

# Contents

<b>1</b>	<b>Python</b>	<b>3</b>
1.1	Fundamentos . . . . .	3
1.1.1	Scope y variables del entorno . . . . .	3
1.2	Tipos de datos, estructuras, objetos incorporados, funciones . . . . .	5
1.2.1	variables y su almacenamiento . . . . .	5
1.2.2	Funciones . . . . .	5
1.2.3	Tupla . . . . .	5
1.2.4	Lista . . . . .	6
1.2.5	Diccionario . . . . .	7
1.2.6	<code>open()</code> . . . . .	8
1.3	Errores y debuggeo de los mismos . . . . .	8
1.3.1	Tipos de testeo . . . . .	8
1.3.2	Errores . . . . .	9
1.3.3	Handlers . . . . .	10
1.3.4	<code>assertions</code> . . . . .	10
1.4	POO . . . . .	11
1.4.1	Métodos importantes . . . . .	11
1.5	Sobre algoritmos . . . . .	11
1.6	Sobre desarrollo de Software . . . . .	11
1.6.1	Requerimientos . . . . .	12
1.6.2	Modelo WRSPM . . . . .	12
1.6.3	Arquitectura de Software . . . . .	13
1.6.4	Diseño del software . . . . .	14
1.6.5	Modularidad . . . . .	15
1.6.6	Implementación . . . . .	17
1.6.7	Testeo del Software . . . . .	18
1.6.8	Modelos de desarrollo de software . . . . .	21
<b>2</b>	<b>Git and github</b>	<b>22</b>
2.1	Algunas extensiones de VSCode para Git . . . . .	22
2.1.1	Git History . . . . .	22
2.1.2	Git Blame . . . . .	22
2.1.3	Git Lens . . . . .	23
2.2	Fundamentos de Git: cómo funciona por dentro . . . . .	23
2.2.1	Crear un nuevo repositorio . . . . .	23
2.2.2	Objetos en Git . . . . .	23
2.2.3	JSON . . . . .	24
2.2.4	Hash . . . . .	25
2.2.5	Exploración de objetos de git mediante <code>cat</code> . . . . .	25
2.2.6	Creación de objetos mediante comandos de git . . . . .	25
2.2.7	Tree . . . . .	26
2.2.8	Permisos de objetos de git . . . . .	26
2.2.9	Creación de objetos Tree . . . . .	26
2.2.10	Tres pilares importantes . . . . .	27
2.2.11	Git checkout index . . . . .	27
2.3	Operaciones básicas de Git . . . . .	27

2.3.1	¿Qué es commit?	27
2.3.2	Creación de las primeras versiones	28
2.3.3	Comandos básicos de git	29
2.3.4	Historial de un archivo	30
2.4	Las ramas	30
2.4.1	HEAD	31
2.5	Repositorios remotos	32
2.5.1	git diff	32
2.6	Fusión o combinación de ramas	32
2.6.1	conflictos de fusión	33
2.7	Comandos para los repositorios remotos	34
2.7.1	Git push	34
2.7.2	Git fetch y pull	34
2.7.3	Ramas rastreadas	36
2.7.4	Proceso pull	36
2.7.5	Fetch head	37
2.7.6	Git pull con modificacion de repositorio remoto	37
2.7.7	Subida de cambios al repositorio remoto	38
2.7.8	De rama local a rama nueva remota	38
2.7.9	Actualización de estados de ramas	38
2.8	Pull Requests	39
2.8.1	Paso a paso del proceso de solicitudes de integración	39
<b>3</b>	<b>Datos</b>	<b>41</b>
3.1	Modelo de datos	41
3.1.1	Normalización de los datos	41
3.1.2	Expresiones de análisis de datos	41
3.1.3	Parámetros 'what if'	45
3.2	SQL	46
3.2.1	Conceptos	46
3.2.2	Pasos para crear una database y usarla	47
3.2.3	Tipos de datos	47
3.2.4	Variables de puntos flotantes y puntos fijos	48
3.2.5	Otros tipos de datos	48
3.2.6	Creación de tablas	48
3.2.7	Restricciones de SQL	49
3.2.8	Buenas prácticas	52
3.2.9	Manipulación de datos	52
3.2.10	Uniones	58
<b>4</b>	<b>Essentials for Cibersecurity</b>	<b>61</b>
4.0.1	Herramientas de gestión	61
<b>5</b>	<b>Docker</b>	<b>63</b>
5.1	Comandos de Docker	63
5.1.1	Mapeos de puertos	64
5.1.2	Persistencia de datos	65

# Chapter 1

# Python

## 1.1 Fundamentos

### 1.1.1 Scope y variables del entorno

El término "scope" en el contexto de la programación se puede traducir al español como "ámbito" o "alcance". El alcance o ámbito de una variable en Python se refiere a la región del programa donde esa variable es válida y puede ser accedida.

En Python, existen diferentes niveles de alcance, como el alcance global y el alcance local. El alcance global se refiere a las variables que están definidas fuera de cualquier función o clase y son accesibles desde cualquier parte del programa. El alcance local se refiere a las variables que están definidas dentro de una función o clase y solo son accesibles dentro de esa función o clase. El ejemplo más claro puede ser visto a continuación

```
enemies = 1

def increase_enemies():
    enemies = 2
    print("enemies is", enemies)

increase_enemies()
print("enemies is", enemies)
```

En este caso se llama a una función que cambia la variable "enemies" de 1 a 2. Sin embargo, la salida de este programa es

```
enemies is 2
enemies is 1
```

Podemos ver que fuera de la función, que fue donde se declaró la variable `enemies` esta no cambió de valor; solamente fue cambiada dentro de la función. Si por ejemplo tratáramos de escribir un programa que declare una variable solamente dentro de la función y la intentamos imprimir fuera de ella,

```
def drink():
    var = 1
    print(var)

print(var)
```

Saltará un error `NameError: name 'var' is not defined`. Esto es porque en este caso y en el anterior, las variables `enemies` y `var` son variables locales y están dentro del ámbito o alcance de la función que las declara o modifica.

Por su parte, cualquier variable declarada fuera de todas las funciones y clases, son denominadas como variables locales y el ámbito o alcance de estas será global. Y es accesible desde cualquier parte del programa. Este concepto de ámbito o alcance no solamente aplica para las variables sino que también aplica para funciones, entre otro tipo de elementos.

#### 1.1.1.1 Espacio de nombres

El concepto de espacio de nombres hace referencia a la manera que se tiene de organizar los diferentes nombres; sean de clases, funciones, variables, etc. Cada espacio de nombre puede ser visto como un contenedor con todos los nombres definidos. Existen diferentes tipos de namespaces:

1. Namespace Global: Es el namespace de nivel superior y contiene los nombres definidos en el alcance global, es decir, fuera de cualquier función o clase. Los nombres definidos en el namespace global son accesibles desde cualquier parte del programa.
2. Namespace Local: Es el namespace creado cuando se define una función o clase. Contiene los nombres definidos dentro de esa función o clase y solo son accesibles desde su interior.
3. Namespace de Módulo: Cada archivo de Python se considera un módulo y tiene su propio namespace. Los nombres definidos en un módulo son accesibles desde otros módulos si se realiza una importación.
4. Namespace Incorporado (Built-in): Contiene los nombres predefinidos que son proporcionados por Python de manera predeterminada. Estos nombres incluyen funciones y tipos incorporados como `print()`, `len()`, `str()`, etc.

Una característica importante de Python es que no tiene un ámbito de bloque, a diferencia de otros lenguajes de programación. Esto quiere decir, por ejemplo,

```
if var1:
    <code>
    <code>
    <code>
    <code>
```

Si el lenguaje tiene ámbito de bloque, entonces cualquier definición dentro de las líneas subordinadas al `if` anterior solamente existirán dentro del `if`. En python esto no pasa, cualquier variable que este definida aquí estará dentro del ámbito en que se encuentre el `if`.

#### 1.1.1.2 Cómo modificar una variable global

Una cosa importante dentro del tema de los entornos de que siempre que tenemos dos ámbitos diferentes; por ejemplo el entorno global y un entorno local, una variable global comparada con una variable local son dos variables completamente diferentes, aunque tengan el mismo nombre. Es muy mala idea nombrar dos variables con el mismo nombre, aunque estén en dos ámbitos diferentes. En el caso dado de que se requiera modificar el valor de una variable global dentro de una variable local, es imprescindible indicar explícitamente que se trata de una variable global:

```
var = 2
def fun():
    global var
    var = 1
    print(var)
fun()
print(var)
```

En este caso, no se crea una nueva variable en el entorno local de la función, sino que se modifica la variable global declarada al principio.

En general no es muy recuente el uso de este método de cambio de variables globales, porque se presta para confusiones y facilita de cierta manera los errores. Sin embargo, el uso de las variables globales es importante por ejemplo cuando se necesita declarar una variable constante dentro del programa. Siempre se usa como convención usar letras mayúsculas y barras bajas para declarar constantes dentro del programa (ejemplo `MY_EMAIL = "sebas@unal.edu"`).

## 1.2 Tipos de datos, estructuras, objetos incorporados, funciones

### 1.2.1 variables y su almacenamiento

al entrar en una funcion, normalmente el entorno se reinicia, con lo cual la mayoría de las variables que se declaren o se modifiquen solamente lo hará en ese entorno. Sin embargo, algunas funciones de algunos tipos de datos (mayoritariamente modificar) como `list.append()`

### 1.2.2 Funciones

Una manera de especificar mejor los argumentos de las funciones, es mediante la siguiente forma:

```
my_fun(a=1, b=2, c=3)
```

### 1.2.3 Tupla

De manera similar que una secuencia de caracteres, una tupla es una secuencia de datos de distintos tipos. Colección de distintos datos. Este no se puede modificar una vez se declara. Esto es, no son mutables y por tanto se almacenan en un solo bloque de memoria.

```
tupla = (2,"hola",False)
```

Si se suman las tuplas a y b, el resultado es una concatenación

```
tupla = (1,2,3)
tupla2 = (4,5,True)
tupla3 = tupla + tupla2
print(tupla3)
La salida es
(1,2,3,4,5,True)
```

Multiplicar una tupla:

```
a = (1,2)
print(a*5)
```

```
output: (1, 2, 1, 2, 1, 2, 1, 2, 1, 2)
```

Recorrer una tupla:

```
for i in tuple:
    print(i)
```

Verificar si es miembro

```
3 in (1,2,3)
```

La tuplas son útiles para hacer cambios de variables en una linea misma:

```
x = 2
y = 5
(x,y) = (y,x)
print(x)
output: 5
```

También para definir una función en la que retornan varios valores

```
def funcion(x,y):
    q = x // y
    r = x % y
```

```
    return (q,r)

(cociente,residuo) = funcion(47,11)
print(residuo)
```

output: 3

Para retornar el *enésimo* elemento de una tupla se pone `tuple[i]`. Recordar que los índices van desde cero hasta `length(tuple)-1`.

**Nota:** `tuple[1:5]` retornará los valores de la tupla desde el índice **2** hasta el **4**.

- `len(tuple)`: Retorna el tamaño de la tupla
- `max(tuple)`: Retorna el valor máximo de la tupla. Si en la tupla hay cadenas de caracteres, tuplas o listas, retorna un error.
- `tuple(List)`: Retorna una tupla conformada con los elementos de la lista `List`.

### 1.2.4 Lista

Otro tipo de arreglo de datos es la lista. La principal diferencia entre tupla y lista es que la lista sí es mutable y también ocupa dos bloques de memoria; esto hace que trabajar con tuplas sea más rápido pero la ventaja de la lista es que es modificable.

```
list1 = ['physics', 'chemistry', 1997, 2000]
```

El tipo de acceso de los elementos de una lista es el mismo que para las tuplas. De la misma forma las operaciones; ver la sección anterior en las operaciones de listas.

#### 1.2.4.1 Lista de funciones

`len(list)` Retorna el tamaño de la lista.

`max(list)` Retorna el valor del máximo, si la lista contiene combinaciones de numeros y caracteres o listas o tuplas, genera error.

`min(list)` Retorna el valor mínimo dentro de la lista.

`list(seq)` Retorna una lista compuesta por los elementos de `seq`.

`sorted(list)` Retorna una lista con los elementos de `list` ordenados de menor a mayor.

#### 1.2.4.2 Lista de métodos

`list.append(obj)` Añade el objeto `obj` al final de la lista

`list.count(obj)` Retorna el número de veces que el objeto `obj` ocurre en la lista.

`list.extend(seq)` Añade el contenido de `seq` a la lista. Lo añade al final de la lista.

`list.index(obj)` Retorna el índice más pequeño en la lista en que el objeto `obj` aparece.

`list.insert(index, obj)` Inserta el objeto `obj` en la casilla `index` de la lista, moviendo el resto una posición.

`list.pop(obj = list[-1])` Remueve y retorna el objeto que se encuentre en la posición `obj`. Por defecto si no se ingresa argumento, remueve el último objeto de la lista.

`list.remove(obj)` Remueve el primer objeto `obj` que encuentre en la lista.

`list.reverse()` Invierte el orden de los componentes de la lista.

`list.sort(key=None)` Organiza los elementos de la lista, si hay una directiva de ordenamiento, se puede ingresar como el argumento `key`, por defecto, organiza de menor a mayor.

`list.clear()` Elimina todos los componentes dentro de la lista dejándola como una lista vacía.

`list.copy()` Retorna una copia de la lista.

**Nota** Si se declara una lista como sigue

```
A = [1,2,3]
B = A
```

Tanto A como B están apuntando al mismo objeto, o dirección de memoria. Para realizar una copia de A en otro espacio de memoria se utiliza

```
B = A[:]
```

### 1.2.5 Diccionario

Es una lista especial en la que se puede dar un identificador especial al índice de la misma

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
print(dict["Name"])
```

El primer objeto del diccionario tiene un identificador llamado "Name" y un valor asociado a él que en este caso es "Zara". La salida del código anterior será

```
Zara
```

Se puede modificar el valor de una entrada en el diccionario y se puede también añadir una nueva entrada

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
dict['Age'] = 8; # update existing entry
dict['School'] = "DPS School" # Add new entry
```

Para borrar elementos de un diccionario

```
del(dict["Name"])
```

Una propiedad importante de los diccionarios es que los elementos pueden ser cualquier objeto de python. Las 'llaves' o identificadores deben ser objetos inmutables como cadenas de caracteres o tuplas.

#### 1.2.5.1 Funciones

`len(dic)`

`str(dic)` Retorna una cadena de caracteres imprimible de los elementos que contiene el diccionario `dic`

#### 1.2.5.2 Métodos

`dic.clear()` Elimina todo el contenido del diccionario

`dic.copy()` Retorna una copia del diccionario, sirve para asignar a otra variable y copiar el diccionario.

**dic.fromkeys(iterable,value)** Retorna un nuevo diccionario cuyas 'llaves' estarán determinadas por los elementos de **iterable** y con un valor asociado (único) de **value**

**dic.get(key, default=None)** Retorna el valor correspondiente a la llave **key**. El segundo argumento es el retorno cuando no hay una llave que corresponda.

**dic.items()** Este método retorna una lista de tuplas, cada tupla corresponde a la llave y al valor correspondiente.

**dic.keys()** Retorna una vista de todas las llaves del diccionario. Se puede conseguir la lista nativa a través de **list()**.

**dic.setdefault(key, default = None)** Es similar a **get()** con la diferencia de que si la llave **key** no está en el diccionario, entonces la añade y su valor correspondiente será **default**

**dic.update(dict2)** Actualiza el diccionario añadiendo todos los pares (llave-valor) de **dict2**.

**dic.values()** Retorna una vista de los valores del diccionario. Se puede conseguir la lista nativa a través de **list()**.

## 1.2.6 open()

El retorno de esta función es un objeto **file**.

**open (file, mode='r', buffering=- 1, encoding=None, errors=None, newline=None, closefd=True, open**

- **file** es un objeto **path-like**, es el nombre del archivo a abrir (incluyendo la ruta si es necesario) o crear.
- **mode** es un string opcional que especifica el modo en el que el archivo se abre, por defecto **'r'** para leer **'w'** para escribir, truncando el archivo primero **'x'** para creación de archivo nuevo, falla si ya hay un archivo con ese nombre, **'a'** para escribir en el archivo, anexando al final del archivo si este existe, **'b'** modo binario, **'t'** es modo de texto que está por defecto, **'+'** para abrir y actualizar (leer y escribir)

Los archivos abiertos en el modo binario retornan el contenido como objetos byte sin ninguna decodificación; en el modo texto, el contenido se lee como string.

### 1.2.6.1 Concatenación especial de cadenas de caracteres.

una forma interesante de hacer concatenación de caracteres es mediante la siguiente forma. Sea **score=0** una variable del tipo **int**. Si hacemos **print(f"your score is score")** se hará la conversión de entero a caracter automáticamente sin la necesidad de hacer **print("your score is" + str(score))**

## 1.3 Errores y debuggeo de los mismos

### 1.3.1 Tipos de testeo

#### 1.3.1.1 Test unitario

Si el programa es modular, es posible hacer tests que aseguren que cada función hace lo que se supone que debe hacer según las especificaciones.

#### Test de regresión

Cada vez que se soluciona un error, se realiza testeo nuevamente del código, con el objetivo de asegurarse que al realizar la corrección no se agregaron nuevos errores.



## Test de integración

Realizar testeo del programa como un todo. Se ponen juntas cada una de las partes individuales

### 1.3.1.2 back box testing

Se tiene el código y se realizan las pruebas con diferentes casos con el fin de encontrar todas las rutas posibles que hay en el código.

Se determina el docstring de una función, ejemplo:

```
def sqrt(x,eps)
    """Asume x y eps como flotantes, mayores que cero o igual para x, y retorna un res tal que x
```

La idea es entonces realizar testeos de diferentes casos dadas las especificaciones del docstring.

En el caso del ejemplo anterior, se puede hacer un conjunto de pruebas con valores como raíces cuadradas perfectas, números irracionales, menores que 1, o por ejemplo con valores extremos como muy pequeño y muy grandes de ambos `x` y `eps`.

### 1.3.1.3 glass box testing

En este caso lo que se hace es utilizar directamente el código para guiar los casos de prueba. En este caso se pueden llegar a presentar muchas posibilidades de caminos disponibles, teniendo en cuenta la posible presencia de bucles y repeticiones en el código. Por ejemplo, para ramas en los que hay diferentes casos, es importante lograr hacer la prueba para todos y cada uno de los posibles caminos o casos. Para bucles `for`, se deben preparar pruebas en las que no se entra a dicho bucle, también pruebas en las que se entra una vez, dos veces, tres y así sucesivamente. Para bucles `while` es de manera similar, pero asegurándose de tener casos de prueba que puedan cubrir todas las formas posibles de romper el bucle.

Hacer el debugging tiene una variedad grande de posibilidades. Utilizar `print`, por ejemplo, dentro de funciones o bucles.

## 1.3.2 Errores

### 1.3.2.1 IndexError

```
test[1,2,3]
test[4]
```

### 1.3.2.2 TypeError

```
int(test)
```

### 1.3.2.3 NameError

cuando no se encuentra un nombre ya sea local o global.  
a una variable inexistente.

### 1.3.2.4 SyntaxError

Errores de sintaxis. Cuando python no puede interpretar o analizar gramaticalmente el código.

### 1.3.2.5 AttributeError

las referencias a atributos falla.

### 1.3.2.6 ValueError

el tipo de operador está correcto, pero el valor del mismo es imposible.

#### 1.3.2.7 IOError

El sistema IO reporta una malfunción (por ejemplo un archivo no encontrado). Los errores son llamados excepciones.

ahora por alguna razón esto no deja seguir y continuar

### 1.3.3 Handlers

Los llamados manejadores, son lo que se encargan de llevar a cabo la rutina o ejecución necesaria cuando determinada cosa ocurre, sean interrupciones o excepciones.

```
try:
    xxxxxxxx
    xxxxxxxx
except (exception_type1):
    xxxxxxxx
    xxxxxx
except (exception_type2):
    xxxxxxxx
    xxxxxx
    .
    .
    .

else:
```

lo anterior se ejecuta cuando el cuerpo del `try` asociado se ejecuta sin ninguna excepción.

```
finally:
```

Siempre se ejecuta después del `try`, `else`, y `except`, incluso cuando existen `break`, `continue` o `return`.

```
raise <ExceptionName> (<Arguments>)
raise <ValueError> ("Uis, algo esta mal")
```

### 1.3.4 assertions

programación "defensiva".

```
assert <lo_que_se_espera>, <mensaje>
```

Da un error de ejecución del tipo `AssertionError`. dando la explicación pertinente.

Ahora hay errores lógicos que son más complicados de tratar.

Cosas que no se deben hacer:

1. Escribir el código entero para hacer pruebas sobre él
2. Hacer debug en el programa entero
3. Olvidar en qué lugar estaba el bug
4. Olvidar cuáles fueron los cambios que se hicieron

Por el contrario es más recomendable escribir una función, probarla, hacer depuración, y así con cada función nueva que se escriba. Hacer Testeo de integración. También hacer copias de seguridad del código, cambiarlo, advertir mediante comentarios los cambios, y al realizar pruebas, hacer comparaciones.

## 1.4 POO

`object` es el tipo más básico en python.

```
class <name>(<parent_class>):  
  
class coordinate(object):  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y
```

El `self` es un parámetro para referirse a la instancia de la clase, la que esté ejecutándose. El constructor siempre será `def __init__()`:

### 1.4.1 Métodos importantes

Estos métodos sustituyen operadores o funciones importantes.

- `__str__()`: Su retorno es lo que se muestra cuando se ejecuta la función `print()`
- `__add__()`: Su retorno es el valor de `'a+b'`.
- `__sub__()`: Su retorno es el valor de `'a-b'`.
- `__mul__()`: Su retorno es el valor de `'a*b'`.
- `__eq__()`: Su retorno es el valor de `'a==b'`.
- `__lt__()`: Su retorno es el valor de `'a<b'`.
- `__len__()`: Su retorno es el valor de `'len(a)'`.

**nota** las **variables de clase** son variables cuyo valor se comparte entre todas las instancias de la clase

## 1.5 Sobre algoritmos

¿Cómo se puede establecer o sabe qué tan eficiente es mi algoritmo?

Los razonamientos que surgen a partir de este punto dan como resultado un análisis interesante sobre las siguientes cuestiones:

- Cómo podemos razonar sobre un algoritmo con el objetivo de predecir la cantidad de tiempo que este necesitará para resolver un problema de un tamaño en particular.
- Cómo podemos relacionar las opciones en el diseño de algoritmos con la eficiencia en tiempo del resultado.

## 1.6 Sobre desarrollo de Software

Cuando se requiere crear un Software, similar a como se planea la construcción de una casa, el primer paso siempre es tener un esquema que qué es exactamente lo que se quiere construir.Cuál es el objetivo principal que dicho Software quiere cumplir. Después de este paso, viene la fase de diseño; está compuesta por los desarrolladores y los arquitectos. Se determina cómo se trabajará, los lineamientos, etc. Una vez el diseño está finalizado, viene la parte del código; la implementación de aquello que se quiere implementar. Cada sub equipo va a realizar pruebas y tests de cada componente. Una vez todos los componentes están listos, es el momento de realizar la unión de dichos componentes, y se realizan las pruebas de integración, pruebas de funcionalidad. Cuando todo está listo, se viene la fase de producción, operación y mantenimiento. Esto implica que el usuario empezará a utilizar el Software, y es cuando pueden venir requerimientos como cambios, mejoras, etc. Las fases se pueden resumir como sigue

- Requerimientos
- Diseño.
- Implementación.
- Verificación.
- Operación y mantenimiento.

En muchas ocasiones puede presentarse situaciones en las que los desarrolladores o los arquitectos pueden malinterpretar los requerimientos del usuario, este malentendido puede extenderse a las fases posteriores. Esto hace importante tener muy en cuenta diferentes modelos que permitan disminuir al máximo este tipo de situaciones. Un concepto importante de algo que parece ser más que un modelo es el llamado 'Agile'.

Se trata de un enfoque de desarrollo que se enfatiza en la flexibilidad, la colaboración y el desarrollo iterativo. Este desarrollo implica la división del proyecto en varias iteraciones que típicamente tienen una duración de una a cuatro semanas. Cada iteración se concentra en entregar un incremento en la producción del trabajo. Esto permite una alimentación más frecuente y la habilidad de adaptarse y realizar cambios en caso de ser necesario. Agile también promueve la colaboración cercana entre los desarrolladores y el cliente. El cliente está más involucrado en el proceso de desarrollo, proveyendo retroalimentación y clarificando los requerimientos. También reconoce que los requerimientos y las prioridades pueden cambiar a lo largo del tiempo. En lugar de tratar de predecir y planear cada detalle, la idea es abrazar los cambios y ajustar los planes a medida de las nuevas informaciones. Esto permite una gran flexibilidad, y la habilidad de responder a las necesidades cambiantes de los clientes. Los miembros de los equipos colaboran de cerca, comparten responsabilidades y toman decisiones de manera colectiva. Siempre se enfocan en el mejoramiento continuo.

### 1.6.1 Requerimientos

El hecho de que un Software sea un objeto intangible, hace bastante más complicado comunicar ideas de manera exacta, sobre verdaderamente qué es lo que se pretende y delimitar los requerimientos de un software. Un requerimiento tiene esencialmente dos definiciones. El primero es definido como un proceso, en el cual se elabora una idea compartida sobre el problema que existe y eventualmente la potencial solución al mismo. Se construye un conjunto de descripciones de alto nivel de cada parte que compone el problema. El principal objetivo es elaborar un documento que pueda describir detalladamente qué es lo que el sistema deberá hacer y qué es lo que el sistema no deberá hacer. Es importante tener en cuenta más el 'qué' que el 'cómo', se quiere determinar el comportamiento que tendrá la solución sin tomar decisiones prematuras que puedan afectar la habilidad de diseñar la solución. Este diseño no se realiza todavía en este paso. El segundo concepto de especificación de requerimiento, es el producto de este proceso; la documentación que sale como producto de este proceso.

La especificación de los requerimientos son muy importantes debido a dos aspectos particulares: Por la parte de ingeniería, la importancia recae en el hecho de que así se evita cometer diversos errores que pueden desencadenar en pérdidas de tiempo. Está demostrado que un mayor porcentaje de tiempo invertido en el proceso de especificación de requerimientos da como resultado un mejor porcentaje de costos por imprevistos.

### 1.6.2 Modelo WRSPM

El modelo WRSPM, también conocido como el modelo mundo-máquina, es un modelo que permite esquematizar y entender los requerimientos del usuario y determinar las especificaciones necesarias de Software para resolver el problema. El modelo consiste en cinco elementos: W (world), R (requirements), S (specifications), P (program), y M (machine). Las suposiciones del mundo son aquellas cosas que ya están dadas por sentadas dentro del universo del problema. Los requerimientos son los objetivos del usuario, aquello que el usuario quiere lograr. Las especificaciones 'S' definen cómo el sistema va a cumplir esos requerimientos. El programa ya está inmerso en el conjunto del sistema, y es, de hecho, el código o conjuntos de códigos escritos por los programadores; el programa que cumplirá las especificaciones. Finalmente M es la máquina; el conjunto de hardware que compone la

solución.

En este sistema tenemos cuatro variables interesantes:  $e_h$ ,  $e_v$ ,  $s_v$ , y  $s_h$ .  $e_h$  son los elementos del entorno que están ocultos al sistema, fuera del sistema, pero aún nos preocupa. Un ejemplo puede ser la tarjeta de crédito que el usuario necesita para poder retirar del cajero. El conjunto  $e_v$  es el de las partes visibles para el sistema en el entorno. En nuestro ejemplo, son los datos generados al leer la cinta magnética de la tarjeta de crédito, y el número PIN introducido. En esencia, cualquier dato que puede ser leído o introducido en el sistema, pues este lo puede leer y es visible. Los  $s_v$  son los elementos del sistema que están visibles en el entorno. Esto puede comprender los botones del cajero, la información en pantalla, etc. Finalmente, los  $s_h$  son los elementos del sistema que no son visibles para usuarios; que están escondidos internamente en el código, en el hardware, y demás.

### 1.6.2.1 Ejemplo WRSPM

Un ejemplo para ilustrar los elementos que componen el modelo. Sea un monitor de paciente, capaz de leer signos vitales como frecuencia cardíaca, pulso, presión arterial, etc. El deseo u objetivo del monitor, es tener un sistema de alerta que notifique a la enfermera si el corazón del paciente se detiene. Esto da como requerimiento real el siguiente: si el corazón del paciente se detiene, se debe avisar a la enfermera. Eso se traduce dentro del sistema de la siguiente manera: si el sonido de un sensor cae por debajo de un umbral establecido, se activará una alarma. Un elemento clave es analizar una de las posibilidades de la parte 'W' del entorno. Se da por sentado que si la alarma suena, siempre habrá una enfermera que escuche y entienda que el corazón se detuvo. Este y muchos otros elementos que se asumen del entorno (y que por tanto no hacen parte del sistema) deben ser cuidadosamente estudiados y tenidos en cuenta dentro de la elaboración de los requerimientos y posteriores especificaciones.

## 1.6.3 Arquitectura de Software

Tal como sugiere el concepto de arquitectura en construcciones de edificios, un arquitecto es una clase de interfaz entre el cliente, lo que quiere; y el contratista, el implementador, la persona que construye. De manera similar, también es importante mencionar que el tipo de arquitecto debe cambiar y es diferente en función del tipo de producto que se desea diseñar. Como ejemplo en construcciones, el arquitecto de un rascacielos es muy diferente a un arquitecto de una represa, o el de un reactor nuclear. Una definición de arquitectura de software sería la siguiente: Se define como la estructura de cada componente del software y la manera en la que estos componentes se relacionan entre sí para formar el software como tal. Un elemento clave de este concepto cae en el elemento de particionar el software en partes más pequeñas, independientes, funcionales y con un valor empresarial; de modo que puedan ser fácilmente integrados entre sí para conformar el sistema completo.

### 1.6.3.1 Modelos de arquitectura

Algunos modelos arquitectónicos como los siguientes son bastante usados e implementados en la industria

- pipe and filter: Este modelo de manera secuencial lo que se podría denominar filtros (o transformación) de datos que se conectan a otros filtros mediante los conectores (pipes).
- client-server: Este modelo puede ser fácilmente ejemplificado mediante sistemas basados en internet, como servicios web, sistemas bancarios en internet, etc.
- layers: Es una forma de separar la estructura en diferentes capas independientes entre sí, pero que se correlacionan de manera que el sistema funciona correctamente. Cada capa puede ser modificada sin afectar el funcionamiento del resto de las capas.
- blackboard: Este modelo es un estilo arquitectónico para datos compartidos. Se trata de un sistema o módulo central (puede ser un programa compartido, o una fuente común de información) y un conjunto de componentes que precisan de este módulo central para operar, ya sea mediante la consulta de datos, o mediante procesamiento compartido.

### 1.6.3.2 Proceso en la arquitectura

El proceso de diseño de una arquitectura se puede desglosar a tres preocupaciones principales:

- Estructura del sistema. Se refiere a cómo el sistema se descompone en estos varios subsistemas principales, y cómo estos se comunican entre sí.
- Modelamiento de control. Es la forma en la que la arquitectura realiza un modelo de las relaciones de control entre las diferentes partes del sistema
- Descomposición modular. Es la forma en que se identifican las particiones de los subsistemas

Otra cosa importante a nivel arquitectónico es cómo podemos evaluar la calidad de un software. Los siguientes son algunos atributos de calidad que se tienen en cuenta para el Software:

1. Desempeño
2. Confiabilidad
3. Capacidad de testeo
4. seguridad
5. Usabilidad

### 1.6.4 Diseño del software

Recordando el modelo de las etapas de diseño de un software:

1. Requerimientos
2. especificaciones
3. Arquitectura
4. Diseño
5. Implementación

Caemos ahora en la cuarta etapa. Esta etapa se encuentra entre las decisiones a nivel empresarial y el esfuerzo del desarrollo. El diseño del Software lo definimos nuevamente de dos maneras: el proceso en sí de transformación del problema en una solución, en nuestro caso es transformar la especificación de requisitos en una descripción detallada de software que está listo para codificar; y el producto o sustantivo, que indica la descripción documentada de esta solución y las restricciones y explicaciones utilizadas para llegar a ella.

El primer paso es entender bien los problemas que surgen respecto al diseño. Esta información debe provenir de la documentación de especificaciones y de los requisitos. Un acrónimo muy común dentro de la tecnología es 'TMTOWDI'; esto implica que se deben identificar más de una solución. Luego, está la parte de descripción de la abstracción de la solución, que comprende la utilización de gráficos que incluyan maquetas o estructuras alámbricas, descripciones formales como el lenguaje de modelado unificado o diagramas UML, como diagramas de clases y diagramas de secuencia, y otras anotaciones descriptivas. Todo este proceso debe repetirse para cada una de las abstracciones, subsistemas, componentes, etc. hasta que todo el diseño esté expresado en términos primitivos. Una manera de evaluar esta etapa es asegurarse de poder entregar estas descripciones de diseño a un equipo de desarrollo desconocido y que este pueda dar con la solución completa. Aquí cosas como lenguajes de programación no deben estar concretadas, pues hace parte de la etapa de desarrollo y no de diseño.

En arquitectura y diseño se siguen las siguientes etapas

1. System Arch
2. Component Spec
3. Component Interface Spec

4. Component Design
5. Data Structure Design
6. Algorithm Design

Las primeras tres conciernen al apartado de arquitectura, mientras que las pultimas tres son de la parte de diseño. La parte arquitectónica comprende de separar todo el sistema en diferentes componentes y expecificar cómo es la interacción entre los componentes mediante las interfaces. En la parte de diseño, cada componente está diseñado de forma aislada, y luego cualquier estructura de datos que sea intrínsecamente compleja, importante, o compartida por varias clases o componentes, debe ser diseñada para que sea eficiente. Lo mismo ocurre para los algoritmos; si se trata de uno complejo o importante, se diseñará con pseudocódigo para garantizar que el algoritmo se construye correctamente.

El diseño del software toma los requerimientos abstractos y crea los detalles listos para ser desarrollados. Se deciden cosas como las clases, los métodos, los tipos de datos que se usarán en la solución, pero no las optimizaciones específicas de lenguaje, porque esto es parte del desarrollo. Dará detalles que están listos para implementación, pero no incluyen los detalles de la implementación. El diseño también consiste en crear entregables y documentación necesarios para que el equipo de desarrollo pueda crear algo que satisfaga las necesidades del usuario o del cliente.

### 1.6.5 Modularidad

Dentro del diseño del software es importante tener en cuenta que este tenga aspectos de modularidad. Con modularidad nos referimos principalmente a lo siguiente:

1. Acople
2. Cohesión
3. Ocultamiento de información
4. Encapsulación de datos

Los dos primeros son una medida de qué tan bien funcionan juntos los diferentes módulos y qué tan bien cumple un módulo particular una tarea bien definida. La ocultación de información determinar cómo podemos extraer información y conocimientos de manera que podamos cumplir o completar trabajos complejos en paralelo sin tener que conocer todos los detalles de la implementación relacionados con la forma en que finalmente se completará la tarea. Por su parte, la encapsulación de datos se refiere a la idea de que podemos incluir construcciones o conceptos dentro de un módulo, que nos permite entender o manipular el concepto con mucha más facilidad cuando lo analizamos de forma relativamente aislada. Dado que un software es un sistema muy complejo y difícil de evaluar como un todo, sobre todo porque no es un elemento tangible, es imprescindible que este goce de una buena modularidad de manera que la complejidad del sistema pueda ser dividido en partes más pequeñas. Para esto, los objetivos principales son los siguientes:

1. Descomponibilidad
2. Componibilidad
3. Facilidad de entendimiento

#### 1.6.5.1 Acoplamiento en el diseño

El acoplamiento se refiere a qué tan estrechamente un módulo está vinculado a otro en un sistema. Para mantener la modularidad y manejar la complejidad, es crucial mantener un bajo acoplamiento entre módulos. Esto significa que cuando se hacen cambios en los requisitos durante el desarrollo, estos no deberían afectar significativamente a otros módulos. El objetivo es que los cambios en el código se contengan dentro de un único módulo, minimizando su impacto en el resto del sistema.

Existen diferentes niveles de acoplamiento, que varían desde los más fuertes hasta los más débiles

**Acoplamiento de contenido y común** Ocurre cuando dos módulos dependen de la misma información subyacente. El acoplamiento de contenido se da cuando un módulo depende directamente de los datos de otro, mientras que el acoplamiento común ocurre cuando ambos módulos dependen de datos globales.

**Acoplamiento externo** Se refiere a la dependencia de un formato, protocolo o interfaz impuestos externamente. Aunque a veces es inevitable, es un acoplamiento fuerte que puede afectar a muchos módulos.

**Acoplamiento de control** Se presenta cuando un módulo controla el flujo lógico de otro mediante la transmisión de información sobre qué hacer o en qué orden hacerlo.

**Acoplamiento de estructura de datos** Sucede cuando dos módulos dependen de la misma estructura de datos compuesta. Si la estructura cambia, podría afectar negativamente a los módulos involucrados.

**Acoplamiento de datos** Es un acoplamiento más débil y ocurre cuando solo se comparten parámetros simples entre módulos.

**Acoplamiento de mensajes** Es el acoplamiento más débil, logrado principalmente a través de la descentralización del estado y la comunicación entre componentes mediante el paso de parámetros o mensajes.

En sistemas complejos, siempre habrá acoplamiento, pero lo importante es enfocar la atención en mantener bajo acoplamiento y alta cohesión entre los módulos que necesitan comunicarse, lo que resulta en mejores diseños y soluciones más robustas.

#### 1.6.5.2 Cohesión en el diseño

La cohesión se refiere a qué tan bien encaja todo dentro de un módulo y funciona junto para lograr el propósito del mismo. Definimos también varios niveles de cohesión. Hay niveles de cohesión débil, de cohesión media y de cohesión fuerte. Los siguientes son los niveles de cohesión débiles

**Cohesión coincidente** Es el nivel más débil e indica que los elementos del módulo se encuentran dentro del mismo archivo, no hay nada más que los una.

**Cohesión temporal** Significa que el código se activa al mismo tiempo, que son llamados en el mismo momento.

**Cohesión procedural** Muy similar al anterior también se basa en el tiempo y no es una cohesión muy fuerte, suele suceder cuando un procedimiento tenga lugar justo después de otro porque así lo indica el algoritmo, pero no tienen más relación ni cohesión que esa.

**Asociación lógica** Sucede cuando se agrupan los componentes que realizan funciones similares. Los siguientes son los niveles de cohesión media:

**Cohesión comunicacional** Sucede cuando los componentes funcionan con la misma entrada o producen la misma salida

**Cohesión secuencial** Esta se logra cuando se agrupan funciones o elementos que secuencialmente dependen unos de otros. Por ejemplo cuando una parte del componente es la entrada de otra parte. Por último, los siguientes son los niveles de cohesión más fuertes y mejores o deseados:

**Cohesión de objetos** Aquí se presenta que cada operación de un módulo se proporciona para permitir que los atributos del objeto sean modificados o inspeccionados. Aquí cada una de las partes del módulo está diseñada específicamente para un propósito dentro del propio objeto.



**Cohesión funcional** En este agrupamiento, cada uno de los elementos están diseñados y son necesarios para la ejecución de una única función o comportamiento bien definido.

## 1.6.6 Implementación

La implementación corresponde al trabajo de desarrollo del software. Desde la perspectiva del proceso, es esencial tener en cuenta que una persona tiene que tener un período de descanso suficiente y un periodo de trabajo adecuado sin excederse en las horas de trabajo. Por otro lado, desde la perspectiva del código, es imprescindible que este esté debidamente documentado. Por ejemplo, es mejor que los comentarios del código expliquen el porqué y que el código por sí sea el que explique el cómo.

Un consejo siempre útil es el siguiente: escribe tus comentarios, tests, y manejo de excepciones antes de escribir el código funcional. Ya tenemos el diseño completado, ya sabemos cuál es la solución en nuestra mente, es mejor escribirla en los comentarios para que el código que viene tenga sentido. Siempre es recomendado documentar cualquier problema que surga para no volver a repetirlos más adelante. La guía de estilo de Google para C++ recomienda que si una función tiene más de 40 líneas, es mejor pensar en desglosarla. Las funciones cortas y compactas encapsulan la complejidad y la aíslan mediante el uso de muchas funciones. Es preferible que el uso de métodos sea para tareas particulares y evitar que estos tengan efectos secundarios en otros métodos. Otro consejo es que si nos damos cuenta de que un código se utiliza dos veces, es mejor convertirlo en un método, pues es muy probable que este sea utilizado más veces más adelante.

### 1.6.6.1 Despliegue

El despliegue más que una etapa en sí, es un evento que se encuentra entre las etapas de prueba y mantenimiento. Un concepto importante es el llamado rollback o retroceso, y se define como el retroceso o inversión de acciones que han sido completadas durante el despliegue con la idea de revertir un sistema a algún estado anterior. Esto es algo que ocurre cuando la implementación no funciona según lo previsto, es importante tener un plan para poder revertir las acciones realizadas cuando las cosas no salen según lo esperado.

Hay tres estrategias de despliegue, conocidas como estrategias de cambio (cutover strategies), que se utilizan para asegurar un despliegue confiable de sistemas y actualizaciones. Estas estrategias son:

- Cold Backup
- Warm Standby
- Hot Failover

**Cold Backup** Es la estrategia más básica, en la que se tiene hardware separado listo, pero sin ningún software instalado ni datos replicados. En caso de una falla en el centro de datos principal, se activa el servidor de respaldo, se instalan las aplicaciones, se configuran y se transfieren los datos. Este proceso puede tomar alrededor de 24 horas, lo cual puede ser inaceptable para sistemas con alta frecuencia de transacciones.

**Warm Standby** En esta estrategia, se tiene una máquina lista y configurada, pero no está activa ni replicando datos en tiempo real. Se puede activar rápidamente, pero aún se requiere replicar datos y realizar configuraciones menores antes de que el sistema esté listo para producción. Esta estrategia puede llevar entre 2 y 4 horas, dependiendo del tiempo necesario para identificar y aprobar la activación del sistema de respaldo.

**Hot Failover** Es la estrategia más avanzada y rápida. El sistema de respaldo está completamente activo, con datos replicados constantemente en un retraso mínimo (por ejemplo, 5 minutos). Si el sistema principal falla, se puede redirigir el tráfico al sistema de respaldo en menos de 30 minutos. Esta estrategia asegura una mínima pérdida de datos y es probada regularmente para garantizar su efectividad.

Es importante mencionar la importancia de probar estas estrategias de cambio como parte de un plan de continuidad del negocio (BCP, por sus siglas en inglés), ya que la peor situación sería necesitar realizar un cambio y que la estrategia falle por falta de pruebas. Además, diferencia entre

la distribución de carga (load balancing) y el hot failover, destacando que el hot failover implica una verdadera separación geográfica sin que el sitio de respaldo reciba datos continuamente.

Estas estrategias ayudan a mitigar los riesgos asociados con fallas en los sistemas y aseguran que las interrupciones sean lo más cortas posible.

### **1.6.7 Testeo del Software**

El propósito principal del testeo dentro de la ingeniería de software es de encontrar errores dentro del software, ya sea en todo el programa o en partes de este.

Respondamos a la pregunta ¿Qué es una prueba? Iniciando con el software que se está probando, se debe tener en cuenta que entendemos por software, no al producto que se está desarrollando en su totalidad, sino a una parte o subconjunto del programa que haya sido completado, y que pueda ser ejecutado para probarlo. Se trata de algún módulo o unidad del código. Se habla entonces de pruebas unitarias. Cada unidad en este punto se entiende como un método, una rutina, función o procedimiento. Los datos de prueba son aquellos que insertamos a la unidad como entrada al realizar la prueba. Una vez se han introducido los datos de entrada, se debe hacer una evaluación del dato de salida, el comportamiento producto de ejecutar dicha unidad con los datos de entrada ingresados. Se debe verificar si dicha salida corresponde a un resultado correcto. Algo tiene que realizar este cuestionamiento y dar un veredicto de la correctitud de la salida. Este algo se llama oráculo. El oráculo tradicionalmente es el desarrollador o el tester, pero también pueden haber oráculos automatizados que mejoran el rendimiento y la calidad del veredicto. Estos deben evaluar con base en resultados conocidos, esperados, determinados o recuperados. En conjunto, los datos de entrada con los de salida conforman lo que llamamos casos de prueba.

#### **1.6.7.1 EL bug**

¿Qué es un bug? Un bug es cualquier error o falla en la unidad. En el sistema ocurre una falla cuando el resultado o retorno entregado se desvía del retorno especificado. Significa que algo no ocurrió de la manera que debía ocurrir. La especificación es una descripción detallada y consensuada del retorno esperado. El error se produjo porque el sistema era erróneo. Un error es la parte del estado del sistema que puede provocar una falla. Un error latente es el que se encuentra escrito en el código y se materializa o hace efectivo cuando se activa (ejecuta). Cuando el error provoca ese desvío o cambio en el comportamiento esperado, es cuando se provoca el fallo. Error es la manifestación de una falta, y la falla es la manifestación del error. Como ejemplo, la falla es la equivocación del programador, como consecuencia, de esta falta, se produce un error latente en el código; finalmente, solo cuando se ejecuta este código el error se hace efectivo, y es cuando se produce la falla en la unidad. Otro ejemplo: un error de mantenimiento o redacción del manual del operador es una falta, puesto que lo pusieron de manera incorrecta en el manual. Como consecuencia, se produce el error en el manual correspondiente, instrucciones erróneas sobre cómo utilizar el software, que permanecerá latente hasta que alguien lea el manual e indique cómo ejecutar el código. En ese momento se produce la falla.

#### **1.6.7.2 Verificación**

Según la IEEE, la verificación es el proceso de evaluación de un sistema o componente para determinar si los productos satisfacen las condiciones impuestas. Con condiciones impuestas nos referimos a los documentos y especificaciones realizados en el diseño, por tanto la definición puede mejorarse como el testeo del programa en comparación con los documentos de diseño o especificaciones más cercanamente relacionados. Con esto hablamos específicamente a lo que anotamos. Una manera de definirlo interesante es la siguiente: un intento de encontrar errores mediante la ejecución del programa en un entorno simulado de prueba. En las palabras más concretas o completas, la verificación la confirmación de que el software se comporta conforme a sus especificaciones, respondiendo a la pregunta ¿Estamos construyendo la cosa correctamente?

#### **1.6.7.3 Validación**

La validación viene más de la mano con los requerimientos de sistema dados por el usuario, aquí se tiene en cuenta el requisito dicho en el idioma del usuario. Una forma interesante de definir la

verificación es como un intento de encontrar errores mediante la ejecución de un programa en un entorno real. La definición más completa es la siguiente: validación es la confirmación de que el software se comporta tal que el usuario puede estar satisfecho, asegurándose de que el sistema cumple con las necesidades del cliente, respondiendo la pregunta ¿Estamos construyendo la cosa correcta?

#### 1.6.7.4 Estrategias de testeo

Existen diferentes estrategias de pruebas, cada una tiene utilidades para encontrar diferentes tipos de errores.

**Prueba incremental** Consideremos dos módulos A y B, y tres casos de pruebas T1, T2 y T3. Se realizan las pruebas de A y B, una vez terminadas, no se desechan las pruebas sino que nos quedamos con ellas. Luego, si se añade un tercer módulo C, también se añadirá un caso de prueba T4 necesario para probar C de forma aislada, como prueba unitaria. Añadimos esta prueba a las pruebas de A y B, y entonces las ejecutamos todas. De esta manera se puede verificar y determinar si algo cambió en el código correcto anterior con base en el nuevo módulo añadido, así como comprobar que los módulos actuales siguen funcionando según lo previsto. Se siguen añadiendo módulos y pruebas, y volviendo a ejecutar todas las pruebas a medida que se avanza. La técnica de volver a ejecutar pruebas antiguas en un conjunto más grande se denomina prueba de regresión.

**Prueba top-down** Cuando se está desarrollando software utilizando un enfoque de arriba hacia abajo (top-down), se necesita crear algo que supla los elementos de niveles inferiores que aún no has creado. Estos elementos se llaman "Stubs".

Imaginemos que se está construyendo software de Nivel 1, pero este software depende de otros componentes que pertenecen al Nivel 2 y que aún no han sido desarrollados. Por ejemplo, se podría necesitar instanciar un objeto para realizar alguna tarea específica, pero si esos objetos todavía no existen, se necesita algo que permita continuar el desarrollo del Nivel 1 y verificar que el programa funcione correctamente.

Aquí es donde entran los "Stubs". Un "Stub" es un fragmento de código muy simple, a menudo de una sola línea o unas pocas líneas, que cuando es llamado, devuelve un valor fijo (hard coded). Este valor simula lo que sería un valor de retorno real cuando el software de Nivel 2 esté completamente desarrollado.

Además de los "Stubs", existe algo similar llamado "Mock". Mientras que un "Stub" devuelve un valor predefinido, un "Mock" no devuelve necesariamente un valor, sino que se utiliza para verificar si un método ha sido llamado correctamente.

A medida que se continúan desarrollando los niveles inferiores del software, se podría tener que crear "Stubs" para el Nivel 3 y así sucesivamente. De esta manera, se avanza en el desarrollo, creando el software de cada nivel y usando "Stubs" para los niveles aún no implementados. Esto permite que el proceso de desarrollo continúe sin interrupciones, asegurando que cada nivel funcione correctamente antes de que los niveles inferiores estén completamente desarrollados.

**Prueba down-top** En el enfoque de desarrollo de software de abajo hacia arriba (bottom-up), el proceso es inverso al desarrollo top-down. Aquí, primero se desarrollan e implementan los componentes más básicos o de nivel inferior antes de construir las capas superiores que los integran. El Nivel 3 representa los componentes de nivel más bajo en la jerarquía del software. Estos son elementos fundamentales o funciones básicas que ya han sido completamente implementados. Sin embargo, en este punto, no existe una capa superior (Nivel 2 o Nivel 1) que coordine o integre estos elementos en un sistema funcional completo. Para probar y utilizar estos componentes de Nivel 3 antes de que el software de los niveles superiores esté construido, se emplean Drivers. Un "Driver" es un fragmento de código que simula el comportamiento de los niveles superiores. Su función es ejecutar llamadas a los componentes de Nivel 3 para asegurarse de que operan correctamente. Los Drivers esencialmente "conducen" el comportamiento del software de nivel inferior (Nivel 3) como si formaran parte de un sistema completo.

Una de las principales dificultades al construir Drivers es que, sin tener los componentes de Nivel 2 (y posiblemente de Nivel 1) completamente desarrollados, puede ser complicado determinar qué entradas o secuencias de operaciones serán necesarias para utilizar correctamente los elementos de Nivel 3. Es posible que no se conozca con precisión el orden de las operaciones o los tipos de datos que se necesitarán. A menudo, estos Drivers también están hard coded (con código fijo), y se basan

en suposiciones sobre cuáles serán los casos de uso más comunes o importantes para garantizar que todas las operaciones de Nivel 3 estén completas.

Una vez que los Drivers han garantizado que los componentes de Nivel 3 funcionan correctamente, se procede a desarrollar el software de Nivel 2. Estos son los componentes que empiezan a integrar y coordinar las funcionalidades de Nivel 3. Finalmente, se continúa con el desarrollo hacia arriba, pudiendo crear Drivers de Nivel 1 que coordinen el software de Nivel 2, y así sucesivamente, hasta que todo el sistema esté completo.

En el enfoque bottom-up, los Drivers son esenciales para verificar que los componentes de los niveles inferiores funcionan correctamente antes de que las capas superiores del software estén construidas. Este enfoque permite asegurar que cada nivel inferior esté bien implementado y operativo antes de integrar los componentes en un sistema más complejo.

**Prueba back to back** Es una técnica utilizada en el desarrollo de software para comparar la salida de dos versiones de un programa, generalmente una versión anterior y una nueva, para verificar que el comportamiento del software sigue siendo correcto después de modificaciones.

El objetivo principal de Back to Back Testing es asegurar que las funcionalidades que funcionaban correctamente en una versión anterior sigan funcionando igual en la nueva versión. Esto es especialmente útil cuando no se dispone de pruebas automatizadas preexistentes y se quiere expandir el conjunto de datos de prueba sin tener que definir resultados esperados manualmente.

1. Uso de Iteraciones Anteriores: Se ejecutan los datos de prueba que se utilizaron en la versión anterior del programa, la cual se supone que funcionaba correctamente, tanto en la versión antigua como en la nueva.
2. Comparación de Salidas: Si la funcionalidad no ha sido modificada, se espera que las salidas de ambas versiones sean idénticas. Cualquier discrepancia sugiere un error o un cambio inesperado.
3. Verificación de Cambios: Para las partes del programa que han sido modificadas (por ejemplo, para corregir errores o agregar nuevas características), se comparan nuevamente las salidas de ambas versiones. Aquí, se espera que las salidas sean diferentes, reflejando el cambio realizado.

**Axiomas en las pruebas** A medida que el número de defectos detectados en una pieza de software incrementa, también incrementa la probabilidad de la existencia de más defectos no detectados. Otro axioma es que siempre se debe asignar al mejor programador para realizar pruebas. También es importante saber y tener en cuenta que las pruebas exhaustivas al cien por ciento no existen. No se pueden ejecutar todas las combinaciones posibles de entradas. Por lo anterior es importante proporcionar algún tipo de estrategia que ataque los aspectos más importantes o críticos de los programas mientras los probamos. Recordemos que las pruebas solo pueden encontrar errores y no probar la ausencia. No se puede saber si hay algún otro error remanente. No se podrá saber nunca si se encontró el último error. Por otro lado, siempre tomará mas tiempo del necesario para probar menos de lo que nos gustaría, en otras palabras, siempre se acabará el tiempo antes de que se nos acaben los casos de prueba.

#### 1.6.7.5 Algunas perspectivas

**Prueba de caja negra** Está diseñada sin un conocimiento de la estructura interna del programa o unidad. Está basado en requerimientos funcionales, solo se puede conocer entrada - salida.

**Prueba de caja blanca** Aquí sí se conoce y examina el diseño interno del programa, se requiere un conocimiento detallado de la estructura.

#### 1.6.7.6 Etapas de pruebas

Las siguientes son las diferentes etapas de pruebas

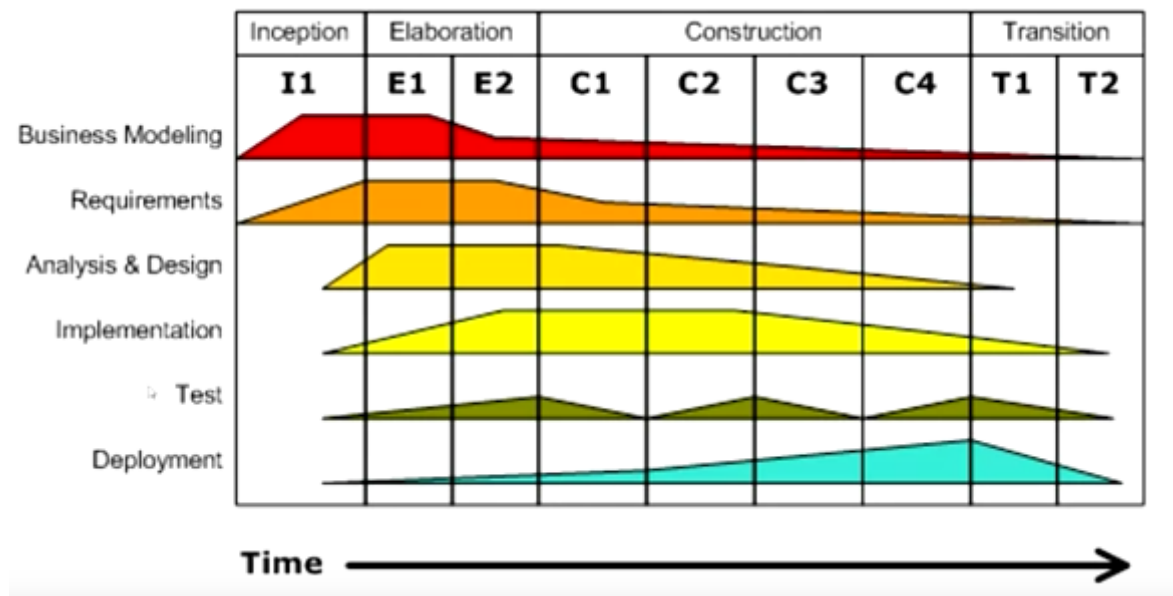
1. Prueba unitaria

2. Prueba de módulo: aquí tenemos la prueba de una colección de unidades que son dependientes entre sí y que forman un módulo.
3. Prueba de subsistemas: aquí tenemos una de las primeras pruebas de integración, porque también estamos probando que los componentes funcionen juntos. Normalmente aquí se integran probablemente trabajos de más de un equipo de desarrollo.
4. Prueba de sistema: aquí ya se hacen pruebas diferentes de todo el sistema, cosas como rendimiento, usabilidad, seguridad, etc.
5. Prueba de aceptación: aquí ya tenemos las pruebas realizadas por el usuario. Normalmente se les puede llamar pruebas alfa, beta.

## 1.6.8 Modelos de desarrollo de software

### 1.6.8.1 Modelos iterativos

Si nos fijamos en el modelo cascada, visto anteriormente, es probablemente uno de los modelos más usados y populares. Sin embargo, existen modelos diferentes que pueden tener ciertas ventajas. Uno de ellos es el modelo unificado.



Las áreas de colores indican el nivel de esfuerzo que según este modelo se debe poner en cada una de las fases. Podemos ver que hay cuatro fases: inicio, elaboración, construcción y transición. El inicio está dividido en una sola iteración, la elaboración está dividida en dos iteraciones, y así.

El inicio se centra principalmente en el modelado empresarial y los requisitos; por tanto es la fase más corta de todas. En esta fase se establece el modelo de negocio, también se define el alcance. Luego se hace un estudio de viabilidad para determinar si el proyecto es posible desde el punto de vista del mercado y de la capacidad de ejecución de la compañía. Se determina también qué elementos del proyecto serán construidos y cuáles serán comprados.

La siguiente es la fase de la elaboración; en esta fase se desarrollan y llevan a cabo muchas actividades en torno a los requisitos. El objetivo clave de esta fase es que se puedan determinar dos objetivos: abordar todos los riesgos conocidos; se hace mucho enfoque en lo que puede salir mal y en cómo puede resolverse. El segundo objetivo es validar la arquitectura del sistema; determinar cómo se va a construir el sistema. Todo esto deriva en una estimación muy creíble para la siguiente fase.

Luego viene la fase de construcción. Esta es la fase más grande de todo el proyecto. En esta fase se construye el software mediante múltiples iteraciones, y de cada iteración se produce un lanzamiento; esto quiere decir que se publica algo en cada iteración y se recibe feedback.

## Chapter 2

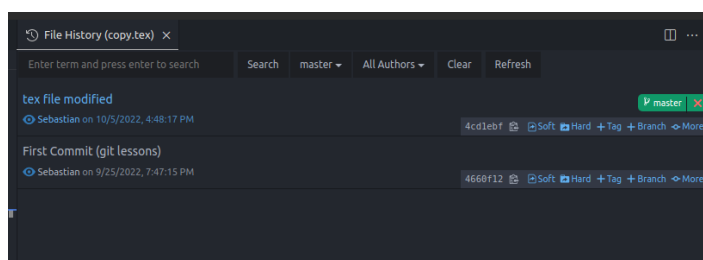
# Git and github

## 2.1 Algunas extensiones de VSCode para Git

### 2.1.1 Git History

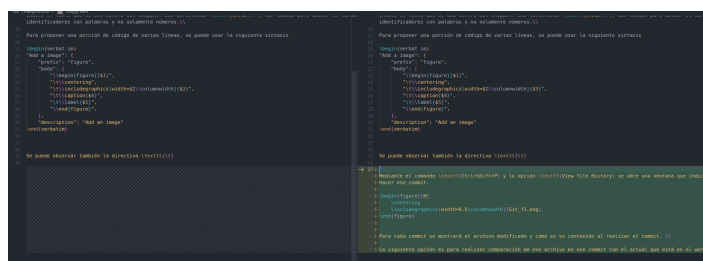
Es una extensión que sirve para visualizar los cambios históricos que se han hecho a los diferentes archivos.

Mediante el comando **Ctrl+Shift+P** y la opción **View file History** se abre una ventana que indica las versiones del proyecto (commits) y cómo han sido los archivos al momento de hacer la confirmación de esa versión.



Para cada versión se mostrará el archivo modificado y cómo es su contenido al realizar la confirmación.

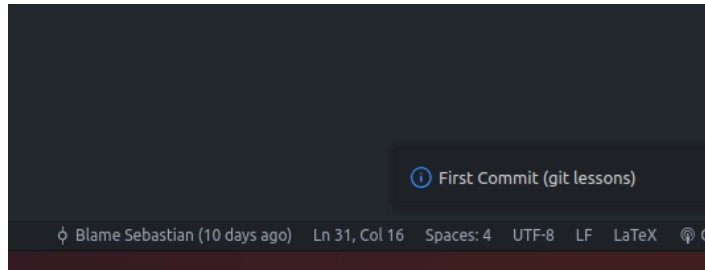
La siguiente opción es para realizar comparación de ese archivo en esa versión con la actual que está en el espacio de trabajo.



Las siguientes opciones permiten ver la comparación del archivo su versión anterior. Y la última opción permite ver la historia completa del archivo

### 2.1.2 Git Blame

Esta extensión sirve para realizar revisión línea por línea de por quién, hace cuánto y en qué versión se realizó esa línea de código



### 2.1.3 Git Lens

Esta extensión es similar a las anteriores, en que ayuda a verificar las identidades de las personas que están modificando archivos, muestra línea por línea información del autor, y versión de la línea en cuestión.

## 2.2 Fundamentos de Git: cómo funciona por dentro

### 2.2.1 Crear un nuevo repositorio

La sentencia primaria y básica para inicializar un nuevo repositorio es

```
git init
```

Este comando se realiza dentro de la carpeta en la que se desea realizar el repositorio. Una vez creado el repositorio, se pueden añadir las carpetas correspondientes dentro. Al crearse el repositorio, se crea una carpeta oculta llamada `.git` de manera automática.

dentro de esta carpeta tenemos diferentes carpetas y archivos. Uno de ellos es el archivo `config`, en este archivo tenemos una serie de cadenas que nos indica las configuraciones que tiene el repositorio. El siguiente es un ejemplo de un archivo de configuración:

```
[core]
  repositoryformatversion = 0
  filemode = false
  bare = false
  logallrefupdates = true
  ignorecase = true
[remote "origin"]
  url = https://github.com/JhoAraSan/Process.git
  fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"]
  remote = origin
  merge = refs/heads/master
```

Otro archivo es el de descripción `description`, el cual se puede editar para añadir una descripción al repositorio. Finalmente el archivo `HEAD`, el cual puede arrojar la siguiente cadena:

```
ref: refs/heads/master
```

Más adelante se dará la explicación correspondiente a este archivo.

Git tiene su propio sistema de archivos; en este sistema de archivos git guarda o almacena objetos, y estos objetos son guardados dentro de la carpeta correspondiente `objects`.

### 2.2.2 Objetos en Git

En git existen cuatro tipos de objetos:

```
Blob
Tree
Commit
Annotated Tag
```

Estos cuatro objetos son los suficientes para poder realizar todo el seguimiento de los archivos del repositorio.

#### 2.2.2.1 Blob

Este es el tipo de objeto en el que git guarda **archivos**. Todo tipo de archivos con la extensión que sea. Cualquier tipo de archivo serpa guardado como un blob.

#### 2.2.2.2 Tree

Este es el tipo de objeto en el que git almacena la información sobre los directorios. Dicho de otra fforma, un objeto Tree es una representación de una carpeta o un directorio.

#### 2.2.2.3 Commit

A través de este tipo de objeto, Git es capaz de almacenar diferentes versiones de uno o varios archivos a través del tiempo.

#### 2.2.2.4 Annotated Tag

Este objeto es esencialmente un texto que está apuntando a un commit versión específica.

Para poder gestionar o administrar objetos en git se usan los comandos de git de bajo nivel:

`git hash-object` Con esre comando podemos crear nuevos objetos con la estructura de git. `git cat-file` Con este comando se pueden leer los objetos git. `git mktree` con este comando se puede crear un nuevo objeto de tipo Tree

A manera de ejemplo, si colocamos un texto string cualquiera mediante el siguiente comando:

```
echo "Hello, Git" | git hash-object --stdin -w
```

Vamos a obtener como salida un hash, además de que se creará el objeto correspondiente en la carpeta `objects`; la carpeta será los dos primeros caracteres del hash, y dentro habrá un archivo con el resto de caracteres del hash. Solamente se crea el objeto, el repositorio seguirá estando vacío. Importante remarcar que el hash retornado es el hash del string que le metimos de entrada.

### 2.2.3 JSON

Las siglas significan "JavaScript Object Notation". Es un formato que permite el intercambio de datos entre diferentes servidores. Como ejemplo, podemos extraer datos mediante una API desde un servidor a una página web. La siguiente es un ejemplo de una estructura JSON:

```
{
  "id": "12345667",
  "name": "Mike",
  "age": 25,
  "city": "New York",
  "hobbies": ["Skate", "Running"]
}
```

Siempre será recomendado que las llaves en un archivo Json sean únicas. La estructura de datos que hay en git es muy similar a JSON; Git tambien almacena nombres "llave" y valores. Las "llaves" en git son los hashes de cada objeto. En git, el hash generado (el cual es equivalente a la key) es función o depende del valor.



## 2.2.4 Hash

Al realizar el comando de la seccion anterior, vimos que `ek string "Hello, Git"` generó un hash `b7aec520dec0a7516c18eb4c68b64ae1eb9b5a5e`. Esto significa que se aplicó una funcion hash al string o al dato ingresado.

Una función hash es una función que toma una entrada de cualquier tamaño (longitud) y tiene una salida de un tamaño fijo. Es importante también notar que el hash es una función unidireccional, es decir, que si tenemos un hash generado no vamos a poder saber cuál fue la entrada que la generó. Para la misma función hash, la misma entrada siempre va a generar la misma salida.

Las funciones o algoritmos para generar hashes más importantes son las siguientes:

- MD5 (128 bit)
- SHA1 (160 bit)
- SHA256 (256 bit)
- SHA384 (384 bit)
- SHA512 (512 bit)

El algoritmo usado por git para generar sus hashes es **SHA1**.

Cada caracter de 4 bits de longitud está en formato hexadecimal. Por tanto, un hash de git tiene una longitud de 40 caracteres hexadecimales.

Dado lo anterior, surge la pregunta de cuantos archivos diferentes podemos guardar en el mismo repositorio.

La cantidad total de hashes diferentes es  $16^{40} \approx 1.46 \cdot 10^{48}$ . Por otro lado podemos hacernos la pregunta de cual es la posibilidad de que dos archivos diferentes produzcan el mismo hash?

La probabilidad de encontrar un hash específico es  $\frac{1}{16^{40}} \approx 6.84 \cdot 10^{-49}$ . Por tanto para saber la probabilidad de que dos archivos produzcan el mismo hash, tenemos

$$P = \frac{1}{16^{40}} \frac{1}{16^{40}} = \frac{1}{16^{80}} \approx 4.68 \cdot 10^{-97}$$

Como elemento adicional, tenemos que la probabilidad de que teniendo  $n$  archivos diferentes, dos de ellos generen el mismo hash, es el siguiente:

$$P = \frac{(2^{160} - n)!(2^{160})^{n-1} - (2^{160} - 1)!}{(2^{160} - n)! 2^{160(n-1)}}$$

Para que haya una probabilidad de 1 de que haya una colision de hash es necesario que en un repositorio hayan más archivos que número diferente de hashes.

## 2.2.5 Exploración de objetos de git mediante cat

Recordemos que todo objeto de git tiene su correspondiente hash. Podemos usar el comando `git cat` para obtener información de cualquier objeto. Las opciones del comando son:

`git cat-file -p <hash>` Retorna el contenido del objeto.  
`git cat-file -t <hash>` Retorna el tamaño del objeto.  
`git cat-file -s <hash>` Retorna el tipo del objeto. El tamaño lo retorna en bytes.

## 2.2.6 Creación de objetos mediante comandos de git

el comando `git hash-object` se usa para crear nuevos objetos, luego tenemos varias opciones adicionales:

```
echo "Hello, Git!" | git hash-object --stdin -w
```

La primera opción es para tomar la entrada como entrada estándar, la segunda es importante porque es la que hace se cree el objeto. También podemos crear objetos en git basados en archivos locales:

```
git hash-object <filename> -w
```

Una cosa importante de notar es que en git cuando almacenamos archivos del tipo blob, estos objetos no tienen un nombre de archivo. Como se podrá ver en los anteriores comandos, ninguno de ellos retorna el nombre del archivo puesto que este no se almacena. Otra cosa importante de notar es que tanto el tamaño como el tipo de objeto se almacenan dentro del mismo hash. La estructura con la que lo hace es la siguiente:

contenido + tipo de objeto + tamaño del objeto = hash. Entre el tipo de objeto más tamaño, y el contenido del objeto hay un delimitador. En esencia, el hash se genera a partir de lo siguiente:

```
blob 11\0Hello
```

El delimitador es `\0`, antes del delimitador tenemos el tamaño que para el ejemplo es 11 bytes, y antes el tipo de objeto seguido de un espacio. Luego del delimitador está el contenido del archivo.

### 2.2.7 Tree

Este tipo de objeto es el que representa las direcciones y los directorios. Un objeto Tree puede tener tanto blobs como otros trees. La estructura es la misma de los demás objetos (tipo, tamaño, delimitador y contenido). En este caso el contenido será diferencial al de un blob:

```
100644 blob 57537e1d8fba7d80c5bcca8b04e49666b1c1790f .babelrc
100644 blob 602c57ffb51af99d6f3b54c0ee9587bb110fb990 .flow config
040000 tree 80655da8d80aaaf92ce5357e7828dc09adb00993 dist
100644 blob 06a8a51a6489fc2bc982c534c9518f289089f375 package.json
040000 tree fc01489d8afd08431c7245b4216ea9d01856c3b9 src
```

Como podemos ver un tree puede contener blobs y más trees, tenemos tres secciones importantes: el primer número representa los permisos, el segundo es el tipo de objeto, luego va el hash, y por último el nombre o directorio.

### 2.2.8 Permisos de objetos de git

El primer número representa los permisos de los objetos de Git, estos permisos se pueden ver en la siguiente lista.

```
040000 Directorio
100644 Archivo regular no ejecutable
100664 Archivo de escritura de grupo no ejecutable normal
100755 Archivo ejecutable regular
120000 Link simbólico
160000 Gitlink
```

La razón de que existan estos permisos es porque los repositorios de git deben ser independientes de cualquier sistema de archivos del SO en el que está.

### 2.2.9 Creación de objetos Tree

Teniendo un ejemplo de dos objetos blob, cada uno con su respectivo hash, podemos crear un archivo del tipo tree que nos proporcione apuntadores para cada uno de los blobs y que nos proporcione la información de los nombres de los archivos de los blobs. Si los dos archivos blobs tienen los siguientes hashes

```
284a47ff0d9b952bab8ccbae29b97b5beb700e82
814d2ecd90a29b25b12880623d82e727f9a650cb
```

Los cuales representan los archivos `file1.txt` y `file2.txt`; en este caso, el contenido del tree será el siguiente:

```
100644 blob 284a47ff0d9b952bab8ccbae29b97b5beb700e82 file1.txt
100644 blob 814d2ecd90a29b25b12880623d82e727f9a650cb file2.txt
```

Para crear un nuevo tree usamos el comando `git mktree`, primero creamos un objeto del tipo texto con el contenido de arriba y lo guardamos en cualquier carpeta.

### 2.2.10 Tres pilares importantes

Dentro de los repositorios tenemos y podemos identificar tres áreas fundamentales: directorio de trabajo (working directory), staging area o index, y git repository.

El staging area que también es llamado index es el área responsable de preparar los archivos para ser insertados en un repositorio limpio, y del mismo modo, prepara los archivos tomados del repositorio para ser puestos en el directorio del trabajo. El proceso en el que los archivos pasa por el area de staging es siempre obligatorio. Es el puente entre el directorio de trabajo y el repositorio de git. Si un objeto tree está representando el nombre de dos objetos blob, significa que esta representando un directorio raíz. Es decir que se hace necesario describir otro tree que represente una carpeta con su respectivo nombre en la que estén alojados los dos archivos blob.

`git ls-files -s` es un comando que sirve para listar los archivos que se encuentran en el staging area. Si queremos enviar cualquier objeto tree desde el repositorio de git hasta el area de staging, usamos el comando `git read-tree <hash>`.

### 2.2.11 Git checkout index

Teniendo los dos objetos blob creados a mano y el objeto tree también creado a mano con los nombres de los anteriores blobs, y también ateniendo estos dentro del staging area, se pueden añadir dentro del directorio de trabajo, que corresponde a la carpeta física (dentro del sistema de archivos de cada SO) en la que se ven los archivos. Esto se hace con el siguiente comando: `git checkout -index -a`. Con la opción `-a` decimos que agregue todos los archivos.

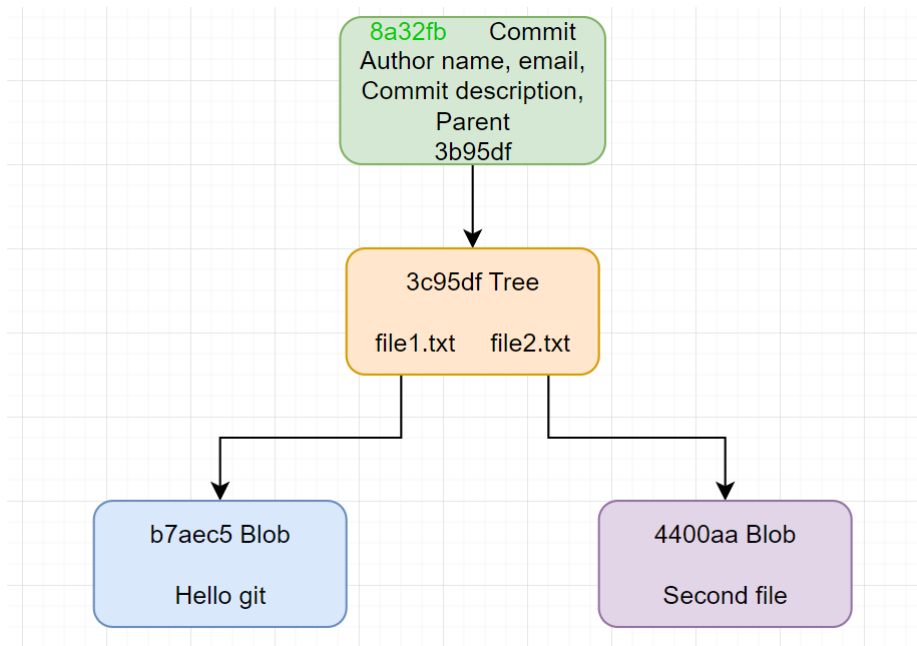
## 2.3 Operaciones básicas de Git

### 2.3.1 ¿Qué es commit?

Uno de los cuatro tipos de objetos principales es el denominado "commit". Lo primero de todo es que `commit` tiene la misma estructura que los otros tipos de objetos; es decir, que contiene la estructura de los objetos en git: un hash del tipo sha1 que consiste es tipo de objeto + tamaño + delimitador + contenido.

La diferencia esta en que el contenido de un commit es el siguiente: nombre de autor, correo de autor, descripción de la versión, y como opcional, la versión padre. La confirmación de versión (commit) sirve esencialmente para guardar diferentes versiones de los proyectos. **Cada commit es una versión diferente del proyecto.**

La siguiente imagen muestra cómo son los apuntadores de cada objeto de git:



Como se puede ver, el archivo de versión (commit) es una especie de envoltorio para el objeto tree, y tiene un apuntador hacia el tree. Cada uno de los commits, puede ser llevado al directorio de trabajo para ver esa versión del proyecto. Los siguientes comandos sirven para establecer en git el nombre y dirección de correo electrónico:

```
git config --global user.name <name>
git config --global user.email <Email>
```

Y para leer la configuración establecida, usamos el siguiente comando: `git config --list`

### 2.3.2 Creación de las primeras versiones

En primer lugar, debemos estar pendientes de cuál es el estado del repositorio.

```
git status
```

Con el comando anterior podemos ver los cambios realizados para ser confirmados. También podemos ver si hay algún cambio sin seguimiento para añadir y posteriormente ser enviados/confirmados.

Una vez tengamos listos los cambios realizados para enviar, usamos el comando `git commit -m "comment"`. Con la opción `-m` podemos asignar un comentario a la versión. Es muy importante tener en cuenta que cuando hacemos confirmación de versión, estamos enviando información del 'staging area' al 'git repository', y cuando hacemos el proceso contrario (desde 'staging area' a 'working directory'), el proceso se llama 'checkout'.

El commit como archivo hash contiene lo siguiente:

- tree (es el hash del tree principal al que apunta el commit)
- parent (es el hash del commit anterior)
- Usuario autor de los cambios
- Usuario que confirmó el cambio
- Comentario

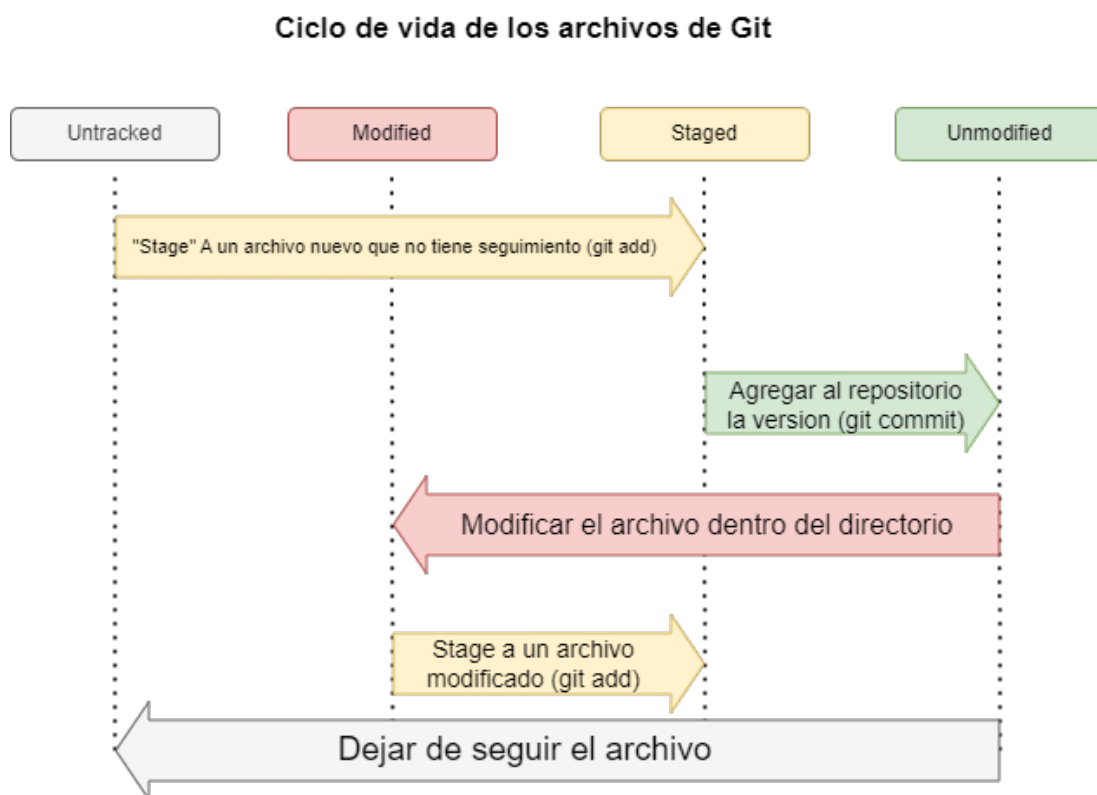
### 2.3.3 Comandos básicos de git

- `git status`
- `git add` con este se agregan archivos al área de staging
- `git commit` con este se escriben los cambios al repositorio ya como objetos
- `git log` con este se muestra el historial de cambios o commits
- `git checkout` este comando sirve para poner en el directorio actual (working directory) un commit o un branch específico.
- `git cat-file -p <hash>` Con este comando, como se mostró arriba, se visualiza el contenido del archivo correspondiente al hash ingresado.
- `git ls-files -s` Lista todos los archivos (blob) indicando el directorio donde están e indicando su hash.

Cuando se agrega un nuevo archivo al directorio, los archivos pueden tener cuatro estados diferentes:

- Untracked
- Modified
- Staged
- Unmodified

Si un archivo recién creado se adiciona al directorio, directamente está en el estado "untracked". En la siguiente figura se puede observar cómo va cambiando el estado de los archivos.



Para listar los archivos que están dentro del staging area, se usa el comando `git ls-files -s`. Al agregar archivos al área de staging, tenemos varias opciones

- `git add <name>` Agrega el archivo especificado a la zona de preparación.

- `git add -A` Agrega todos los archivos modificados, eliminados y nuevos al área de preparación. La opción `-A` incluye archivos en subdirectorios.
- `git add .` solo incluye los archivos en el directorio actual.
- `git add -u` Agrega todos los archivos modificados y eliminados al área de preparación, pero no los nuevos.
- `git add -p` Abre una sesión interactiva que permite agregar solo partes seleccionadas de los cambios realizados en un archivo.

Además podemos quitar archivos del area de preparación, mediante el comando `git rm --cached <name>`, esta última opción sirve para quitar un archivo en específico.

Una de las propiedades importantes de ver es que cuando realizamos un commit, este tiene un apuntador a su commit padre, es el hash de su commit padre. Según el tipo de commit este tendrá uno u otro commit padre (más adelante se verá que para pull requests pueden haber punteros distintos).

### 2.3.4 Historial de un archivo

Un comando útil para analizar la historia de un archivo en nuestro repositorio es el siguiente

```
git log --pretty=oneline <archivo_con_su_ruta>
```

Este comando retorna una lista de hashes que corresponden a los commits que han modificado dicho archivo:

```
13bde112178bbf94d9f83a2a14397c14d8cb973b UpdateBrowser
288cb6fee465de9e1bf682c7b47a25c4c75dd9e7 UpdateBrowser
2f7e1498f72e5d77b2773938f8516dd4c576b922 UpdateDictJson
23835e0a0d35796e708c50cfc71c523ef1b941d5 UpdateDictJson
f9dd3785bbc39a5e1ae9f8fed003f7b696f17f6b Se cambia apertura de navegador para form OVH
e1b1a7f0632ef745d02749468aa02eee016e62f9 Merge branch 'test2' of https://github.com/JhoAraSan/Proo
863641d145bdf2d4546a9b8f0328afed95d74ac Update code
47cf1a72d0ee58077a61558072c0866c46295547 cloudflare form bus ixed
673268edaff5d407e7de309c14e8a81829adf137 update
bdc232b7cb40c41c70eacfe7182d510145f26ce2 check bug
.
.
.
```

Como se puede ver, se muestra el hash del commit y a continuación se muestra el comentario realizado. El primer commit en la lista es el más reciente. El último de la lista será el commit que creó el archivo.

De forma alternativa, se puede ver solamente el commit creador del archivo en cuestión mediante el comando

```
git log --pretty=oneline --diff-filter=A -- .\Consola_3000\Consola3000.py
```

El cual devuelve

```
04bf774da85b9db1a059da4111887240ad2b3d4e renombramiento de carpeta a sugerencia de Sebastian
```

## 2.4 Las ramas

Como introducción a la sección, recordemos la capacidad de llevar un commit (screenshot de una versión del proyecto) al directorio actual. Esto se realiza mediante el comando "git checkout". En otras palabras, es algo así como saltar hacia una versión específica del proyecto.

Una definición general de lo que es una rama de github, es que es una referencia textual a un commit. Las siguientes son características de las ramas en git:

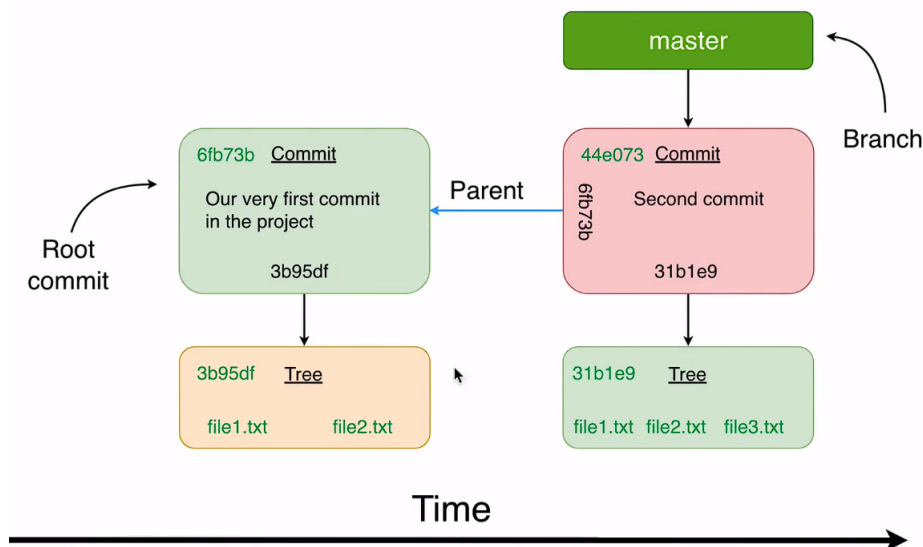
1. La rama por defecto es la master.
2. En un mismo repositorio pueden existir varias ramas.
3. Los apuntadores a todas las ramas se localizan en `.git/refs/heads`
4. Cada rama maneja sus propios commits.
5. El puntero de la rama se mueve automáticamente después de cada nuevo commit.
6. Para cambiar de branch se usa el comando `git checkout <branch>`

El puntero de la rama será el último commit realizado en dicha rama, el archivo es un texto que contiene el hash de dicho commit.

### 2.4.1 HEAD

El concepto de head es útil para especificar al sistema cuál es la rama en la que me encuentro actualmente. Básicamente HEAD es el apuntador que apunta hacia la rama/commit **actual**. Solamente existe un solo HEAD en cada proyecto.

1. El puntero se guarda en `.git/HEAD`
2. El Puntero por defecto es `refs:/heads/master`
3. Para cambiar la referencia a una rama específica se usa `git checkout <branch>`
4. Para cambiar la referencia a un commit específico se usa `git checkout <sha1>`



Cada vez que se crea una nueva rama en un repositorio de Git, se crea una referencia a la cabeza (HEAD) de esa rama en el sistema de archivos de Git. Esta referencia se guarda en el directorio `.git/refs/heads/` dentro del repositorio.

Para la administración de las ramas disponemos de los siguientes comandos:

- `git branch` Lista todas las ramas locales
- `git branch <name>` Crea una nueva rama
- `git checkout <branch>` Se dirige a la rama especificada
- `git branch -d <name>` Borrar la rama especificada
- `git branch -m <old> <new>` Renombrar la rama especificada

Un comando muy útil para crear una rama nueva y dirigirse directamente a ella es la siguiente:

```
git checkout -b <branch name>
```

## Ejemplo

Para un repositorio de un proyecto cualquiera como ejemplo vamos a la carpeta `.git/refs/heads`.

Si se lista el contenido del directorio se obtiene la siguiente salida

```
Directory: C:\Users\seb-c\OneDrive\Documentos\Project_Process\Process\.git\refs\heads
```

Mode	LastWriteTime	Length	Name
----	-----	-----	----
1a---	3/25/2023 3:49 PM	41	master
1a---	9/6/2023 9:31 AM	41	speechGen
1a---	9/10/2023 4:00 PM	41	test2

El cual muestra que en el repositorio de ejemplo hay tres ramas: `master`, `speechGen` y `test2`. Si queremos ver el contenido del archivo `speechGen` se obtiene

```
866c48dd5d96c7ba7ae730dbd3dc85896bc6b576
```

Este es el hash correspondiente al **último** commit de esta rama. De aquí se puede concluir que la rama puede verse como un apuntador hacia el commit.

Para ver dónde se guarda el apuntador general `HEAD` hacia la rama (o commit) en el que se está actualmente. Vamos al directorio que guarda el puntero:

```
cd .git
cat HEAD
```

Se obtiene lo siguiente

```
ref: refs/heads/test2
```

El cual indica que en el momento de realizar el comando, el usuario estaba en la rama `test2`

## 2.5 Repositorios remotos

En esta sección se describirán algunas características no antes vistas sobre los procesos asociados a los repositorios remotos.

### 2.5.1 git diff

Este es un comando que puede ser útil para ver y evidenciar las diferencias entre un archivo modificado su anterior versión dentro de la consola. Al usar el comando podemos ver el hash provisional del nuevo archivo (el que tendría si se realiza el commit), las líneas agregadas- quitadas-modificadas.

## 2.6 Fusión o combinación de ramas

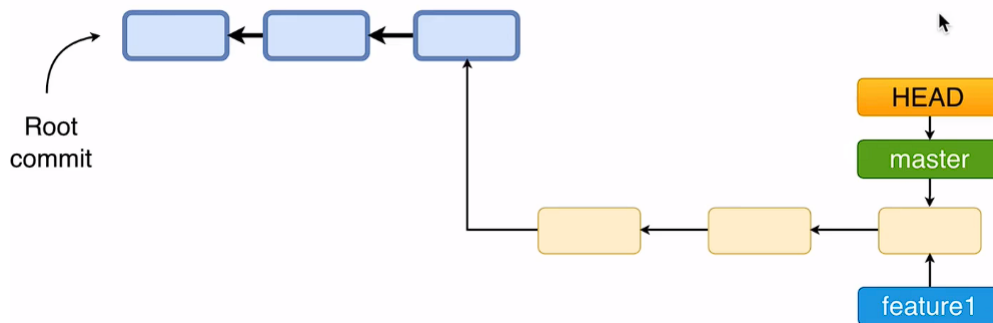
Es importante tener clara la perspectiva de dónde está apuntando `HEAD`. De ello depende la información que vamos a obtener al llamar al comando `git log`. Si estamos visualizando commits anteriores, ese commit será el actual para la vista que tengamos en el momento.

Ahora teniendo en cuenta lo anterior, suponemos que hemos creado una rama para agregar cualquier especificación; hemos creado esa rama desde la rama principal. Luego hemos realizado cambios en dicha rama y hemos confirmado dichos cambios. Posteriormente volvemos a cambiar la vista hacia la rama principal y **desde esta rama traemos o unimos los cambios realizados en la rama secundaria**; ese es el proceso de fusión o combinación de ramas.

```
git merge <feature-branch>
```

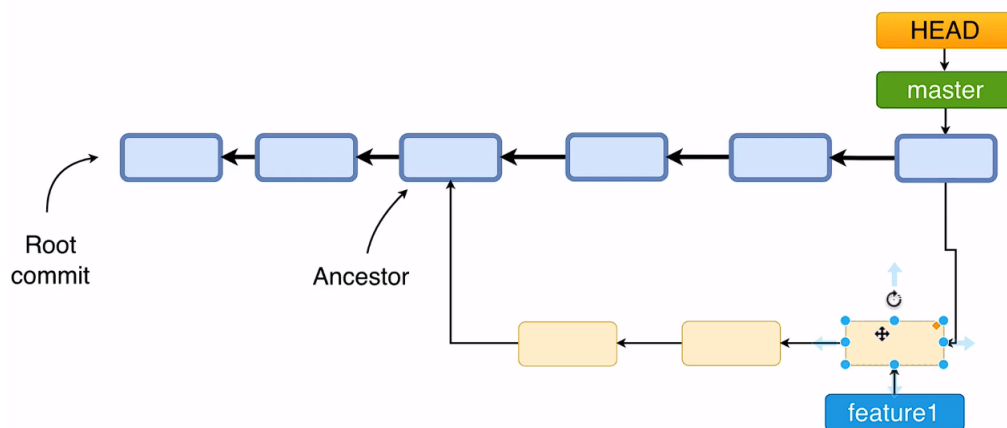


En esencia, cuando hacemos la fusión de ramas, lo que estamos realizando es un cambio del apuntador de la rama hacia la que fusionamos (la principal) para apuntar ahora al último commit realizado en la rama que estamos trayendo. Este caso aplica cuando en la rama principal no hay cambios después de haber creado la rama secundaria.



Supongamos ahora que tenemos nuestra rama principal, creamos una rama para trabajar en características secundarias, pero al mismo tiempo también realizamos cambios en la rama principal. Si en este momento queremos hacer una fusión de las ramas, ya no podemos simplemente cambiar el puntero de la rama principal; ahora es necesario realizar una fusión de 3 direcciones.

En la fusión de tres direcciones tenemos tres versiones importantes: la versión ancestro, que corresponde a la última versión en común que tienen las dos ramas a unir; la última versión de la rama secundaria y la última versión de la principal. Como en la fusión anterior, también se debe ir a la versión de la rama receptora; se crea una nueva versión de fusión en esta rama; **dicha versión tendrá como versiones padres la última de la rama receptora y la última de la rama secundaria.** Si no existen conflictos entre archivos que se hayan modificado en ambas ramas, simplemente la nueva versión combinará los archivos nuevos.



Una vez realizado este proceso, se puede borrar la rama secundaria, lo cual significa borrar el apuntador de la rama, las versiones permanecen.

### 2.6.1 conflictos de fusión

Los conflictos ocurren cuando se intenta fusionar dos ramas y en ambas se ha modificado el mismo archivo. Estos conflictos siempre deben ser arreglados manualmente. Si intentamos unir dos ramas y se generan conflictos, el estado actual del repositorio cambiará a tener dos caminos sin fusionar. Git le pedirá al usuario que corrija los conflictos y que confirme dichos cambios. Están las opciones de

dejar los cambios de la rama principal, dejar los cambios de la rama secundaria, o de dejar ambos cambios.

Algo interesante de observar, es que en este momento se habrán creado tres objetos blob diferentes en el staging area. tres objetos que tienen el nombre del archivo que contiene el conflicto. El primero corresponde a la versión ancestro de ambas ramas, el segundo corresponde a la modificación de la rama principal, y el ultimo corresponde a la modificación de la rama secundaria. Existen varias formas de resolver estos conflictos; el primero se puede hacer mediante la consola:

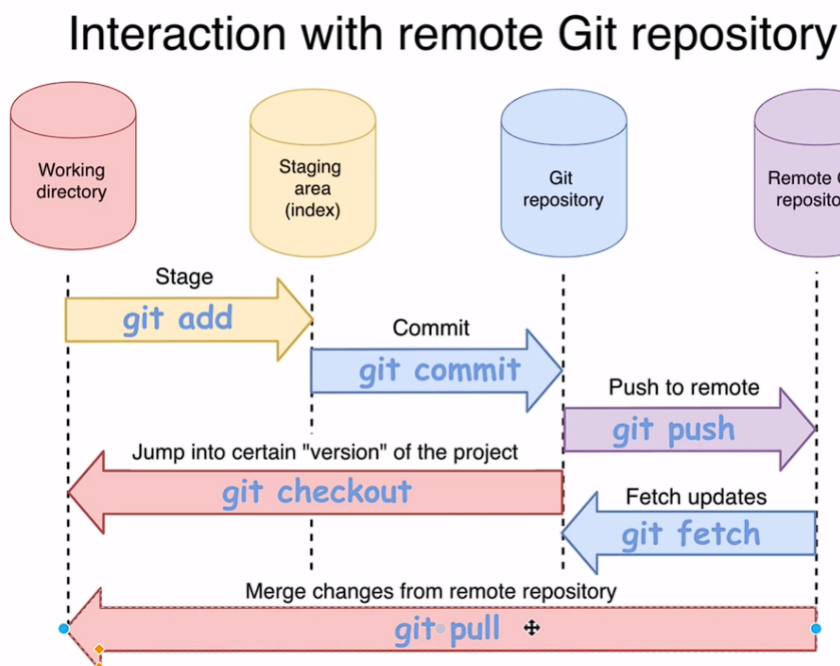
Abrimos el archivo que contiene los conflictos mediante **nano** por ejemplo y seleccionar manualmente cuál de la(s) líneas van a ser conservadas. guardar el archivo y de esta manera el o los conflictos habrán sido resueltos. También se tiene a opción de incluso volver a modificar el archivo si ninguna de las dos versiones es la que queremos. Finalmente cuando las modificaciones sean las deseadas, podremos concluir la fusión de las ramas mediante la confirmación (commit).

## 2.7 Comandos para los repositorios remotos

Dentro de los comandos más importantes para la clonación de los repositorios remotos, los siguientes son los más importantes:

### 2.7.1 Git push

Añadido a las tres áreas de los repositorios, tenemos un área adicional que corresponde al repositorio remoto. El primer comando envía toda la información desde el área de repositorio local y lo envía directamente al repositorio remoto. Solamente los cambios que están confirmados son los que efectivamente se ven reflejados en el repositorio remoto.



### 2.7.2 Git fetch y pull

Una vez el repositorio está actualizado, y es necesario pasar esa información al repositorio local, se pueden hacer dos comandos, el primero es `git fetch`: este comando es para enviar toda la información del repositorio remoto al repositorio local, pero sin enviarlo al área del directorio de trabajo. Por su parte, si queremos enviar directamente la información del repositorio remoto al directorio de trabajo usamos el comando `git pull`.

Para entender un poco la diferencia entre los dos comandos, supóngase que se crea una nueva rama en el repositorio remoto. Después de realizar `git fetch` dicha rama será creada en el repositorio local. En otras palabras, con `git fetch` básicamente estoy actualizando la información del repositorio remoto en mi repositorio local.

Por su parte, el siguiente ejemplo muestra el funcionamiento de `pull`:

1. Se clona el repositorio remoto
2. Se hace `checkout` a la rama master en el repositorio local
3. Se realizan cambios y se confirman en la rama master del repositorio remoto
4. Después de realizar `git pull` el repositorio local extraerá los cambios del remoto.
5. Git realiza la fusión de la rama master en el repositorio local
6. Tanto el área de staging como el directorio de trabajo se actualizan automáticamente luego de la fusión.

Cuando uno clona un repositorio remoto en uno local, git automáticamente crea un enlace entre ambos repositorios. Por defecto para el repositorio local, el nombre del repositorio remoto es `origin`.

Siempre que se clona un repositorio, git solamente crea una rama local con el mismo nombre de la rama por defecto del repositorio remoto. Por tanto, si tenemos dos ramas en el repositorio remoto, y clonamos este repositorio, en nuestro repositorio local solamente habrá una rama que es la rama principal del repositorio remoto. Para poder traer una rama remota que no es la principal a nuestro repositorio local, solamente necesitamos hacer un `checkout` a dicha rama, de esta forma esta rama aparecerá en nuestro repo local.

Para los repositorios remotos, un comando importante y útil puede ser el siguiente `git remote show origin`. Con este comando podemos ver mucha más información sobre el repositorio remoto como las ramas remotas, las ramas locales, cuáles están configuradas o trackeadas, etc. El siguiente es un ejemplo de la información total de un repositorio remoto:

```
PS C:\Users\seb-c\OneDrive\Documentos\Project_Process\Process> git remote show origin
* remote origin
Fetch URL: https://github.com/JhoAraSan/Process.git
Push URL: https://github.com/JhoAraSan/Process.git
HEAD branch: master
Remote branches:
  master                tracked
  refs/remotes/origin/clases  stale (use 'git remote prune' to remove)
  refs/remotes/origin/test    stale (use 'git remote prune' to remove)
  refs/remotes/origin/virtual stale (use 'git remote prune' to remove)
  speechGen              tracked
  test2                  tracked
Local branches configured for 'git pull':
  master    merges with remote master
  speechGen merges with remote speechGen
  test2     merges with remote test2
Local refs configured for 'git push':
  master    pushes to master      (up to date)
  speechGen pushes to speechGen   (up to date)
  test2     pushes to test2       (up to date)
```

En este ejemplo "stale" indica que la rama fue borrada del repositorio remoto, y se puede quitar de la lista con el comando `git remote prune origin`

Por su parte, cuando queremos sincronizar los cambios entre el repositorio remoto y el local, las ramas siempre harán un merge, es decir que se fusionarán y habrá que manejar los posibles conflictos que hayan entre la rama local y su correspondiente remota.

Para el siguiente ejemplo podemos listar todas las ramas en los repositorios local y remoto:

```
git branch -a

master
speechGen
* test2
remotes/origin/HEAD -> origin/master
remotes/origin/clases
remotes/origin/master
remotes/origin/speechGen
remotes/origin/test
remotes/origin/test2
remotes/origin/virtual
```

Como se puede ver, la línea `remotes/origin/HEAD -> origin/master` indica que en el repositorio remoto, la rama por defecto es la master.

### 2.7.3 Ramas rastreadas

Las ramas rastreadas son todas las ramas del repositorio remoto que tienen su correspondencia con la rama en el repositorio local. En otras palabras, son las ramas que aparecen tanto en el repositorio remoto como en el repositorio local.

Cuando se clona un repositorio remoto, solamente se crea la rama principal en el repositorio local. Estando en repositorio local se realiza el comando `git checkout` a una rama del repositorio remoto, y en ese momento se crea la rama rastreada. El comando `git branch -vv` muestra información de las ramas rastreadas:

```
git branch -vv

master      5d42019 [origin/master] Unificando y dejando la verdadera principal
speechGen   866c48d [origin/speechGen] first ver apps module in window DONE!
* test2     3c3e9ae [origin/test2] Cambio del nombre de la Appstore
```

Como se ve, se muestra el nombre, el hash y el comentario de las ramas que están rastreadas.

### 2.7.4 Proceso pull

Para realizar un `git pull`, se combinan varios conceptos previamente discutidos:

1. Rama de rastreo local: Es necesario tener una rama local que esté rastreando una rama remota para poder realizar un `pull`
2. Funcionamiento del `merge`: Es necesario entender cómo funcionan las fusiones. Recordar que puede hacerse con el enfoque de avance rápido o con una fusión de tres vías.
3. `git fetch` antes del `pull`, git efectúa un `fetch` para obtener todos los cambios desde el repositorio remoto

El proceso de `git pull` se realiza en dos pasos:  
Primero se ejecuta un `git fetch`, que toma todas las actualizaciones del repositorio remoto y las escribe en el repositorio local. Esto incluye todas las nuevas ramas y versiones creadas en el repositorio remoto.

Luego se ejecuta un `git merge`. Esta fusión es local, es decir, se hace en el repositorio local sin interactuar con el repositorio remoto. Durante este proceso, se utilizan dos ramas: la rama receptora será la rama local y la rama "feature" será la rama remota correspondiente. Git mezclará la rama remota en la rama local.

Es relevante notar que Git usa un término especial, "Fetch Head", en lugar del nombre de la rama remota durante este proceso. Como limitación, `git pull` actualiza solo la rama local que está actualmente en uso. No afecta ninguna otra rama local.

### 2.7.5 Fetch head

Pongamos un ejemplo: sea un repositorio tal que al revisar las ramas tanto locales como remotas obtenemos el siguiente resultado:

```
git branch -a
```

```
*   feature-1
    master
    remotes/origin/HEAD -> origin/master
    remotes/origin/master
```

Vemos que tenemos las dos ramas remotas master y feature-1 con sus correspondientes ramas locales. Si hacemos el siguiente código

```
git branch -vv
*   feature-1    ccc9d7b [origin/feature-1]
    master       f38cf54 [origin/master]
```

Vemos nuevamente la correspondencia entre las ramas. Escribiendo el comando `fetch` y luego `pull` tenemos lo siguiente:

```
git fetch -v
```

```
From https://github.com/yo/myrepo
 = [up to date] feature-1    -> origin/feature-1
 = [up to date] master      -> origin/master
```

```
git pull
Already up to date
```

```
git pull -v
```

```
From https://github.com/yo/myrepo
 = [up to date] feature-1    -> origin/feature-1
 = [up to date] master      -> origin/master
Already up to date
```

Con lo anterior podemos ver que `pull` siempre hace `fetch` antes de hacer algunos cambios adicionales para intentar fusionar las ramas.

Si vemos los archivos que están dentro de la carpeta `.git` podremos ver que hay un archivo llamado `FETCH_HEAD`, cuyo contenido sería el siguiente

```
cat FETCH_HEAD
73acf27141929dd1f890236317cb54009914c35a      branch 'test2' of https://github.com/JhoAraS
629fab88a1c5954a9389c827f0e7b4829ce734ba      not-for-merge tag '1' of https://github.com/JhoAraS
```

Este contenido corresponde a las ramas que están en el repositorio remoto. Si cambiamos de rama, al hacer este mismo comando, veremos en primer lugar la rama que está actualmente. Cuando hacemos `git pull`, Git primero ejecuta "git fetch". Después del fetch se actualiza la lista de `.git/FETCH_HEAD` y la primera rama de esta lista será la rama actual. Finalmente Git ejecuta `git merge FETCH_HEAD` que busca la primera rama en el fetch head sin la etiqueta "not-for-merge" y la fusiona con la rama local actual.

### 2.7.6 Git pull con modificación de repositorio remoto

Si después de hacer `git pull` el repositorio remoto es modificado, se crea un nuevo commit en el repositorio remoto. Git actualiza el repositorio y realiza la actualización del apuntador en el repositorio local, del commit antiguo al nuevo commit.

Ahora supongamos que se realizan cambios tanto en el repositorio remoto como en el repositorio local, y queremos traer o hacer pull al repositorio remoto. Tengamos en cuenta que al revisar con `git log`, veremos el último commit que acabamos de hacer, y anterior a ese veremos el último commit sincronizado; el que se supone que es el último commit del repositorio remoto. Sabemos que esta es una información falsa, pues este commit está desactualizado. Mediante el comando `git fetch`, vamos a actualizar la información del cambio realizado en el repositorio remoto, aunque Git ya creó los objetos dentro del repositorio local, estos aún no se ven en el directorio de trabajo porque aún no se ha fusionado con la rama local.

Después de esto podemos fusionar el cambio remoto con el cambio local. Mediante `git merge FETCH_HEAD`

### 2.7.7 Subida de cambios al repositorio remoto

Ahora que tenemos cambios en el repositorio local que están ausentes en el repositorio remoto, es hora de poner la operación push en acción. Recordar siempre que es necesario tener permisos de escritura. Una vez estén sincronizados los repositorios, podemos ver que los apuntadores de los repositorios local (que está en la carpeta `.git/refs/head`) y la remota (que está en la carpeta `.git/refs/remotes/origin`) tienen el mismo commit de destino.

Si necesitamos crear una rama nueva, los pasos a realizar para que también se vea reflejada en el repositorio remoto es lo siguiente:

1. Se crea la nueva rama local mediante `git checkout -b nueva`
2. Se hacen los cambios correspondientes.
3. Se confirman los cambios con `commit`
4. Se sube la nueva rama al repositorio remoto con el comando `git push -v -u origin feature-2`

### 2.7.8 De rama local a rama nueva remota

Cuando se crea una rama nueva en el repositorio local y se requiere tener dicha rama en el repositorio remoto es necesario crear la rama nueva en ambos lados de forma manual. Si intentamos ejecutar el comando `push` a una rama nueva que no está en el repositorio remoto, obtendríamos el siguiente error:

```
fatal: The current branch nombre has no upstream branch.}
To push the current branch and set the remote as upstream, use
  git push --set-upstream origin nombre
```

el comando sugerido `git push --set-upstream origin nombre` efectivamente realiza la creación de la rama. Sin embargo, un comando equivalente y más corto es el siguiente

```
git push -v -u origin nombre
```

Y así queda creada la rama en el repositorio remoto con el mismo nombre que la rama local nueva.

### 2.7.9 Actualización de estados de ramas

Cuando una rama del repositorio remoto se elimina, es necesario realizar una actualización en el repositorio local. Si se ejecuta el comando `git branch -vv` se observará que aunque en el repositorio remoto ya no exista, la rama local todavía sigue a la rama remota. Incluso después de actualizar el repositorio mediante `git fetch`, vemos que la rama sigue siguiendo a la desaparecida rama remota. Para este caso se usa el siguiente comando

```
git remote update origin --prune
```

Esto asegura que el repositorio local reconozca la eliminación de la rama remota. Finalmente se ejecuta `git branch -D nombre`. Para eliminar también la rama local.

Ahora, también es posible eliminar la rama remota desde la consola local:

Recordemos que cuando se crea una nueva rama local, se debe especificar al momento de publicar el repositorio remoto:

```
git push -u origin nombre-rama
```

El comando para borrar una rama remota es

```
git push origin -d nombre-rama
```

Y luego se borra la rama local

```
git branch -D nombre-rama
```

### 2.7.9.1 comando git show-ref

Este comando es útil porque indica todas las referencias a las correspondientes versiones de las ramas tanto locales como remotas:

```
git show-ref
5d42019b30082902c9dd6dc8ebfcb5f63c75095d refs/heads/master
866c48dd5d96c7ba7ae730dbd3dc85896bc6b576 refs/heads/speechGen
4b8c370d8bbe8d2769cdb69e91d90fc9a2a7338e refs/heads/test2
5d42019b30082902c9dd6dc8ebfcb5f63c75095d refs/remotes/origin/HEAD
5d42019b30082902c9dd6dc8ebfcb5f63c75095d refs/remotes/origin/master
866c48dd5d96c7ba7ae730dbd3dc85896bc6b576 refs/remotes/origin/speechGen
4b8c370d8bbe8d2769cdb69e91d90fc9a2a7338e refs/remotes/origin/test2
d5c7ebc8c4cdf48f3395a092f5616a33f0aab274 refs/stash
```

Es un indicador muy útil para saber si un repositorio está debidamente actualizado. El comando puede especificar una sola rama solo añadiendo el nombre de la misma al comando.

## 2.8 Pull Requests

Una manera de definir las peticiones o solicitudes de integración "Pull requests" sería la siguiente: Una solicitud de integración es una propuesta de potenciales cambios en el repositorio. La idea principal detrás de trabajar con Git es desarrollar múltiples características de manera simultánea, usualmente en diferentes ramas y por diferentes personas. Cuando un colaborador, como Bob o Mike, está listo para aplicar cambios a la rama principal, inician comunicación con otros desarrolladores a través de "pull requests". Un "pull request" es simplemente una propuesta de cambios potenciales en el código. Estos cambios son "potenciales" porque después de una revisión por parte de otros desarrolladores, pueden ser rechazados o aprobados. Si se rechazan, el "pull request" se cierra y la rama correspondiente se elimina. El objetivo principal es aplicar los cambios para avanzar en el desarrollo.

El término "Pull Request" o "Merge Request" depende del contexto y del flujo de trabajo en desarrollo de software. En un entorno donde todos los desarrolladores trabajan en el mismo repositorio y tienen acceso de escritura, "Merge Request" sería más apropiado porque el objetivo es fusionar cambios en la rama principal tras la aprobación de revisores.

En cambio, en proyectos de código abierto con múltiples repositorios (uno principal y otros bifurcados), "Pull Request" es más adecuado. Aquí, un desarrollador que no tiene acceso de escritura al repositorio principal puede solicitar que el propietario del repositorio principal "jale" y revise los cambios de una rama en un repositorio bifurcado.

Por lo tanto, si los desarrolladores trabajan en el mismo proyecto, "Merge Request" es más apropiado. Si los desarrolladores crean bifurcaciones del repositorio y el propietario debe jalar cambios de esas bifurcaciones, entonces "Pull Request" es más adecuado.

### 2.8.1 Paso a paso del proceso de solicitudes de integración

El siguiente ejemplo expone de manera clara el proceso.

En un flujo de trabajo de desarrollo, dos desarrolladores, Mike y Bob, colaboran en un proyecto. Mike crea una nueva rama llamada "feature-one" en su computadora local y realiza cambios. Una vez satisfecho, sube estos cambios al repositorio remoto. A continuación, abre un "Pull Request" para iniciar el proceso de revisión por parte de otros colaboradores. Él puede iniciar el proceso de solicitud de integración una vez su rama esté en el repositorio remoto.

Mike solicita a Bob revisar esta solicitud. Él puede añadir dentro de la petición algunas descripciones de los cambios que realizó en su rama. Bob puede optar por descargar la rama para probar los cambios localmente o revisarlos directamente en línea. Tras la revisión, Bob puede añadir comentarios o solicitar cambios adicionales. Mike puede hacer los cambios necesarios y actualizar el Pull Request existente sin necesidad de crear uno nuevo. El Pull Request se actualiza automáticamente cuando él confirma los cambios en su rama y los sube al repositorio remoto. Cuando estas actualizaciones son subidas, a Bob se le notifica mediante correo electrónico sobre las nuevas versiones.

Una vez que Bob aprueba los cambios y se alcanza el número requerido de aprobaciones, se permite la fusión del Pull Request en la rama principal. Dependiendo de los permisos, uno de los desarrolladores o un administrador realiza la fusión.

Este proceso permite la colaboración efectiva y revisiones detalladas antes de que los cambios se integren en las ramas principales del proyecto. En resumen, el Pull Request es una herramienta central para revisar e implementar características en un entorno de desarrollo colaborativo.

Dentro de la página del repositorio remoto está la opción para crear una nueva solicitud de integración. Es importante que para que pueda haber una fusión, no pueden haber conflictos de fusión, por tanto cualquier conflicto de fusión debe resolverse antes de iniciara la solicitud de integración.

En la creación de la solicitud de integración se deber realizar una descripción del cambio, actualización, componente, característica, etc, realizado. Siempre es recomendable realizar la mejor descripción posible, ser muy claro y conciso con la descripción.

Una vez creada la solicitud, las personas pueden revisarla, añadir comentarios, ver los cambios que se han realizado, etc. Por ejemplo, dentro de la pestaña de archivos modificados, uno puede insertar un comentario sobre una línea de código específica.

Existen dos opciones para publicar el comentario: uno es añadir un solo comentario a la línea de código, y la otra es iniciar una review, que implica un grupo de líneas o una sección de código. Por otro lado está la opción de aprobar la solicitud dejando un comentario de la aprobación, dejar un simple comentario sin indicar aprobación o declinación, y sugerir más cambios para una nueva revisión.

Una vez la solicitud está aprobada, cualquier usuario con los permisos correspondientes puede fusionar la rama. Es importante notar que en algunas ocasiones Github por defecto permite a usuarios fusionar la rama incluso si no hay ninguna review.



# Chapter 3

## Datos

### 3.1 Modelo de datos

¿Qué es un modelo de datos?

Es un conjunto de tablas de datos que tienen relaciones y conexiones entre sí. Lo importante es que existan relaciones entre las diferentes tablas; relaciones de conexión entre datos como columnas o datos claves en común.

#### 3.1.1 Normalización de los datos

Es el proceso de organización de las tablas y las columnas en una base de datos relacional para reducir la redundancia y preservar la integridad de los datos.

- la eliminación de los datos para reducir el tamaño de las tablas y aumentar la velocidad y eficiencia.
- Minimizar las anomalías y los errores de las modificaciones de los datos.
- simplificación de las consultas y estructuración de la base de datos para análisis más útiles.

una buena manera de entender una base de datos normalizada es definiéndola de la siguiente manera: en una base de datos normalizada, cada tabla debe tener un **propósito único y específico**.

los modelos tienen generalmente dos tipos de tablas

1. Tablas de datos: contiene números o valores, típicamente a nivel granular, con una columna de identificación o Id que puede ser
2. Tablas de Vista: provee atributos descriptivos normalmente basados en texto sobre cada dimensión de la tabla. En este caso, este tipo de tablas provee mucha información sobre objetos como pueden ser "clientes" o "Productos".

#### 3.1.2 Expresiones de análisis de datos

Las expresiones de análisis de datos o DAX son un lenguaje de fórmulas de PowerBI. Con estas fórmulas se puede hacer lo siguiente:

- Añadir columnas con cálculos y mediciones al modelo, usando sintaxis intuitiva.
- Ir más allá de las capacidades de las formulas tradicionales, con funciones potentes y flexibles construidas específicamente para trabajar con modelos de datos relacionales.

Existen dos maneras de realizar acciones DAX: mediante creación de columnas nuevas, y mediante mediciones de un solo valor.

### 3.1.2.1 Columnas calculadas

Se pueden agregar columnas nuevas basadas en fórmulas a las tablas. Cosas importantes a remarcar:

1. No hay referencias a celdas, todas las columnas calculadas se referencian a tablas enteras o a columnas enteras
2. Las columnas calculadas generan valores por cada fila, los cuales son visibles dentro de las tablas en la vista de datos.
3. Las columnas calculadas entienden el conexto de filas, lo que significa que son muy útiles para definir propiedades que estén basadas en la información de cada fila.

Como un tip adicional y útil, use las columnas calculadas para **filtrar datos** más que para cear valores numéricos.

#### Ejemplo

```
Parent = IF(Customer_Lookup[TotalChildren]>0, "Yes", "No")
```

En este ejemplo se puede usar un if condicionante para agregar una columna nueva de control o de filtro.

### 3.1.2.2 Mediciones

En este caso son usados para generar nuevos valores calculados.

1. Igual que la anterior, no hay forma de referenciar celdas aisladas; solo para tablas enteras.
2. Estos valores no pueden ser visibles dentro de las tablas, solamente se pueden ver en elementos de visualización como cartas, o matrices.
3. Estos cálculos siempre estarán regidos por el contexto del filtro; es decir, el valor se recalcula siempre que sea aplicado algún filtro en el elemento de visualización que se agregue o configure.

A manera de regla general, use este tipo de cálculo cuando una única fila no puede entregarle el valor solicitado. En otras palabras, cuando necesite más de una fila.

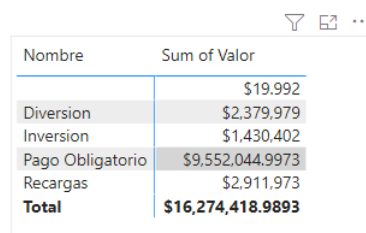
### 3.1.2.3 Mediciones implícitas y explícitas

Las mediciones implícitas están en los elementos de visualización cuando uno arrastra un campo numérico a dicho elemento. Ahí es cuando se puede seleccionar el modo de agregación (suma, media, mínimo, máximo, etc).

Las mediciones explícitas, son por el contrario las que se definen de manera literal con la sintaxis de DAX

Recordando que las mediciones son evaluadas con base en el contexto de filtrado, según el filtro aplicado el resultado de la medición sera distinta.

Por ejemplo, para la siguiente imagen, el valor sombreado es la suma del campo 'Valor' está canculado con base en el siguiente contexto de filtro: `IDLookup[Nombre]="Pago Obligatorio"`



Nombre	Sum of Valor
	\$19.992
Diversion	\$2,379,979
Inversion	\$1,430.402
Pago Obligatorio	\$9,552,044.9973
Recargas	\$2,911.973
<b>Total</b>	<b>\$16,274,418.9893</b>

`CALCULATE(Measure, DATEADD(Calendar[Date], -1, MONTH))` Aquí la función `DATEADD` devuelve un rango de fechas desplazado en la cantidad y con intervalos puestos en sus argumentos

`CALCULATE(Measure, DATESINPERIOD(Calendar[Date], MAX(Calendar[Date]), -1, DAY))`

### 3.1.2.4 Sintaxis de las expresiones DAX

Las sentencias de las expresiones se pueden separar en tres partes diferentes: nombre de la medición realizada, nombre de la función, y las referencias o argumentos de las funciones, estas pueden referirse al nombre de la tabla con su respectiva columna. También se puede estar refiriendo a una medición anteriormente definida. Como tip de utilidad, cuando nos vayamos a referir a referencias de columnas, usamos el nombre: `Table[Column]`. Y para referencias a mediciones, solo usamos el nombre de la medición: `Measure`.

Algunos operadores importantes los podemos ver en la siguiente tabla

Operador	Significado	Ejemplo
+	Suma	2 + 7
-	Resta	5 - 3
*	Multiplicación	2 * 6
/	División	2 / 4
^	Exponente	5 ^ 5
=	Igual a	[City] = "Boston"
>	Mayor	[Quantity] > 10
<	Menor que	[Quantity] < 10
>=	Mayor o igual	[Unit_Price] >= 2.5
<=	Menor o igual	[Unit_Price] <= 2.5
<>	Diferente a	[Country] <> "Mexico"
&	Concatena dos caracteres o cadenas para formar una nueva	[City] & " " & [State]
& &	AND	([State]="MA") & & ([Quantity]>10)
	OR	([State]="MA")    ([State]="CT")
IN	Crea una condición lógica OR basada en una lista dada.	'Store Lookup'[State] IN "MA", "CT", "NY"

Las funciones también las podemos clasificar en diferentes categorías:

#### 1. Math and stats

Tenemos aquí las funciones matemáticas de agregación más básicas y también algunas iteradoras:

- SUM
- AVERAGE
- MAX/MIN
- DIVIDE
- COUNT/COUNTA
- COUNTROWS
- DISTINCTCOUNT
- SUMX
- AVERAGEX
- MAXX/MINX
- RANKX
- COUNTX

#### 2. Logical functions

Estas funciones generalmente tienen la tarea de retornar información sobre valores, con base en una expresión condicional dada. usualmente o mayoritariamente basadas en condicionales "if":

- IF
- IFERROR
- AND
- OR
- NOT
- SWITCH
- TRUE
- FALSE

### 3. Text functions

Son funciones que sirven para manipular texto o formatos de control para fechas, horas o números

- CONCATENATE
- FORMAT
- LEFT/MID/RIGHT
- UPPER/LOWER
- PROPER
- LEN
- SEARCH/FIND
- REPLACE
- REPT
- SUBSTITUTE
- TRIM
- UNICHAR

### 4. Filter functions

Estas son funciones que están basadas en tablas relacionadas y para filtrar funciones para cálculos dinámicos.

- CALCULATE
- FILTER
- ALL
- ALLEXCEPT
- RELATED
- RELATEDTABLE
- DISTINCT
- VALUES
- EARLIER/EARLIEST
- HASONEVALUE
- HASONEFILTER
- ISFILTERED
- USERELATIONSHIP

### 5. Date and time functions

Estas son funciones especializadas en fechas y horas, así como operaciones inteligentes avanzadas

- DATEDIFF
- YEARFRAC
- YEAR/MONTH/DAY

- HOUR/MINUTE/SECOND
- TODAY/NOW
- WEEKDAY/WEEKNUM
- DATESYTD
- DATESQTD
- DATESMTD
- DATEADD
- DATESINPERIOD

Un par de funciones muy útiles para el manejo de las fechas son `weekday/weeknum()` y `eomonth()`. La primera función retorna el número del día de la semana de la fecha que se ponga como argumento. Por defecto, el número 1 será el día domingo, pero esto se puede cambiar mediante una opción. La segunda retorna la fecha correspondiente al último día del mes, más o menos un número especificado de meses.

Una de las funciones más importantes o útiles puede ser `RELATED` la cual retorna valores relacionados directamente con cada fila con base en la relación que haya con otras tablas. La sintaxis de la función es:

La función `DATEADD` Devuelve la fecha que resulta de restar o sumar la cantidad especificada en el argumento.

`=RELATED(ColumnName)`

el nombre de la columna es la columna de la cual se requiere extraer la información o el dato. Como tip es recomendable tratar de no crear nuevas columnas basadas en esta función, en algunas ocasiones esto puede ser redundante; es más recomendable usar este tipo de funciones dentro de otras como iteradores del tipo `SUMX`.

`DATEADD(Calendar[date], -1, MONTH)`

La anterior devuelve la fecha restada en un mes

Teniendo en cuenta la tabla de gastos personal, si usamos la siguiente función:

`YTD Gastos = CALCULATE(SUM(Gasto[Valor]), DATESYTD(Calendario_Lookup[Fecha]))`

obtenemos la suma acumulativa de los gastos y se "renueva" en año nuevo. Observar como las barras van aumentando y el valor acumulado en el año.

Start of Month	Sum of Valor	YTD Gastos
Friday, July 01, 2022	\$1,385,890	\$1,385,890
Monday, August 01, 2022	\$2,677,236.256	\$4,063,126.256
Thursday, September 01, 2022	\$2,809,979.404	\$6,873,105.66
Saturday, October 01, 2022	\$2,622,030.812	\$9,495,136.472
Tuesday, November 01, 2022	\$2,756,051.46	\$12,251,187.932
Thursday, December 01, 2022	\$5,171,846.0376	\$17,423,033.9696
Sunday, January 01, 2023	\$5,508,035.1006	\$5,508,035.1006
Wednesday, February 01, 2023	\$826,739.9978	\$6,334,775.0984
<b>Total</b>	<b>\$23,757,809.068</b>	<b>\$6,334,775.0984</b>

### 3.1.3 Parámetros 'what if'

Este tipo de parámetros son esencialmente parámetros DAX preconfigurados que producen valores dentro de un rango dado. Este tipo de medidas pueden ser muy útiles por ejemplo para estudios de pronósticos o para estudios de posibles escenarios. para más información revisar la lección 98 del curso.

## 3.2 SQL

### 3.2.1 Conceptos

#### 3.2.1.1 Query

Es una porción de código que induce a la computadora a ejecutar una serie de operaciones y que entregará la salida requerida.

SQL Es un lenguaje de programación declarativo no procedural. No se focaliza tanto en el procedimiento como en cuál es la tarea que se está solicitando. Su sintaxis se compone principalmente de

- Lenguaje de definición de datos
- Lenguaje de manipulación de datos
- Lenguaje de control de datos
- Lenguaje de control de transacciones

```
CREATE object_type object_name
```

```
CREATE TABLE object_name (column_name data_type)
```

`CREATE TABLE sales (purchase_number INT data_type)` creará una tabla llamada sales con una columna llamada purchase number del tipo entero.

```
ALTER TABLE sales  
ADD COLUMN date_of_purchase DATE;
```

Se modifica la tabla agregando una columna nueva.

`DROP TABLE customers;` borra la tabla entera

`RENAME TABLE customers TO customer_data;` cambia el nombre de la tabla.

`TRUNCATE` borra los datos enteros de una tabla, pero la tabla sigue existiendo

Ahora algunas palabras reservadas de DML (lenguaje de manipulación de datos)

`SELECT .. FROM sales` selecciona lo indicado de la tabla de ventas. Se usa para extraer información de la tabla

`INSERT INTO sales (purchase_number, date_of_purchase) VALUES(1, '2017-10-11');` añade datos a la tabla

```
UPDATE sales  
SET date_of_purchase = '2017-12-11'  
WHERE purchase_number = 1;
```

Cambia directamente una entrada de la tabla

```
DELETE FROM sales  
WHERE  
    purchase_number = 1;
```

Sentencias para control de datos

```
GRANT type_permission ON database_name.table_name TO 'username'@'localhost'
```

Otorga permisos determinados a los usuarios

```
REVOKE type_permission ON database_name.table_name TO 'username'@'localhost'
```

Quita los permiso.

Finalmente los comandos de control de transacciones

COMMIT solamente funciona cuando se hacen cambios del tipo INSERT, DELETE, o UPDATE.

```
UPDATE customers
SET las_name = "Johnson"
WHERE customer_id = 4
COMMIT;
```

ROLLBACK permite devolver al estado anterior de commit

### 3.2.2 Pasos para crear una database y usarla

```
create database if not exists Sales;
use Sales
```

EL lenguaje SQL no es sensible a mayúsculas-minúsculas, ni para los nombres de los diferentes objetos, ni para las solicitudes. Esto quiere decir que **sales** y **Sales** serán iguales para SQL

### 3.2.3 Tipos de datos

Siempre es necesario especificar el tipo de datos que se insertará en cada columna de la tabla.

#### 3.2.3.1 Cadenas de caracteres

Existen varios tipos de cadenas string.

1. **caracter: CHAR**, tiene un tamaño de almacenamiento fijo y depende de qué tamaño sea declarado. **CHAR(5)** tendrá un tamaño de 5 bytes aunque el string tenga menos de 5 símbolos.
2. **caracter variable: VARCHAR** en este caso el tamaño de la variable no está fijo, así un **VARCHAR(5)** = 'bob' tendrá un tamaño de 3 bytes.
3. **ENUM** Es un tipo de variable que contiene un conjunto definido total de cadenas. Por ejemplo, para el género, cuando solamente hay dos opciones para seleccionar, se puede declarar la variable **ENUM('M', 'F')**.

Una cadena de caracteres char puede tener un tamaño máximo de 255 bytes. Un **varchar**, por su parte, puede tener un tamaño máximo de 65535 bytes. El procesamiento de las variables *char* es más rápido y eficiente que el de las **varchar**, razón por la cual a veces es más conveniente declarar las primeras.

#### 3.2.3.2 Enteros

La siguiente tabla resume los diferentes tipos de enteros que se pueden declarar y sus tamaños máximos así como los valores máximos que pueden tomar

Tipo	Tamaño	Valor mínimo (con signo/sin signo)	Valor máximo (con signo/sin signo)
TINYINT	1	-128/0	127/255
SMALLINT	2	-32 768/0	32 767/65 535
MEDIUMINT	3	-8 388 608/0	8 388 607 / 16 777 215
INT	4	-2 147 483 648 /0	2 147 483 647 / 4 294 967 295
BIGINT	8	-9 223 372 036 854 775 808 / 0	9 223 372 36 854 775 807 / 18 446 744 73 709 551 615

### 3.2.4 Variables de puntos flotantes y puntos fijos

En este contexto la precisión de un número se refiere a la cantidad de dígitos que hay en el mismo. Por ejemplo, 10.523 tiene una precisión de 5. Por su parte, la escala de la variable se refiere a la cantidad de dígitos que hay después del punto decimal. En el ejemplo anterior, la escala es de 3.

Es importante saber la diferencia entre datos de punto fijo y datos de punto flotante. Los datos de punto fijo son los que representan valores exactos. Existen dos tipos de variables para los puntos fijos:

```
DECIMAL(5,3)
NUMERIC
```

Para el ejemplo `decimal(5,3)` cualquier número insertado en esta variable contendrá esta estructura. Si se inserta un 10.5 el número en realidad será 10.500, y si se agrega un número con más decimales, entonces se redondeará para que pueda entrar en la variable, se saltará una advertencia.

Un buen ejemplo de uso de este tipo de datos son los valores monetarios: Salarios, gastos, etc. `NUMERIC(p,s)` donde `p` representa la precisión y `s` representa la escala.

Por su parte, los datos de coma flotante se usan para aproximar valores solamente y tiene como objetivo equilibrar el alcance y la precisión. Simplemente aproxima el valor y guarda esa aproximación. En este caso tenemos los siguientes tipos de variables para los números de coma flotante.

```
FLOAT
DOUBLE
```

`float` tiene un tamaño de 4 bytes. Es de precisión sencilla y cuenta con un máximo de 23 dígitos. `double` tiene un tamaño de 8 bytes. Es de precisión doble y cuenta con un máximo de 53 dígitos.

### 3.2.5 Otros tipos de datos

- **DATE** el formato es YYYY-MM-DD. La hora no hace parte de esta representación
- **DATETIME** El formato es YYYY-MM-DD HH:MM:SS[.fraction]. El rango de la hora es 0 - 23:59:59.999999
- **TIMESTAMP** Es el conteo en segundos desde una fecha inicial, sirve como punto de comparación entre dos fechas y establecer la diferencia en tiempo
- **BLOB** Significa binary large object. Se puede usar para guardar archivos de diferentes tipos: objetos binarios de gran tamaño. como fotos.
- 

### 3.2.6 Creación de tablas

```
CREATE TABLE table_name
(
    column_1 data_type constraints,
    .
    .
    .
    column_n data_type constraints
);
```

Ejemplo:



```
CREATE TABLE Sales
(
    purchase_name int not null primary key auto_increment,
    date_of_purchase date not null,
    customer_ID int,
    item_code varchar(10) not null
);
```

la sentencia `auto_increment` asigna el número de ID a esta columna y la aumenta automáticamente.

```
CREATE TABLE Customers
(
    customer_ID int not null primary key auto_increment,
    first_name varchar(255) not null,
    last_name varchar(255) not null,
    email_address varchar(255),
    number_of_complaints int
);
```

### 3.2.7 Restricciones de SQL

Las restricciones son especificaciones de reglas o límites que se imponen a cada una de las columnas de la tabla. Los términos "primary key" (clave primaria) y "foreign key" (clave foránea) son fundamentales para entender cómo se relacionan las tablas entre sí y cómo se mantiene la integridad de los datos.

#### 3.2.7.1 primary key

Una clave primaria es un campo (o conjunto de campos) que identifica de manera única cada fila en una tabla de una base de datos. Las características principales de una clave primaria son:

1. **Unicidad:** Ningún valor duplicado es permitido en una clave primaria. Cada fila debe tener un valor único para la clave primaria.
2. **No nula:** Una clave primaria no puede tener valores nulos. Cada fila debe tener un valor para la clave primaria.
3. **Identificación:** La clave primaria se utiliza para identificar de manera exclusiva una fila en la tabla.

Una forma de asignar una clave primaria a una tabla es la siguiente:

```
CREATE TABLE Sales
(
    purchase_number int auto_increment,
    date_of_purchase date,
    customer_ID int,
    item_code varchar(10),
    primary key (purchase_number)
);
```

Creemos las demás tablas con sus respectivas claves primarias:

```
create table customer
(
    customer_id int,
    first_name varchar(255),
```

```

        last_name varchar(255),
        number_of_complaints int,
primary key c(ustomer_id)
);

```

### 3.2.7.2 Clave foránea

Una clave foránea es un campo (o conjunto de campos) en una tabla que se utiliza para referenciar la clave primaria de otra tabla. La relación entre la clave foránea y la clave primaria es lo que permite establecer relaciones entre tablas. Las características principales de una clave foránea son:

1. **Correspondencia** con la clave primaria: Una clave foránea en una tabla corresponde a una clave primaria en otra tabla.
2. **Integridad referencial**: La clave foránea ayuda a mantener la integridad referencial entre las tablas, asegurando que la relación entre ellas sea válida. Por ejemplo, si una fila en una tabla A hace referencia a otra fila en una tabla B, entonces la fila referenciada debe existir en la tabla B.
3. **Valores nulos**: A diferencia de las claves primarias, las claves foráneas pueden tener valores nulos, lo que indica que no hay una relación con otra tabla.

En el contexto de bases de datos relacionales, los términos "tabla padre" y "tabla hija" son fundamentales para entender cómo se estructuran y relacionan los datos. Estos conceptos están intrínsecamente vinculados a las relaciones entre tablas, en particular a través del uso de claves primarias y foráneas.

**Tabla Padre:** Una tabla padre, en una relación de base de datos, es aquella que tiene una clave primaria referenciada por otra tabla. La característica clave de una tabla padre es que proporciona el registro principal o la fuente de datos que otras tablas referencian. Las propiedades de una tabla padre son :

1. **Posee una Clave Primaria**: Es esencial que tenga una clave primaria, que es un identificador único para cada fila de la tabla .
2. **Independencia**: La tabla padre puede existir en la base de datos sin la necesidad de tener una tabla hija asociada. Sus registros no dependen de la existencia de registros en otra tabla.
3. **Fuente de Referencia**: Otras tablas (tablas hijas) hacen referencia a la tabla padre a través de sus claves foráneas, estableciendo así una relación.

**Tabla Hija:** Una tabla hija es aquella que contiene una clave foránea que hace referencia a la clave primaria de otra tabla (la tabla padre). Las características de una tabla hija son

1. **Posee una Clave Foránea**: Contiene al menos una clave foránea que establece una relación con la clave primaria de la tabla padre.
2. **Dependencia Referencial**: La existencia de registros en la tabla hija generalmente depende de los registros en la tabla padre. Por ejemplo, no se puede insertar un registro en la tabla hija si no existe un registro correspondiente en la tabla padre, a menos que la clave foránea permita valores nulos.
3. **Integridad Referencial**: La clave foránea asegura que las relaciones entre la tabla hija y la tabla padre sean válidas y coherentes.

La forma de definir una clave foránea es

```

create table sales
(
    purchase_number int auto_increment,
    date_of_purchase date,

```

```

        customer_id int,
        item_code varchar(10),
primary key(purchase_number),
foreign key(customer_id) references customers(customer_id)
);

```

Donde la referencia se hace a la tabla que tenga esa clave como la primaria. Hay una restricción adicional llamada **on delete cascade**. Esta cláusula indica al sistema que si se elimina en la tabla padre una entrada o fila, entonces todas las entradas de la tabla hija que hagan referencia a esa clave también deberán ser borradas.

Se pueden modificar las características de las tablas sin necesidad de borrarlas para añadirlas nuevamente, mediante **alter**

```

alter table name
add foreign key;
drop primary key

```

De la misma forma se puede eliminar cualquier restricción o característica.

### 3.2.7.3 Clave única

La restricción **UNIQUE KEY** es un tipo de restricción que se puede aplicar a una o más columnas de una tabla para asegurar que cada fila de esa columna (o combinación de columnas) tenga un valor único dentro de la tabla. Es decir, no se pueden tener dos o más filas con el mismo valor en la(s) columna(s) marcada(s) como **UNIQUE**. Un ejemplo clásico de esta restricción es una columna de correos electrónicos; que no necesariamente constituye la clave principal, pero no puede ser duplicado, aunque pueden haber filas con valores en blanco en esta columna.

Para borrar una clave única de una tabla se debe usar

```
drop index name;
```

La instrucción **DROP INDEX** para eliminar una clave única **UNIQUE KEY** puede parecer inicialmente confusa, pero tiene sentido si consideramos cómo se implementan las claves únicas internamente en los sistemas de gestión de bases de datos.

**Implementación de Claves Únicas como Índices** Cuando se define una **UNIQUE KEY** en una tabla, la mayoría de los sistemas de bases de datos automáticamente crean un índice único para esa clave. Este índice único es lo que efectivamente garantiza la restricción de unicidad, asegurando que no se puedan insertar valores duplicados en la columna o conjunto de columnas designadas. pero ¿Por Qué Se usa **DROP INDEX**?

1. **Índices Únicos Subyacentes:** Dado que las claves únicas son implementadas internamente como índices únicos, la eliminación de una restricción de clave única implica la eliminación del índice asociado.
2. **Consistencia con Otros Tipos de Índices:** Los RDBMS suelen tratar las claves únicas de manera similar a otros índices (como los índices normales o los índices de texto completo). Por lo tanto, se utiliza un comando común **DROP INDEX** para eliminar cualquier tipo de índice, ya sea que haya sido creado para una clave primaria, una clave única, o para optimización de consultas.
3. **Simplificación de la Sintaxis SQL:** Utilizar un comando común para eliminar índices, independientemente de su tipo, simplifica la sintaxis del lenguaje SQL y hace que el manejo de la base de datos sea más uniforme y predecible.

**Consideraciones Adicionales:** Es importante mencionar que la sintaxis exacta para eliminar una clave única puede variar entre diferentes sistemas de bases de datos. Algunos sistemas pueden ofrecer una sintaxis que permite eliminar directamente una clave única sin referirse explícitamente al índice. Por su parte, eliminar una clave única (y por lo tanto, el índice único asociado) debe hacerse con cuidado, ya que altera las restricciones de integridad de la tabla y puede afectar el rendimiento de las consultas.

#### 3.2.7.4 Por defecto

La restricción `default` sirve para asignar un valor particular a cualquier fila de una columna; una forma de establecer un valor predeterminado para los casos en que no se especifica un valor particular al momento de añadir filas en la tabla. `number_of_complaints int default 0`. Para modificar una columna y configurar su valor por defecto, se puede mediante la siguiente forma:

```
alter table name
change column old_name new name new constraint;
alter column col_name drop constraint;
```

Las sentencias `ALTER TABLE ... CHANGE COLUMN` y `ALTER TABLE ... ALTER COLUMN` son utilizadas para modificar las características de las columnas, pero tienen propósitos y comportamientos distintos.

`CHANGE COLUMN` se usa principalmente para cambiar el nombre de la columna, y también permite cambiar el tipo de datos de la columna y agregar o modificar restricciones. Por su parte, `ALTER COLUMN` se usa para modificar restricciones o configuraciones de la columna ; específicamente para cambiar aspectos como el valor por defecto o para modificar la capacidad de aceptar valores nulos. A diferencia de `change column`, `alter column` no se usa para cambiar el nombre de la columna o su tipo de datos.

Hay una tercera, llamada `MODIFY COLUMN`: Usado para cambiar el tipo de datos de una columna y modificar o agregar restricciones, pero sin cambiar el nombre de la columna.

#### 3.2.7.5 NOT NULL

Esta restricción para una columna impide que una entrada nueva esté vacía; es decir, debe tener algún valor siempre.

### 3.2.8 Buenas prácticas

Es evidente la importancia de realizar siempre un código limpio y bien entendible, organizado, cuyas secciones sean fácilmente intercambiables. Al momento de declarar nombres, asegurarse de que sean nombres cortos, con significado acorde a su función, que brinde información sobre sí mismo, que sea pronunciable. La convención clásica indica que aunque SQL no distingue mayúsculas de minúsculas, es buena práctica usar mayúsculas para escribir palabras reservadas y tipos de instrucciones, y minúscula para los nombres, nunca usar espacios, y separar palabras por guión bajo.

### 3.2.9 Manipulación de datos

Una de las declaraciones más importantes en el mundo de SQL es `SELECT`.

#### 3.2.9.1 SELECT

Es una declaración que permite extraer una fracción del conjunto de datos. Permite obtener datos de las tablas. Básicamente se trata de una instrucción que se usa para solicitar datos de la base de datos. La sintaxis es la siguiente:

```
SELECT col_1, col_2, ...
FROM table_name;
```

Hay una cláusula dentro de la declaración denominada `WHERE`, la cual permite establecer una condición sobre la cual se puede especificar qué parte de los datos se requiere obtener

```
SELECT col_1, col_2, ...
FROM table_name
WHERE condition;
```

La condición puede ser cualquier valor de verdad: `WHERE first_name = 'Denis'`. Los operadores de verdad son

- =
- AND
- OR
- IN
- NOT IN
- LIKE
- NOT LIKE
- BETWEEN... AND..
- EXISTS
- NOT EXISTS
- IS NULL
- IS NOT NULL

Uno de interés de los anteriores es el LIKE, utilizando

```
SELECT * FROM table WHERE first_name LIKE('seb%')
SELECT * FROM table WHERE first_name LIKE('%an')
```

Indica una búsqueda de los nombres que empiecen por 'seb' ('seb%') o que terminen con 'an' (%an). Si se usa %ak% se selecciona todas las que tengan 'ak' en cualquier parte. Si se usa por ejemplo 'Mar\_', entonces solo se buscará todo lo que contenga 'Mar' y **un solo caracter más**. Por parte de las fechas, SQL es capaz de realizar comparaciones entre fechas con operadores simples, por ejemplo la comparación

```
hire_date > '2000-01-01'
```

La sentencia SELECT DISTINCT tiene la capacidad de retornar solamente valores únicos.

### 3.2.9.2 Funciones agregadas

Se puede obtener mucha información de las tablas a través de las funciones de agregación

1. COUNT
2. SUM
3. MIN
4. MAX
5. AVG

La sintaxis es la siguiente

```
SELECT COUNT(col_name)
FROM table_name
```

Las operaciones de verdad se deben hacer directamente con las tablas:

```
SELECT
    COUNT(DISTINCT col_name)
FROM
    table_name;
```

Las funciones de gregación ignora los elementos NULL a menos que se le indique lo contrario. Por su parte, seleccionamos operaciones especializadas mediante la forma

```
SELECT
    COUNT(*)
FROM
    table_name
WHERE
    col_name > 100000;
```

**Ordenación** Podemos ordenar el resultado con base en cualquier columna, para esto usamos **ORDER BY col\_name**:

```
SELECT
    *
FROM
    table_name
ORDER BY first_name;
```

Por defecto el ordenamiento es ascendente, para configurar lo contrario usamos

```
SELECT
    *
FROM
    table_name
ORDER BY first_name DESC;
```

Se pueden poner varias columnas para ordenar, la primera tendré la prioridad.

```
SELECT
    *
FROM
    table_name
ORDER BY first_name other_col ASC;
```

**GROUP BY** Todos los resultados de SQL pueden ser agrupados de acuerdo a uno o más campos específicos. La sentencia debe ser escrita inmediatamente después de las condiciones dadas por **WHERE** si están, y justo antes de la cláusula **ORDER BY**. El comando agrupa las filas que contienen los mismos valores en columnas especificadas en conjuntos de resumen. Estos cagrupamientos pueden usarse para realizar operaciones de agregación.

```
SELECT
    COUNT(first_name)
FROM
    employees
GROUP BY first name;
```

Este filtro devuelve una lista con el número de veces que cada nombre se repite en toda la lista. se puede añadir el nombre para que quede más completo:

```
SELECT
    first_name, COUNT(first_name)
FROM
    employees
GROUP BY first name;
```

De manera más compacta, escribimos un resumen de una solicitud para las tablas mediante **SELECT**:

```
SELECT col_name(s)
FROM table_name
WHERE conditions
GROUP BY col_name(s)
ORDER BY col_name(s)
```

**alias** Se usa para renombrar las columnas que extraigamos mediante alguna operación

```
SELECT
    first_name, COUNT(first_name) AS name_freq
FROM
    employees
GROUP BY first_name;
```

**HAVING** Se utiliza a menudo junto con **GROUP BY** para refinar condiciones, la estructura es

```
SELECT col_name(s)
FROM table_name
WHERE conditions
GROUP BY col_name(s)
HAVING conditions
ORDER BY col_name(s)
```

Es el equivalente al **WHERE** pero aplicado al bloque **GROUP BY**. Un ejemplo es el siguiente

```
SELECT
    first_name, COUNT(first_name) AS name_count
FROM
    employees
GROUP BY first_name
HAVING COUNT(first_name) > 280
ORDER BY first_name
```

Es usual encontrarse en situaciones en las que no es claro la diferencia de usar **HAVING** y **WHERE**. La cláusula **WHERE**

1. Filtrado de Filas: **WHERE** se utiliza para filtrar filas individuales antes de que se realice cualquier agrupación de datos (**GROUP BY**).
2. Operaciones en Datos Crudos: Funciona directamente sobre los datos crudos (raw data) de las tablas. Por ejemplo, puedes usar **WHERE** para filtrar registros basados en condiciones específicas de las columnas de la tabla
3. No Aplicable a Funciones de Agregación: No puedes usar **WHERE** para filtrar basándote en el resultado de una función de agregación como **SUM()**, **AVG()**, etc.

La cláusula **HAVING**

1. Filtrado de Grupos: **HAVING** se utiliza para filtrar grupos creados por la cláusula **GROUP BY**.
2. Operaciones en Datos Agrupados: Funciona sobre el resultado de las funciones de agregación. Es útil cuando necesitas aplicar condiciones a un conjunto agrupado de registros.
3. Uso Posterior a **GROUP BY**: Se usa después de **GROUP BY** para filtrar grupos según una condición de agregación.

Un ejemplo en el que se puede usar ambos: Extraer una lista de todos los nombres que se encuentren menos que 200 veces. Los datos se deben referir a las personas que fueron contratadas después de primero de enero de 1999.

```
SELECT
    first_name, COUNT(first_name) AS names_count
FROM
    employees
WHERE
    hire_date > '1999-01-01'
GROUP BY first_name
HAVING COUNT(first_name) < 200
ORDER BY first_name DESC;
```

Por último, se puede limitar el número de resultados mediante la cláusula `LIMIT`.

```
SELECT *
FROM salaires
ORDER BY salary DESC
LIMIT 10;
```

### 3.2.9.3 Inserción de datos `INSERT`

Recordemos la sintaxis para insertar datos nuevos:

```
INSERT INTO table_name (col_1, ..., col_n)
VALUES (val_1, ..., val_n)
```

Es posible no especificar la lista de columnas que se insertan, pero en este caso es necesario siempre poner todos los valores (es decir, no omitir ninguno) y agregarlos en el mismo orden en que salen en la tabla.

Utilizando la palabra `INTO` se pueden agregar datos de columnas de una tabla en otra, basándose en alguna condición si es necesario. Puede ser útil en la creación de tablas duplicadas para realizar algún tipo de prueba.

```
INSERT INTO table_2 (col_1, ..., col_n)
SELECT col_1, ..., col_n
FROM table_1
WHERE condition;
```

### 3.2.9.4 Declaración de actualización

Como introducción, el concepto de Transaction Control Language (TCL) se refiere a un conjunto de instrucciones en SQL que se utilizan para manejar las transacciones en una base de datos. Una transacción es una secuencia de operaciones de base de datos que se tratan como una unidad única de trabajo. Estas operaciones deben cumplir con las propiedades ACID (Atomicidad, Consistencia, Aislamiento, Durabilidad) para asegurar la integridad y confiabilidad de la base de datos. Las instrucciones TCL permiten controlar estas transacciones para mantener la integridad de los datos y gestionar la concurrencia. Los componentes principales de TCL son:

1. **COMMIT:** Esta sentencia se utiliza para guardar permanentemente los cambios realizados por las transacciones en la base de datos. Una vez que se ejecuta `COMMIT`, los cambios realizados por la transacción se hacen permanentes y visibles para otras transacciones.
2. **ROLLBACK** Este comando deshace todas las operaciones realizadas en la transacción actual y devuelve los datos a su estado anterior. `ROLLBACK` se utiliza en situaciones donde se detecta un error o se necesita cancelar una transacción antes de hacer los cambios permanentes con `COMMIT`.



Evidentemente, con la declaración `UPDATE` se puede actualizar cualquier entrada de cualquier tabla de la base de datos. La sintaxis es

```
UPDATE table_name
SET col_1 = val_1, ..., col_n = val_n,
WHERE conditions
```

### 3.2.9.5 Declaración de borrado

su sintaxis es la siguiente:

```
DELETE FROM table_name
WHERE conditions
```

Es muy importante siempre tener en cuenta la relación de conexiones que existen entre dos o más tablas, pues borrar una entrada en una tabla, puede ocasionar la eliminación de sus correspondientes filas en otras tablas que estén relacionadas con ella. Esto se puede verificar en las propiedades de tabla (DDL), si existe la propiedad `ON DELETE CASCADE`.

`DROP`, `TRUNCATE`, y `DELETE` son comandos en SQL utilizados para eliminar datos, pero difieren en su funcionalidad y efecto. `DROP` elimina completamente una tabla de la base de datos, borrando tanto su estructura como sus datos, y no se puede deshacer, lo que lo convierte en la opción más drástica. `TRUNCATE` también elimina todos los datos de una tabla, pero a diferencia de `DROP`, mantiene la estructura de la tabla; es más rápido que `DELETE` y reinicia cualquier contador de identidad, pero no permite condiciones y es, en general, irreversible. `DELETE`, por otro lado, es más flexible ya que permite especificar condiciones para seleccionar qué filas eliminar, mantiene la integridad de los datos y es reversible con `ROLLBACK` si se usa dentro de una transacción, aunque es más lento comparado con `TRUNCATE` para eliminar grandes cantidades de datos.

### 3.2.9.6 Funciones de agregación

Estas funciones toman la información que está contenida en varias filas de una tabla y retorna o devuelve un solo valor.

1. **COUNT**: Determina el número de filas que cumplen un criterio específico. Por ejemplo, `COUNT(*)` cuenta todas las filas en una tabla, mientras que `COUNT(column_name)` cuenta las filas donde la columna especificada no tiene un valor `NULL`.
2. **SUM**: Calcula la suma total de una columna numérica. Es útil para obtener el total de valores, como el total de ventas en una tabla de transacciones.
3. **MIN**: Encuentra el valor mínimo en una columna dada. Esta función es empleada para identificar el valor más bajo en un conjunto de datos, como el precio más bajo de un producto.
4. **MAX**: Obtiene el valor máximo en una columna. Se usa para determinar el valor más alto en un conjunto de datos, como el salario más alto entre los empleados.
5. **AVG**: Calcula el valor promedio de una columna numérica. Esta función es útil para encontrar la media de valores, como el promedio de precios de productos.
6. **ROUND**: Redondea un número a un número específico de decimales. Es comúnmente usada para formatear la salida de los cálculos numéricos, especialmente en los datos financieros.
7. **COALESCE**: Devuelve el primer valor no `NULL` de una lista de expresiones. Es útil para manejar valores `NULL`, proporcionando una forma de definir valores predeterminados.
8. **IFNULL**: Evalúa dos expresiones y devuelve la primera si no es `NULL`; de lo contrario, devuelve la segunda. Esta función es particularmente útil para tratar con valores `NULL`, permitiendo definir un valor de reemplazo cuando un campo es `NULL`.

Una de las anteriores que pueden causar alguna confusión es **COALESCE**, esta función devuelve el primer valor no nulo en una lista de argumentos. Como ejemplo, sea una tabla de empleados con columnas **ID\_empleado**, **Nombre**, **Telefono**, **Email**. En algunos casos, los empleados no han proporcionado su número de teléfono o email, por lo que estas columnas pueden contener valores nulos. Supóngase que se desea generar una lista de contactos de los empleados prefiriendo usar el teléfono, pero en caso de no estar disponible, usar el email. Es aquí donde **COALESCE** resulta útil.

```
SELECT
    ID_empleado,
    Nombre,
    COALESCE(Telefono, Email) AS contacto,
FROM
    Empleados;
```

Esta consulta intentará primero mostrar el teléfono, pero en caso de que este valor sea nulo, entonces intentará mostrar el email. En caso de que ambos estén nulos, el resultado también será nulo. La diferencia entre **COALESCE()** y **IFNULL()** en SQL radica en su funcionalidad y flexibilidad al manejar valores nulos. Aunque ambas funciones se utilizan para lidiar con valores nulos, tienen diferencias clave en su comportamiento y uso. **COALESCE()** es más flexible y se adhiere al estándar SQL, siendo útil para múltiples argumentos, mientras que **IFNULL()** es una función más limitada y específica de MySQL, adecuada para situaciones simples de dos valores. La elección entre las dos dependerá de tus necesidades específicas y del sistema de base de datos que estés utilizando.

### 3.2.10 Uniones

La unión es la herramienta de SQL que permite construir una relación entre las tablas. La mejor herramienta que ayuda en la búsqueda de estrategias para enlazar las tablas es el esquema relacional. Como resultado, una junta muestra un conjunto de resultados que contienen los campos derivados de dos o más tablas.

La sintaxis para la solicitud **JOIN** es la siguiente:

```
SELECT
    table_1.column_name(s), table_2.column_name(s)
FROM
    table_1
JOIN
    table_2 ON table_1.column_name = table_2.column_name;
```

En el primer **SELECT** se deben especificar las columnas que queremos extraer, las que queremos ver. Una forma abreviada y muy útil para este comando es:

```
SELECT
    t1.column_name(s), t2.column_name(s)
FROM
    table_1 t1
JOIN
    table_2 t2 ON t1.column_name = t2.column_name;
```

Como consejo general, el nombre de la columna de la tabla 2 se refiere a la tabla referencia o 'Lookup'. La siguiente forma también es válida y se usa cuando se quiere primero asignar los alias de las tablas

```
FROM dept_manager_dup m
INNER JOIN
departments_dup d ON m.dept_no = d.dept_no
```

Anteriormente, para realizar junturas en las tablas y por ende, relaciones entre ellas, se usaba la cláusula `WHERE`:

```
SELECT
    t1.col1, t2.col2, t3.col3
FROM
    table1 t1
    table2 t2
WHERE
    t1.col_name = t2.col_name
```

Actualmente, realizar uniones de esta manera está desaconsejado, pues consume más procesamiento y por ende más tiempo.

### 3.2.10.1 Entradas duplicadas

Cuando hay presencia de filas que tienen exactamente los mismos valores para todas las columnas. Generalmente, no se permiten filas duplicadas en una base de datos o tabla de datos, es una práctica común para muchas empresas limpiar sus datos de tales ocurrencias. Sin embargo, pueden encontrarse, especialmente en datos nuevos, sin procesar o no controlados. Para manejar duplicados y evitar filas duplicadas en los resultados de las consultas, se recomienda agrupar por el campo que más difiere entre los registros. En este ejemplo, sería el número de empleado. Este enfoque apilará todas las filas que tengan el mismo número de empleado, devolviendo la salida inicial sin valores duplicados. Se resalta la importancia de esta herramienta, especialmente en bases de datos con millones de filas, donde no se debe asumir la ausencia de filas duplicadas y se aconseja acostumbrarse a agrupar las uniones por el campo que más varía entre los registros.

### 3.2.10.2 Función `LEFT JOIN`

La función `LEFT JOIN` recupera todas las filas correspondientes a la tabla derecha (la segunda tabla). Si no hay coincidencias entre las filas de la tabla izquierda y la derecha, el resultado aún incluirá todas las filas de la tabla izquierda, pero con valores `NULL` en las columnas de la tabla derecha. Recordemos que la tabla derecha es hacia donde relacionamos (puede ser la tabla Lookup)

Esta función es particularmente útil cuando se necesita incluir todos los registros de una tabla, independientemente de si tienen una coincidencia en la otra tabla. Por ejemplo, en una consulta que une una tabla de empleados (tabla izquierda) mediante `LEFT JOIN`, se obtendrían todos los empleados incluyendo aquellos que no estén asignados a ningún departamento. Las columnas correspondientes a la tabla de departamentos mostrarían un `NULL` para estos empleados sin departamento asignado.

Por su parte, tenemos la función `RIGHT JOIN`. Se trata de la función análoga a la anterior, mostrando todos los resultados de la tabla de la derecha.

Cuando en una solicitud SQL se usan tanto `JOIN` como `WHERE`, esta última es utilizada para definir la condición o condiciones que determinarán cuáles van a ser los puntos de conexión entre las dos tablas

### 3.2.10.3 Uniones cruzadas

Las uniones cruzadas van a tomar los valores de cierta tabla y conectarlos con todos los valores de las tablas con las que se requiere realizar la unión. La función `'CROSS JOIN'` en SQL se utiliza para realizar un producto cartesiano entre dos tablas, es decir, combina cada fila de la primera tabla con cada fila de la segunda tabla. A diferencia de otros tipos de `JOIN` como `'INNER JOIN'` o `'LEFT JOIN'`, `'CROSS JOIN'` no requiere una condición de unión explícita, ya que simplemente empareja cada fila de una tabla con todas las filas de la otra tabla.

El resultado de un `'CROSS JOIN'` es una tabla que contiene todas las posibles combinaciones de filas entre las dos tablas originales. Si la primera tabla tiene  $N$  filas y la segunda tabla tiene  $M$  filas, entonces el resultado del `'CROSS JOIN'` tendrá  $N \times M$  filas.

Por ejemplo, si se realiza un `'CROSS JOIN'` entre una tabla de `'Productos'` y una tabla de `'Vendedores'`, el resultado incluirá todas las combinaciones posibles de productos y vendedores, lo cual puede ser útil para análisis que requieren considerar todas las posibles asociaciones entre dos conjuntos de datos.

‘CROSS JOIN’ se utiliza en situaciones específicas donde se necesita explorar o analizar todas las combinaciones posibles entre dos conjuntos de datos. Sin embargo, debido a que puede generar un gran número de filas, su uso debe ser cuidadoso y bien justificado. La sintaxis sería la siguiente

```
SELECT
    dm.*, d.*
FROM
    dept_manager dm
    CROSS JOIN
    departments d
ORDER BY dm.emp_no , d.dept_no;
```

## Chapter 4

# Essentials for Cibersecurity

La guía de manejo de incidentes en la seguridad de sistemas NIST 800-61 indica un proceso de respuesta ante incidentes.

El primer paso para cualquier respuesta a incidentes es la preparación. Es muy importante tener la seguridad y la certeza de cómo se debe actuar ante un incidente. Es necesario tener muy bien establecido un proceso de respuesta par acad atipode incidente. Una vez se presenta el incidente, el siguiente paso para el manejo es la detección y el análisis; cosas como determinar el nivel de criticidad del incidente, cuál es el tipo de impacto que potencialmente se tiene con este incidente, si se trata de un positivo real o un falso positivo. El siguiente paso es la contención, erradicación y recuperación; la contención implica el aislamiento, de ser posible, de los dispositivos afectados. Es necesario a veces volver al paso anterior de análisis y detección, porque dentro de la mitigación se pueden determinar nuevos elementos no detectados anteriormente y ser analizados. Finalmente el último paso es la actividad post-incidente: esta está estrechamente relacionada con el primer paso de preparación porque en este paso se obtienen nuevos elementos y lecciones que se implementan en la preparación par evitar nuevos incidentes.

### 4.0.1 Herramientas de gestión

Es importante mencionar las herramientas más implementadas dentro de un equipo de operaciones de seguridad.

#### 4.0.1.1 EDR

El significado de EDR es (Endpoint Detection and Response). Es una tecnología dieñada para monitorear continuamente los endpoint (que son dispositivos finales como computadoras, teléfonos y dispositivos IoT) con el objetivo de realizar detección y respuesta ante diferentes incidentes de seguridad. Es una tecnología que recopila datos de actividad de los enspoints, los analiza para identificar patrones de amenazas y si se detecta una amenaza, responder automáticamente en tiempo real para contener o eliminar la amenaza y notificar al personal de seguridad. Las principales funcionalidades que puede desempeñar un EDR son las siguientes:

- Monitoreo y análisis en tiempo real
- Detección de amenazas
- Respuesta automática
- Análisis forense

#### 4.0.1.2 XDR

El XDR significa Respuesta y detección extendida. La principar diferencia con EDR es su alcance y capacidad de integración. EDR se centra exclusivamente en la protección y respuesta a incidentes en los endpoints. XDR amplía el enfoque al integrar la detección y erspuesta a través de útiples coponentes de la infraestructura, incluyendo endpoints, redes, servidores, aplicaciones den la nube,

etc. Con un análisis enfocado en la correlación de datos y análisis de eventos en toda la infraestructura, y una respuesta coordinada y automatizada.

#### **4.0.1.3 SIEM**

Security Information and Event Management. Es un sistema centralizado cuyo objetivo es recolectar, correlacionar y analizar todos los datos como logs y eventos de todos los incidentes de seguridad. Se centra en la agregación de datos y posterior análisis. Esta tecnología implementa reglas predefinidas y personalizables para correlacionar eventos y detectar posibles incidentes de seguridad. Genera alertas en tiempo real cuando se detectan eventos que coinciden con las reglas de correlación permitiendo una respuesta rápida a incidentes.

#### **4.0.1.4 SOAR**

Es una tecnología cuyo objetivo es implementar automatizaciones y mejorar la eficiencia y eficacia de las operaciones. Permite la integración de múltiples herramientas y sistemas de seguridad, facilitando la coordinación de flujos de trabajo de seguridad a través de diferentes sistemas y equipos. Implementa la automatización de procesos repetitivos y rutinarios, tales como la recopilación de datos, análisis de eventos y respuesta inicial a incidentes.

Normalmente los equipos de SOC en grandes empresas se dividen en dos sub equipos más pequeños: Equipo azul y equipo Rojo. El equipo azul se encarga mayormente de realizar las tareas más típicas de analista de SOC: monitoreo de seguridad, respuesta a incidentes, análisis forense, seguimiento de amenazas, etc. Por otro lado, el equipo Rojo está más enfocado en la evaluación de vulnerabilidades, testeos de penetración, ingeniería social, y en general, simulación de procedimientos, técnicas y tácticas de adversarios.

## Chapter 5

# Docker

Empezamos dando una definición de qué es docker. En un proyecto que tiene utiliza tantas tecnologías y diferentes, cada una con sus dependencias y librerías necesarias, es difícil poder realizar despliegues y ejecuciones de los mismos de manera estandarizada cuando cambiamos de plataforma. Docker entrega una solución a esto. Se trata de una plataforma que permite configurar contenedores y empaquetar toda la aplicación y sus dependencias y librerías de forma aislada para poder ser ejecutada en cualquier entorno o máquina. Dicho empaquetamiento se realiza en lo que se conoce como contenedor.

Un contenedor es una unidad estándar de software que empaqueta el código de una aplicación y todas sus dependencias para que esta pueda ejecutarse de manera rápida y confiable en diferentes entornos. A diferencia de una máquina virtual, un contenedor no incluye un sistema operativo, por tanto son más ligeros y rápidos.

Docker puede configurar también algo llamado imagen. Una imagen no es más que un paquete que funciona como plantilla con la que se pueden crear uno o más contenedores. Por tanto, un contenedor es una instancia de la imagen que ha sido aislada y tiene su propio entorno y conjunto de dependencias

### 5.1 Comandos de Docker

El primer comando a aprender es el de **run**. Este comando ejecuta un contenedor a partir de una imagen.

**docker run nginx** ejecuta e instancia un contenedor con base en la imagen **nginx**. Docker buscará la imagen de manera local, y si no la encuentra, entonces buscará en el docker hub y descargará esta imagen para ejecutarla.

**docker ps** Lista todos los contenedores que están siendo ejecutados. Imprime también alguna información como el ID del contenedor, el nombre de la imagen de la que está instanciado el contenedor, el estado actual y el nombre del contenedor. En cada ejecución, docker asigna un identificador ID y nombre al contenedor de manera aleatoria.

**docker ps -a** lista todos los contenedores que han sido ejecutados, aunque hayan parado ya. **docker stop <container\_name>** Con este comando detenemos el contenedor seleccionado.

**docker rm <container\_name>** Con este comando quitamos o removemos el contenedor. De este modo ya no aparecerá en la lista de contenedores anteriores. **docker images** Con este comando se imprime una lista de las imágenes disponibles en el dispositivo local. **docker rmi name** Este comando se utiliza para eliminar o remover una imagen que ya no es necesaria. Es importante tener en cuenta que no deben haber contenedores ejecutándose, o en la lista de anteriormente ejecutados al momento de intentar eliminar su imagen. **docker pull name** Este comando sirve para 'halar' (descargar) una imagen dada, pero sin ejecutar un contenedor a partir de dicha imagen.

Es importante notar que un contenedor solo está ejecutándose tanto tiempo como su proceso interno.

Hay un comando que sirve para ejecutar otros comandos dentro del contenedor que se está ejecutando. Sea un contenedor **practical.taussig**, que es una instancia de la imagen de ubuntu. Se puede ejecutar un comando dentro de dicho contenedor, si ejecutamos el contenedor de tal forma que esté activo durante 100 segundos

```
docker run ubuntu sleep 100
```

Luego el comando para ejecutar cualquier comando dentro de mi contenedor es

```
docker exec practical_taussig ls -la
```

De esta manera se ejecutará el comando `ls -la` dentro del contenedor que se está ejecutando.

Una opción para poder ejecutar un contenedor y quedarse en segundo plano es mediante la opción 'detached' `-d`

```
docker run -d kodekloud/simple-webapp
```

De esta forma la terminal seguirá disponible para ejecutar otros comandos. Si dentro de la terminal quiero adjuntar este contenedor para ver su estado actual, simplemente ejecutamos el comando

```
docker attach name
```

Existe una combinación de opciones para ejecutar un contenedor que tenga la imagen de centos por ejemplo, y permita interactuar con su terminal:

```
docker run -it centos bash
```

La opción `-it` es una combinación de `-i` e `-t`. La primera, indica interactivo, lo cual indica a Docker que mantenga la entrada estándar 'stdin' abierta, de modo que se pueda interactuar de manera activa con el contenedor. Por su parte, la opción `-t` indica a Docker que asigne un terminal pseudo-TTY al contenedor. und TTY es lo que proporciona una interfaz de terminal dentro del contenedor.

Para ejecutar un contenedor desde una imagen y darle un nombre específico, se usa el siguiente comando:

```
docker run --name mycontainer image
```

Desde la terminal, el comando para ingresar las credenciales de docker hub es el siguiente

```
docker login
```

Por otro lado, es posible ejecutar contenedores desde versiones anteriores de imágenes. Por ejemplo, si quiero ejecutar una versión anterior de **redis** utilizamos

```
docker run redis:4.0
```

La parte `:4.0` se denomina etiqueta. Si no se utiliza etiqueta, por defecto Docker utiliza la última versión. En la página web de docker hub, están todas las etiquetas soportadas para cada imagen.

### 5.1.1 Mapeos de puertos

En docker, cuando se inicia un contenedor que ejecuta un servicio que escucha un puerto, este puerto está por defecto solo accesible dentro del contenedor. Para que este puerto sea accesible desde el host o desde una red externa, es necesario mapearlo a un puerto del host:

```
docker run -p 8080:5000 mysql
```

Este ejemplo se ejecuta un contenedor con la imagen de sql cuyo puerto para comunicación con el host de docker es **8080**, y el puerto de la instancia del contenedor **mysql** es **5000**. la opción `-p` indica la configuración del puerto. Dentro de un host de docker, pueden haber diferentes contenedores instanciados. Cada uno tendrá su respectivo puerto y siempre será único.



### 5.1.2 Persistencia de datos

Sea una instancia de **mySQL** `docker run mysql`. Los datos cualesquiera dentro de esta base de datos estarán dentro del sistema de archivos del host de docker en `/var/lib/mysql`. Si se detiene la instancia del contenedor, entonces todos los datos presentes en la base también desaparecerán. Para solucionar esto, se puede mantener una persistencia de los datos desde una localización fuera del contenedor pero dentro del host.

```
docker run -v /opt/datadir:/var/lib/mysql mysql
```

Con este comando, la opción `-v` monta un volumen virtual en el contenedor. La dirección `/opt/datadir` es el volumen creado dentro del host (la máquina).

Se puede inspeccionar un contenedor, averiguando detalles y datos característicos en formato **JSON** mediante el siguiente comando

```
docker inspect nameorID
```

Por otro lado, también se pueden ver los logs de un contenedor aunque esté ejecutándose en segundo plano:

```
docker logs nameorID
```