último, `git` tiene comandos `mv` y `rm` que deben de ser usados cuando queremos mover o borrar un archivo del repositorio, i.e. `git mv` y `git rm`.

Ejercicio 10

Crea un archivo `adios.txt`, comitealo, has cambios, comitea y luego bórralo. No olvides hacer el último `commit` también.

3.4 Git en la web

[30] Otra opción, popular luego de qu Github fue adquirido por Microsoft es Gitlab.

Github[30] aparenta ser un **repositorio central**, con una interfaz web bonita, pero recuerda que `git` es un sistema distribuito de control de versiones y **ningún nodo** tiene preferencia sobre los demás, pero por comodidad, podemos usar **Github*/*Gitlab** para colaborar en proyectos.

El repositorio de la clase está en:

`https://github.com/ITAM-DS/programming-for-data-science-2`

Para obtener una copia de trabajo en su computadora deberán de **clonar** su repositorio:

```
mkdir /repositorios
git clone https://github.com/ITAM-DS/programming-for-data-scien
    /repositorios/programming-for-data-science-2019
```

Esto creará una carpeta `programming-for-data-science` en `$HOME`.

¡Ya no tendrás que descargar el archivo `zip` cada vez que empiece la clase!

3.5 *Github flow*

Ahora trabajaremos en equipo, el flujo cambia respecto al *solo flow*. El cambio se debe a la introducción de nuevos conceptos: **clonar**, **push/pull**, **issue** y **branch**.

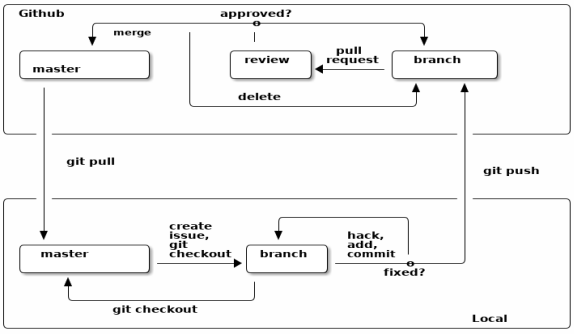

Figura 3.2
Github flow

Para entender este diagrama, necesitamos explicar estos nuevos **verbos**.

En *github flow* es muy importante que:

- Crees *issues* y trabajes en ellos.
- Idealmente un *issue* está relacionado con una sóla cosa (arreglar algo, agregar algo)
- Idealmente un *issue* debe de durar menos de 8 horas.
- Hagas muchos *commits* al día

Hacer *commits* frecuentemente permite que podamos hacer control de versiones de manera granular. Si un *bug* aparece, podemos regresar a versiones pasadas dónde el *bug* no existía. Otra ventaja es que *commits* pequeños y frecuentes ayudan a tus compañeros de trabajo (y tu futuro tú) entiendan mejor cuál es tu intención.

- ¡Usa mensajes que tengan sentido!

En serio, escribe mensajes que tengan sentido

Figura 3.3
Imagen de XKCD

3.6 *Branches*

Los *Branches* son usados para aislar el trabajo, así podrás trabajar en resolver algún *issue* y no afectar el trabajo de los demás. En el *github flow* sólo una persona está trabajando en un *branch*.
Para ver las *branches* locales

```
git branch
```

Tu *branch* actual está marcado con un asterisco (*)
Si quieres trabajar en algún *branch* existente:

```
git checkout branch_name
```

Si ejecutas de nuevo `git branch` notarás que el asterisco ahora está marcando `branch_name`
Para crear un nuevo *branch*:

```
git checkout -b branch_name
```

3.7 *Push, Pull* y *Pull request*

Es importante que tu *branch* `master` este al día. Para traer los cambios más recientes:

```
git pull origin master
```

Cuando hayas resuelto el *issue* en el que estas trabajando,

```
git push
```

Por último, para solicitar que se integren tus cambios a la rama `master`, debes de hacer un *pull request* (**PR**). Esto se hace en el sitio web de Github.

Algunas cosas importantes que debes de hacer:

- Asígnate el **PR**
- Asigna a un miembro de tu equipo (al menos) como revisor.
- Describe los cambios que hiciste.
- Responder/Resolver los comentarios que haga el *revisor*.
- Hacer *merge* del **PR** una vez que haya sido aprobado.

El *revisor* por su parte debe de hacer:

- Revisar los cambios
- Hacer comentarios en línea o en el *thread* del **PR**.
- Verificar y discutir los cambios con la persona que hizo el **PR**.
- Aprobar el **PR** una vez que los cambios sean atendidos.

3.8 *Merges*

Al estar trabajando en el *github flow*, lo que buscamos es poder colaborar y combinar el trabajo de varias personas. `Git` es una herramienta fantástica para esto.

La operación de integrar los cambios realizados a un archivo en una rama, se llama *merge*. `Git` realizará (o intentará) hacerlo de manera automática.

En el *github flow* el *merge* ocurre al analizar el *pull request*.

Existen ocasiones[31] donde el *merge* automático no funcionará bien.

[31] Por ejemplo, si están editando varias personas el mismo archivo en el mismo renglón.

Esto se le llama *conflicto*, cuando esto ocurre `git` mostrará un mensaje de error como el que sigue:

```
CONFLICT (content): Merge conflict in file_name
Automatic merge failed; fix conflicts and then commit the result.
```

[32] O simplemente use `git diff`

Al abrir[32] el archivo con conflicto verás algo como lo que sigue:

```
++<<<<<<<<<< HEAD
...
...
++==========
...
...
++>>>>>>>>>> Branch my_branch
```

La manera de resolver los conflictos de manera *manual* es como sigue:

1. Inspecciona los cambios. La sección entre `++<<<<<<< HEAD` y el separador `++==========` es el código de la rama actual. Lo que está entre el separador y `++>>>>>>>` son los cambios de la rama que quieres integrar.
2. Borra la sección que no quieras quedarte.
3. Cierra el archivo y has *commit*.

Una vez que hayas resuelto los conflictos, has *commit* de tus cambios para completar el *merge*.

3.9 Algunos comandos útiles

Para más comandos, ve este `gist`.

Cuadro 3.1: *Algunos comandos que es bueno tener a la mano*

Comando `git`	Resultado
`git checkout my_file`	Descarta los cambios hecho al archivo `my_file`
`git reset --hard`	Descarta todos los cambios hechos (incluidos aquellos que no han sido *commiteados*)
`git stash`	Descarta los cambios pero los guarda para su uso posterior.
`git clean`	Remueve los archivos que no están *trackeados* por `git`
`git diff`	Muestra todos los cambios hecho desde el último *commit*
`git diff my_file`	Muestra los cambios en el archivo `my_file` desde el último *commit*
`git checkout --ours my_file`	Acepta los cambios de la rama actual
`git checkout --theirs my_file`	Acepta los cambios de la rama que queremos integrar

4 Desarrollo de software

> The way to learn to program
> is by programming
>
> Nathan Myhrvold

4.1 El zen de python

La regla de oro de programación nos dice que, programamos para comunicarnos con humanos, no con computadoras. Si alguna vez la olvidas o tienes dudas, siempre puedes consultar una versión más elaborada en el zen de python.

```
import this
```

4.2 Paradigmas de programación

4.2.1 Procedural

La unidad más importante para diseñar es el **verbo**. La lógica guiada por los verbos se coloca dentro de *funciones* , *procedimientos* o *subrutinas*.

Si decides esta opción para programar, lo que tienes que hacer es que tu código siga una serie **secuencial** de pasos.

Suena obvio ¿Cierto? Pero esta decisión tendrá los siguientes efectos:

- La salida de una función no necesariamente tiene una correlación directa con la entrada
- Todo se tiene que hacer en un orden específico
- La ejecución de la rutina tendrá efectos laterales (*side effects*)
- La solución se tiende a implementar de forma lineal.

4.2.2 Orientado a Objetos

Cuando diseñas la solución a un problema usando este paradigma te enfocas en los **sustantivos** en lugar de los verbos. Cada sustantivo se mapeará a un *objeto*.

Los objetos contienen la información sobre su estado (*state*) y su comportamiento (*behavior*).

El estado de un objeto queda descrito por las características del mismo, es decir, qué palabras usarías para describirlo. Para identificar el estado buscarás relaciones tiene (*has*) o es (*is a*) en la descripción del problema. El estado de un objeto se programará en variables llamadas *atributos*.

Comportamientos es aquello que el objeto puede hacer, la lógica de este comportamiento se codifica en *métodos*. Regularmente los nombres de los métodos se ponen en infinitivo: `correr`, `beber`, etc.

4.2.3 Funcional

Idealmente, un lenguaje *funcional* permite escribir funciones matemáticas, es decir funciones que tienen n argumentos y regresan un valor. Las funciones matemáticas, siempre regresan el mismo valor si la función es aplicada en los mismos argumentos.

Los impactos más importantes de este paradigma son:

- Siempre devuelven el mismo valor para una entrada dada
- El orden de evaluación **no** está definido
- No hay un estado (*stateless*)
- Son fácilmente paralelizables

4.2.4 Lógico / Declarativo

Iniciaron su popularidad en lo que se conoció como Inteligencia Artificial de Quinta generación, aunque fueron desarrollados desde mucho antes. Este tipo lenguajes se caracteriza por que le indicas a la computadora lo que *quieres* obtener y no como *tiene que hacerlo*.

Ejemplos famosos son `Prolog` y `SQL`.

4.2.5 *Spaghetti* o *Lasagna*

4.3 Diseñar una solución: *Semantic design*

Al diseñar cualquier pieza de software, incluyendo un producto de datos ten siempre en mente los siguientes tres principios:

- Los *tipos* y *objetos* de tu programa *deben de significar algo*
- El dominio semántico debe de ser modular (*composable*)
- El dominio semántico debe de ser tan simple como sea posible.

5 Bases de datos

> I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships.
>
> Linus Torvalds

5.1 ¿Por qué usarlas?

Muchos científicos de datos (o analistas o estadísticos o economistas) cuando inician sus carreras, están acostumbrados a trabajar con conjuntos de datos pequeños[35], estáticas o "muertas"[36]. Son las que les dan en la escuela, las que ven en blogs, las que vienen como ejemplos en los libros o son las "bases de datos" abiertas[37] por el gobierno.

El trabajo a realizar se puede realizar en estos casos usando archivos de texto, y utilizando `R` o `python` u otro lenguaje de *scripting*[38] basta. Algunos menos afortunados, enfrentan el mismo flujo de trabajo usando lenguajes estadísticos como `SPSS`, `SAS`, `STATA` o `Minitab`.

[35] Entendemos por "conjuntos de datos pequeños", como aquellos que puedes poner en la memoria de tu laptop y aún tienes espacio para ejecutar algunos análisis.

[36] Sólo por seguir el tema de día de muertos

[37] Más adelante veremos como esto es una muy mala elección de palabras

[38] Por ejemplo, `bash`

Este *workflow* aprendido, sirve cuando queremos hacer análisis simples[39], cuando no tienes planeado que vayas a repetir el análisis (ni tú, ni nadie más)[40], cuando no vas a recibir datos actualizados, etc.

En la sección pasada vimos que para paliar el dolor futuro, la utilización de un lenguaje de verdad (como `R` o `pyhton`, por ejemplo) y acomodar el código en funciones, pensando en comunicarse con humanos ayuda. ¿Pero qué hacer con los datos? ¿O qué hacer en los casos siguientes?

- Tienes fuentes datos muy grandes [41]
- Provienen de diversas fuentes
- Pueden o son actualizados
- Quieres compartir los datos con otros para que reproduzcan tu análisis

En estos casos lo mejor es usar un "Sistema de administración de bases de datos" (*Database Management System*) o **DBMS**. Los **DBMS** están optimizados para ordenar, organizar, administrar y analizar datos. Mitigan el problema de escalamiento y de complejidad cuando tus datos aumentan de volumen y en variedad. Además, los **DBMS** facilitan la creación de un *data model* que permitirá que los datos sean almacenados, consultados (*queried*) y actualizados de una manera eficiente y **concurrente** por múltiples usuarios (!).

Recuerda: Las bases de datos no son únicamente para **almacenamiento**, ellas están para manipular de una manera eficiente tus datos: /trata de hacer la mayor cantidad de manipulación de datos cerca de donde los datos están localizados/

[39] Aquellos análisis que mapean muy bien a las capacidades o diseño de tu lenguaje de programación estadístico

[40] Esto es siempre falso

[41] Que no caben en tu memoria …

5.2 RDBMS: Bases de datos relacionales

En la década de 1970, Edward F. Codd, desarrolló, usando las matemáticas del *álgebra relacional*[42], el modelo de bases de datos conocidas como Gestor de bases de datos relacionales (RDBMS).

Por simplicidad, a las RDBMS las llamaremos *bases de datos*.

Las bases de datos relacionales contienen *relaciones* (tablas), las cuales tienen un conjunto de *tuplas* (renglones), las cuales mapean *atributos* a valores atómicos, los cuales quedan definidos por una *tupla header* mapeado a un *dominio* (columnas).

Originalmente booleanas, la implementación actual es lógica trivaluada (`TRUE`, `FALSE`, `NULL`).

El lenguaje usado para manipular datos se conoce como `SQL`: *Structured Query Language*.

Ejemplos de bases de datos relacionales son:

- sqlite
- MySQL
- PostgreSQL
- Microsoft SQL Server
- Oracle
- IBM DB2
- Teradata
- …

En esta clase usaremos `sqlite` y `PostgreSQL`.

[42] Por eso las RDBMS son relacionales y no por lo que se cree popularmente de que son relacionales por que las tablas están "relacionadas" por identificadores y restricciones.

5.3 *Data model*

Un modelo de datos (*data model*) es una colección de conceptos que describen los datos.

La descripción de una colección particular de datos usando el *modelo de datos* se conoce como *esquema*, la cual, en las RDBMS incluye nombre de los atributos, tipo, restricciones, reglas de negocio, etc.

5.4 Las RDBMS son ACID

Las bases de datos relacionales, satisfacen las propiedades conocidas como **ACID**:

atomicidad Todo el trabajo de una transacción o se completa en su totalidad (`commit`) o *nada* del trabajo se completa (`rollback`).

consistencia Cada transacción transforma la base de datos de un estado consistente a otro estado consistente.

aislado (*isolated*) Los resultados de los cambios realizados en una transacción no son visibles hasta que la transacción es *committed*.

durable Los cambios resultantes de una transacción sobreviven a fallas.

5.4.1 Agregar imagen de cliente, motor, datos

5.5 Ejemplo: `sqlite`

`SQLite` es una base de datos relacional y local. `SQLite` está instalada en cada uno de los dispositivos Android, iPhone, Mac OS o Windows 10. También está instalada en todos los navegadores Firefox, Chrome, Safari. Cada `python` lo instala también. Como pueden ver está en todos lados.

No requiere manejo de usuarios, así que puedes inmediatamente a usarla. En tu `vagrant` teclea:

```
sqlite3 turista.db
```

Este comando crea una *base de datos* que se vivirá en el archivo `turista.db`

Puedes ver las tablas dentro de la base de datos `turista.db`

```
.tables
```

Para salir teclea `Ctrl+d`. Si quieres volver a entrar a la base de datos, teclea de nuevo

```
sqlite3 turista.db
```

Si queremos usar de manera programática `sqlite`, tenemos dos opciones: usar el cliente de base de datos desde la terminal o usando un lenguaje de programación, como `python` o `R`. En todos los casos utilizaremos `SQL`

5.5.1 Desde un archivo SQL

El siguiente archivo `SQL` crea una tabla llamada `games`, realiza un query y luego destruye la tabla

```
create temporary table if not exists games (
  game integer
);

insert into games(game) values(1);

select game from games;
```

```
.read games_test.sql
```

Al ser una tabla temporal, no permanece luego de que acaba la transacción.

```
.tables
```

5.5.2 Python

```
import sqlite3
import logging

with sqlite3.connect('turista.db') as conn: # Utilizando un context manager
```

```
    conn.execute('create table if not exists games (game integer)
    conn.execute('insert into games(game) values (?)', ('1',))
```

Veamos si funcionó. En la terminal, ingresa a la base de datos `turista.db` y teclea:

```
select game from games;
```

Podemos replicar este comportamiento desde `pyhton` también:

```
query = "select game from games"

with  sqlite3.connect('turista.db') as conn:
    cursor=conn.execute(query) # Obtenemos un cursor para recorrer
    games = cursor.fetchall()

for game in games:
    print(f"Game: {game[0]}") # Es una tupla, tomamos la primera c
```

5.5.3 R

La manera más fácil y moderna de conectarse a `sqlite` desde `R` es usando la librería `tidyverse=m en particular =dbplyr`

```
install.packages('tidyverse')
install.packages('dbplyr')
install.packages('RSQLite')
```

Observa como en lugar de usar un lenguaje declarativo como `SQL`, usamos un lenguaje tipo *data flow*, que es de un corte más imperativo.

```
library(tidyverse)

con ← DBI::dbConnect(RSQLite::SQLite(), "turista.db")

games  ←tbl(con, 'games')

games %>% select(game)
```

Aunque no veamos `SQL` en este ejemplo, nota que `dbplyr` está generando códigi `SQL` por nosotros.

```
games  %>%
  select(game) %>%
  show_query()
```

Agregar diagrama

5.6 Ejemplo: PostgreSQL y psql

PostgreSQL (o simplemente postgres) es una de las bases de datos relacionales más poderosas disponibles.
Entre las ventajas de Postgresql están:

- Open-source
- Varios tipos de índices
- 5 tipos diferentes de joins
- *subqueries* en cualquier cláusula
- *Window functions*
- Soporte geoespacial
- Soporte para minería de texto
- *queries* recursivos
- Tipos de datos como XML, json, arreglos, rangos, etc …
- Extensible: lenguajes externos, puedes crear tipos de datos, funciones, agregadores, operadores, etc.
- *foreign data wrappers*
- Creación concurrente de índices
- *queries* en paralelo
- Listen/notify

psql es su cliente de la base de datos.
Primero debemos de crear una *base de datos*[43] y para hacerlo debes de ser el administrador del servidor **RDBMS**. Por *default* el usuario es postgres.

```
sudo su postgres
```

El *prompt* de vagrant debió de cambiar a algo como

[43] Este será uno de los mucho ejemplos en los cuales un término en Ciencia de datos está sobrecargado (overloaded). En particular, aquí me refiero a crear un conjunto de tablas que tengan un data model que será manejado por el RDBMS.

```
postgres@ubuntu1904:/home/vagrant$
```

Teclea `psql` para iniciar el cliente de base de datos.

```
psql
```

El *prompt* debería de cambiar de nuevo:

```
postgres=#
```

Iniciemos creando la base de datos `turista`:

```
create database turista;
```

`psql` está lleno de pequeños atajos que puedes consultar con `\?`, por ejemplo, para ver las bases de datos creadas

```
\l
```

Ejercicio 11

- Algunos comandos útiles: `\l`, `\connect`, `\d`, `\dt`, `\a`, `\x`, `\i`, `\o`, `\g`, `\!`, `\timing on/off`, averigua que hacen cada uno de ellos.
- `\help` adelante de una sentencia `SQL` les muestra la ayuda de la sentencia Intenten `\help select` y ejecútenlo cada vez que veamos un comando de SQL que desconozcan.

Acto seguido, creemos un usuario[44], llamado `turista` y asignémosle (`GRANT`) todos los privilegios en la base de datos `turista`

[44] En postgres no hay usuarios, hay roles

```
create role turista login ; -- Permitimos que el rol se pueda cone
alter role turista with encrypted password 'some_password'; -- Agr
grant all privileges on database turista to turista; -- Asignamos
```

Puedes ver los roles en el servidor PostgreSQL mediante

```
\du+
```

Teclea Ctrl+d (para salir de psql) y luego Ctrl+d (para salir de la sesión del usuario postgres). Con la base de datos creada es posible conectarte desde la sesión del usuario vagrant

```
psql -U turista -d turista -h 0.0.0.0 -W
```

La sintaxis es la siguiente:

```
psql -h host -U user -d base_de_datos -W
```

Para evitar que pregunte la contraseña, creen un archivo .pgpass en el $HOME con la siguiente sintaxis:

```
host:port:*:username:password
```

El archivo debe de ser visible sólo para el usuario vagrant por lo que hay que guardarlo con permisos 0600:

```
chmod 0600 .pgpass
```

Otra adición que hará tu vida fácil, es tener un archivo .pg_service_conf

```
[turista]
host=0.0.0.0
port=5432
user=turista
```

5.6.1 Archivo SQL

De la misma manera que en sqlite podemos usar PostgreSQL de manera programática.

Si quieres ejecutar un archivo `.sql`:

```
psql -f script.sql
```

También es posible ejecutar un comando `SQL` (muy útil en *scripts* de `bash`)

```
psql -c "SELECT * from pg_tables limit 1;"
```

5.6.2 Python

Muy similar al caso anterior:

```
import psycopg2
import psycopg2.extras

DB_URL="postgres://service=turista"

with psycopg2.connect(DB_URL) as conn:
    cursor = conn.cursor()
```

5.6.3 R

```
install.packages(RPostgres)
```

```
library(tidyverse)

con <- DBI::dbConnect(RPostgres::Postgre("postgres://service=turi

games  <-tbl(con, 'games')

games %>% select(game)
```

5.7 SQL

`SQL` es un lenguaje de tipo *declarativo*, al usarlo tienes que declarar en el resultado que quieres obtener respecto a una modelo de datos *conocido* el cual contiene el conjunto de datos.

La declaración se conoce como *query*. El resultado del *query* se conoce como *result set* y es una relación, la cual ya existía en el modelo de datos o fue determinada al ejecutar el *query*. El estándar actual es `SQL 2016`.

`SQL` puede ser dividido en `DDL` y `DML`.

Data definition language (DDL) es usado para cambiar el esquema de la base de datos, i.e. crear y destruir tablas, cambiar (alterar) columnas, etc.

Por su parte *Data manipulation language* (DML) se usa para consultar las tablas y para modificar los renglones de una tabla: insertar, borrar, modificar.

5.7.1 Datos para jugar

Para algunos de los ejercicios de esta sección usaremos los datos conocidos como **berka**. Esta base de datos fue liberada por un banco de Europa Oriental, para el **PKDD'99 Discovery Challenge**.

Puedes descargar una copia de la base aquí

Cada cuenta tiene tanto características estáticas (eg. fecha de creación) , contenidas en la relación `account`, como dinámicas (e.g. pagos debitados o acreditados, balances) dados en las relaciones `permanent_order` y `transactions`.

La relación `client` describe las características de la persona que puede manipular esas cuentas. Un cliente puede tener una o más cuentas, pero también se puede dar el caso que muchos clientes pueden manipular una cuenta; los clientes y cuentas están relacionadas a través de la relación `disposition`.

Las relaciones `loan` y `credit_card` describen algunos de los servicios que el banco ofrece a sus clientes:

- Varias tarjetas de crédito pueden ser asignadas a una cuenta.
- Hay un máximo de un préstamo por cuenta.

La relación `demographic_data` contiene información pública sobre los distritos (e.g. tasa de desempleo).
Pueden consultar más información aquí

5.7.2 Diagrama ERD

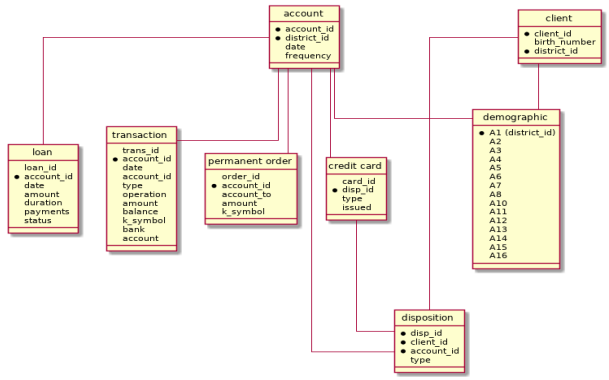

Figura 5.1
Diagrama de entidad-relación(ERD) para la base de datos `berka`. No se muestran todos los atributos de las relaciones.

5.7.3 Importar/Cargar los datos

sqlite

Para jugar con estos datos usaremos una base de datos `sqlite` a la que llamaremos `berka.db`
Activamos el modo `csv` en `sqlite`.

```
.mode csv
```

Esto le indica al cliente de `sqlite` que interprete el archivo de entrada como un archivo csv.
Luego de descomprimir el archivo `data_berka.zip`, podemos usar `head` en uno de los archivos para ver su contenido:

```
head account.asc
```

Los archivos tienen como separador `";"`. Debemos indicarle esa información a `sqlite`

```
.separator ";"
```

Acto seguido importemos el archivo `client.asc`. Nota que lo estoy guardando a una tabla llamada `clients`, si en plural.

```
.import client.asc clients
```

Podemos ver que la tabla está ahí con el comando `.tables`

```
.tables
```

Y su esquema con `.schema`

```
.schema clients
```

Ejercicio 12

Importa las tablas restantes

PostgreSQL

Para jugar con estos datos usaremos una base de datos `PostgreSQL` a la que llamaremos `berka`.
Siendo el usuario `postgres`, podemos crear la base de datos como antes, o podemos usar los siguientes comandos:

```
createuser berka -D -l -P # Crear el usuario berka
```

Las banderas usadas `-l` (puede hacer login), `-P` (pregunta por la contraseña a asignar) y `-D`, le remueve los permisos para crear bases de datos.

```
createdb -O berka berka # Crear la bd berka y vuelve dueño al usuario berka
```

En la carpeta `berka/sql` está el archivo con el código necesario para crear el esquema `raw`.

En lugar de ejecutar nuestro *pipeline* desde un archivo `bash`, lo haremos con una aplicación en `python`. Lo puedes ver en `berka/berka.py`.

Ejercicio 13

- Crea la estructura de directorios recomendada en este curso para `berka`
- Agrega logging al archivo `berka.py`
- Crea el archivo `create_schemas.sql` en la carpeta `sql`
- Completa las funciones `create_schemas` y `create_raw_tables` en archivo `berka.py`

5.8 Usando SQL para Ciencia de Datos

`SQL` es un lenguaje *declarativo*, es decir, no le decimos a la computadora (en este caso al RDBMS) **como** debe proceder paso a paso para obtener nuestro resultado deseado (como en un lenguaje imperativo como `python` o `R`), al contrario, le especificamos el resultado (lo que queremos) y el RDBMS debe de averiguar/calcular por si mismo los pasos para obtenerlo[45].

Dicho esto, la manera en la que construyes el *query* es describiendo lo que quieres, no como lo obtienes.

Para ejemplificar esto, exploremos la base de datos `berka`.

Por ejemplo podemos contar el número de renglones

```
select count(*) from clients;
```

[45] Técnicamente, postgres (y otras bases de datos) lo hacen con un componente llamado planner

Ejercicio 14

Calcula el número de renglones de cada una de las tablas

5.8.1 SELECT

`SELECT` es usado (no hay sorpresa aquí) para seleccionar datos de la base de datos. Los datos obtenidos son almacenados en una tabla (todo son tablas) llamada *result set*.

En su forma más sencilla es:

```
SELECT <atributos>
FROM <una o más relaciones>
WHERE <condiciones>
```

Ejercicio 15

Parece sencillo ¿cierto?, ejecuta

```
\help select
```

```
select * from transs limit 1;
```

trans_id	account_id	date	type	operation	amount	balance	k_symbol	bank	account
695247	2378	930101	PRIJEM	VKLAD	700.0	700.0			

Hay dos cosas a notar en el *query* pasado: primero usamos el modificador `limit`. El como supondrás limita el número de renglones en el *result set*. Si no lo haces, obtendrás más de un

millón de renglones en tu pantalla. El segundo punto es el uso de " * " en el *query*. Este comodín le dice a la base de datos tomar **todas** las columnas de la tabla. Esto es muy ineficiente y se considera mala práctica: *Sólo regresa las columnas que te interesan*:

```
select
  trans_id, account_id, date, amount
  from
      transs
 limit 3;
```

trans_id	account_id	date	amount
695247	2378	930101	700.0
171812	576	930101	900.0
207264	704	930101	1000.0

En su forma (más o menos) completa:

```
SELECT DISTINCT column, AGG_FUNC(column_or_expression), …
FROM mytable
JOIN another_table
      ON mytable.column = another_table.column
WHERE constraint_expression
GROUP BY column
HAVING constraint_expression
ORDER BY column ASC/DESC
LIMIT count;
```

Modificadores básicos

1. LIMIT
   Como recién vimos, las tablas pueden tener millones de renglones, y la transferencia desde el servidor así como la impresión a pantalla puede tomar muchísimo tiempo. En la etapa de exploración es mejor *limitar* el número de renglones.

2. ORDER BY Ordena el *result set* de forma ascendente (por omisión, o usando el atributo `asc`) o descendente (con `desc`) en la(s) columna(s) especificada(s).
   Si udas `order by` y `limit`, puedes responder preguntas como el *top 10 de ...* o el *bottom 5 de ...*
3. WHERE
   La cláusula `where` es usada para filtrar *records* o renglones. Sólo las tuplas que cumplen con la condición especificada son preservados.

```
select
  trans_id, account_id, date, amount
  from
      transs
where account_id = '3450'
order by date desc
limit 5;
```

trans_id	account_id	date	amount
3465829	3450	981231	176.4
1009993	3450	981230	2700.0
1009947	3450	981225	39232.0
1009982	3450	981209	18191.0
1009983	3450	981203	11332.0

4. GROUP BY
   Es usado con funciones de agregación (`count`, `max`, `min`, `sum`, `avg`) para agrupar el *result set* basado en una o más columnas.

```
select account_id, count(*) as txs
from transs
group by account_id
order by count(*) desc
limit 3;
```

account_id	txs
8261	675
3834	665
96	661

5. HAVING

   `GROUP BY` fue usado para crear agregaciones de renglones en grupos, para luego aplicar en ellos alguna función como `min, max, sum, count, avg`.

   Si quieres especificar una condición de filtrado en los resultados necesitas usar el *keyword* `HAVING`:

```
select
  account_id as account,
  count(*) as txs,
  round(avg(amount),2)  as amount_per_tx
from transs
group by account_id
having amount_per_tx ⩾ 20000
order by amount_per_tx desc
```

account	txs	amount_per_tx
1935	143	21062.61
2384	235	21011.14
2211	65	20842.65
3634	101	20287.1

   Entonces, `WHERE` aplica la condición a los renglones *antes* de ser agregados, `HAVING` lo hace *después*, i.e. aplica la condición a los renglones agregados.

Orden de ejecución

El orden de ejecución de un *query* no es de arriba hacia abajo (como en un lenguaje imperativo), es algo un *poco* más complicado:

paso	instrucción
5	`SELECT`
6	`DISTINCT column, AGG_FUNC(column_or_expression), …`
1	`FROM mytable`
1	`JOIN another_table`
1	`ON mytable.column = another_table.column`
2	`WHERE constraint_expression`
3	`GROUP BY column`
4	`HAVING constraint_expression`
7	`ORDER BY column ASC/DESC`
8	`LIMIT count;`

FILTER

Si quieres ejecutar *varias* agregaciones en un sólo *query*, tienes que recurrir a condicionales en SQL: `case when ... then ... else ... end`

```
select
  count(*) as total,
  count(prijem) as num_prijem,
  count(vyber) as num_vyber,
  count(vydaj) as num_vydaj
  from (
    select
      case when type = 'PRIJEM' then 1 end as prijem,
      case when type = 'VYBER' then 1 end as vyber,
      case when type = 'VYDAJ' then 1  end as vydaj
      from raw.trans
  ) as trans;
```

El modificador `filter` es un sustituto del `case when ... then ... else ... end`, así el *query* anterior puede ser escrito dela manera más compacta

```
select
  count(*) as total,
  count(*) filter(where type = 'PRIJEM') as num_prijem,
  count(*) filter(where type = 'VYBER') as num_vyber,
  count(*) filter(where type = 'VYDAJ') as num_vydaj
  from raw.trans;
```

Funciones de agregación (agregaciones)

Mencionamos (y usamos) funciones de agregación en las secciones de GROUP BY y HAVING .

Una manera de caracterizar[46] a las funciones de agregación, es que son funciones que reciben n valores regresan 1.

[46] Y que ayudará a compararlas con las *Window functions*.

La tabla 5.1 muestra algunos de las funciones más usadas

Cuadro 5.1: *Agregaciones básicas soportadas en PostgreSQL*

Básicos	Estadísticos	Lógicos	Grupos
avg	corr	bool_and	array_agg
count	stdenv	bool_or	json_agg
max	variance	every	xmlagg
min			string_agg
sum			

Como siempre, la documentación de PostgreSQL sobre agregaciones es una excelente referencia para aprender sobre las opciones disponibles.

Además de las funciones recién mostradas SQL define funciones de agregación que dependen del order (*ordered-set*) y funciones aplicadas a la ventana[47] hipotética formada por los argumentos.

[47] Ver *Window functions* más adelante.

Estas funciones de agregación se usan por ejemplo:

```
select
mode() within group(order by amount::float) as "moda",
percentile_cont(0.5) within group(order by amount::float) as "medi
percentile_cont(ARRAY [0.25,0.75]) within group(order by amount::f
from raw.trans;
```

En general las funciones de agregación se pueden escribir como:

Cuadro 5.2: *Agregaciones avanzadas soportadas en PostgreSQL*

ordered-set	hypothetical-set
mode()	rank(args)
percentile_cont(fractions)	dense_rank(args)
percentile_disc(fractions)	percent_rank(args)
	cume_dist(args)

```
aggregate_name (expression [ , ... ] [ order_by_clause ] ) [ FILTER ( WHERE filter_clause ) ]
aggregate_name (ALL expression [ , ... ] [ order_by_clause ] ) [ FILTER ( WHERE filter_clause )
aggregate_name (DISTINCT expression [ , ... ] [ order_by_clause ] ) [ FILTER ( WHERE filter_clau
aggregate_name ( * ) [ FILTER ( WHERE filter_clause ) ]
aggregate_name ( [ expression [ , ... ] ] ) WITHIN GROUP ( order_by_clause ) [ FILTER ( WHERE fi
```

Subqueries

Un *subquery* es simplemente un *query* dentro de otro *query*. El *query* no tiene que ser necesariamente un `SELECT`, el *query* puede ser un `INSERT`, `UPDATE`, `DELETE`, etc. El valor del *subquery* es asignado al *query* exterior.

El valor de retorno nos permite clasificar a los *subqueries*, y al mismo tiempo saber dónde es posible usarlos.

```
select trans_id, account_id, date, amount
from raw.trans
 where amount::float >
       (select max(amount::float)
          from raw.trans
         where account_id = '2378')
 limit 5;
```

Código 5.1: *Datos sobre transacciones que fueron mayores al máximo gastado por la cuenta 2378.*

Los *subqueries* se pueden clasificar de varias formas, pero para nuestros objetivos basta con la siguiente clasificación:

1. Escalares
2. Correlacionados

   Los *subqueries* correlacionados son aquellos que hacen referencia a variables del *query* superior. Esta referencia

```
select
  trans_id, account_id, date, amount,
  /**
      Este es el subquery escalar. Calcula el promedio de las txs.
      También se muestra el uso de la función TRUNC para truncar a
      dos decimales.
  */
  (select trunc(avg(amount::numeric),2) from raw.trans) as "AVG(am
from raw.trans
 where amount::float >
       (select max(amount::float)
          from raw.trans
         where account_id = '2378')
 limit 5;
```

Código 5.2: *Datos sobre transacciones que fueron mayores al máximo gastado por la cuenta 2378. Se muestra, además, la cantidad promedio por transacción.*

hace que el *subquery* se ejecute **una** vez por cada renglón del *query* superior.

```
select
  "A2" as district_name, "A4" as inhabitants, "A3" as region
  from
      raw.district as t1
 where "A4"::float > (
   /** Este subquery es correlacionado */
   select
     avg(t2."A4"::float)
     from
         raw.district as t2
    where t2."A3" = t1."A3"
 ) limit 5;
```

Código 5.3: *Distritos con mayor número de habitantes que el promedio de la región.*

Se pueden lograr optimizaciones si agregas índices o si lo reescribes como un join.

Operaciones de conjuntos

Otra manera de unir tablas o relaciones, además de usar *joins*, es utilizar operaciones de conjuntos.

Es posible conceptualizar que los *joins* pegan las tablas de manera horizontal (agregan columnas) y los operadores de conjuntos en forma vertical (agregan renglones).

Los operadores de conjuntos van en medio de dos `selects`:

```
(select * from tablaA)
operador de conjuntos
(select * from tablaB)
```

Código 5.4: *Pseudocódigo mostrando el uso de los operadores de conjuntos.*

Donde el operador de conjuntos puede ser uno de:

- `union` / `union all`
- `intersect` / `intersect all`
- `except`

Ejercicio 16

Prueba la diferencia entre `union` y `union all` usando la `tablaA`.

Ejercicio 17

- Crea el archivo `to_clean.sql`, en él:
  - Arregla los nombres de las tablas (en plural)
  - Arregla los nombres de los identificadores (en singular)
  - Crea los catálogos correspondientes (ve el apéndice)
  - Crea las columnas faltantes (`sexo`, `edad` o `fecha_nacimiento`)
- Completa la función `to_clean` en archivo `berka.py`

Exportar datos

Ahora que entendemos como seleccionar datos desde nuestra base de datos, podemos usar la combinación de `\copy` y `select` para exportar datos hacia afuera de la *base de datos*.

```
\copy (select columns from table1 where conditions) to 'archivo.ps
```

5.8.2 Joins

La operación `join` *combina* tablas horizontalmente (i.e. agregando las columnas). Es, quizá, la operación más poderosa de `SQL`. La combinación de las tablas se da a nivel renglón si se cumple la condición de unión. Existen 6 diferentes tipos de *joins*: `cross`, `inner`, `outer`, `semi`, `anti` y `lateral`.

En esta sección discutiremos los primeros 5 tipos y dejaremos la discusión de `lateral join` para una sección posterior.

Para explicar mejor estos *joins* usaremos las dos tablas siguientes:

Cuadro 5.3: *Tabla A para ejemplificar las operaciones* join

num	color
1	negro
2	azul
3	blanco

Cuadro 5.4: *Tabla B para ejemplificar las operaciones* join

letra	color
A	azul
B	blanco
C	naranja

CROSS JOIN

Este *join* es el bloque de construcción básico de la mayoría de los *joins* siguientes. También es el más fácil de comprender. El `cross join` es un *producto cartesiano*.

Usando las tablas recién definidas

```
select * from  tablaA cross join tablaB;
```

num	color	letra	color
1	negro	A	azul
1	negro	B	blanco
1	negro	C	naranja
2	azul	A	azul
2	azul	B	blanco
2	azul	C	naranja
3	blanco	A	azul
3	blanco	B	blanco
3	blanco	C	naranja

Nota que el tamaño del resultado es *renglones en A* $\times$ *renglones en B*.

INNER JOIN

Este *join* es de las operaciones originales de la base de datos de Edward F. Cobb. En álgebra relacional a esta operación es conocida como Θ-join.

Esta operación *filtra* el resultado del cross join mediante el uso de un predicado[48]. En este filtrado, sólo se retienen las combinaciones para los cuales existen colores en ambas tablas.

[48] Es decir cualquier enunciado que sea verdadero o falso dependiendo del valor de sus variables. Formalmente $P : X \to \{\mathcal{T}, \mathcal{F}\}$, i.e. una función booleana.

```
select * from
tablaA inner join tablaB
on tablaA.color = tablaB.color;
```

Nota que como es una operación de filtrado, si una tabla está vacía, el resultado final será vacío.

Cuadro 5.5: *inner join*

num	color	letra	color
2	azul	A	azul
3	blanco	B	blanco

El *inner join* tiene varias sintaxis alternativas, por ejemplo, con el *keyword* `USING`, el cual se puede aplicar si **ambas** tablas tienen una o más columnas con el mismo nombre[49].

[49] Existe el caso también del **NATURAL JOIN** el cual considero monstruoso y no debiera de ser usado *nunca*, ya que viola el precepto de *explícito es mejor que implícito*.

```
select *
from tablaA
inner join
tablaB using(color);
```

Cuadro 5.6: *using*

num	color	letra
2	azul	A
3	blanco	B

Este *join* soporta **cualquier** tipo de predicado. En el caso en el que el predicado está probando una igualdad, algunas personas le llaman a este *join* **Equijoin**.

Es importante recordar que el *inner join* ocurre *después* del cross join. Por lo tanto, el siguiente *query* es equivalente[50]:

[50] Puedes verificar esto usando `explain query plan` como prefijo al *query* en `sqlite`, o `explain` en PostgreSQL.

```
select *
from
tablaA cross join tablaB
where tablaA.color = tablaB.color;
```

num	color	letra	color
2	azul	A	azul
3	blanco	B	blanco

Atención: Una creencia (principalmente entre las personas con *background* matemático) es que el *inner join*, remueve duplicados. Esto **NO** es cierto.

Para obtener ese efecto hay que utilizar `DISTINCT` o el modificador exclusivo de PostgreSQL `DISTINCT ON`, o utilizar *window functions*.

OUTER JOIN

El *outer join*, además de los renglones que devuelve el inner join, retiene del cross join renglones de la tabla del lado izquierdo/derecho que no satisficieron el predicado. Los renglone que no satisfacen el predicado son llenados con `NULLs`.

```
select * from
tablaA left outer join tablaB
on tablaA.color = tablaB.color;
```

Cuadro 5.7: *left join*

num	color	letra	color
1	negro		
2	azul	A	azul
3	blanco	B	blanco

El *keyword* `OUTER` puede ser omitido[51]. El ejemplo mostrado en **??** es un *left outer join*, el cual retiene **todos** los renglones de la tabla izquierda en el resultado.

[51] Y es la práctica recomendada.

Existe también, el *right outer join*, el cual devuelve todos los renglones de la tabla derecha, aún si no tuvieron un *match* .

```
select * from
tablaA right outer join tablaB
on tablaA.color = tablaB.color;
```

El cual, por motivos de comunicabilidad, no se recomienda[52].

[52] De hecho, en `sqlite` no está implementado.

Usa siempre un *left outer join*, todos los *right joins* pueden ser convertidos a *left joins*.

FULL (OUTER) JOIN

[53] Tampoco es soportado en `sqlite`.

En este *join*[53], se retienen *todos* los renglones de ambas tablas, hayan o no hayan satisfecho el predicado.

```
select * from
tablaA full join tablaB
on tablaA.color = tablaB.color;
```

Es posible también usar el *keyword* `USING`.

SEMI JOIN

[54] Si esperabas ver `select * from tablaA left semi join tablaB ...`, tendrás que usar Cloudera Impala.

Cuando sólo queremos la *mitad* del resultado en una operación *join*, debemos de usar el modificador[54] `EXISTS` e.g. (en nuestras tablas para ejemplificar) *números que tienen letras*, pero **no** queremos en el resultado las letras.

```
select *
from
    tablaA
where exists (
  select *
    from
        tablaB
   where
     tablaA.color = tablaB.color
);
```

num	color
2	azul
3	blanco

Es decir, *queremos los renglones de la tabla A que tienen un color que* **existe** *en la tabla B*.

Una alternativa equivalente es usar el *keyword* `IN`:

```
select *
from
    tablaA
where color in (
  select color
    from
        tablaB
);
```

num	color
2	azul
3	blanco

ANTI JOIN

El opuesto del semi join es el *anti join*: queremos **todos** los números que **no** tienen color.

```
select *
from
    tablaA
where not exists (
  select *
    from
        tablaB
   where
     tablaA.color = tablaB.color
);
```

num	color
1	negro

Atención: Es totalmente razonable pensar que como `EXISTS` e `IN` son equivalentes, `NOT EXISTS` y `NOT IN` también lo serán. Pero, en el caso general, **NO** lo son.

```
select * from
tablaA
where color not  in ('rojo', 'verde', NULL);
```

Recuerda, `NULL` en `SQL` significa *desconocido*. En este caso, el predicado no puede ser evaluado a verdadero o falso y por lo tanto , *query* retorna vacío.

¡Ten mucho cuidado con esto!

Otra alternativa peligrosa (o por lo menos lenta) es la siguiente:

```
select tablaA.* from
tablaA
left join
tablaB  using(color)
where letra is null;
```

num	color
1	negro

El *query* anterior **NO** es tratado como un *join* por la base de datos, lo cual lo hace ineficiente, tiene los mismos peligros ocultos que usar `NOT IN` y además es difícil de leer.

5.8.3 Índices

- No son necesariamente `id` (identificadores de renglón), pero pueden serlo
- No son necesariamente `primary/foreign keys`, pero pueden serlo

Sintáxis y propuesta de nombrado:

[52] *Full text search*

[53] Geographical Information System

Cuadro 5.8: *Principales tipos de índices en PostgreSQL*

tipo	cuándo usarlo
b-tree	Valor por omisión
gin	hstore, array, json
gist	FTS[55], GIS[56]
sp-gist	GIS
hash	

```
create index esquema.tabla.columna(s)_ix on esquema.tabla(columna(s));
```

5.8.4 *SQL Analítico:Window Functions*

Las funciones de ventana (window functions) convierten a las funciones de agregación en algo más poderoso. En particular le permiten a la función de agregación tener acceso a un conjunto de renglones desde el renglón actual. Este conjunto de renglones es definido mediante el *keyword* `OVER`.

`OVER` define que renglones son visibles en cada renglón.

- `over()` hace visibles todos los renglones para cada renglón.
- `over(partition by algo)` segrega de manera parecida a un `group by`.

Las funciones de ventana (también conocidas como analíticas), regresan un valor por cada renglón, a diferencia de las funciones de agregación que regresan un renglón por grupo. Las ventanas son *particionadas* en grupos mediante el *keyword* `PARTITION BY` . El valor de retorno es calculado en los renglones de la partición de la ventana. Para la mayoría de las funciones el *orden* de los renglones en la ventana es importante.

Sintáxis

De la documentación

```
function_name ([expression [, expression ... ]]) [ FILTER ( WHERE
function_name ([expression [, expression ... ]]) [ FILTER ( WHERE
function_name ( * ) [ FILTER ( WHERE filter_clause ) ] OVER windo
function_name ( * ) [ FILTER ( WHERE filter_clause ) ] OVER ( win
```

donde `window_definition` es:

```
[ existing_window_name ]
[ PARTITION BY expression [, ...] ]
[ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { F
[ frame_clause ]
```

Cuadro 5.9: *Funciones de ventana en PostgreSQL*

función	¿Qué hace?
row_number()	Regresa el número de la fila actual.
rank()	Rango de la fila actual con gaps.
dense_rank()	Lo mismo pero sin gap.
percent_rank()	Rango relativo del renglón actual $\frac{rank-1}{renglones-1}$.
cume_dist()	Rango relativo del renglón actual.
ntile()	Divide los renglones en una partición equitativamente
lag()	Regresa el valor de la fila anterior (partición).
lead()	Regresa el valor de la fila siguiente (partición).
first_value()	Regresa el primer valor del marco.
last_value()	Regresa el último valor del marco.
nth_value()	Regresa el n-ésimo valor del marco.

Preguntas analíticas: Viendo a través de la ventana

¿Cómo se compara el número de transacciones de una cuenta con otras cuentas en el mismo distrito? ¿Total de dinero transaccionado?¿Promed de préstamos?

```
with txs_per_account as (
select
        client,
        type,
        extract(year from date) as year,
        count(*) as txs
from semantic.events
where extract(year from date) = 2015 and type is not null
```

```
group by client, type, year
)

select
        year, client,
        type,
        txs,
        sum(txs) over w1 as "total inspections per type",
        100*(txs::decimal/sum(txs) over w1)::numeric(18,1)  as "% of inspections",
        (avg(txs) over w1)::numeric(18,3) as "avg inspections per type",
        txs - avg(txs) over w1 as "distance from avg",
        first_value(txs) over w2 as "max inspections per type",
        inspections - first_value(txs) over w2 as "distance from top 1",
        dense_rank() over w2 as rank,
        (nth_value(txs,1) over w3 / txs::decimal)::numeric(18,1) as "rate to top 1",
        ntile(5) over w2 as ntile
from txs_per_account
where type = 'owner'
window
       w1 as (partition by type, year),
       w2 as (partition by type, year order by txs desc),
       w3 as (partition by type, year order by txs desc rows between unbounded preceding and unb
limit 10;
```

Preguntas analíticas: Usando renglones anteriores

En una fecha dada, ¿Cuántos días desde la última transacción?

```
select
account_id
date::date as tx_date
lag(date::date, 1) over w1 as previous_tx,
age(date::date, lag(date::date,1) over w1) as time_since_last_tx
from raw.trans
where facility_type = 'wholesale'
window w1 as (partition by account_id order by date::date asc)
order by account_id, date::date asc ;
```

Código 5.5: *Días desde la última transacción de tipo retiro (`VYDAJ`). Lo estamos haciendo desde la tabla `raw.trans`. Obvio lo deberíamos de hacer desde el esquema `cleaned` o `semantic`.*

Preguntas analíticas: Restringiendo la ventana

Cantidad de dinero transaccionado en las últimas tres transacciones

```
with txs as (
select
     client,
     account,
     date,
     sum(amount) as total_amount
from cleaned.transactions
group by client, account, date
)

select
        client,
        date,
        total_amount,
        sum(total_amount) over w as running_total,
        array_agg(total_amount) over w as previous_transactions
from txs
where  client = 1
window w as (partition by client order by date asc rows between 3
```

Código 5.6: *Cantidad de dinero transaccionado en las últimas tres transacciones*

5.8.5 *Datawarehousing*

A veces necesitas generar datos para un *dashboard* de BI. Típicamente debes de mostrar el número total de transacciones, sus resultados o tipos, desglosarlo por ciudad, distrito, mes y año. Obviamente incluyendo totales y subtotales.

Funciones de *datawarehousing*

Los desarrolladores de `PostgreSQL` sobrecargadorn el `GROUP BY`, de tal manera que además de su uso regular, lo puedes usar para generar tablas o reporte de métricas de agregación por conjuntos (`GROUPING SETS`), jerarquías (`ROLLUP`) y combinaciones (`CUBE`) en un *query* sencillo.

```
select
        extract(month from date) as month,
        extract(year from date) as year,
        type,
        count(*) as number_of_txs
from semantic.events
where extract(year from date) = 2017 and
      extract(month from date)  = 1
group by month, year, type
```

```
--group by GROUPING SETS (month, year, type, ())
--group by ROLLUP (month, year, type, result)
--group by CUBE (month, year, type)
```

5.8.6 Consejos para hacer búsquedas

- Los problemas principales de un 'query' son:
  - No hay memoria para el sorting.
  - No hay estadísticas.
  - Está escrito con las patas.
- Hagan mucho en cada query
  - PostgreSQL es muy bueno con queries grandes
  - (y no tan bueno con muchos queries pequeños).
- Asegurar que cuando usen una llave o índice los tipos coincidan (o no se usará el índice).
- Eviten búsquedas de texto como `LIKE %Hola%`, usen expresiones regulares o FTS.

Buscanco optimizaciones: EXPLAIN

Muestra el plan de ejecución del *query*

```
explain [analyze]
select * from ...
```

Ejemplos:

Plan hipotético:

```
explain
  select *
  from
  raw.trans;
```

Planea y ejecuta el mejor plan:

```
explain analyze
select *
from
raw.trans;
```

Observa el cambio con filtros:

```
explain analyze
  select *
  from
  raw.trans
  where
  amount::numeric > 5000;
```

Lo que debemos de buscar en este árbol invertido[57] es conteos innecesarios, escaneos secuenciales, loops enormes, etc.

[57] Hay que buscar en el nivel más profundo para ver donde ocurre el problema.

5.8.7 Modificar: Insertar, borrar, actualizar

Usando `psql`:

```
begin; -- Abrimos una transacción
```

Creamos la siguiente tabla

```
create table cosas ( -- tabla en plural
  cosa serial not null, -- identificador en singular
  nombre varchar,
  etiquetas varchar [],
  periodo tsrange,
  created_at timestamp, -- siempre
  updated_at timestamp -- siempre
);
```

Insertar tuplas

`insert` agrega tuplas de valores a una relación

```
insert into r(a1,...., an) values (v1,...., vn);
```

```
insert into cosas
values (1, 'Cosa 1', '{"Algo", "Otra cosa"}', '[2015-03-04 08:00, 2015-03-04 11:00]', now(), now
insert into cosas
values (2, 'Cosa 2', '{"Algo más", "cosa"}', '[2005-03-04 08:00, 2007-03-04 11:00]', now(), now(
insert into cosas
values (3, 'Cosa 3', '{"Algo", "cosa"}', '[2019-03-04 08:00, 2019-10-04 11:00]', now(), now());
```

Borrar tuplas

```
\help delete
```

```
delete from cosas
where cosa = 1;
```

Actualizar

```
\help update
```

```
update cosas
set updated_at = now() where cosa = 3;
```

Seleccionar usando operadores

Por ejemplo, el operador de contención @>

```
select * from cosas where periodo @> '2017-01-01';
```

Finalizando

Al acabar (y para no dejar nada permanente en la base de datos) ejecuta

```
rollback;
```

5.8.8 Algunos *goodies* de PostgreSQL

Fechas

```
select
date_trunc('year', '2014-02-25'::date) as year,
date_trunc('month', '2014-02-25'::date) as month,
date_trunc('day', '2014-02-25'::date) as day;
```

```
select
to_char('2013-02-25'::date, 'YYYY') as year,
to_char('2013-02-25'::date, 'MM-YYYY') as month,
to_char('2013-02-25'::date, 'DD') as day;
```

```
select
date_part('day', '2013-02-25'::date) as day;
```

Generar secuencias

Ejemplo básico

```
select * from generate_series(0,20,5);
```

Usándola con funciones

```
select avg(val)
from generate_series(0,20,5) as val;
```

Una serie de fechas

```
select
current_date + step.i as date_series
from
generate_series(0,15,3) as step(i);
```

5.8.9 Esquema propuesto

- `raw` → `clean` → `semantic` → `features`, `labels`, `cohort` → results
- Algunas otros esquemas: `dwh`, `ts`, `labels`, `cohorts`, `samples`, `burning bus` `stats`

raw o *staging*

- Los datos tal como son
- Si tienes archivos, un archivo -> una tabla
- Si los datos estan muy sucios, tus tablas pueden ser de una columna
- Todos los tipos de columna son `varchar`

clean

- Limpiar los tipos
- Arreglar los formatos
- Escojer una representacion para las cadenas
- Unir tablas
- Limpiar nombres de columnas
- Generar catálogos
- *tidy data*

semantic

- Dado el problema, escoger a las entidades y a los eventos
- Unir en el conjunto minimo de tablas (i.e. una tabla entidad, una tabla eventos)

6 Proyectos

6.1 Datasets para proyectos

Sakila DVD rental store, featuring things like films, actors, film-actor relationships, and a central inventory table that connects films, stores, and rentals.

PGFoundry World, DellStore, Pagila

Land registration (UK) Also here, and here

Adventure Works for PostgreSQL

Mouse Genome sample data set Instructions

Freebase

IMDb

MoMa

hanMusicBrainz

RITA

United States Sentencing Commission Individual Offender Data Sets (from `@kwilson`)

6.2 ¿ Qué debo de hacer?

- Crear un repositorio
- Crear estructura de carpetas para un proyecto en `python`
- Escoger una fuente de datos
- Crear un `README.md`: Describir la fuente de datos. Describir la entidad, Estructura de la base de datos, *Pipeline* .Instalación. Ejecución.
- Cargar la base de datos a `raw`

- Crear una versión limpia en `cleaned`
- Crear el esquema `semantic`
- Crear *features* temporales ligados a la entidad dadas las fechas del evento. Guardarlos en el esquema *features*

Abreviaciones usadas

PCA	Principal component analysis
RDBMS	Relational Data Base Management System
SNF	Smith normal form
SQL	Structured query Language
TDA	Topological data analysis

Glosario

LaTeX	A document preparation system
$\mathbb{R}$	The set of Real numbers

Apéndices

Turista

Implementando el juego del Turista

1 Descripción

Basado en el Monopoly

Agregar historia de Monopoly

Turista Mundial

Instructivo

Deuda Eterna (Versión cubana)

2 Primera iteración

Figura 1
Simulador de juegos de Turista: Primera iteración

```
class Dados:

    def __init__(self, numero_caras=6, numero_dados=2):
        self.numero_dados = numero_dados
        self.total = None
```

```
        self.son_iguales = False
        self.caras = np.arange(1, numero_caras+1)
        self.tirada = None

    def tirar(self):
        self.tirada = np.random.choice(self.caras, self.numero_dad

        self.total = self.tirada.sum()
        self.son_iguales = len(set(self.tirada)) == 1

    def __repr__(self) -> str:
        return f"{self.tirada} (={self.total})"
```

```
dados = Dados()
dados.tirar()
print(dados)
```

Las piezas en el Turista son aviones de colores, pero para darle una mayor variedad, copiaremos las que tiene el juego de Monopoly:

```
Pieza = Enum('Pieza',
             'TOP_HAT BATTLESHIP RACECAR SCOTTIE_DOG CAT TREX PENG
```

En donde usamos un `Enum`.

```
trex = Pieza['TREX']
print(trex)
```

```
class Pais:
    def __init__(self, nombre, posicion, tablero):
        self.nombre:str = nombre
        self.posicion:int = posicion
        self.turista:Turista =  tablero
        self.piezas:List[Pieza] = []

    def quitar(self, pieza:Pieza) -> None:
        pass

    def mover(self, movimientos:int) -> Pais:
        return self.turista.tablero[self.posicion + movimientos %


    def poner(self, jugador:Jugador) -> None:
        self.piezas.append(jugador.pieza)
        jugador.posicion = self

    def __repr__(self) -> str:
        return f"{self.nombre} [{self.posicion}]"
```

```
costa_rica = Pais(nombre="Costa Rica", posicion=2, tablero=None)
print(costa_rica)
```

```
@dataclass
class Jugador:
    pieza: Pieza
    posicion: Pais

    def turno(self, dados:Dados) -> int:
        dados.tirar()

        self.posicion.quitar(self)
        self.posicion = self.posicion.mover(dados.total)
        self.posicion.poner(self)

    @property
    def quebrado(self):
        return False


    def __repr__(self) -> str:
        return f"{self.pieza}@{self.posicion}"
```

```
jugador = Jugador(trex, costa_rica)
print(jugador)
```

```
class Turista:

    NUMERO_PAISES = 40

    def __init__(self, numero_jugadores:int=4, maximo_rondas=50):
        self.numero_jugadores:int = numero_jugadores
        self.maximo_rondas:int = maximo_rondas
        self.jugadores:List[Jugador] = self._crear_jugadores()
        self.tablero:List[Pais] = self._crear_tablero()
        self.rondas:int = 0
        self.jugador_actual = None

    def _crear_tablero(self) -> List[Pais]:
        tablero = [Pais(f"P{posicion}", posicion, self) for posicion in range(Turista.NUMERO_PAI
        return tablero

    def _crear_jugadores(self) -> List[Jugador]:
        return [Jugador(pieza=pieza,
                        posicion=None)
                for pieza in Pieza][:self.numero_jugadores]

    @property
    def posicion_inicial(self):
```

```
        return self.tablero[0]

    def colocar_tablero(self):
        for jugador in self.jugadores:
            self.posicion_inicial.poner(jugador)

        self.ganador = None

        self.rondas = 0

    def jugar(self):
        self.dados = Dados()

        self.colocar_tablero()

        print(self)

        while(self.continuar()):

            for jugador in self.jugadores:
                self.jugador_actual = jugador
                if not self.jugador_actual.quebrado:
                    self.jugador_actual.turno(dados)
            self.rondas += 1
            print(self)

        self.ganador = self.jugador_actual

    @property
    def hay_jugadores(self) -> bool:
        return all([not jugador.quebrado for jugador in self.jugad
    #
    def continuar(self) -> bool:
        return self.rondas < self.maximo_rondas and self.hay_jugad


    def __repr__(self) -> str:
        return f"{self.rondas}: {self.jugadores}"
```

```
t = Turista(numero_jugadores=4, maximo_rondas = 2)
print(t)
```

```
t.jugar()
```

```
class SimuladorTurista:
    def __init__(self, numero_rondas=2, numero_simulaciones=2) ->
        self.numero_simulaciones = numero_simulaciones
        self.numero_rondas = numero_rondas
```

```
    def simular(self):
        for simulacion in range(self.numero_simulaciones):
            turista = Turista(maximo_rondas=self.numero_rondas)
            ganador = turista.jugar()
            logger.info(f"Ganador: {ganador}")
```

```
s = SimuladorTurista()
s.simular()
```

3 Antes de continuar

Conseguimos lo que queríamos como primera iteración. Pero no vamos a llegar muy lejos si estamos haciendo todo de esta manera tan desordenada.
Tenemos que ordenar nuestra área de trabajo.

3.1 Estructura de directorios

3.2 Ambiente

El manejo de librerías[61] en `python` es un caos. Siguiendo una filosofía *defensiva*[62] sobre la vida, creemos un ambiente virtual.

[61] *Bibliotecas* es el término correcto en español.

[62] Estoica, más bien

```
pyenv virtualenv 3.7.3 turista
```

```
echo 'turista' > .python-version
```

3.3 Dependencias

```
pyenv shell system
curl -sSL https://raw.githubusercontent.com/sdispater/poetry/master/get-poetry.py | python
pyenv shell --unset
```

```
poetry init
```

```
poetry add numpy --extras all
```

Si ya existe un archivo `poetry.lock` (contiene las versiones *exactas*)[63] o `pyproject.toml` (contiene las especificaciones de versiones *semánticas*), puedes instalar todas las dependencias mediante

[63] Equivalente al archivo resultante de `pip freeze`

```
poetry install -E doc
```

3.4 `TOML`

Instalamos esta biblioteca para interactuar programáticamente con el archivo `pyproject.toml`

```
poetry add toml
```

3.5 Para crear aplicaciones con interfaz de línea de comandos

```
poetry add click
```

3.6 Verificador de violaciones de PEP8

```
poetry add --dev flake8
poetry add --dev flake8-docstrings
poetry add --dev xdoctest
poetry add --dev pydocstyle
```

```
poetry run flake8 .
```

3.7 Formateo de código

```
poetry add --dev black --allow-prereleases
```

Acomodar los imports en el orden correcto

```
poetry add --dev isort -E pyproject
```

```
poetry run black .
```

3.8 Verificación estática

```
poetry add --dev mypy
```

```
poetry run mypy .
```

3.9 Pruebas unitarias

```
poetry add --dev pytest-cov
poetry add --dev pytest-mock
poetry add --dev coverage
poetry add --dev tox
poetry add --dev towncrier
```

3.10 Generador de documentación

```
poetry add --dev sphinx
poetry add --dev sphinx_rtd_theme
```

4 Segunda iteración

> Any sufficiently advanced bug is indistinguishable from a feature.
>
> Rich Kulawiec

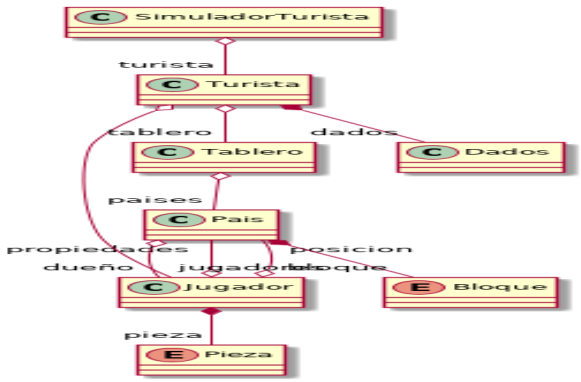

Figura 2
Simulador de juegos de turista: Segunda iteración

Un punto *doloroso* de nuestra primera iteración es causado por nuestra respuesta a la pregunta: ¿Cómo los jugadores mueven sus fichas en el tablero siguiendo las reglas de juego?

```
Bloque = Enum('Bloque',
              'ROJO MORADO VERDE NARANJA AZUL ROSA CAFÉ AMARILLO A
```

Con la `Pieza` sabiendo "dónde" está, `Pais` se vuelve más simple, así podemos enriquecerla con los atributos que faltaban:

```
class Pais:
    def __init__(self, indice:int, nombre:str, precio:int,  bloque
        self.nombre = str(nombre)
        self.indice = int(indice)
        self.precio = int(precio)
        self.renta_inicial = int(renta_inicial)
        self.costo_construccion = int(costo_construccion)
        self.bloque = Bloque[bloque]
        self.dueño:Jugador = None
        self.construcciones: List[int] = None

    @property
```

```
    def hipoteca(self) -> int:
        return self.precio/2

    @property
    def renta(self) -> int:
        numero_construcciones = len(self.construcciones) if self.construcciones else 0
        return self.renta_inicial*self.incrementos(numero_construcciones)

    @property
    def hipotecada(self) -> bool:
        return False

    @property
    def disponible(self) -> bool:
        return False

    @property
    def construible(self) -> bool:
        return False
    #
    def incrementos(self, numero_construcciones: int) -> int:
        INCREMENTOS = [1,5,15,45,80,125]

        return INCREMENTOS[numero_construcciones]

    def colocar(self, jugador:Jugador):
        if not self.dueño:
            jugador.comprar(self)
        elif self.dueño is not jugador:
            jugador.pagar(self.renta)
            self.dueño.cobrar(self.renta)

    def __repr__(self):
        return f"{self.nombre} [{'D' if self.disponible else '' }{'H' if self.hipotecada else ''
```

```
p = Pais(nombre="Costa Rica", indice=3, precio=8000, renta_inicial=1000, bloque='ROJO', costo_co
print(p)
```

El código de la clase `Turista`, en la primera iteración era largo y complicado. La razón de esto[64] es la *asignación de responsabilidades*, es decir, la clase hace muchas cosas.

Vamos a dividirla en dos clases: `Tablero` y `Turista`. La responsabilidad de `Tablero` es contener los países y las piezas. `Turista` es el juego, se encarga de los turnos, contiene los jugadores y verifica si se han cumplido las condiciones para decretar un ganador.

[64] Y de casi todo el diseño orientado a objetos.

```
class Tablero:

    NUMERO_PAISES = 40

    def __init__(self):
        self.paises:List[Pais] = []
        self.ronda = 0
        self._crear_tablero()

    def _crear_tablero(self) -> None:
        with open('data/paises.csv', 'r') as renglones:
            for renglon in renglones:
                if not renglon.startswith('indice'):
                    pais = Pais(*[columna.strip() for columna in r
                    pais.role = pais_role_factory(pais)
                    self.paises.append(pais)

    def siguiente_pais(self, pais_inicio, distancia) -> Pais:
        indice_final = (pais_inicio.indice + distancia) % Tablero.
        return self.paises[indice_final]

    @property
    def posicion_inicial(self) -> Pais:
        return self.paises[0]

    def __repr__(self) -> str:
        return f"{self.paises}"
```

```
tablero = Tablero()
print(tablero)
```

Agreguemos a `Jugador` su lista de propiedades, dinero con el que cuenta y otros atributos que ayudarán a la estadística.

```
class Jugador:
    def __init__(self, pieza:Pieza, tablero:Tablero, dinero_inicia
        self.pieza = pieza
        self.tablero:Tablero = tablero
        self.dinero_inicial:int = dinero_inicial
        self.dinero_actual:int = self.dinero_inicial
        self.vueltas:int = 0
        self.turnos:int = 0
        self.posicion:Pais = self.tablero.posicion_inicial
        self.propiedades:List[Pais] = []

    @property
    def quebrado(self):
        return self.dinero_actual ⩽ 0

    def turno(self, dados:Dados):
        dados.tirar()
```

```
        posicion_actual = self.posicion
        self.posicion = self.tablero.siguiente_pais(posicion_actual, dados.total)
        self.posicion.colocar(self)
        self.turnos += 1

    def comprar(self, pais:Pais):
        if self.dinero_actual >= pais.precio:
            self.pagar(pais.precio)
            pais.dueño = self
            self.propiedades.append(pais)

    def pagar(self, cantidad):
        self.dinero_actual -= cantidad

    def cobrar(self, cantidad):
        self.dinero_actual += cantidad

    def __repr__(self) -> str:
        return f"{self.pieza.name} @{self.posicion.nombre} ${self.dinero_actual} {self.propiedad
```

```
j = Jugador(Pieza(Pieza['CAT']), tablero, 150_000)
print(j)
```

```
j.turno(dados)
print(j)
```

```
print(j.propiedades)
```

La clase `Turista` contiene las reglas *globales* (quién ganó, el sueldo a pagar por cada vuelta, las rondas, etc) y se encarga de manejar la colocación inicial de los jugadores en el tablero.

```
class Turista:

    DINERO_INICIAL = 150_000
    SUELDO = 20_000

    def __init__(self, numero_jugadores=4, maximo_rondas=10):
        self.numero_jugadores:int = numero_jugadores
        self.maximo_rondas = maximo_rondas
        self.tablero:Tablero = Tablero()
        self.jugadores:List[Jugador] = self._crear_jugadores()
        self.rondas:int = 0
        self.jugador_actual = None
        self.dados = Dados()
```

```
    def _crear_jugadores(self) -> List[Jugador]:
        return [Jugador(pieza=pieza, tablero=self.tablero, dinero_
                for pieza in Pieza][:self.numero_jugadores]

    def jugar(self) -> Jugador:

        while(self.continuar()):
            self.ronda()

        return self.ganador

    def ronda(self) -> None:
        for jugador in self.jugadores:
            self.jugador_actual = jugador
            if not self.jugador_actual.quebrado:
                self.jugador_actual.turno(self.dados)
        self.rondas += 1

    @property
    def hay_jugadores(self) -> bool:
        return any([not jugador.quebrado for jugador in self.jugad

    def continuar(self) -> bool:
        return self.rondas < self.maximo_rondas and self.hay_jugad

    @property
    def ganador(self) -> Jugador:
        return self.jugadores[np.argmax([jugador.dinero_actual for

    def __repr__(self) -> str:
        return f"{self.jugadores}"
```

```
t = Turista()
print(t)
```

La clase `Simulador` sigue igual

```
simulador = SimuladorTurista()
simulador.simular()
```

5 Tercera iteración

El objetivo final que estamos buscando es simular el juego para poder analizar su comportamiento, para lograrlo debemos de guardar los datos generados por la simulación.

Una propuesta de qué datos queremos guardar es lo siguiente:

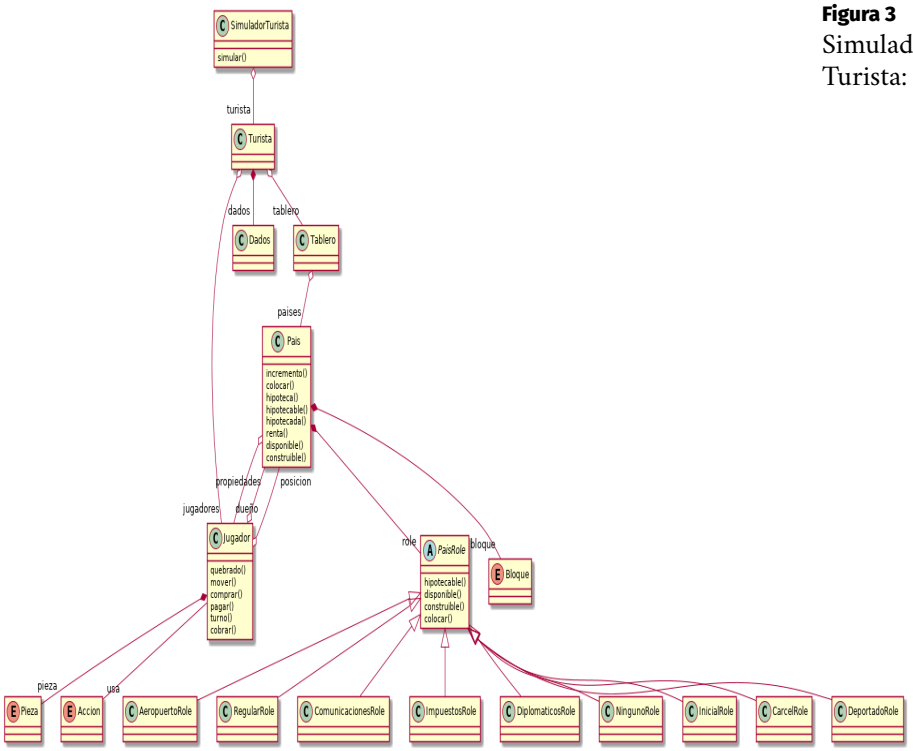

Figura 3
Simulador de juegos de Turista: Tercera iteración

```
juego|ronda|jugador|pieza|turno|posicion|accion
```

Las columnas son:

juego Identificador de la simulación
ronda Identificador de la ronda del juego
jugador Identificador del jugador
pieza Identificador de la pieza usada por el jugador
turno Turno del jugador
posicion País en el que termina el turno el jugador
accion Acción tomada por el jugador en el país

Nota que puede haber más de un renglón por turno por jugador.

Vamos a *decorar* los métodos de la clase `Jugador`. En este caso el *decorador* imprimirá a pantalla lo que necesitamos

```
def log_accion(accion:Accion):
    def _log_accion(function):
        def log(self, *func_args, **func_kwargs):
            function_output = function(self, *func_args, **func_kw
            jugador = self
            print(f"{jugador.pieza.name}|{jugador.turnos}|{jugador
            return function_output # Regresamos lo que la función
        return log
    return _log_accion # Regresamos el decorador
```

Definamos un nuevo `Enum` que contenga las acciones posibles que puede tomar un jugador

```
Accion = Enum('Accion', 'ATERRIZAR DESPEGAR COMPRAR PAGAR CONSTRUI
```

Con estos cambios, la clase `Jugador` ahora luce así

```
class Jugador:
    def __init__(self, pieza:Pieza, tablero:Tablero, dinero_inicia
        self.pieza = pieza
        self.tablero:Tablero = tablero
        self.dinero_inicial:int = dinero_inicial
        self.dinero_actual:int = self.dinero_inicial
        self.vueltas:int = 0
        self.turnos:int = 0
        self.posicion = self.tablero.posicion_inicial
        self.propiedades:List[Pais] = []

    @property
    def quebrado(self) -> bool:
        return self.dinero_actual <= 0

    def turno(self, dados:Dados) -> None:
        self.turnos += 1
        self.mover(dados)
        if self.posicion.disponible:
            logger.debug(f"{self.posicion} está disponible")
            if self.dinero_actual >= self.posicion.precio:
                logger.debug(f"{self} comprando {self.posicion}")
                self.comprar(self.posicion)

    @log_accion(Accion.ATERRIZAR)
    def mover(self, dados:Dados) -> None:
        dados.tirar()
        logger.debug(f"{self} tiró {dados.tirada}")
        posicion_actual = self.posicion
        self.posicion = self.tablero.siguiente_pais(posicion_actua
        logger.info(f"{self} aterrizando en {self.posicion}")
        self.posicion.colocar(self)

    @log_accion(Accion.COMPRAR)
```

```
    def comprar(self, pais:Pais) → None:
        logger.info(f"{self} compró {pais} por ${pais.precio}")
        self.dinero_actual -= pais.precio
        pais.dueño = self
        self.propiedades.append(pais)

    @log_accion(Accion.PAGAR)
    def pagar(self, cantidad:int) → None:
        logger.info(f"{self} pagó ${cantidad}")
        self.dinero_actual -= cantidad

    @log_accion(Accion.COBRAR)
    def cobrar(self, cantidad:int) → None:
        logger.info(f"{self} recibió {cantidad}")
        self.dinero_actual += cantidad

    @log_accion(Accion.CONSTRUIR)
    def construir(self) → None:
        logger.info(f"{self} construye un restaurante por {self.posicion.costo_construccion}")


    def __repr__(self) → str:
        return f"{self.pieza.name} @{self.posicion.nombre} ${self.dinero_actual} {self.propiedad

    def __str__(self) → str:
        return f"{self.pieza.name}"
```

```
tablero = Tablero()
jugador = Jugador(pieza=Pieza.PENGUIN, tablero=tablero)
dados = Dados()
jugador.turno(dados)
```

Diferentes países se comportan diferente, colocaremos este comportamiento en una clase aparte llamada `PaisRole`.

```
class Pais:
    def __init__(self, indice:int, nombre:str, precio:int,  bloque:str, renta_inicial:int, costo
        self.nombre = str(nombre)
        self.indice = int(indice)
        try:
            self.precio = int(precio)
        except ValueError:
            self.precio = None
        try:
            self.renta_inicial = int(renta_inicial)
        except ValueError:
            self.renta_inicial = None
        try:
            self.costo_construccion = int(costo_construccion)
        except ValueError:
            self.costo_construccion = None
```

```
        self.bloque:Bloque = Bloque[bloque]
        self.dueño:Jugador = None
        self.construcciones: List[int] = None
        self.hipotecada:bool = False
        self.role = None

    @property
    def hipoteca(self) -> int:
        return round(self.precio/2)

    @property
    def renta(self) -> int:
        numero_construcciones = len(self.construcciones) if self.c
        return self.renta_inicial*self.incrementos(numero_construc


    @property
    def disponible(self) -> bool:
        return self.role.disponible

    @property
    def construible(self) -> bool:
        return self.role.construible

    def colocar(self, jugador:Jugador) -> None:
        self.role.colocar(jugador)

    def incrementos(self, numero_construcciones: int) -> int:
        INCREMENTOS = [1,5,15,45,80,125]

        return INCREMENTOS[numero_construcciones]

    def __repr__(self) -> str:
        return f"{self.nombre} [{'D' if self.disponible else '' }+

    def __str__(self) -> str:
        return f"{self.nombre}"
```

Concentraremos la creación de las clases en un FactoryMethod, básicamente esta clase aísla la creación de los roles de la clase `Pais`.

```
def pais_role_factory(pais:Pais) -> PaisRole:
    class PaisRole(ABC):
        def __init__(self, pais:Pais):
            self.pais = pais

        @property
        def disponible(self) -> bool:
            return False

        @property
```

```
        def hipotecable(self) -> bool:
            return False

        @property
        def construible(self) -> bool:
            return False

        @abstractmethod
        def colocar(self, jugador:Jugador) -> None:
            pass

    class DiplomaticoRole(PaisRole):
        def colocar(self, jugador:Jugador) -> None:
            logger.info(f"{jugador} llegando a una estación diplomática ... ")

    class ImpuestosRole(PaisRole):
        def __init__(self, pais:Pais, impuesto_fijo:int=10_000, tasa:float=0.10):
            self.impuesto_fijo = impuesto_fijo
            self.tasa = tasa
            PaisRole.__init__(self, pais)

        def colocar(self, jugador:Jugador) -> None:
            logger.info(f"{jugador} debe de pagar impuestos")
            jugador.pagar(round(max(self.impuesto_fijo, jugador.dinero_actual*self.tasa)))

    class RegularRole(PaisRole):
        @property
        def disponible(self) -> bool:
            return self.pais.dueño is None

        @property
        def construible(self) -> bool:
            return True

        def colocar(self, jugador:Jugador) -> None:
            if self.pais.dueño and self.pais.dueño is not jugador:
                jugador.pagar(self.pais.renta)
                self.pais.dueño.cobrar(self.pais.renta)

    class ComunicacionesRole(PaisRole):
        def colocar(self, jugador:Jugador) -> None:
            logger.info(f"{jugador} tomando una carta ... Misteriosamente está en blanco ... No ha

    class AeropuertoRole(PaisRole):
        @property
        def disponible(self) -> bool:
            return self.pais.dueño is None

        def colocar(self, jugador:Jugador) -> None:
            logger.info(f"{jugador} llegando a un Aeropuerto ... ")

    class InicialRole(PaisRole):
        def colocar(self, jugador:Jugador) -> None:
            logger.info(f"¡Bienvenido a México {jugador}! Toma $20,000")
            jugador.cobrar(20_000)
```

```
    class CarcelRole(PaisRole):
        def colocar(self, jugador:Jugador) -> None:
            logger.info(f"¡{jugador} encarcelado!")
            jugador.encarcelado = True

    class DeportadoRole(PaisRole):
        def colocar(self, jugador:Jugador) -> None:
            logger.info(f"¡{jugador} deportado!")
            jugador.deportado = True

    class NingunoRole(PaisRole):
        def colocar(self, jugador:Jugador) -> None:
            logger.info(f"{jugador} Disfruta del paisaje")

    if pais.bloque is Bloque.DIPLOMÁTICOS: return DiplomaticoRole(
    if pais.bloque is Bloque.AEROPUERTOS: return AeropuertoRole(pa
    if pais.bloque is Bloque.COMUNICACIONES: return Comunicaciones
    if pais.bloque is Bloque.INICIAL: return InicialRole(pais)
    if pais.bloque is Bloque.CÁRCEL: return CarcelRole(pais)
    if pais.bloque is Bloque.DEPORTADO: return DeportadoRole(pais)
    if pais.bloque is Bloque.IMPUESTOS: return ImpuestosRole(pais)
    if pais.bloque is Bloque.NINGUNO: return NingunoRole(pais)
    else:
        return RegularRole(pais)
```

6 Algunos detalles finales

> Do one thing, do it well
>
> UNIX Philosophy

6.1 Archivo `__init__.py`

```
"""Simple simulador de turista."""

from .turista import (
    SimuladorTurista,
    Pais,
    Tablero,
    Turista,
    Jugador,
    Dados,
    Bloque,
    Pieza,
    Accion
```

```
)

__all__ = [
    "SimuladorTurista",
    "Pais",
    "Tablero",
    "Turista",
    "Jugador",
    "Dados",
    "Bloque",
    "Pieza",
    "Accion",
]

def setup_logging():
    import logging
    import logging.config
    import coloredlogs
    import yaml

    with open('config/logging.yaml', 'r') as f:
        config = yaml.safe_load(f.read())
        logging.config.dictConfig(config)
        coloredlogs.install()

setup_logging()

def get_pyproject():
    import os
    import toml

    init_path = os.path.abspath(os.path.dirname(__file__))
    pyproject_path = os.path.join(init_path, "../pyproject.toml")

    with open(pyproject_path, "r") as fopen:
        pyproject = toml.load(fopen)

    return pyproject["tool"]["poetry"]


__version__ = get_pyproject()["version"]
__doc__ = get_pyproject()["description"]
```

6.2 Interfaz de línea de comandos

```
""" Una interfaz de línea de comandos para el simulador de juegos de turista"""
import click

from dynaconf import settings

from .turista import (
```

```
    SimuladorTurista,
)

@click.command()
@click.option('--rondas', default=1, help='Número máximo de rondas
@click.option('--simulaciones', default=1, help='Número de simulac
def simular(rondas, simulaciones):
    s = SimuladorTurista(numero_rondas=rondas, numero_simulaciones
    s.simular()

if __name__ == "__main__":
    simular()
```

6.3 Pruebas unitarias

```
""" Pruebas unitarias para el juego de turista """

import pytest

def test_turista():
    assert 3 == 2
```

6.4 Un mejor logger

```
version: 1
disable_existing_loggers: true

formatters:
  simple:
    format: '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
  rich:
    format: '%(name)-30s  %(asctime)s %(levelname)10s %(process)6d
    datefmt: '%d/%m/%Y %I:%M:%S %p'

handlers:
  console:
    class: logging.StreamHandler
    level: DEBUG
    formatter: simple
    stream: ext://sys.stdout

  file:
    class: logging.handlers.RotatingFileHandler
    level: INFO
    formatter: simple
    filename: turista.log
    maxBytes: 10485760 # 10MB
```

```
    backupCount: 20
    encoding: utf8

loggers:
  turista:
    level: DEBUG
    handlers: [file]
    propagate: no

root:
  level: NOTSET
  handlers: [console]
```

Berka

Conjunto de datos Berka

1 Introducción

Para algunos de los ejercicios de esta sección usaremos los datos conocidos como **berka**. Esta base de datos fue liberada por un banco de Europa Oriental, para el **PKDD'99 Discovery Challenge**.

Puedes descargar una copia de la base aquí

2 Diagrama entidad-relación

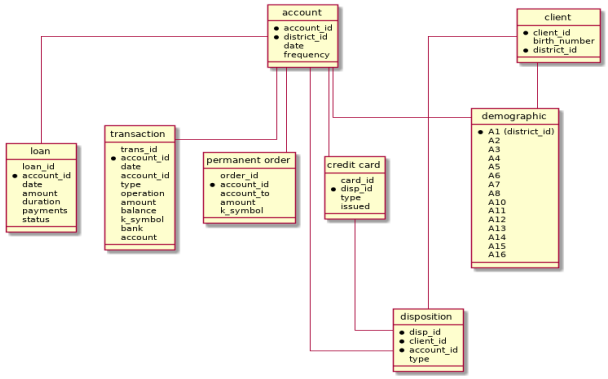

Figura 1
Diagrama de entidad-relación(ERD) para la base de datos `berka`. No se muestran todos los atributos de las relaciones.

3 Descripción de los datos

Cada cuenta tiene tanto características estáticas (eg. fecha de creación) , contenidas en la relación `account`, como dinámicas (e.g.

pagos debitados o acreditados, balances) dados en las relaciones `permanent_order` y `transactions`.

La relación `client` describe las características de la persona que puede manipular esas cuentas. Un cliente puede tener una o más cuentas, pero también se puede dar el caso que muchos clientes pueden manipular una cuenta; los clientes y cuentas están relacionadas a través de la relación `disposition`.

Las relaciones `loan` y `credit_card` describen algunos de los servicios que el banco ofrece a sus clientes:

- Varias tarjetas de crédito pueden ser asignadas a una cuenta.
- Hay un máximo de un préstamo por cuenta.

La relación `demographic_data` contiene información pública sobre los distritos (e.g. tasa de desempleo).

Pueden consultar más información aquí

4 Tablas

Cuadro 1: *Tabla describiendo las columnas de la relación* `account`*, contenida en el archivo* `account.asc`.

item	meaning	remark
`account_id`	identification of the account	
`district_id`	location of the branch	
`date`	date of creating of the account	in the form YYMMDD
`frequency`	frequency of issuance of statements	`POPLATEK MESICNE` stands fo
		`POPLATEK TYDNE` stands for w
		POPLATEK POP OBRATU stan

Cuadro 2: *Tabla describiendo las columnas de la relación* `client`*, contenida en el archivo* `client.asc`*.*

item	meaning	remark
`client_id`	record identifier	
`birth_number`	number identification of client	the number is in the form `YYMMDD` for men, the number is in the form `YYMM+50DD` for women, where YYMMDD is the date of birth
`district_id`	address of the client	

Cuadro 3: *Tabla describiendo las columnas de la relación* `disposition`*, contenida en el archivo* `disp.asc`*.*

item	meaning	remark
`disp_id`	record identifier	
`client_id`	identification of a client	
`account_id`	identification of an account	
`type`	type of disposition (owner/user)	only owner can issue permanent orders and ask for a loan

Cuadro 4: *Tabla describiendo las columnas de la relación* `order`*, contenida en el archivo* `order.asc`*.*

item	meaning	remark
`order_id`	record identifier	
`account_id`	account, the order is issued	each bank has unique two-letter code
	for bank to bank of the recipient	
`account_to`	account of the recipient	
`amount`	debited amount	
`K_symbol`	characterization of the payment	”POJISTNE” stands for insurrance payment ”SIPO” stands for household ”LEASING” stands for leasing ”UVER” stands for loan payment

Cuadro 5: *Tabla describiendo las columnas de la relación* `transcations`*, contenida en el archivo* `tran.asc`*.*

item	meaning	remark
`trans_id`	record identifier	
`account_id`	account, the transation deals with	
`date`	date of transaction in the form YYMMDD	
`type`	+/- transaction	”PRIJEM” stands for credit ”VYDAJ” stands for withdra
`operation`	mode of transaction	”VYBER KARTOU” credit ca ”VKLAD” credit in cash ”PR collection from another b withdrawal in cash ”PREV remittance to another ban
`amount`	amount of money	
`balance`	balance after transaction	
`k_symbol`	characterization of the transaction	”POJISTNE” stands for insu ”SLUZBY” stands for paym ”UROK” stands for interes UROK” sanction interest if ”SIPO” stands for househo for old-age pension ”UVER
`bank`	bank of the partner	each bank has unique two
`account`	account of the partner	

Cuadro 6: *Tabla describiendo las columnas de la relación* `loan`*, contenida en el archivo* `loan.asc`*.*

item	meaning	remark
`loan_id`	record identifier	
`account_id`	identification of the account	
`date`	date when the loan was granted	in the form YYMMDD
`amount`	amount of money	
`duration`	duration of the loan	
`payments`	monthly payments	
`status`	status of paying off the loan	’A’ stands for contract finished, no ’B’ stands for contract finished, loan not pay running contract, OK so far, ’D’ sta contract, client in debt

Cuadro 7: *Tabla describiendo las columnas de la relación* `card`*, contenida en el archivo* `card.asc`*.*

item	meaning	remark
`card_id`	record identifier	
`disp_id`	disposition to an account	
`type`	type of card	possible values are ”junior”, ”classic”, ”gold”
`issued`	issue date	in the form YYMMDD

Cuadro 8: *Tabla describiendo las columnas de la relación* `demographic`*, contenida en el archivo* `district.asc`*.*

item	meaning	remark
`A1`	`district_id`	district code
`A2`	district name	
`A3`	region	
`A4`	no. of inhabitants	
`A5`	no. of municipalities with inhabitants < 499	
`A6`	no. of municipalities with inhabitants 500-1999	
`A7`	no. of municipalities with inhabitants 2000-9999	
`A8`	no. of municipalities with inhabitants >10000	
`A9`	no. of cities	
`A10`	ratio of urban inhabitants	
`A11`	average salary	
`A12`	unemploymant rate ’95	
`A13`	unemploymant rate ’96	
`A14`	no. of enterpreneurs per 1000 inhabitants	
`A15`	no. of commited crimes ’95	
`A16`	no. of commited crimes ’96	