

Introducción a Bases de Datos

DigitalHouse >
Coding School



Certified Tech
Developer
The Ultimate Degree

Índice

1. Modelado de datos
2. Modelo físico vs Modelo lógico
3. Extra: Cliente/Servidor

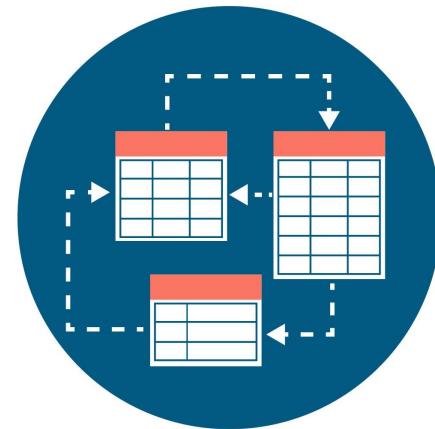
1

Modelado de datos

Modelado de datos

Un modelo es un conjunto de herramientas conceptuales para describir datos, sus relaciones, su significado y sus restricciones de consistencia.

El modelado de datos es una manera de estructurar y organizar los datos para que se puedan utilizar fácilmente por las bases de datos.



Beneficios

- Registrar los requerimientos de datos de un proceso de negocio.
- Se puede descomponer un proceso complejo en partes.
- Permite observar patrones.
- Sirve de plano para construir la base de datos física.
- El modelo de datos ayuda a las empresas a comunicarse dentro y entre las organizaciones.
- Proporciona soporte ante los cambios de requerimientos del negocio o aplicaciones.



Tipos de modelos

Existen 3 tipos de modelos que podrían implementarse:

Conceptual

Es un modelo con un diseño muy general y abstracto, cuyo objetivo es explicar la visión general del negocio o sistema.

Lógico

El modelo lógico es una versión completa que incluye todos los detalles acerca de los datos. Explica qué datos son importantes, su semántica, relaciones y restricciones. Explica el “qué”.

Físico

Es un modelo que implementa el modelo lógico. Es un esquema que se va a implementar dentro de un sistema de gestión de bases de datos. Explica el “cómo”.

Nosotros vamos a hacer énfasis en el modelo lógico para entender cómo implementarlo dentro del modelo físico.

2

Modelo lógico vs Modelo físico

Modelo lógico

Un **modelo de datos lógico** describe los datos con el mayor detalle posible, independientemente de cómo se implementarán físicamente en la base de datos.

Describe los elementos importantes del negocio, qué significa cada objeto, su nivel de detalle, cómo se relacionan entre sí y sus restricciones.

Modelo lógico

Las características de un modelo de datos lógicos incluyen:

- Se definen cuáles son los conceptos importantes sobre los que hay que almacenar información. Estos elementos se denominan **entidades**.
- Se especifican todos los **atributos** para cada una de las entidades.
- Se conectan las entidades mediante **relaciones**.
- Se especifica la clave principal para cada entidad.
- Se especifican las claves externas (claves que identifican la relación entre diferentes entidades).
- La normalización ocurre en este nivel.

Modelo físico

El modelo de datos físicos representa **cómo** se construirá el modelo en la base de datos.

Un modelo de base de datos física muestra todas las estructuras de tablas, incluidos el nombre de columna, el tipo de datos de columna, las restricciones de columna, la clave principal, la clave externa y las relaciones entre las tablas.

Tipos de modelos

Las características de un modelo de datos físicos incluyen:

- Especificación de todas las tablas y columnas.
- Las claves externas se usan para identificar relaciones entre tablas.
- La desnormalización puede ocurrir según los requisitos del usuario.

Las consideraciones físicas pueden hacer que el modelo de datos físicos sea bastante diferente del modelo de datos lógicos.

El modelo físico puede diferir de un motor de bases de datos a otro —no es lo mismo implementar en Ms. SQL Server que en MySQL—.

Tipos de modelos

Los pasos para el diseño del modelo de datos físicos son los siguientes:

- Convertir entidades en tablas.
- Convertir relaciones en claves externas.
- Convertir atributos en columnas.
- Modificar el modelo de datos físicos en función de las restricciones / requisitos físicos.

Modelo lógico vs Modelo físico

MODELO LÓGICO

- Describe el “qué”.
- Explica el negocio.
- Es independiente de la implementación.
- Responsable: el analista.

MODELO FÍSICO

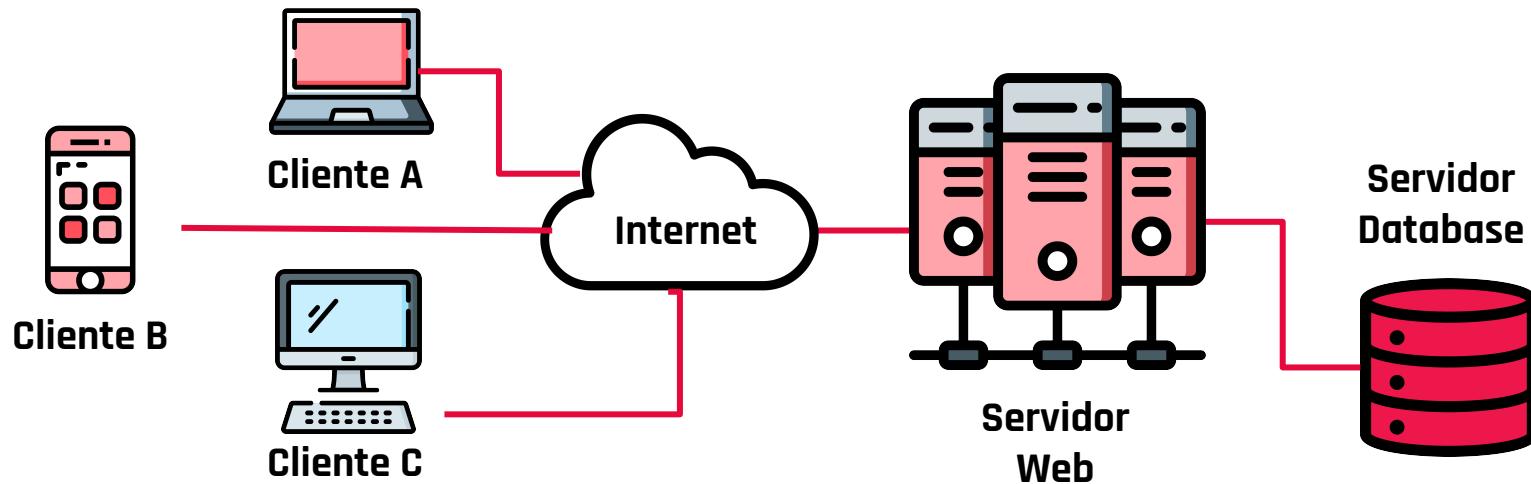
- Describe el “cómo”.
- Explica técnicamente cómo se van a almacenar los datos.
- Explica la implementación en el sistema de gestión de bases de datos.
- Responsable: administrador de bases de datos o simil.

3

Extra: Cliente/Servidor

Cliente/Servidor

Las bases de datos se implementan bajo una **arquitectura Cliente - Servidor**. Esta arquitectura tiene dos partes claramente diferenciadas, por un lado, la parte del **servidor** y, por otro, la parte de **cliente o grupo de clientes**.

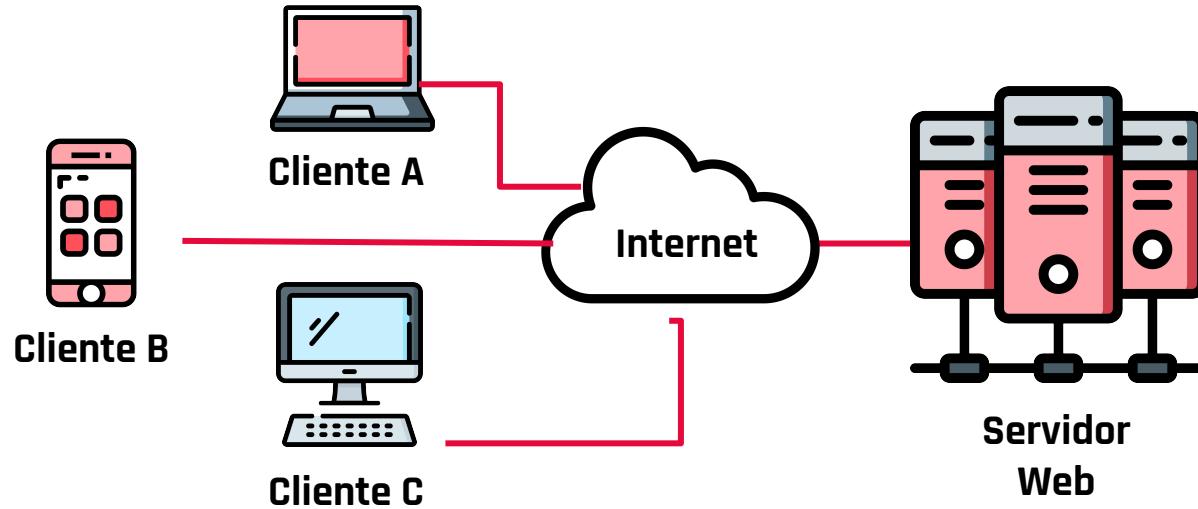


Cliente/Servidor

Esto se realiza así porque:

- El **servidor**, normalmente, es una “máquina” bastante potente con un hardware y software específico que actúa de depósito de datos y funciona como un sistema gestor de base de datos o aplicaciones.
- Los **clientes** suelen ser estaciones de trabajo que solicitan varios servicios al servidor. Son los que necesitan los servicios del servidor. Físicamente se pueden ver como distintas computadoras, celulares y dispositivos que se conectan con el servidor.

Cliente/Servidor



¿Porqué es importante entender la arquitectura?

- Porque el servidor, va a ser el motor de bases de datos.
En nuestro caso, **MySQL Server**.
- Nuestro cliente van a ser las aplicaciones que consulten o almacenen los datos. Por ejemplo, **MySQL Workbench** o, en el futuro, la aplicación que desarrollemos que va a interactuar con la base de datos.

Modelos de bases de datos

DigitalHouse >
Coding School



**Certified Tech
Developer**
The Ultimate Degree

Índice

1. Objetivos
2. Modelos
3. Modelo entidad-relación
4. Entidades
5. Atributos
6. Claves

1 | Objetivos

El objetivo de cualquier sistema de información es representar mediante abstracciones del mundo real toda la información necesaria para el cumplimiento de los fines.

Para describir la estructura de una base de datos es necesario definir el concepto de modelo de datos, es una colección de herramientas conceptuales para describir datos, relaciones entre ellos, semántica asociada a los datos y restricciones de consistencia.



2 | Modelos

Tipos

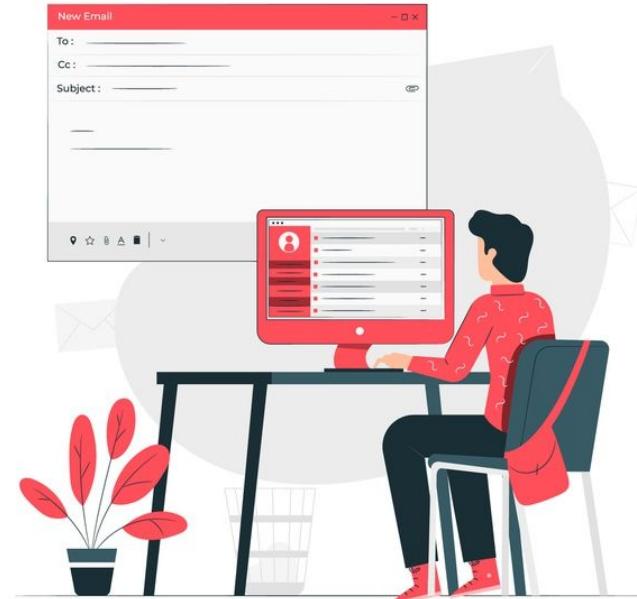
1

Modelo conceptual basado en objetos: Fue definido por **Peter Chen** en 1976. Se utiliza para la representación de la realidad No comprometida con ningún entorno informático: Sería el **Modelo Entidad-Relación** propiamente dicho.



2

Modelo lógico basado en objetos: determinan algunos criterios de almacenamiento y de operaciones de manipulación de los datos dentro de un entorno informático.



3

Modelo entidad-relación

“

El modelo de datos entidad-relación se basa en una percepción del mundo real, que consiste en un conjunto de objetos básicos llamados entidades y de relaciones entre ellos. Se emplea para interpretar, especificar y documentar los requerimientos para los sistemas de bases de datos.



”

Modelo entidad-relación

Por lo tanto un modelo entidad-relación es un método de representación abstracta del mundo real centrado en las restricciones o propiedades lógicas de una bases de datos.



4 | Entidades

“

Una **entidad** es un objeto, real o abstracto, acerca del cual se recoge información de interés para la base de datos.



”

Tipos

- **Entidades fuertes:** tienen existencia por sí mismas.
(alumnos, empleados, departamento.)
- **Entidades débiles:**
dependen de otra entidad para su existencia (hijos de empleados)



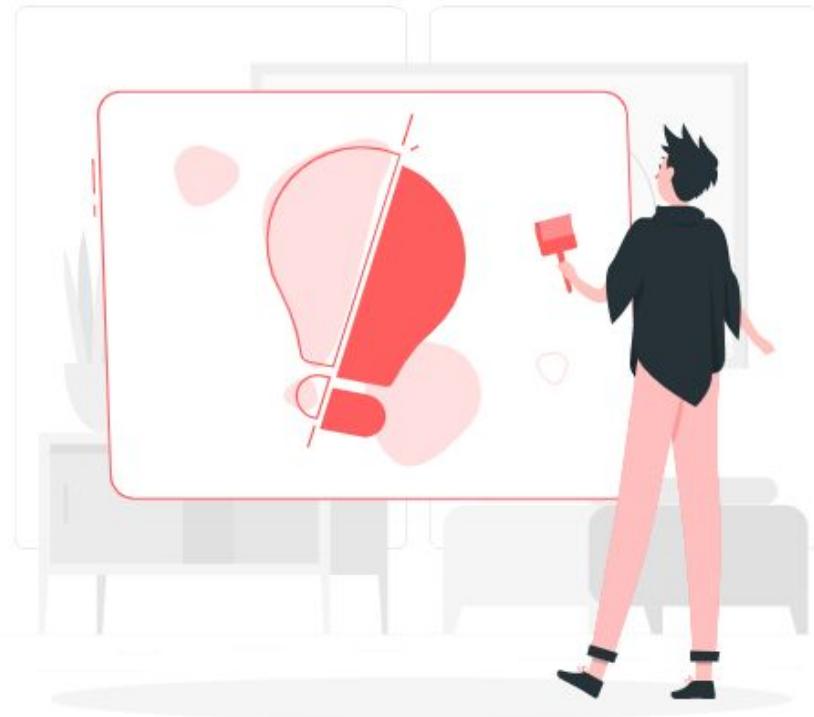
Entidades

Se define como **ocurrencia** de entidad al conjunto de datos para una entidad en particular.

Por ejemplo:

125, Juan Pérez, casado, 23 años.

Cada entidad tiene propiedades particulares llamadas **atributos**.



5 | Atributos

“

Los atributos describen las características de una entidad.

Por ejemplo:

Entidad: clientes

Atributos: legajo, nombre, domicilio, etc.



”

Tipos

1. Atributo con simple valor
2. Atributo multivalor
3. Atributos derivados
4. Atributo clave
5. Atributos nulos



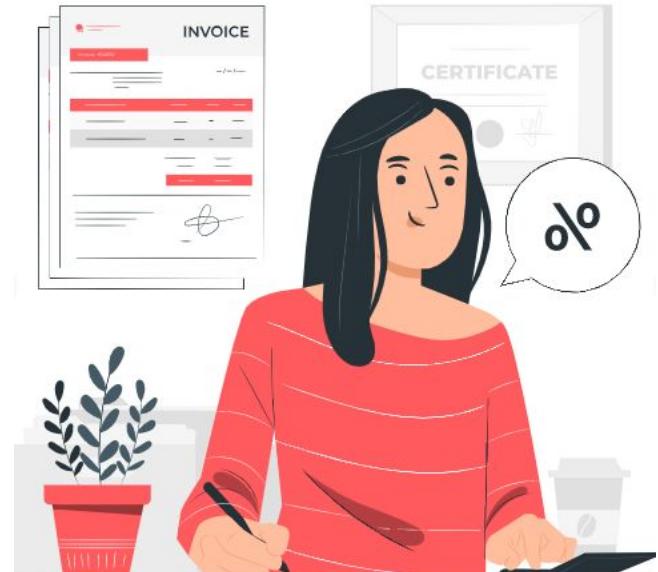
1

Atributo con simple valor:

Cuando un atributo tiene un simple valor para una identidad particular.

Por ejemplo:

Una persona que tiene un valor por su fecha de nacimiento y la fecha de nacimiento es un simple valor de la persona.



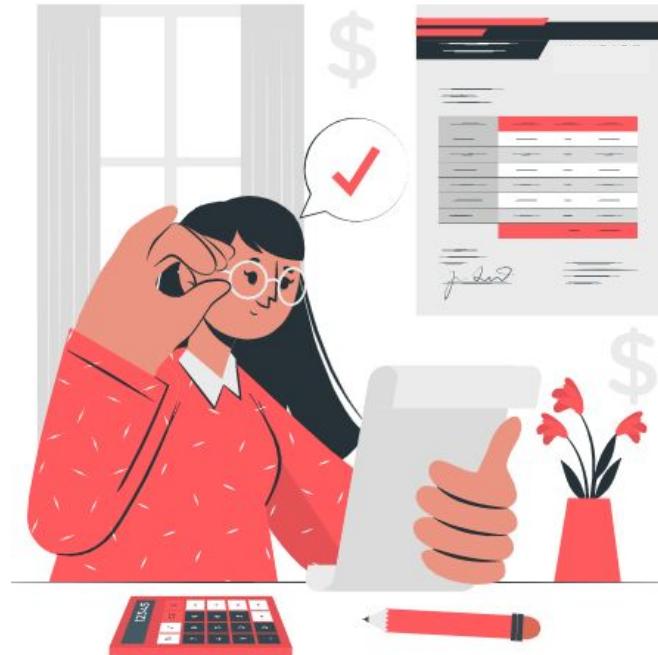
2

Atributo multivalor:

Cuando un atributo tiene una serie de valores para identificarse.

Por ejemplo:

El atributo teléfonos de un cliente que puede contener uno o más números de teléfono.



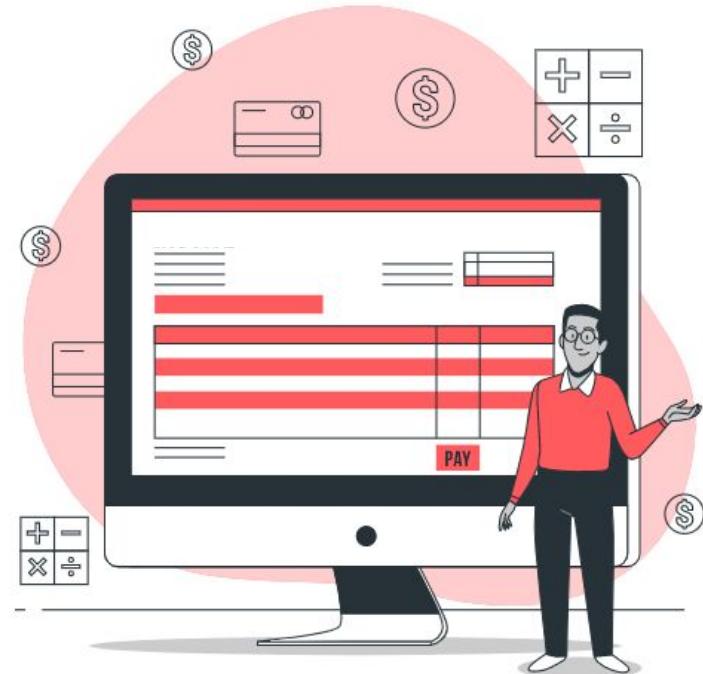
3

Atributos derivados:

Cuando los valores de un atributo son afines y el valor para este tipo de atributo se puede derivar de los valores de otros atributos.

Por ejemplo:

La edad y fecha de nacimiento de una persona; si conocemos la fecha de nacimiento, podemos calcular su edad, en este caso se dice que la edad es un atributo derivado del atributo fecha de nacimiento.



4

Atributo clave:

Las entidades pueden contener un atributo que identifica cada una de las ocurrencias de la entidad. Es decir, usualmente contienen un atributo que diferencia los ítems entre sí.

Por ejemplo:

En la entidad clientes el atributo documento puede ser un atributo clave. No necesariamente el atributo clave debe ser un solo atributo, hay casos en que varios atributos forman una llave. Por ejemplo: tipo más número de factura.

5

Atributos nulos:

Se usa cuando una entidad no tiene valor para un atributo o que el valor es desconocido.

6 | Claves

Tipos

1. Clave candidata
2. Clave primaria
3. Superclave



1

Clave candidata:

Se compone por uno o más atributos cuyos valores **identifican únicamente** a cada **ocurrencia** de la entidad, sin que ningún subconjunto de ellos pueda realizar esta misma función. Una clave candidata es una **posible clave primaria**. Pueden definirse varias claves candidatas para luego seleccionar la más adecuada.



2

Clave primaria:

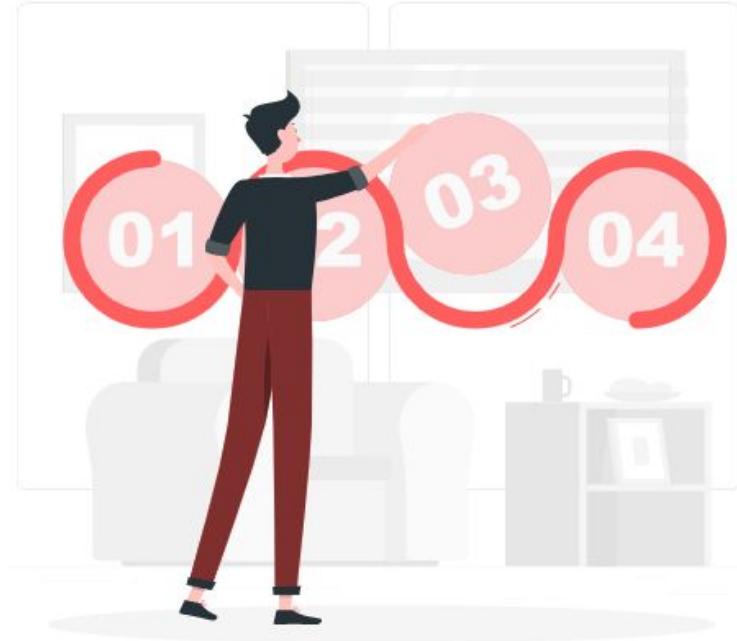
Está compuesta por uno o más atributos cuyos valores **identifican únicamente** a cada **ocurrencia** de la entidad. No pueden contener valores **nulos** ni **repetidos**. Esta clave es una de aquellas que anteriormente se seleccionaron como candidata.



3

Superclave:

es el conjunto de uno o más **atributos** que, tomados **colectivamente**, permiten identificar de forma **única** a la **ocurrencia** de una entidad. Se utiliza generalmente en las **tablas de relación**, este concepto se desarrollará en las próximas clases.



Entidades y atributos

DigitalHouse >
Coding School



**Certified Tech
Developer**
The Ultimate Degree

Entidad

Dentro de nuestro sistema tendremos **entidades** y para integrarlos en nuestro diagrama los representaremos usando un rectángulo. Estas entidades son todos los objetivos sobre los cuales tenemos un interés de almacenar información.

Pelicula

Actor

Genero

Atributos

Son las **características** que van a definir a cada entidad. Por ejemplo, la entidad **Películas** podría tener estos atributos.

Pelicula

titulo
rating
fecha_estreno
premios

Convención de nombres

En los nombres de **entidades** y **atributos** siempre se debe utilizar **sustantivos** en singular o plural. No se puede utilizar **eñes**, **espacios** ni **acentos**. Si el nombre se compone por más de una palabra, se deben reemplazar los espacios con guiones bajos (snake case) o eliminar dicho espacio y colocar una mayúscula en la inicial de cada palabra (camel case).

Ej. Así podríamos **asignar** un **nombre** para la siguiente frase:

- “costos anuales” → costo_anual → costoAnual
- “costos anuales” → costos_anuales → costosAnuales

Convención de nombres

Cabe aclarar que, debido a que MySQL utiliza directorios y archivos para almacenar bases de datos y tablas, los nombres se distinguen entre mayúsculas y minúsculas solo si el sistema operativo posee un sistema de archivo sensible al tipo.

- **Windows:** No distingue entre mayúsculas y minúsculas.
- Algunas versiones de **Unix** y **Linux:** Distinguen entre mayúsculas y minúsculas. (Case Sensitive)

Clave primaria

Una **Clave Primaria** o **Primary Key** es un campo que identifica a cada fila de una tabla de **forma única**. Es decir que No puede haber dos filas en una tabla que tengan la misma **PK**.

id	título	rating	fecha_estreno	premios
1001	Pulp Fiction	9.8	1995-02-16	10
1002	Kill Bill Vol. 1	9.5	2003-11-27	25

En este caso los id de las películas no se pueden repetir.

Para identificar la clave primaria en una entidad, podemos escribir el atributo en negrita seguido de las iniciales **PK** entre paréntesis.

PELÍCULAS

id (PK)

título

rating

fecha_estreno

país

¿Entidades o atributos?

Con algunos atributos vamos a tener la duda si es un atributo o una entidad. Por ejemplo, el teléfono de un cliente. De qué depende:

- Del escenario que se modela.
- La semántica asociada con el atributo en cuestión.

Por ejemplo, para un *call center* es importante registrar todos los posibles teléfonos del cliente. "Teléfonos" es una entidad relevante dentro de su modelado.

Datos

Los **datos** son los **valores** que pueden tener los **atributos**. Nos permiten entender si son datos numéricos, textos, fechas, si tienen un formato en particular, si son obligatorios u opcionales.

No se modelan, pero nos permite entender mejor las entidades:

Peliculas

id	título	rating	fecha_estreno	premio
1001	Pulp Fiction	9.8	1995-02-16	10
1002	Kill Bill Vol. 1	9.5	2003-11-27	25

Tipos de datos

DigitalHouse >
Coding School



**Certified Tech
Developer**
The Ultimate Degree

Índice

1. [Definición](#)
2. [Datos de tipo texto](#)
3. [Datos de tipo numérico](#)
4. [Datos de tipo fecha](#)
5. [Datos de tipo boolean](#)

1

Definición

Tipos de datos

Los datos o atributos de cada registro de una tabla tienen que ser de un tipo de dato concreto.

Cuando diseñamos una base de datos tenemos que pensar qué tipo de datos requerimos para nuestro modelo.

Cada tipo de dato tiene un tamaño determinado y cuanta más precisión apliquemos en su definición, más **rápido** y **performante** va a funcionar MySQL.

Tipo de datos			
Texto	Numérico Entero	Numérico Decimal	Fecha
Juan Pérez	12345	120,50	2021-03-15

2

Datos de tipo texto

Datos de tipo **texto**

Almacenan datos **alfanuméricos** y **símbolos**.

CHAR(*n*)

n → 1 a 255 caracteres.

Se recomienda utilizar en cadenas de texto de **longitud poco variable**.

Ej.: La **longitud** de la palabra “*HOLA*” es **4** y se define: **Char(4)**.

VARCHAR(*n*)

n → 1 a 21.845 caracteres.

Se recomienda utilizar en cadenas de texto de **longitud muy variable**.

Ej.: La **longitud** de la palabra “*HASTA LUEGO*” es **11** y se define: **Varchar(11)**.

Nota: La letra ***n*** indica la **longitud máxima de caracteres** a utilizar.

Datos de tipo **texto**

Almacenan datos **alfanuméricos** y **símbolos**.

TINYTEXT

0 a 255 caracteres.

TEXT

0 a 65,535 caracteres.

MEDIUMTEXT

0 a 16.777.215 caracteres.

LONGTEXT

0 a 4.294.967.295 caracteres.

3

Datos de tipo numérico

Datos de tipo numérico entero

TINYINT

-128 a 127 (sin signo de 0 a 255)

SMALLINT

-32.768 a 32.767 (sin signo de 0 a 65.535)

MEDIUMINT

-8.388.608 a 8.388.607 (sin signo de 0 a 16.777.215)

INT

-2.147.483.648 a 2.147.483.647 (sin signo de 0 a 4.294.967.295)

BIGINT

-9.223.372.036.854.775.808 a 9.223.372.036.854.775.807

(sin signo de 0 a 18.446.744.073.709.551.615)

Datos de tipo numérico decimal

FLOAT(*n,d*)

Almacenan números de coma flotante pequeño.

Tienen precisión simple para la parte decimal (máx. 7 dígitos).

n → 1 a 24 dígitos (incluyendo la parte decimal).

d → 0 a 7 dígitos dependiendo de cuánto se asigne en *n*.

DOUBLE(*n,d*)

Almacenan números de coma flotante grande.

Tienen precisión doble para la parte decimal (máx. 15 dígitos).

n → 25 a 53 dígitos (incluyendo la parte decimal).

d → 0 a 15 dígitos dependiendo de cuánto se asigne en *n*.

Nota: Las letras *n* y *d* indican la **longitud máxima de dígitos** a utilizar.

Datos de tipo numérico decimal

FLOAT(24,7)

12345678901234567,1234567
|—————||—————|
Parte entera (17 dígitos) Parte decimal (7 dígitos)

DOUBLE(53,15)

12345678901234567890123456789012345678,123456789012345
|—————||—————|
Parte entera (38 dígitos) Parte decimal (15 dígitos)

4 | Datos de tipo fecha

Datos de tipo **fecha**

A la hora de almacenar fechas, hay que tener en cuenta que MySQL no comprueba de una manera estricta si una fecha es válida o no.

DATE

Almacena solamente la fecha en formato **YYYY-MM-DD**.

Valores permitidos: '0001-01-01' a '9999-12-31'.

La fecha se debe colocar entre **comillas** simples o dobles y se separa por **guiones**. Ejemplo: '2021-05-15'.

Datos de tipo **fecha**

TIME

Almacena solamente la hora en formato **HH:MM:SS**.

Valores permitidos: '00:00:00' a '23:59:59'.

La hora se debe colocar entre **comillas** simples o dobles y se separa por **dos puntos**. Ejemplo: '11:50:55'.

DATETIME

Almacena la fecha y hora en formato **YYYY-MM-DD HH:MM:SS**.

Valores permitidos: '0001-01-01 00:00:00' a '23:59:59 9999-12-31'.

La fecha se debe colocar entre **comillas** simples o dobles, un espacio y la hora que se debe separar por **dos puntos**. Ejemplo: '2021-05-15 11:50:55'.

5 | Datos de tipo boolean

Datos de tipo **boolean**

BOOLEAN

MySQL guarda los booleanos como un **cero** o como un **uno**. Por cuestiones de performance, este tipo de dato viene desactivado y no se recomienda su uso.

En el caso de querer guardar valores "verdaderos" y "falsos", se recomienda utilizar el tipo de dato **TINYINT**, donde:

- **0** para representar el **false**.
- **1** para representar el **true**.

Relaciones

DigitalHouse >
Coding School



**Certified Tech
Developer**
The Ultimate Degree

Cardinalidad

Es la forma en que se relacionan las entidades.

Cardinalidad	Se lee	Representación	
1:1	Uno a uno	+	—————+—————
1:M	Uno a muchos	+	—————+—————
M:1	Muchos a uno	—————+————— 	
N:M	Muchos a muchos	—————+————— 	



Muchas veces vemos las notaciones como 1 a N o 1 a M. Son iguales, la letra se utiliza para representar "muchos".

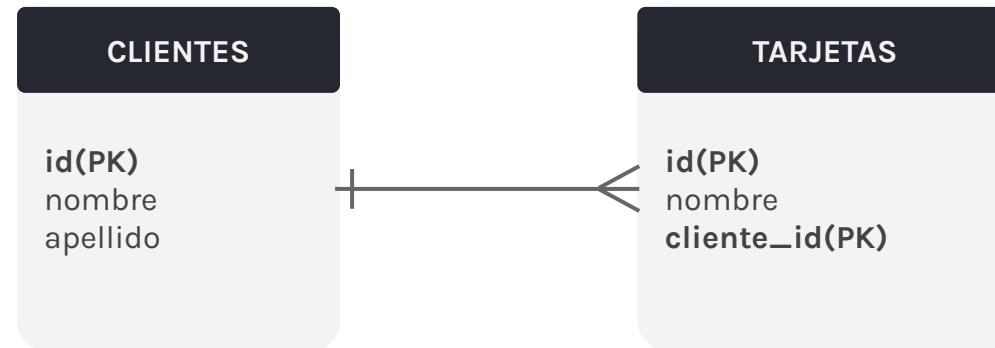
Uno a uno (1:1)

Un usuario **tiene** solo una dirección. Una dirección **pertenecce** solo a un **usuario**. Para establecer la relación colocamos la **clave primaria** de la dirección en la tabla de **usuarios**, indicando que **esa** dirección está asociada a **ese** usuario (clave foránea).



Uno a muchos (1:N)

Un **cliente** puede tener muchas **tarjetas**. Una **tarjeta** pertenece solo a **un cliente**. Para establecer la relación colocamos la **clave primaria** del **cliente** en la tabla de **tarjetas**, indicando que **esas** tarjetas están asociadas a un usuario en particular.



Muchos a muchos (N:M)

Un **cliente** puede comprar **muchos productos**. Un **producto** puede ser comprado por **muchos clientes**. En las relaciones **N:M**, en la base de datos, la relación en sí pasa a ser una **tabla**. Esta tabla intermedia —también conocida como tabla pivot— puede tener 3 datos: una clave primaria (**PK**) y dos claves foráneas (**FK**), cada una haciendo referencia a cada tabla de la relación.



Muchos a muchos (N:M)

En este ejemplo, **cliente_producto** sería nuestra tabla intermedia. Cada fila de esta tabla representa un cruce entre cliente y producto. Podría ser, en este caso, una compra:

- La fila 1 indica que el **cliente 1** (Juan) compró el **producto 1** (Pelota).
- La fila 2 indica que **Juan** también compró el **producto 2** (Laptop).
- La fila 3 indica que una **Laptop** también fue comprada por el cliente 3 (Marta).

CLIENTES

id	nombre	apellido
1	Juan	Pérez
2	Clara	Sánchez
3	Marta	Ríos

CLIENTE_PRODUCTO

id	producto_id	cliente_id
1	1	1
2	2	1
3	2	3

PRODUCTOS

id	nombre
1	Pelota
2	Laptop

Relaciones

DigitalHouse >
Coding School



**Certified Tech
Developer**
The Ultimate Degree

Índice

1. Definición
2. Tipos de relaciones

1

Definición

¿Qué son?

Las relaciones indican cómo se van a relacionar dos tablas. Dentro de una base de datos existen 3 tipos de relaciones:

- **Uno a uno.**
- **Uno a muchos.**
- **Muchos a muchos.**

¿Cómo podemos saber cómo se relaciona una entidad con otra?

Planteando un ejemplo concreto que nos ayude a definir cómo interactúan esas dos entidades entre sí.

Cardinalidad

Es la forma en que se relacionan las entidades.

Cardinalidad	Se lee	Representación
1:1	Uno a uno	+
1:M	Uno a muchos	+
N:M	Muchos a muchos	Y

Nota!

Muchas veces vemos las notaciones como 1 a N o 1 a M. Son iguales, la letra se utiliza para representar "Muchos".

2 | TIPOS DE RELACIONES

Uno a uno (1:1)

Un usuario **tiene** solo una dirección. Una dirección **pertenecce** solo a un **usuario**.

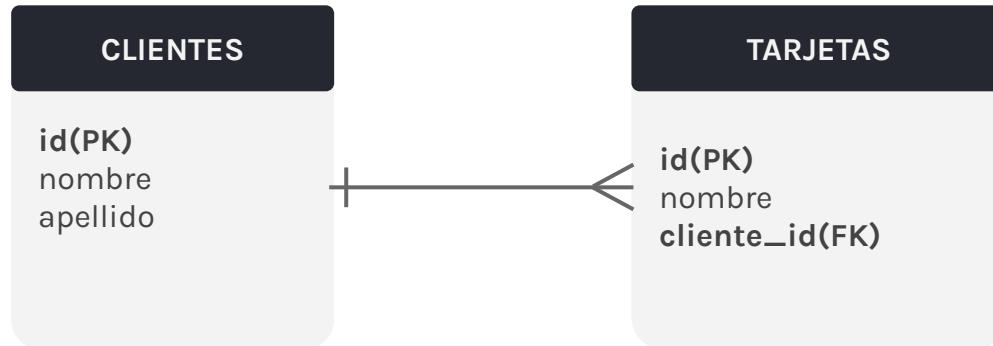
Para establecer la relación colocamos la **clave primaria** de la dirección en la tabla de **usuarios**, indicando que **esa** dirección está asociada a **ese** usuario (Clave foránea).



Uno a muchos (1:N)

Un **cliente** puede tener muchas **tarjetas**. Una **tarjeta** pertenece solo a **un cliente**.

Para establecer la relación colocamos la **clave primaria** del **cliente** en la tabla de **tarjetas**, indicando que **esas** tarjetas están asociadas a un usuario en particular.



Muchos a muchos (N:M)

Un **cliente** puede comprar **muchos productos**. Un **producto** puede ser comprado por **muchos clientes**.

En las relaciones **N:M**, en la base de datos, la relación en sí pasa a ser una **tabla**. Esta tabla intermedia —también conocida como tabla pivot— puede tener 3 datos: una clave primaria (**PK**) y dos claves foráneas (**FK**), cada una haciendo referencia a cada tabla de la relación.



Muchos a muchos (N:M)

En este ejemplo, **cliente_producto** sería nuestra tabla intermedia. Cada fila de esta tabla representa un cruce entre cliente y producto. Podría ser, en este caso, una compra:

- La fila 1 indica que el **cliente 1** (Juan) compró el **producto 1** (Pelota).
- La fila 2 indica que **Juan** también compró el **producto 2** (Laptop).
- La fila 3 indica que una **Laptop** también fue comprada por el cliente 3 (Marta).

CLIENTES

id	nombre	apellido
1	Juan	Perez
2	Clara	Sanchez
3	Marta	Ríos

CLIENTE_PRODUCTO

id	producto_id	cliente_id
1	1	1
2	2	1
3	2	3

PRODUCTOS

id	nombre
1	Pelota
2	Laptop

Cómo Construir una base de datos

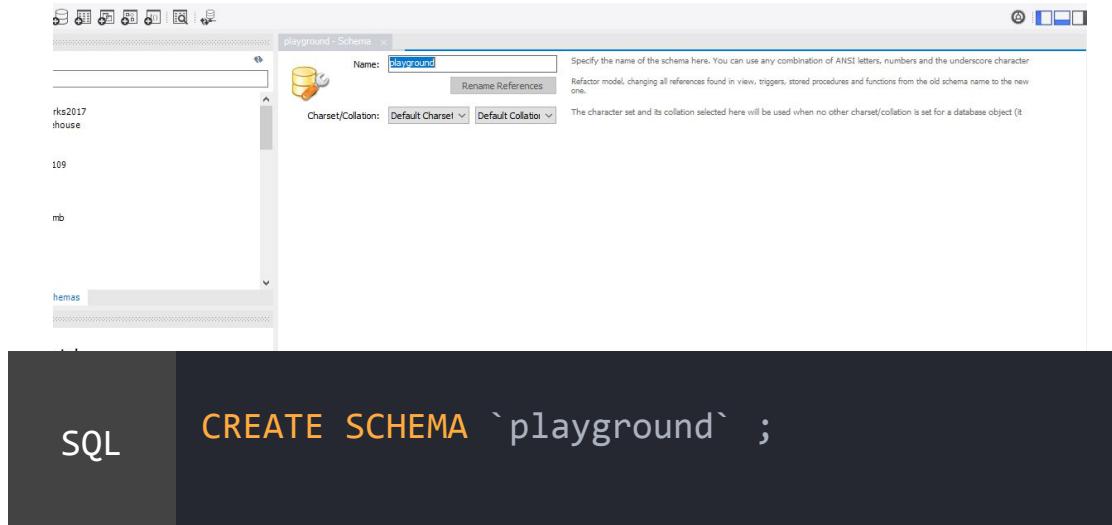
DigitalHouse >
Coding School



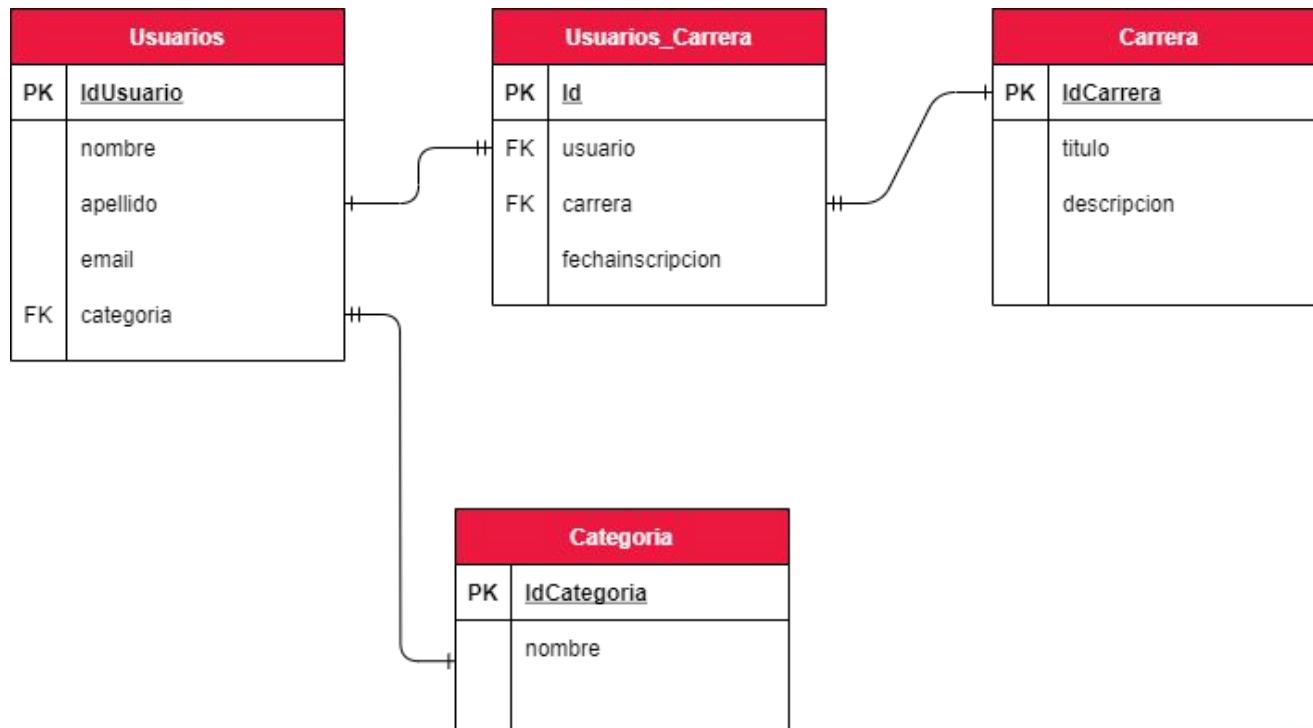
**Certified Tech
Developer**
The Ultimate Degree

¡Empecemos!

Tomando como base el ejercicio de DER Playground realizado, vamos a realizar el paso a paso para **crear una parte de la base de datos**:



DER - Playground



Ejemplo CREATE TABLE “categorias”

SQL

```
CREATE TABLE `playground`.`categorias` (
  `idcategoria` INT NOT NULL,
  `nombre` VARCHAR(100) NULL,
  PRIMARY KEY (`idcategoria`));
```

Ejemplo CREATE TABLE “usuarios”

usuarios - Table

Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G	Default/Expression
idusuario	INT	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>						
nombre	VARCHAR(100)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
apellido	VARCHAR(100)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
email	VARCHAR(50)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
categoria	INT	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	

Column Name: categoria Data Type: INT
Charset/Collation: Default Charset Default Collation
Comments:

Storage: Virtual Stored
 Primary Key Not Null Unique
 Binary Unsigned Zero Fill
 Auto Increment Generated

Columns Indexes Foreign Keys Triggers Partitioning Options

Apply Revert

SQL

```
CREATE TABLE `playground`.`usuarios` (
    `idusuario` INT NOT NULL,
    `nombre` VARCHAR(100) NULL,
    `apellido` VARCHAR(100) NULL,
    `email` VARCHAR(50) NULL,
    `categoria` INT NULL,
    PRIMARY KEY (`idusuario`),
    INDEX `FKcategoria_idx` (`categoria` ASC) VISIBLE,
    CONSTRAINT `FKcategoria`
        FOREIGN KEY (`categoria`)
        REFERENCES `playground`.`categorias` (`idcategoria`)
);
```

Ejemplo CREATE TABLE “carrera”

SQL

```
CREATE TABLE `playground`.`carrera` (
  `idcarrera` INT NOT NULL,
  `titulo` VARCHAR(45) NULL,
  `descripcion` VARCHAR(100) NULL,
  PRIMARY KEY (`idcarrera`));
```

Ejemplo CREATE TABLE “usuarios_carrera”

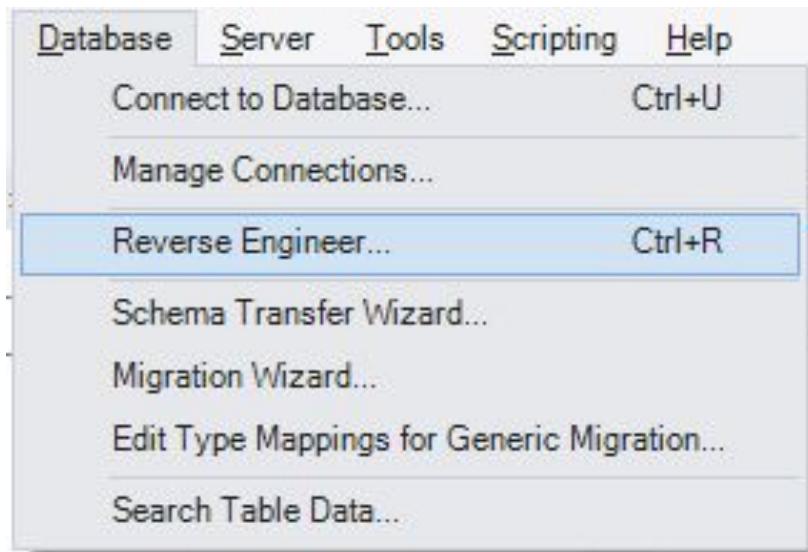
SQL

```
CREATE TABLE `playground`.`usuarios_carrera` (
  `id` INT NOT NULL,
  `usuario` INT NULL,
  `carrera` INT NULL,
  `fechainscripcion` DATE NULL,
  PRIMARY KEY (`id`),
  INDEX `usuario_idx` (`usuario` ASC) VISIBLE,
  INDEX `carrera_idx` (`carrera` ASC) VISIBLE,
  CONSTRAINT `usuario`
    FOREIGN KEY (`usuario`)
    REFERENCES `playground`.`usuarios` (`idusuario`),
  CONSTRAINT `carrera`
    FOREIGN KEY (`carrera`)
    REFERENCES `playground`.`carrera` (`idcarrera`));
```

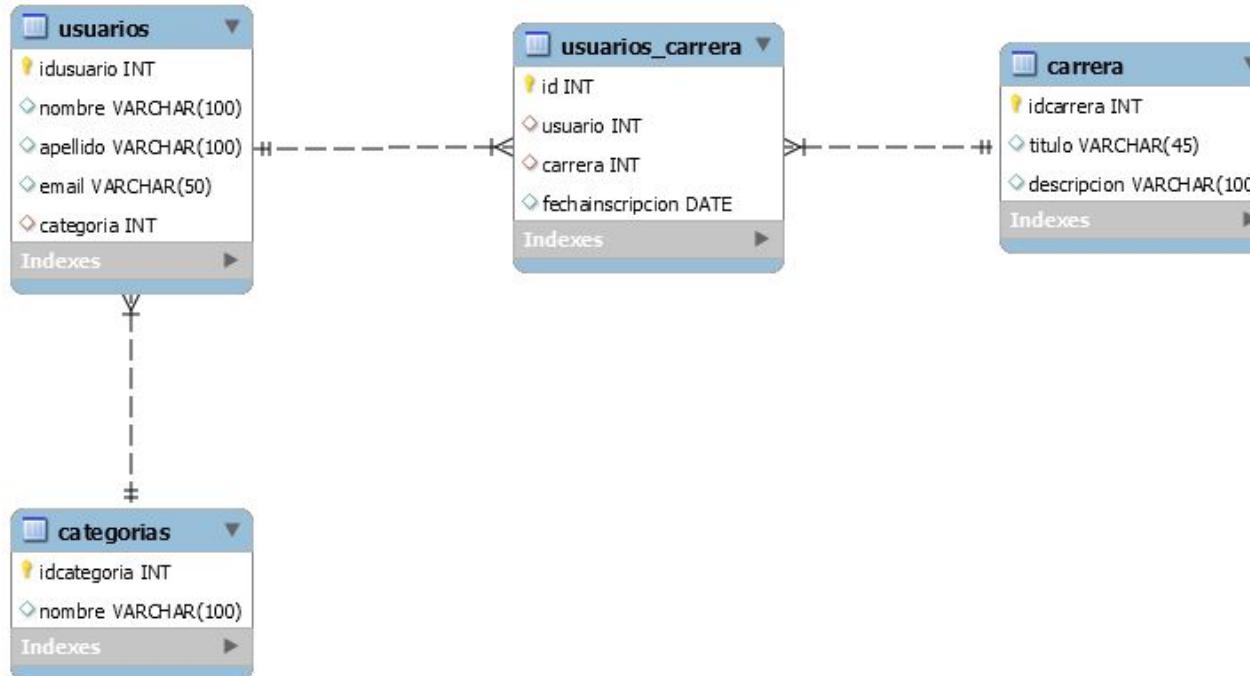
Validar el modelo creado

Continuamos con las tablas de **carrera** y de **Usuarios_Carrera** con las claves foráneas a las tablas de **usuarios** y **carrera**.

Luego, podemos realizar ingeniería inversa para controlar que el modelo relacional es el correcto.



WORKBENCH - DER - Playground



Insertar datos

Vamos a insertar datos en las tablas:

- Categorías: “Alumno”, “Docente”, “Editor” y “Administrador”.
- Usuario: “Juan Perez jperez@gmail.com categoria alumno”.
- Carrera: “Certified Tech Developer Carrera de programación y desarrollo de productos digitales”.
- Juan se inscribió el 1/3/2021 a CTD.



Ejemplo INSERT - Categorías

```
SQL      INSERT INTO `playground`.`categorias`  
              (`idcategoria`, `nombre`)  
            VALUES  
              (1, "Alumno"),  
              (2, "Docente"),  
              (3, "Editor"),  
              (4, "Administrador");
```

Desde la parte de administración nos avisan que no existe más la categoría “Editor”.

¿Qué tenemos que hacer?

SQL

```
DELETE FROM `playground`.`categorias`  
WHERE nombre = "Editor";
```

Ahora, ¿Qué sucede si intentamos eliminar la categoría “Alumno”?

SQL

```
DELETE FROM `playground`.`categorias`  
WHERE nombre = "Alumno";
```



Nos va a generar un error similar a “*Cannot delete or update a parent row: a foreign key constraint fails*”.

Esto indica que no se puede eliminar la categoría “Alumno” dado que debe haber algún alumno bajo esta categoría.

CREATE, DROP y ALTER

DigitalHouse >
Coding School



**Certified Tech
Developer**
The Ultimate Degree

Índice

1. [CREATE BASE DE DATOS](#)
2. [CREATE TABLE](#)
3. [DROP TABLE](#)
4. [ALTER TABLE](#)

1 | CREATE Base de Datos

Comando **CREATE DATABASE**

Con **CREATE DATABASE** podemos crear una base de datos desde cero.

SQL

```
CREATE DATABASE miprimerabasededatos;  
USE miprimerabasededatos;
```

2

CREATE TABLE

Comando **CREATE TABLE**

Con **CREATE TABLE** podemos crear una tabla desde cero, junto con sus columnas, tipos y constraints.

SQL

```
CREATE TABLE nombre_de_la_tabla (
    nombre_de_la_columna_1 TIPO_DE_DATO CONSTRAINT,
    nombre_de_la_columna_2 TIPO_DE_DATO CONSTRAINT
)
```

SQL

```
CREATE TABLE post (
    id INT PRIMARY KEY AUTO_INCREMENT,
    titulo VARCHAR(200)
)
```

Ejemplo CREATE TABLE

SQL

```
CREATE TABLE peliculas (
    id INT PRIMARY KEY AUTO_INCREMENT,
    title VARCHAR(500) NOT NULL,
    rating DECIMAL(3,1) NOT NULL,
    awards INT DEFAULT 0,
    release_date DATE NOT NULL,
    length INT NOT NULL
);
```

FOREIGN KEY

Cuando creamos una columna que contenga una id foránea, será necesario usar la sentencia **FOREIGN KEY** para aclarar a qué tabla y a qué columna hace referencia aquel dato.

Es importante remarcar que la tabla “**clientes**” deberá existir antes de correr esta sentencia para crear la tabla “ordenes”.

SQL

```
CREATE TABLE ordenes (
    orden_id INT NOT NULL,
    orden_numero INT NOT NULL,
    cliente_id INT,
    PRIMARY KEY (orden_id),
    FOREIGN KEY (cliente_id) REFERENCES clientes(id)
);
```

3

DROP TABLE

Comando **DROP TABLE**

DROP TABLE borrará la tabla que le especifiquemos en la sentencia.

SQL

```
DROP TABLE IF EXIST peliculas;
```

4 | ALTER TABLE

Comando **ALTER TABLE**

ALTER TABLE permite alterar una tabla ya existente y va a operar con tres comandos:

- **ADD**: para agregar una columna.
- **MODIFY**: para modificar una columna.
- **DROP**: para borrar una columna.

Ejemplos ALTER TABLE

SQL

```
ALTER TABLE peliculas  
ADD rating DECIMAL(3,1) NOT NULL;
```

Agrega la columna **rating**, aclarando tipo de dato y constraint.

SQL

```
ALTER TABLE peliculas  
MODIFY rating DECIMAL(4,1) NOT NULL;
```

Modifica el decimal de la columna **rating**. Aunque el resto de las configuraciones de la tabla no se modifiquen, es necesario escribirlas en la sentencia.

SQL

```
ALTER TABLE peliculas  
DROP rating;
```

Borra la columna **rating**.

INSERT, UPDATE, DELETE

DigitalHouse >
Coding School



**Certified Tech
Developer**
The Ultimate Degree

Índice

1. [INSERT](#)
2. [UPDATE](#)
3. [DELETE](#)

1 | INSERT

INSERT

Existen dos formas de agregar datos en una tabla:

- Insertando datos en **todas las columnas**.
- Insertando datos en las **columnas que especifiquemos**.

Todas las columnas

Si estamos insertando datos en todas las columnas, no hace falta aclarar los nombres de cada columna. Sin embargo, el orden en el que insertemos los valores, deberá ser el mismo orden que tengan asignadas las columnas en la tabla.

SQL

```
INSERT INTO table_name (columna_1, columna_2, columna_3, ...)
VALUES (valor_1, valor_2, valor_3, ...);
```

SQL

```
INSERT INTO artistas (id, nombre, rating)
VALUES (DEFAULT, 'Shakira', 1.0);
```

Columnas específicas

Para insertar datos en una columna en específico, aclaramos la tabla y luego escribimos el nombre de la o las columnas entre los paréntesis.

SQL

```
INSERT INTO artistas (nombre)  
VALUES ('Calle 13');
```

SQL

```
INSERT INTO artistas (nombre, rating)  
VALUES ('Maluma', 1.0);
```

2 | UPDATE

UPDATE

UPDATE modificará los registros existentes de una tabla. Al igual que con **DELETE**, es importante no olvidar el **WHERE** cuando escribimos la sentencia, aclarando la condición.

```
SQL    UPDATE nombre_tabla  
          SET columna_1 = valor_1, columna_2 = valor_2, ...  
          WHERE condición;
```

```
SQL    UPDATE artistas  
          SET nombre = 'Charly Garcia', rating = 1.0  
          WHERE id = 1;
```

3 | DELETE

DELETE

Con **DELETE** podemos borrar información de una tabla. Es importante recordar utilizar siempre el **WHERE** en la sentencia para agregar la condición de cuáles son las filas que queremos eliminar. Si no escribimos el **WHERE**, estaríamos borrando **toda** la **tabla** y no un registro en particular.

SQL `DELETE FROM nombre_tabla WHERE condición;`

SQL `DELETE FROM artistas WHERE id = 4;`

DigitalHouse >
Coding School

SELECT



**Certified Tech
Developer**
The Ultimate Degree

Cómo usarlo

Toda consulta a la base de datos va a empezar con la palabra **SELECT**.

Su funcionalidad es la de realizar consultas sobre **una o varias columnas** de una tabla.

Para especificar sobre qué tabla queremos realizar esa consulta usamos la palabra **FROM** seguida del nombre de la tabla.

SQL

```
SELECT nombre_columna, nombre_columna, ...
FROM nombre_tabla;
```

Ejemplo - Tabla Peliculas

id	título	rating	fecha_estreno	país
1001	Pulp Fiction	9.8	1995-02-16	Estados Unidos
1002	Kill Bill	9.5	2003-11-27	Estados Unidos

De esta tabla completa, para conocer solamente los títulos y ratings de las películas guardadas en la tabla **películas**, podríamos hacerlo ejecutando la siguiente consulta:

SQL

```
SELECT id, titulo, rating  
FROM peliculas;
```

DigitalHouse >
Coding School

WHERE y
ORDER BY



Certified Tech
Developer
The Ultimate Degree

Índice

1. WHERE
2. ORDER BY

1 | WHERE

WHERE

La funcionalidad del **WHERE** es la de condicionar y filtrar las consultas **SELECT** que se realizan a una base de datos.

```
SQL   SELECT nombre_columna_1, nombre_columna_2, ...
      FROM nombre_tabla
      WHERE condicion;
```

Teniendo una tabla **clientes**, podría consultar primer nombre y apellido, filtrando con un **WHERE** solamente los usuarios **que su país es igual a Argentina** de la siguiente manera:

```
SQL   SELECT primer_nombre, apellido
      FROM clientes
      WHERE pais = 'Argentina';
```

Operadores

=> Igual a
>> Mayor que
>=> Mayor o igual que
<> Menor que
<=> Menor o igual que
<>> Diferente a
!=> Diferente a

IS NULL> Es nulo
BETWEEN> Entre dos valores
IN> Lista de valores
LIKE> Se ajusta a...

Queries de ejemplo

SQL

```
SELECT primer_nombre, apellido  
FROM clientes  
WHERE pais <> 'Argentina';
```

SQL

```
SELECT primer_nombre, apellido  
FROM clientes  
WHERE id < 15;
```

SQL

```
SELECT primer_nombre, apellido  
FROM clientes  
WHERE id > 5;
```

SQL

```
SELECT *
FROM canciones
WHERE id >= 3
AND id < 8;
```

SQL

```
SELECT *
FROM canciones
WHERE id = 2
OR id = 6;
```

SQL

```
DELETE FROM usuarios  
WHERE id = 2;
```



Si en esta query quitáramos el WHERE...
iBorraríamos toda la tabla!

2 | ORDER BY

ORDER BY

ORDER BY se utiliza para ordenar los resultados de una consulta **según el valor de la columna especificada**. Por defecto, se ordena de forma ascendente (ASC) según los valores de la columna. También se puede ordenar de manera descendente (DESC) aclarándolo en la consulta.

SQL

```
SELECT nombre_columna1, nombre_columna2  
FROM tabla  
WHERE condicion  
ORDER BY nombre_columna1;
```

Query de ejemplo

Teniendo una tabla **usuarios**, podría consultar los nombres, filtrar con un **WHERE** solamente los usuarios **mayores de 21 años** y ordenarlos de forma descendente tomando como referencia la columna nombre.

SQL

```
SELECT nombre, rating  
FROM artistas  
WHERE rating > 1.0  
ORDER BY nombre DESC;
```

QUERIES ML

DigitalHouse>
Coding School



**Certified Tech
Developer**
The Ultimate Degree

SELECT - Cómo usarlo

Toda consulta a la base de datos va a empezar con la palabra **SELECT**.

Su funcionalidad es la de realizar consultas sobre **una o varias columnas** de una tabla.

Para especificar sobre qué tabla queremos realizar esa consulta usamos la palabra **FROM** seguida del nombre de la tabla.

SQL

```
SELECT nombre_columna, nombre_columna, ...
FROM nombre_tabla;
```

Ejemplo - Tabla Peliculas

id	título	rating	fecha_estreno	país
1001	Pulp Fiction	9.8	1995-02-16	Estados Unidos
1002	Kill Bill	9.5	2003-11-27	Estados Unidos

De esta tabla completa, para conocer solamente los títulos y ratings de las películas guardadas en la tabla **películas**, podríamos hacerlo ejecutando la siguiente consulta:

SQL

```
SELECT id, titulo, rating  
FROM peliculas;
```

Where

La funcionalidad del **WHERE** es la de condicionar y filtrar las consultas **SELECT** que se realizan a una base de datos.

SQL

```
SELECT nombre_columna_1, nombre_columna_2, ...
FROM nombre_tabla
WHERE condicion;
```

Teniendo una tabla **usuarios**, podríamos consultar nombre y edad, filtrando con un **WHERE** solamente los usuarios **mayores de 17 años** de la siguiente manera:

SQL

```
SELECT nombre, edad
FROM usuarios
WHERE edad > 17;
```

Operadores

- = ➤ Igual a
- > ➤ Mayor que
- >= ➤ Mayor o igual que
- < ➤ Menor que
- <= ➤ Menor o igual que
- <> ➤ Diferente a
- != ➤ Diferente a

Operadores

- IS NULL** → Es nulo
- BETWEEN** → Entre dos valores
- IN** → Lista de valores
- LIKE** → Se ajusta a...

Queries de ejemplo

SQL

```
SELECT nombre, edad  
FROM usuarios  
WHERE edad > 17;
```

SQL

```
SELECT *  
FROM movies  
WHERE title LIKE 'Avatar';
```

Queries de ejemplo

SQL

```
SELECT *
FROM movies
WHERE awards >= 3
AND awards < 8;
```

SQL

```
SELECT *
FROM movies
WHERE awards = 2
OR awards = 6;
```

Query de ejemplo

SQL

```
DELETE FROM usuarios  
WHERE id = 2;
```



Si en esta query quitáramos el WHERE...
iBorraríamos toda la tabla!

Order By

ORDER BY se utiliza para ordenar los resultados de una consulta **según el valor de la columna especificada**. Por defecto, se ordena de forma ascendente (ASC) según los valores de la columna. También se puede ordenar de manera descendente (DESC) aclarándolo en la consulta.

SQL

```
SELECT nombre_columna1, nombre_columna2  
FROM tabla  
WHERE condicion  
ORDER BY nombre_columna1;
```

Query de ejemplo

Teniendo una tabla **usuarios**, podría consultar los nombres, filtrar con un **WHERE** sólo los usuarios **mayores de 21 años** y ordenarlos de forma descendente tomando como referencia la columna nombre.

SQL

```
SELECT nombre, edad  
FROM usuarios  
WHERE edad > 21  
ORDER BY nombre DESC;
```

BETWEEN y LIKE

DigitalHouse>
Coding School



**Certified Tech
Developer**
The Ultimate Degree

BETWEEN

Cuando necesitamos obtener valores **dentro de un rango**, usamos el operador BETWEEN.

- BETWEEN **incluye** los **extremos**.
- BETWEEN funciona con **números, textos y fechas**.
- Se usa como un filtro de un WHERE.

Por ejemplo, coloquialmente:

- Dados los números: 4, 7, 2, 9, 1

Si hicieramos un BETWEEN entre 2 y 7 devolvería 4, 7, 2 (*excluye el 9 y el 1, e incluye el 2*).

Query de ejemplo

Con la siguiente consulta estaríamos seleccionando **nombre** y **edad** de la tabla **alumnos** sólo cuando las edades estén **entre** 6 y 12.

SQL

```
SELECT nombre, edad  
FROM alumnos  
WHERE edad BETWEEN 6 AND 12;
```

LIKE

Cuando hacemos un filtro con un **WHERE**, podemos especificar un patrón de búsqueda que nos permita especificar algo concreto que queremos encontrar en los registros. Eso lo logramos utilizando **comodines** (*wildcards*).

Por ejemplo, podríamos querer buscar:

- Los nombres que tengan la letra 'a' como segundo carácter.
- Las direcciones postales que incluyan la calle 'Monroe'.
- Los clientes que empiecen con 'Los' y terminen con 's'.

“

COMODÍN %

Es un sustituto que representa **cero, uno, o varios** caracteres.

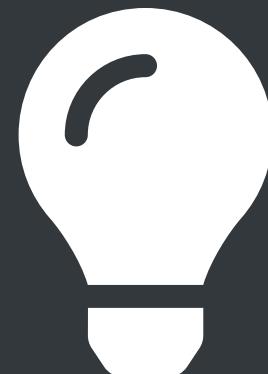


”

“

COMODÍN _

Es un sustituto para **un solo** carácter.



”

Queries de ejemplo

SQL

```
SELECT nombre  
FROM usuarios  
WHERE nombre LIKE '_a%';
```

Devuelve aquellos nombres que tengan la letra 'a' como segundo carácter.

SQL

```
SELECT nombre  
FROM usuarios  
WHERE direccion LIKE '%Monroe%';
```

Devuelve las direcciones de los usuarios que incluyan la calle 'Monroe'.

Queries de ejemplo

SQL

```
SELECT nombre  
FROM clientes  
WHERE nombre LIKE 'Los%s';
```

Devuelve los clientes que empiecen con 'Los' y terminen con 's'.

LIMIT y OFFSET

DigitalHouse>
Coding School



**Certified Tech
Developer**
The Ultimate Degree

Limit

Su funcionalidad es la de **limitar el número de filas** (registros/resultados) devueltas en las consultas SELECT. También establece el **número máximo** de registros a eliminar con DELETE.

SQL

```
SELECT nombre_columna1, nombre_columna2  
FROM nombre_tabla  
LIMIT cantidad_de_registros;
```

Query de ejemplo

Teniendo una tabla **peliculas**, podríamos armar un top 10 con las películas que tengan más de 4 premios usando un **LIMIT** en la siguiente consulta:

SQL

```
SELECT *
FROM peliculas
WHERE premios > 4
LIMIT 10;
```

Offset

- En un escenario en donde hacemos una consulta de todas las películas de la base de datos, la misma nos devolvería muchos registros. Usando un **LIMIT** podríamos aclarar un límite de 20.
- *¿Pero cómo haríamos si quisieramos recuperar sólo 20 películas pero saltando las primeras 10 de la tabla?*
- **OFFSET** nos permite especificar a partir de qué fila comenzar la recuperación de los datos solicitados.

{código}

```
SELECT id, nombre, apellido  
FROM alumnos  
LIMIT 20  
OFFSET 20;
```

{código}

```
SELECT id, nombre, apellido  
FROM alumnos  
LIMIT 20  
OFFSET 20;
```

Seleccionamos las columnas id, nombre y apellido.

{código}

```
SELECT id, nombre, apellido  
FROM alumnos  
LIMIT 20  
OFFSET 20;
```

de la tabla alumnos.

{código}

```
SELECT id, nombre, apellido  
FROM alumnos  
LIMIT 20  
OFFSET 20;
```

Limitamos los registros de la tabla resultante a **20** registros.

{código}

```
SELECT id, nombre, apellido  
FROM alumnos  
LIMIT 20  
OFFSET 20;
```

Desplazamos los resultados 20 posiciones para que se muestre desde la posición **21**.

Alias

DigitalHouse>
Coding School



Certified Tech
Developer
The Ultimate Degree

Alias

Los **alias** se usan para darle un nombre temporal y más amigable a las **tablas, columnas y funciones**. Los **alias** se definen durante una consulta y persisten **solo** durante esa consulta.

Para definir un alias usamos las iniciales **AS** precediendo a la columna que estamos queriendo asignarle ese alias.

SQL

```
SELECT nombre_columna1 AS alias_nombre_columna1  
FROM nombre_tabla;
```

Alias para una columna

```
SELECT razon_social_cliente AS nombre  
FROM cliente  
ORDER BY nombre;
```

Alias para una columna

```
SELECT razon_social_cliente AS nombre  
FROM cliente  
ORDER BY nombre;
```

Seleccionamos la columna *razon_social_cliente* y le asignamos el **alias** nombre.

Alias para una columna

```
SELECT razon_social_cliente AS nombre  
FROM cliente  
ORDER BY nombre;
```

En el **FROM** elegimos tabla cliente.
Con el **ORDER BY** ordenamos los registros con la columna nombre.

Alias para una tabla

```
SELECT nombre, apellido, edad  
FROM alumnos_comision_inicial AS alumnos;
```

Alias para una tabla

```
SELECT nombre, apellido, edad  
FROM alumnos_comision_inicial AS alumnos;
```

Seleccionamos las columnas nombre, apellido y edad.

Alias para una tabla

```
SELECT nombre, apellido, edad  
FROM alumnos_comision_inicial AS alumnos;
```

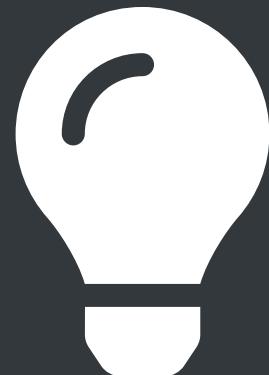
Hacemos la consulta sobre la tabla *alumnos_comision_inicial* y le **asignamos** el alias *alumnos*.

No es recomendable asignar más de una palabra dentro de un alias. En el caso de necesitarlo, utilizar “_”.

“

De este modo, podemos darle alias a las **columnas** y **tablas** que vamos trayendo y hacer más legible la manipulación de datos, teniendo siempre presente que los alias **no modifican** los nombres originales en la base de datos.

”



BETWEEN y LIKE

DigitalHouse>
Coding School



**Certified Tech
Developer**
The Ultimate Degree

BETWEEN

Cuando necesitamos obtener valores **dentro de un rango**, usamos el operador BETWEEN.

- BETWEEN **incluye** los **extremos**.
- BETWEEN funciona con **números, textos y fechas**.
- Se usa como un filtro de un WHERE.

Por ejemplo, coloquialmente:

- Dados los números: 4, 7, 2, 9, 1

Si hicieramos un BETWEEN entre 2 y 7 devolvería 4, 7, 2 (*excluye el 9 y el 1, e incluye el 2*).

Query de ejemplo

Con la siguiente consulta estaríamos seleccionando **nombre** y **edad** de la tabla **alumnos** solo cuando las edades estén **entre** 6 y 12.

SQL

```
SELECT nombre, edad  
FROM alumnos  
WHERE edad BETWEEN 6 AND 12;
```

LIKE

Cuando hacemos un filtro con un **WHERE**, podemos especificar un patrón de búsqueda que nos permita especificar algo concreto que queremos encontrar en los registros. Eso lo logramos utilizando **comodines** (*wildcards*).

Por ejemplo, podríamos querer buscar:

- Los nombres que tengan la letra 'a' como segundo carácter.
- Las direcciones postales que incluyan la calle 'Monroe'.
- Los clientes que empiecen con 'Los' y terminen con 's'.

“

COMODÍN %

Es un sustituto que representa **cero, uno, o varios** caracteres.



”

“

COMODÍN _

Es un sustituto para **un solo** carácter.



”

Queries de ejemplo

SQL

```
SELECT nombre  
FROM usuarios  
WHERE nombre LIKE '_a%';
```

Devuelve aquellos nombres que tengan la letra 'a' como segundo carácter.

SQL

```
SELECT nombre  
FROM usuarios  
WHERE direccion LIKE '%Monroe%';
```

Devuelve las direcciones de los usuarios que incluyan la calle 'Monroe'.

Queries de ejemplo

SQL

```
SELECT nombre  
FROM clientes  
WHERE nombre LIKE 'Los%s';
```

Devuelve los clientes que empiecen con 'Los' y terminen con 's'.

LIMIT y OFFSET

DigitalHouse >
Coding School



**Certified Tech
Developer**
The Ultimate Degree

Limit

Su funcionalidad es la de **limitar el número de filas** (registros/resultados) devueltas en las consultas SELECT. También establece el **número máximo** de registros a eliminar con DELETE.

SQL

```
SELECT nombre_columna1, nombre_columna2  
FROM nombre_tabla  
LIMIT cantidad_de_registros;
```

Query de ejemplo

Teniendo una tabla **peliculas**, podríamos armar un top 10 con las películas que tengan más de 4 premios usando un **LIMIT** en la siguiente consulta:

SQL

```
SELECT *
FROM peliculas
WHERE premios > 4
LIMIT 10;
```

Offset

- En un escenario en donde hacemos una consulta de todas las películas de la base de datos, la misma nos devolvería muchos registros. Usando un **LIMIT** podríamos aclarar un límite de 20.
- *¿Pero cómo haríamos si quisieramos recuperar sólo 20 películas pero saltando las primeras 10 de la tabla?*
- **OFFSET** nos permite especificar a partir de qué fila comenzar la recuperación de los datos solicitados.

{código}

```
SELECT id, nombre, apellido  
FROM alumnos  
LIMIT 20  
OFFSET 20;
```

{código}

```
SELECT id, nombre, apellido  
FROM alumnos  
LIMIT 20  
OFFSET 20;
```

Seleccionamos las columnas id, nombre y apellido.

{código}

```
SELECT id, nombre, apellido
```

```
FROM alumnos
```

```
LIMIT 20
```

```
OFFSET 20;
```

de la tabla alumnos.

{código}

```
SELECT id, nombre, apellido  
FROM alumnos  
LIMIT 20  
OFFSET 20;
```

Limitamos los registros de la tabla resultante a **20** registros.

{código}

```
SELECT id, nombre, apellido  
FROM alumnos  
LIMIT 20  
OFFSET 20;
```



Desplazamos los resultados 20 posiciones para que se muestre desde la posición **21**.

DigitalHouse >
Coding School

Alias



**Certified Tech
Developer**
The Ultimate Degree

Alias

Los **alias** se usan para darle un nombre temporal y más amigable a las **tablas, columnas y funciones**. Los **alias** se definen durante una consulta y persisten **solo** durante esa consulta.

Para definir un alias usamos las iniciales **AS** precediendo a la columna que estamos queriendo asignarle ese alias.

SQL

```
SELECT nombre_columna1 AS alias_nombre_columna1  
FROM nombre_tabla;
```

Alias para una columna

```
SELECT razon_social_cliente AS nombre  
FROM cliente  
ORDER BY nombre;
```

Alias para una columna

```
SELECT razon_social_cliente AS nombre  
FROM cliente  
ORDER BY nombre;
```

Seleccionamos la columna *razon_social_cliente* y le asignamos el **alias** nombre.

Alias para una columna

```
SELECT razon_social_cliente AS nombre  
FROM cliente  
ORDER BY nombre;
```

En el **FROM** elegimos tabla cliente.
Con el **ORDER BY** ordenamos los registros con la columna nombre.

Alias para una tabla

```
SELECT nombre, apellido, edad  
FROM alumnos_comision_inicial AS alumnos;
```

Alias para una tabla

```
SELECT nombre, apellido, edad  
FROM alumnos_comision_inicial AS alumnos;
```

Seleccionamos las columnas nombre, apellido y edad.

Alias para una tabla

```
SELECT nombre, apellido, edad  
FROM alumnos_comision_inicial AS alumnos;
```

Hacemos la consulta sobre la tabla *alumnos_comision_inicial* y le **asignamos** el alias *alumnos*.

No es recomendable asignar más de una palabra dentro de un alias. En el caso de necesitarlo, utilizar “_”.

“

De este modo, podemos darle alias a las **columnas** y **tablas** que vamos trayendo y hacer más legible la manipulación de datos, teniendo siempre presente que los alias **no modifican** los nombres originales en la base de datos.



”

Funciones de agregación

DigitalHouse >
Coding School



**Certified Tech
Developer**
The Ultimate Degree

“

Las funciones de agregación **realizan cálculos** sobre un conjunto de datos y **devuelven** un **único resultado**. Excepto **COUNT**, las funciones de agregación **ignorarán** los valores **NULL**.

”



COUNT

Devuelve un **único** resultado indicando la cantidad de **filas/registros** que cumplen con el criterio.

SQL `SELECT COUNT(*) FROM movies;`

Devuelve la cantidad de registros de la tabla movies.

SQL `SELECT COUNT(id) AS total FROM movies WHERE genre_id=3;`

Devuelve la cantidad de películas de la tabla movies con el genero_id 3 y lo muestra en una columna denominada total.

AVG, SUM

AVG (*average*) devuelve un **único** resultado indicando el **promedio** de una columna cuyo tipo de datos debe ser numérico.

SUM (*suma*) devuelve un **único** resultado indicando la **suma** de una columna cuyo tipo de datos debe ser numérico.

```
SQL    SELECT AVG(rating) FROM movies;
```

Devuelve el promedio del rating de las películas de la tabla movies.

```
SQL    SELECT SUM(length) FROM movies;
```

Devuelve la suma de las duraciones de las películas de la tabla movies.

MIN, MAX

MIN devuelve un **único** resultado indicando el valor **mínimo** de una columna cuyo tipo de datos debe ser numérico.

MAX devuelve un **único** resultado indicando el valor **máximo** de una columna cuyo tipo de datos debe ser numérico.

SQL `SELECT MIN(rating) FROM movies;`

Devolverá el rating de la película menos ranqueada.

SQL `SELECT MAX(length) FROM movies;`

Devolverá el rating de la película mejor ranqueada.

DigitalHouse >
Coding School

GROUP BY



**Certified Tech
Developer**
The Ultimate Degree

Sintaxis

GROUP BY se usa para **agrupar los registros** de la tabla resultante de una consulta por una o más columnas.

SQL

```
SELECT columna_1  
FROM nombre_tabla  
WHERE condition  
GROUP BY columna_1;
```

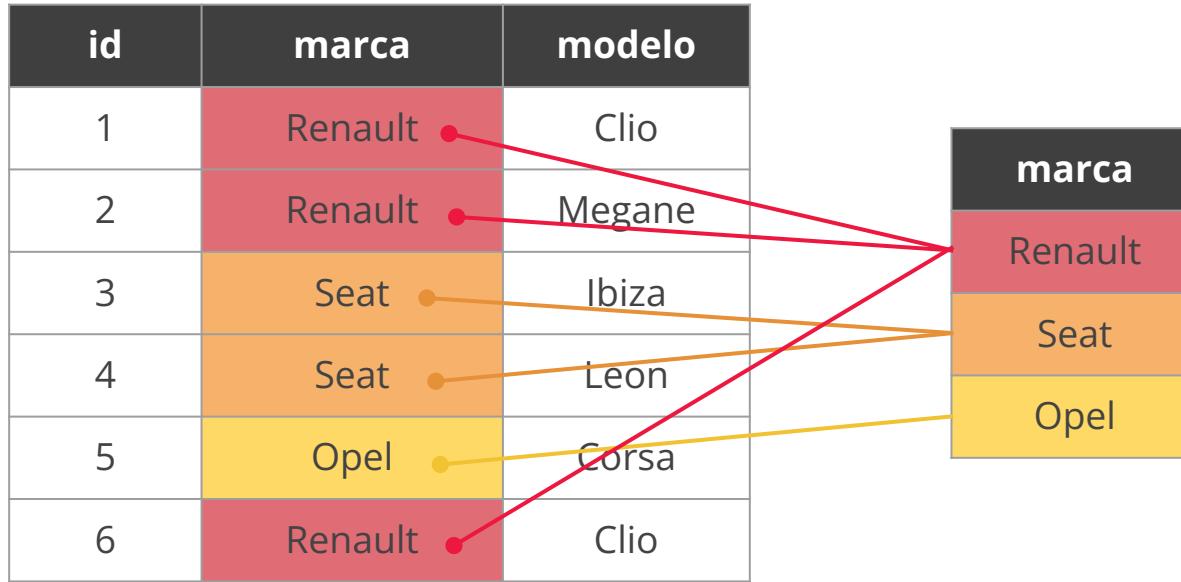
Ejemplo

En el siguiente ejemplo, se utiliza **GROUP BY** para agrupar los **coches** por **marca** mostrando aquellos que tienen el año de fabricación igual o superior al año **2010**.

SQL

```
SELECT marca  
FROM coche  
WHERE anio_fabricacion >= 2010  
GROUP BY marca;
```

Ejemplo (cont.)



Agrupación de datos

Dado que **GROUP BY** agrupa la información, perdemos el detalle de cada una de las filas. Es decir, ya no nos interesa el valor de cada fila, sino un resultado consolidado entre todas las filas. Veamos la siguiente consulta:

SQL

```
SELECT id, marca  
FROM coche  
GROUP BY marca;
```

Si agrupamos los coches por **marca**, ya no podremos visualizar el ID de cada fila. Posiblemente, en la fila nos muestre —para el campo **ID**— el primero de cada grupo de registros.

Veamos algunos ejemplos más...

Ejemplos

SQL

```
SELECT marca, MAX(precio) AS precio_maximo  
FROM coche  
GROUP BY marca;
```

Devuelve la marca y el precio más alto de cada grupo de marcas.

SQL

```
SELECT genero, AVG(duracion) AS duracion_promedio  
FROM pelicula  
GROUP BY genero;
```

Devuelve el género y la duración promedio de cada grupo de géneros.

Conclusión

En resumen, la cláusula **GROUP BY**:

- Se usa para **agrupar filas** que contienen los **mismos valores**.
- Opcionalmente, se utiliza junto con las **funciones de agregación** (SUM, AVG, COUNT, MIN, MAX) con el objetivo de producir reportes resumidos.
- Las consultas que contienen la cláusula GROUP BY se denominan **consultas agrupadas** y solo devuelven una **sola fila** para cada elemento agrupado.

SQL

```
SELECT marca, MAX(precio)
FROM coche
GROUP BY marca;
```

DigitalHouse >
Coding School

HAVING



**Certified Tech
Developer**
The Ultimate Degree

Sintaxis

Cumple la misma función que **WHERE**, a diferencia de que **HAVING** permite la implementación de **alias** y **funciones de agregación** en las condiciones de la selección de **datos**.

SQL

```
SELECT columna_1  
FROM nombre_tabla  
WHERE condition  
GROUP BY columna_1  
HAVING condition_Group  
ORDER BY columna_1;
```

Sintaxis (cont.)

Esta consulta devolverá la cantidad de clientes por país (agrupados por país). Solamente se incluirán en el resultado aquellos países que tengan al menos 3 clientes.

SQL

```
SELECT pais, COUNT(clienteId)
FROM clientes
GROUP BY pais
HAVING COUNT(clienteId)>=3;
```

Estructura de una query

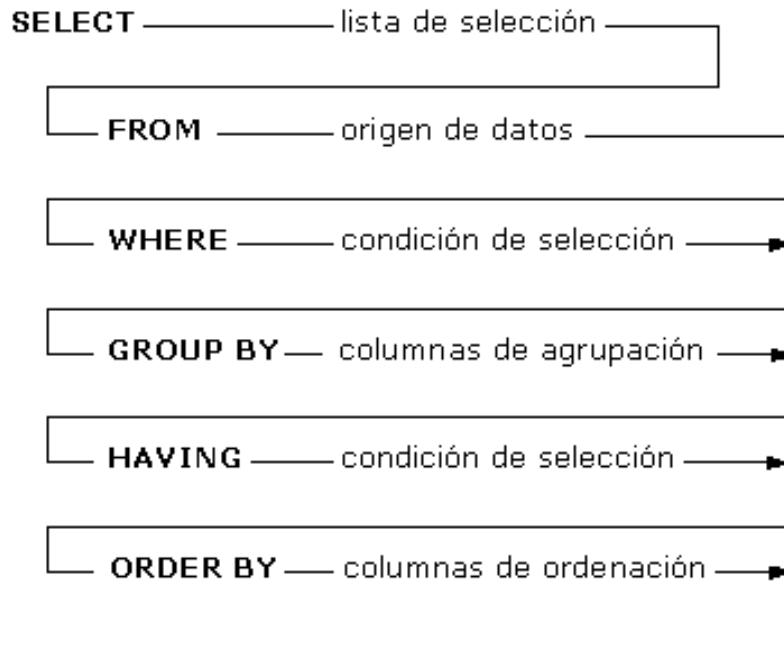


Table reference

DigitalHouse >
Coding School



**Certified Tech
Developer**
The Ultimate Degree

Consultas a más de una tabla

Hasta ahora vimos consultas (SELECT) dentro de una **tabla**. Pero también es posible y necesario hacer consultas a distintas tablas y unir los resultados.

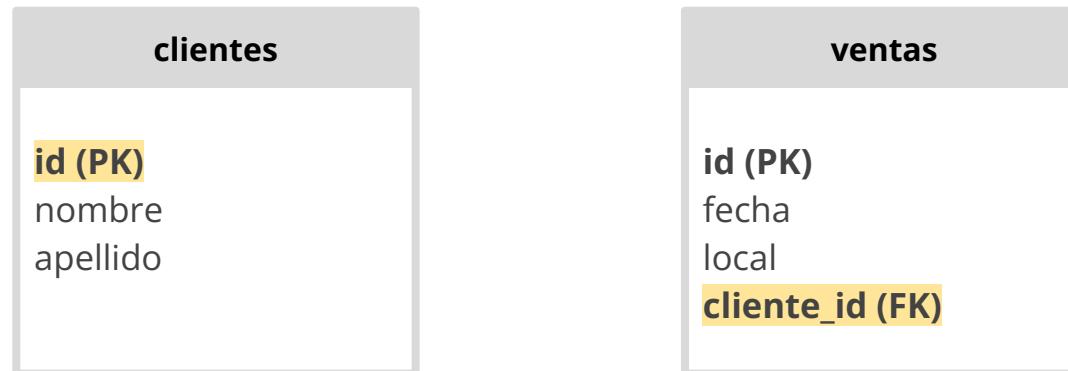
Por ejemplo, un posible escenario sería querer consultar una tabla en donde están los **datos** de los **clientes** y otra tabla en donde están los **datos** de las **ventas a esos clientes**.



Consultas a más de una tabla (cont.)

Seguramente, en la tabla de **ventas**, existirá un campo con el ID del cliente (**cliente_id**).

Si quisiéramos mostrar **todas** las ventas de un cliente concreto, necesitaremos usar datos de **ambas tablas** y **vincularlas** con algún **campo** que **compartan**. En este caso, el **cliente_id**.



Consulta SQL

```
SELECT clientes.id AS ID, clientes.nombre, ventas.fecha  
FROM clientes, ventas  
WHERE clientes.id = ventas.cliente_id;
```

Consulta SQL

```
SELECT clientes.id AS ID, clientes.nombre, ventas.fecha  
FROM clientes, ventas  
WHERE clientes.id = ventas.cliente_id;
```

Seleccionamos:

- La columna **id** de la tabla **clientes** y le asignamos el alias **ID**.
- La columna **nombre** de la tabla **clientes**.
- La columna **fecha** de la tabla **ventas**.

Consulta SQL

```
SELECT clientes.id AS id, clientes.nombre, ventas.fecha  
FROM clientes, ventas  
  
WHERE clientes.id = ventas.cliente_id;
```

El **select** lo hacemos sobre las tablas **clientes** y **ventas**.

Hasta acá la consulta traería **todos los clientes y todas las ventas**. Por eso nos falta todavía agregar un **filtro** que muestre **solo** las ventas de **cada usuario** en particular.

Consulta SQL

```
SELECT clientes.id AS id, clientes.nombre, ventas.fecha  
FROM clientes, ventas  
WHERE clientes.id = ventas.cliente_id;
```

En el WHERE creamos una condición para traer aquellos registros en donde el ID del cliente sea igual en ambas tablas.

DigitalHouse >
Coding School

JOIN



Certified Tech
Developer
The Ultimate Degree

¿Por qué usar JOIN?

Además de realizar consultas dentro de una tabla y de haber empleado **table reference** para consultas en múltiples tablas, existe la herramienta **JOIN** que nos permite hacer consultas a distintas tablas y unir los resultados.

Ventajas del uso de **JOIN**:

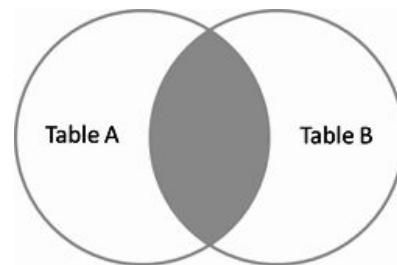
- Su sintaxis es mucho más comprensible.
- Presentan una mejor performance.
- Proveen de ciertas flexibilidades.

INNER JOIN

El **INNER JOIN** es la opción predeterminada y nos devuelve **todos los registros** donde se **cruzan dos o más tablas**. Por ejemplo, si tenemos una tabla cliente y otra factura, al cruzarlas con **INNER JOIN**, nos devuelve aquellos registros o filas donde haya un valor coincidente en ambas tablas.

cliente		
id	nombre	apellido
1	Juan	Perez
2	Clara	Sanchez
3	Marta	García

INNER JOIN



factura		
id	cliente_id	fecha
1	2	12/03/2019
2	2	22/08/2019
3	1	04/09/2019

Consulta a múltiples tablas

Antes con **table reference** escribíamos:

SQL

```
SELECT cliente.id, cliente.nombre, factura.fecha  
FROM cliente, factura;
```

Ahora con **INNER JOIN** escribimos:

SQL

```
SELECT cliente.id, cliente.nombre, factura.fecha  
FROM cliente  
INNER JOIN factura;
```

“

Si bien ya dimos el primer paso (que es **cruzar** ambas tablas), aún nos falta aclarar **dónde** está ese cruce.

Es decir, qué **clave primaria (PK)** se cruzará con qué **clave foránea (FK)**.



”

Definiendo el INNER JOIN

Para definir el **INNER JOIN** tenemos que indicar el filtro por el cual se **evaluará** el **cruce**. Para esto, debemos utilizar la palabra reservada **ON**. Es decir, que lo que antes escribíamos en el **WHERE** de table reference, ahora lo escribiremos en el **ON** de INNER JOIN.

SQL

```
SELECT cliente.id, cliente.nombre, factura.fecha  
FROM cliente  
INNER JOIN factura  
ON cliente.id = factura.cliente_id;
```

DigitalHouse >
Coding School

DISTINCT



**Certified Tech
Developer**
The Ultimate Degree

Cómo funciona

Al realizar una consulta en una tabla, puede ocurrir que en los resultados existan dos o más **filas idénticas**. En algunas situaciones, nos pueden solicitar un listado con registros **no duplicados**, para esto, utilizamos la cláusula **DISTINCT** que devuelve un listado en donde cada fila es distinta.

SQL

```
SELECT DISTINCT columna_1, columna_2  
FROM nombre_tabla;
```

Ejemplo

Partiendo de una tabla de **usuarios**, si ejecutamos la consulta:

SQL

```
SELECT pais FROM usuarios;
```

Obtendremos cinco filas:

usuarios
Perú
Perú
Argentina
Colombia
Argentina

Ejemplo

Si agregamos la cláusula **DISTINCT** en la consulta:

```
SQL SELECT DISTINCT pais FROM usuarios;
```

Obtendremos tres filas:

usuarios
Perú
Argentina
Colombia

{código}

```
SELECT DISTINCT actor.nombre, actor.apellido  
FROM actor  
INNER JOIN actor_pelicula  
ON actor_pelicula.actor_id = actor.id  
INNER JOIN pelicula  
ON pelicula.id = actor_pelicula.pelicula_id  
WHERE pelicula.titulo LIKE '%Harry Potter%';
```

En este ejemplo vemos una query que **pide** los **actores** que hayan actuado en **cualquier película** de **Harry Potter**.

Si no escribiéramos el **DISTINCT**, los actores que hayan participado en más de una película, aparecerán repetidos en el resultado.

Funciones de alteración

DigitalHouse >
Coding School



**Certified Tech
Developer**
The Ultimate Degree

CONCAT

Usamos **CONCAT** para **concatenar** dos o más expresiones:

```
SQL   SELECT CONCAT('Hola', ' a ', 'todos.');
```

> 'Hola a todos.'

```
SQL   SELECT CONCAT('La respuesta es: ', 24, '.');
```

> 'La respuesta es 24.'

```
SQL   SELECT CONCAT('Nombre: ', apellido, ', ', nombre, '.')  
FROM actor;
```

> 'Nombre: Clarke, Emilia.'

COALESCE

Usamos **COALESCE** para sustituir el valor **NULL** en una sucesión de expresiones o campos. Es decir, si la primera expresión es Null, se sustituye con el valor de una segunda expresión, pero si este valor también es Null, se puede sustituir con el valor de una tercera expresión y así sucesivamente.

```
SQL SELECT COALESCE(NULL, 'Sin datos');  
> 'Sin datos'
```

```
SQL SELECT COALESCE(NULL, NULL, 'Digital House');  
> 'Digital House'
```

COALESCE (cont.)

Los tres clientes de la siguiente tabla poseen uno o más datos nulos:

SQL

```
SELECT id, apellido, nombre, telefono_movil, telefono_fijo  
FROM cliente;
```

cliente				
id	apellido	nombre	telefono_movil	telefono_fijo
1	Pérez	Juan	1156685441	43552215
2	Medina	Rocío	Null	43411722
3	López	Matías	Null	Null

COALESCE (cont.)

Usando **COALESCE** podremos sustituir los **datos nulos** en cada registro, indicando la columna a evaluar y el valor de sustitución.

SQL

```
SELECT id, apellido, nombre, COALESCE(telefono_movil, telefono_fijo, 'Sin datos')
AS telefono FROM cliente;
```

cliente			
id	apellido	nombre	telefono
1	Pérez	Juan	1156685441
2	Medina	Rocío	43411722
3	López	Matías	Sin datos

DATEDIFF

Usamos **DATEDIFF** para devolver la **diferencia** entre dos fechas, tomando como granularidad el intervalo especificado.

SQL `SELECT DATEDIFF('2021-02-03 12:45:00', '2021-01-01 07:00:00');`

> 33

Devuelve 33 porque es la cantidad de días de la diferencia entre las fechas indicadas.

SQL `SELECT DATEDIFF('2021-01-15', '2021-01-05');`

> 10

Devuelve 10 porque es la cantidad de días de la diferencia entre las fechas indicadas.

TIMEDIFF

Usamos **TIMEDIFF** para devolver la **diferencia** entre dos horarios, tomando como granularidad el intervalo especificado.

SQL `SELECT TIMEDIFF('2021-01-01 12:45:00', '2021-01-01 07:00:00');`

> `05:45:00`

SQL `SELECT TIMEDIFF('18:45:00', '12:30:00');`

> `06:15:00`

EXTRACT

Usamos **EXTRACT** para **extraer** partes de una fecha:

```
SQL   SELECT EXTRACT(SECOND FROM '2014-02-13 08:44:21');  
> 21
```

```
SQL   SELECT EXTRACT(MINUTE FROM '2014-02-13 08:44:21');  
> 44
```

```
SQL   SELECT EXTRACT(HOUR FROM '2014-02-13 08:44:21');  
> 8
```

```
SQL   SELECT EXTRACT(DAY FROM '2014-02-13 08:44:21');  
> 13
```

EXTRACT (cont.)

```
SQL   SELECT EXTRACT(WEEK FROM '2014-02-13 08:44:21');
```

> 6

```
SQL   SELECT EXTRACT(MONTH FROM '2014-02-13 08:44:21');
```

> 2

```
SQL   SELECT EXTRACT(QUARTER FROM '2014-02-13 08:44:21');
```

> 1

```
SQL   SELECT EXTRACT(YEAR FROM '2014-02-13 08:44:21');
```

> 2014

REPLACE

Usamos **REPLACE** para reemplazar una cadena de caracteres por otro valor. Cabe aclarar que esta función hace distinción entre minúsculas y mayúsculas.

SQL `SELECT REPLACE('Buenas tardes', 'tardes', 'Noches');`

> `Buenas Noches`

SQL `SELECT REPLACE('Buenas tardes', 'a', 'A');`

> `BuenAs tArdes`

SQL `SELECT REPLACE('1520', '2', '5');`

> `1550`

DATE_FORMAT

Usamos **DATE_FORMAT** para cambiar el formato de salida de una fecha según una condición dada.

SQL `SELECT DATE_FORMAT('2017-06-15', '%Y');`

> 2017

SQL `SELECT DATE_FORMAT('2017-06-15', '%W %M %e %Y');`

> Thursday June 15 2017

Para mostrar la fecha escrita en español se debe configurar el idioma con la siguiente instrucción:

SQL `SET lc_time_names = 'es_ES';`
`SELECT DATE_FORMAT('2017-06-15', '%w, %e de %M de %Y');`

> jueves, 15 de junio de 2017

DATE_ADD

Usamos **DATE_ADD** para sumar o agregar un período de tiempo a un valor de tipo DATE o DATETIME.

SQL `SELECT DATE_ADD('2021-06-30', INTERVAL '3' DAY);`

> `2021-07-03`

SQL `SELECT DATE_ADD('2021-06-30', INTERVAL '9' MONTH);`

> `2022-03-30`

SQL `SELECT DATE_ADD('2021-06-30 09:30:00', INTERVAL '4' HOUR);`

> `2021-06-30 13:30:00`

DATE_SUB

Usamos **DATE_SUB** para restar o quitar un período de tiempo a un valor de tipo DATE o DATETIME.

SQL `SELECT DATE_SUB('2021-06-30', INTERVAL '3' DAY);`

> `2021-06-27`

SQL `SELECT DATE_SUB('2021-06-30', INTERVAL '9' MONTH);`

> `2020-09-30`

SQL `SELECT DATE_SUB('2021-06-30 09:30:00', INTERVAL '4' HOUR);`

> `2021-06-30 05:30:00`

CASE

Usamos **CASE** para **evaluar condiciones** y devolver la primera condición que se cumpla. En este ejemplo, la tabla resultante tendrá 4 columnas: id, titulo, rating, calificacion. Esta última columna mostrará los valores: Mala, Regular, Buena y Excelente; **según el rating** de la película.

SQL

```
SELECT id, titulo, rating,
CASE
    WHEN rating < 4 THEN 'Mala'
    WHEN rating BETWEEN 4 AND 6 THEN 'Regular'
    WHEN rating BETWEEN 7 AND 9 THEN 'Buena'
    ELSE 'Excelente'
END AS calificacion
FROM pelicula;
```

CASE (cont.)

A continuación, se muestra la tabla resultante después de haber aplicado la función **CASE**.

pelicula			
id	titulo	rating	calificacion
1	El Padrino	6	Regular
2	Tiburón	4	Regular
3	Jurassic Park	9	Buena
4	Titanic	10	Excelente
5	Matrix	3	Mala

Tipos de JOIN

DigitalHouse >
Coding School



**Certified Tech
Developer**
The Ultimate Degree

Índice

1. INNER
2. LEFT
3. RIGHT
4. Experimentando con LEFT y RIGHT

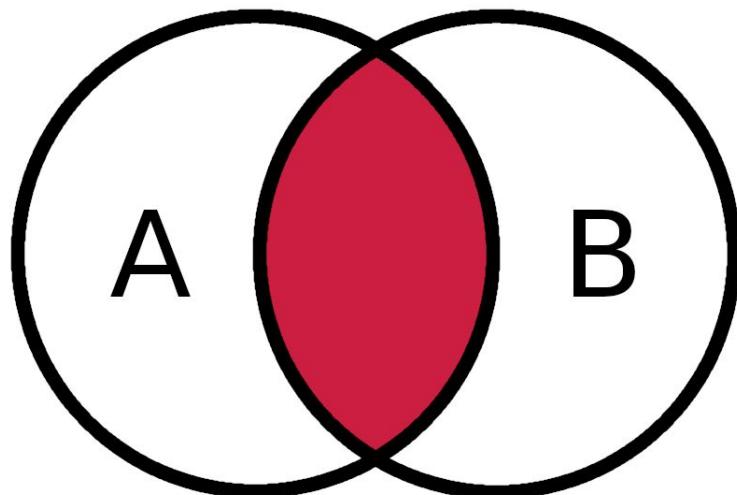
1 | INNER

INNER JOIN

El **INNER JOIN** entre dos tablas devuelve únicamente los registros que cumplen la condición indicada en la cláusula **ON**.

SQL

```
SELECT columna1, columna2, ...
FROM tabla A
INNER JOIN tabla B
ON condicion
```

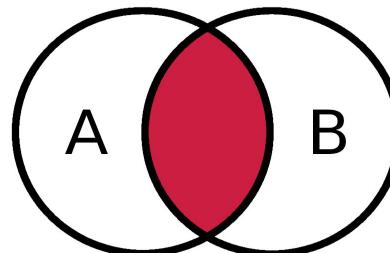


INNER JOIN

El **INNER JOIN** es la opción predeterminada y nos devuelve **todos los registros** donde **se cruzan dos o más tablas**. Por ejemplo, si tenemos una tabla cliente y otra factura, al cruzarlas con **INNER JOIN**, nos devuelve aquellos registros o filas donde haya un valor coincidente en ambas tablas.

cliente		
id	nombre	apellido
1	Juan	Perez
2	Clara	Sanchez
3	Marta	García

INNER JOIN



factura		
id	cliente_id	fecha
11	2	12/09/2019
12	null	20/09/2019
13	1	24/09/2019

INNER JOIN

El ejemplo anterior, se podría construir de la siguiente manera:

SQL

```
SELECT factura.id AS nro_factura, apellido, nombre, fecha  
FROM cliente  
INNER JOIN factura  
ON cliente.id = factura.cliente_id;
```

A continuación, se muestran los datos obtenidos:

nro_factura	apellido	nombre	fecha
11	Sanchez	Clara	12/09/2019
13	Perez	Juan	24/09/2019

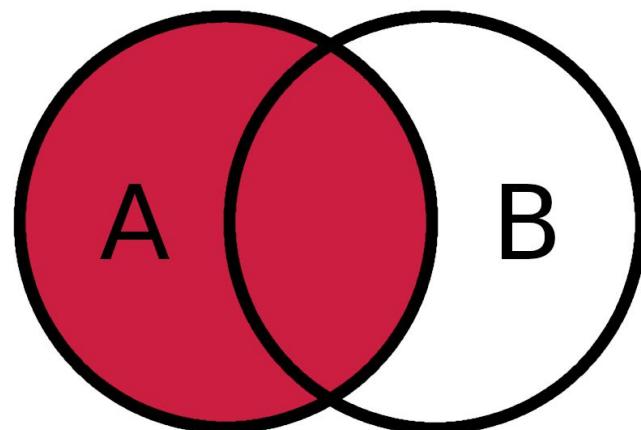
2 | LEFT

LEFT JOIN

El **LEFT JOIN** entre dos tablas devuelve todos los registros de la primera tabla (en este caso sería la tabla A), incluso cuando los registros no cumplan la condición indicada en la cláusula **ON**.

SQL

```
SELECT columna1, columna2, ...
FROM tabla A
LEFT JOIN tabla B
ON condicion
```

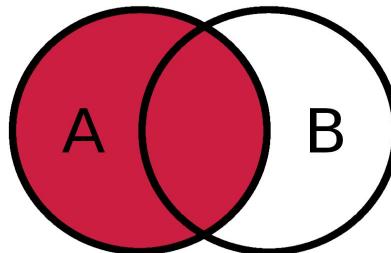


LEFT JOIN

Entonces, **LEFT JOIN** nos devuelve **todos** los registros donde se **cruzan dos o más tablas**. Incluso los registros de una primera tabla (A) que **no cumplen** con la condición indicada en la cláusula **ON**. Por ejemplo, si tenemos una tabla cliente y otra factura, al cruzarlas, nos devuelve aquellos registros donde haya un valor coincidente entre ambas, más los registros de aquellos clientes que no tengan una factura asignada.

cliente		
id	nombre	apellido
1	Juan	Perez
2	Clara	Sanchez
3	Marta	García

LEFT JOIN



factura		
id	cliente_id	fecha
11	2	12/09/2019
12	null	20/09/2019
13	1	24/09/2019

LEFT JOIN

El ejemplo anterior, se podría construir de la siguiente manera:

SQL

```
SELECT factura.id AS nro_factura, apellido, nombre, fecha  
FROM cliente  
LEFT JOIN factura  
ON cliente.id = factura.cliente_id;
```

A continuación, se muestran los datos obtenidos:

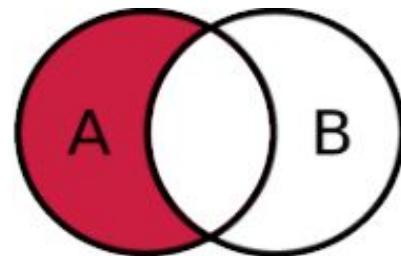
nro_factura	apellido	nombre	fecha
11	Sanchez	Clara	12/09/2019
13	Perez	Juan	24/09/2019
null	García	Marta	null

LEFT Excluding JOIN

Este tipo de **LEFT JOIN** nos devuelve únicamente los **registros** de una primera tabla (A), excluyendo los registros que cumplan con la condición indicada en la cláusula **ON**. Por ejemplo, si tenemos una tabla cliente y otra factura, al cruzarlas, nos devuelve solo aquellos registros de clientes que no tengan una factura asignada.

cliente		
id	nombre	apellido
1	Juan	Perez
2	Clara	Sanchez
3	Marta	García

LEFT Excluding JOIN



factura		
id	cliente_id	fecha
11	2	12/09/2019
12	null	20/09/2019
13	1	24/09/2019

LEFT Excluding JOIN

Continuando con el ejemplo, se podría construir de la siguiente manera:

SQL

```
SELECT factura.id AS nro_factura, apellido, nombre, fecha  
FROM cliente  
LEFT JOIN factura  
ON cliente.id = factura.cliente_id  
WHERE factura.id IS NULL;
```

A continuación, se muestran los datos obtenidos:

nro_factura	apellido	nombre	fecha
null	García	Marta	null

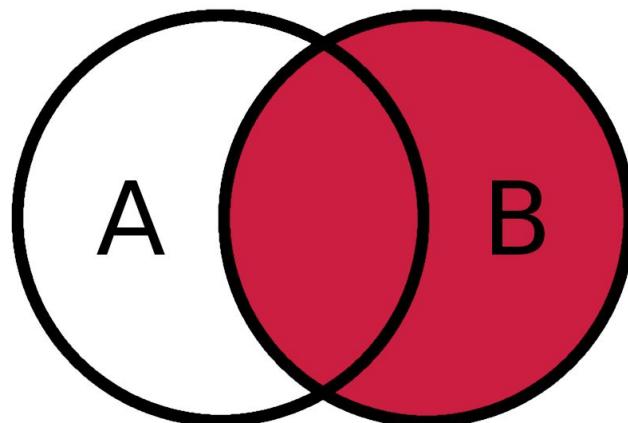
3 | RIGHT

RIGHT JOIN

El **RIGHT JOIN** entre dos tablas devuelve todos los registros de la segunda tabla, incluso cuando los registros no cumplen la condición indicada en la cláusula **ON**.

SQL

```
SELECT columna1, columna2, ...
FROM tabla A
RIGHT JOIN tabla B
ON condicion
```

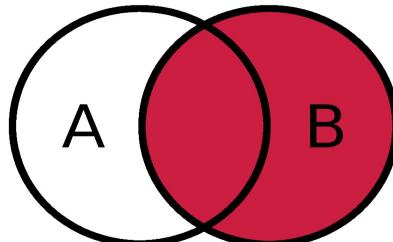


RIGHT JOIN

Entonces, **RIGHT JOIN** nos devuelve **todos** los registros donde se **cruzan dos o más tablas**. Incluso los registros de una segunda tabla (B) que **no cumplan** con la condición indicada en la cláusula **ON**. Por ejemplo, si tenemos una tabla cliente y otra factura, al cruzarlas, nos devuelve aquellos registros donde haya un valor coincidente entre ambas, más los registros de aquellas facturas que no tengan un cliente asignado.

cliente		
id	nombre	apellido
1	Juan	Perez
2	Clara	Sanchez
3	Marta	García

RIGHT JOIN



factura		
id	cliente_id	fecha
11	2	12/09/2019
12	null	20/09/2019
13	1	24/09/2019

RIGHT JOIN

El ejemplo anterior, se podría construir de la siguiente manera:

SQL

```
SELECT factura.id AS nro_factura, apellido, nombre, fecha  
FROM cliente  
RIGHT JOIN factura  
ON cliente.id = factura.cliente_id;
```

A continuación, se muestran los datos obtenidos:

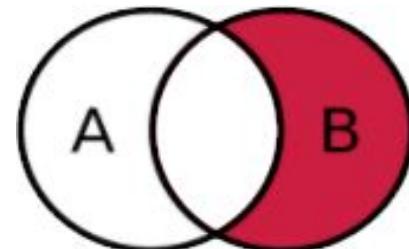
nro_factura	apellido	nombre	fecha
11	Sanchez	Clara	12/09/2019
12	null	null	20/09/2019
13	Perez	Juan	24/09/2019

RIGHT Excluding JOIN

Este tipo de **RIGTH JOIN** nos devuelve únicamente los registros de una segunda tabla (B), **excluyendo** los registros que cumplen con la condición indicada en la cláusula **ON**. Por ejemplo, si tenemos una tabla cliente y otra factura, al cruzarlas, nos devuelve solo aquellos registros de facturas que no tengan asignado un cliente.

cliente		
id	nombre	apellido
1	Juan	Perez
2	Clara	Sanchez
3	Marta	García

RIGHT Excluding JOIN



factura		
id	cliente_id	fecha
11	2	12/09/2019
12	null	20/09/2019
13	1	24/09/2019

RIGHT Excluding JOIN

Continuando con el ejemplo, se podría construir de la siguiente manera:

SQL

```
SELECT factura.id AS nro_factura, apellido, nombre, fecha  
FROM cliente  
RIGHT JOIN factura  
ON cliente.id = factura.cliente_id  
WHERE cliente.id IS NULL;
```

A continuación, se muestran los datos obtenidos:

nro_factura	apellido	nombre	fecha
12	null	null	20/09/2019

4

Experimentando con LEFT y RIGHT

“

¿Qué pasa si **intercambiamos**
de lugar las tablas o si
cambiamos LEFT por **RIGHT**?



”

LEFT JOIN -> RIGHT JOIN

Experimentemos con el último ejemplo que vimos.

S
Q
L

```
SELECT factura.id AS factura, apellido  
FROM cliente  
LEFT JOIN factura  
ON factura.cliente_id = cliente.id;
```

S
Q
L

```
SELECT factura.id AS factura, apellido  
FROM cliente  
RIGHT JOIN factura  
ON factura.cliente_id = cliente.id;
```

Comparemos los resultados:

factura	apellido
11	Sanchez
13	Perez
null	García

factura	apellido
11	Sanchez
12	null
13	Perez

Intercambiando tablas

Ahora, intercambiamos el lugar de las tablas implicadas.

S
Q
L

```
SELECT factura.id AS factura, apellido  
FROM factura  
LEFT JOIN cliente  
ON factura.cliente_id = cliente.id;
```

S
Q
L

```
SELECT factura.id AS factura, apellido  
FROM factura  
RIGHT JOIN cliente  
ON factura.cliente_id = cliente.id;
```

Comparemos los resultados:

factura	apellido
11	Sanchez
12	null
13	Perez

factura	apellido
11	Sanchez
13	Perez
null	García

DigitalHouse >
Coding School

Vistas



**Certified Tech
Developer**
The Ultimate Degree

¿Qué es una vista?

Una **vista** es un elemento de la base de datos que facilita el acceso a los datos de las tablas. Básicamente, su función es **guardar** un **SELECT**. Este es un recurso que nos permite visualizar los resultados abstrayéndonos de cómo esté definida una consulta.

Las vistas tienen la misma **estructura** que una **tabla** (filas y columnas). La diferencia es que solo se almacena la **definición de la consulta**, no así los datos. En otras palabras, una vista es una tabla virtual basada en el conjunto de resultados de una **consulta SQL**.

¿Para qué sirven las vistas?

- Para simplificar el acceso a los datos cuando se requiere la implementación de consultas complejas.
- Para impedir la modificación de datos por terceros.
- Para facilitar la consulta de datos a aquellas personas que no conozcan el modelo de datos o que no sean expertos en SQL.

¿Qué hay que saber sobre las vistas?

- Una vista se **ejecuta** en el momento en que se invoca.
- Los **nombres** de las vistas deben ser **únicos** (no se pueden usar nombres de tablas existentes).
- Solamente se pueden incluir **sentencias SQL** de tipo **SELECT**.
- Los **campos** de las vistas heredan los **tipos de datos** de la tabla.
- El conjunto de resultados que devuelve una vista es **inmodificable**, a diferencia de lo que sucede con el conjunto de resultados de una tabla.

Creación de vistas

Una vista se crea con la cláusula **CREATE VIEW**:

SQL CREATE VIEW nombre_de_la_vista AS consulta SQL;

SQL CREATE VIEW canciones_de_rock AS
 SELECT canciones.id, canciones.nombre, generos.nombre AS genero
 FROM canciones
 INNER JOIN generos
 ON canciones.id_genero = generos.id
 WHERE generos.nombre IN ('Rock', 'Rock And Roll');

Modificación de vistas

Usamos la cláusula **ALTER VIEW** para modificar o reemplazar una vista.

SQL

```
ALTER VIEW nombre_de_la_vista AS consulta SQL;
```

SQL

```
ALTER VIEW vista_coche AS SELECT * FROM coche WHERE marca = 'Fiat';
```

Eliminación de vistas

Utilizamos la cláusula **DROP VIEW** para eliminar una vista.

SQL

```
DROP VIEW nombre_de_la_vista;
```

SQL

```
DROP VIEW vista_coche;
```

Invocación de vistas

Si bien las vistas son elementos diferentes a las tablas, la **invocación** es la misma que la de una tabla. Es decir, se utiliza la cláusula **SELECT**.

Invocando una tabla:

```
SQL SELECT * FROM coche;
```

Invocando una vista:

```
SQL SELECT * FROM vista_coche;
```

Invocando una vista junto con la cláusula **WHERE**:

```
SQL SELECT * FROM vista_coche WHERE id > 10;
```