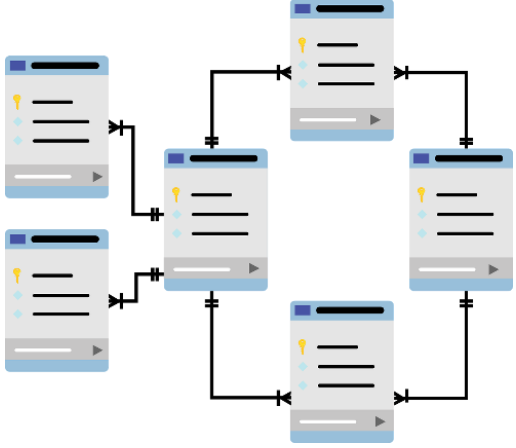


Bases de Datos

Capítulo 1: Introducción a SQL

SQL (Structured Query Language) es un lenguaje de programación estándar diseñado para gestionar y manipular bases de datos relacionales, permitiendo a los usuarios interactuar con las bases de datos para realizar diversas tareas como crear, modificar y eliminar bases de datos y tablas, así como insertar, actualizar y recuperar datos.

Características del lenguaje:

1-. Consultas de datos: Permite realizar consultas simples o complejas para recuperar información específica de una o varias tablas de una base de datos.

2-. Manipulación de datos: Puedes agregar, actualizar y eliminar registros.

3-. Definición de esquemas: Permite definir la estructura de una base de datos, incluyendo la creación de tablas, la especificación de tipos de datos, la definición de restricciones y la creación de índices para mejorar el rendimiento.

4-. Control de acceso: Proporciona funciones para controlar quién puede acceder a la base de datos y qué acciones pueden realizar los usuarios autorizados. Esto incluye la creación de usuarios, la asignación de permisos y la gestión de roles.

Un **RDBMS, o Sistema de Gestión de Bases de Datos Relacionales** (por sus siglas en inglés Relational Database Management System), es un tipo de software diseñado para administrar, almacenar y recuperar datos organizados en una estructura relacional.

Estos sistemas permiten a los usuarios definir la estructura de los datos (estos se organizan en tablas que consisten en filas y columnas), y se establecen relaciones entre las tablas mediante claves primarias y claves foráneas. También permiten realizar consultas complejas, administrar la seguridad de la base de datos y garantizar la integridad y consistencia de los datos almacenados.

Algunos ejemplos populares incluyen MySQL, PostgreSQL, Oracle Database y Microsoft SQL Server.

Curiosidades del lenguaje:

1-. Se convirtió en un estándar del Instituto Nacional Americano de Estándares (ANSI) en 1986 y posteriormente de la Organización Internacional de Normalización (ISO) en 1987.

2-. La mayoría de los sistemas de gestión de bases de datos tienen sus propias extensiones propietarias además del estándar SQL. Estas extensiones pueden incluir características adicionales, funciones específicas del proveedor o mejoras de rendimiento diseñadas para

satisfacer las necesidades particulares de los usuarios o para diferenciar el producto en el mercado.

3-. Las palabras clave no distinguen entre mayúsculas y minúsculas. Por lo tanto, escribir SELECT o select produce el mismo resultado y es válido en términos de sintaxis SQL.

4-. Algunos sistemas de bases de datos requieren que cada instrucción SQL finalice con un punto y coma.

5-. Nombrar los campos de forma coherente y única en una tabla es esencial para una buena gestión de la base de datos. Esto ayuda a identificar claramente cada campo, lo que facilita acceder y trabajar con los datos almacenados. Además, asignar el tipo de datos correcto a cada campo garantiza la integridad de los datos y simplifica la realización de consultas y operaciones en la base de datos.

¿Qué es un campo?

Un **campo** es una columna de una tabla que está diseñada para mantener información específica sobre cada registro de la tabla.

¿Qué es un registro?

Un **registro**, también llamado fila, es cada entrada individual que existe en una tabla.

Capítulo 2: Sentencias SQL

¿Qué son las sentencias?

Las **sentencias** en SQL son instrucciones que se utilizan para interactuar con una base de datos relacional.

SELECT: Se utiliza para especificar qué columnas deseamos extraer o recuperar de una o varias tablas en una base de datos.

FROM: Se utiliza para especificar de qué tabla o tablas deseamos recuperar los datos.

```
SELECT CustomerName, City, ...
FROM Customers;
```

Number of Records: 91

CustomerName	City
Alfreds Futterkiste	Berlin
Ana Trujillo Emparedados y helados	México D.F.
Antonio Moreno Taquería	México D.F.
Around the Horn	London
Berglunds snabbköp	Luleå
Blauer See Delikatessen	Mannheim
Blondel père et fils	Strasbourg
Bólido Comidas preparadas	Madrid

Nota: Si utilizamos la sintaxis **SELECT *** podremos seleccionar todas las columnas de una tabla sin especificar el nombre de cada.

```
SELECT *
FROM Customers;
```

Number of Records: 91

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico

Nota: La instrucción **SELECT DISTINCT** se utiliza para seleccionar valores únicos de una columna en una tabla. En otras palabras, elimina los duplicados de los resultados de la consulta, mostrando solo los valores distintos de esa columna.

```
SELECT DISTINCT Country
FROM Customers;
```

Number of Records: 21

Country
Argentina
Austria
Belgium

WHERE: Se utiliza para filtrar los resultados de una consulta basándose en una condición específica.

```
SELECT *
FROM Customers
WHERE Country = 'Mexico';
```

Number of Records: 5

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
13	Centro comercial Moctezuma	Francisco Chang	Sierras de Granada 9993	México D.F.	05022	Mexico

Nota: Los valores de texto, como nombres, direcciones, etc., generalmente deben ir entre comillas simples, mientras que los valores numéricos no necesitan comillas.

Nota: Los operadores de comparación se utilizar mucho en esta cláusula.

Igual a (**=**)

```
SELECT *
FROM Customers
WHERE Price = 18;
```

Diferente de (**<>**)

```
SELECT *
FROM Customers
WHERE Price <> 18;
```

Mayor que (**>=**)

```
SELECT *
FROM Customers
WHERE Price >= 30;
```

Menor que (**<=**)

```
SELECT *
FROM Customers
WHERE Price <= 30;
```

BETWEEN (Compara si un valor de cualquier tipo de dato está dentro de un rango)

```
SELECT *
FROM Price
WHERE Price BETWEEN 50 AND 60;
```

Nota: Las fechas deben tener guiones medios para separar el año, el mes y el día en el formato de fecha estándar.

IN (Verifica si un valor esta dentro de una lista)

```
SELECT *
FROM Customers
WHERE City IN ('Paris','London');
```

LIKE (Compara si un valor coincide parcialmente con un patrón)

```
SELECT *
FROM Customers
WHERE City LIKE 's%';
```

ORDER BY: Se utiliza para ordenar los resultados de una consulta según los valores de una o más columnas en un orden ascendente (**ASC**) o descendente (**DESC**).

```
SELECT *
FROM Products
ORDER BY Price DESC;
```

Number of Records: 77

ProductID	ProductName	SupplierID	CategoryID	Unit	Price
38	Côte de Blaye	18	1	12 - 75 cl bottles	263.5
29	Thüringer Rostbratwurst	12	6	50 bags x 30 sausgs.	123.79
9	Mishi Kobe Niku	4	6	18 - 500 g pkgs.	97

Nota: Cuando ordenas valores de cadena (texto) utilizando esta cláusula, los valores se ordenarán alfabéticamente de forma predeterminada en orden ascendente (A-Z).

Nota: Cuando especificas múltiples columnas para ordenar, SQL organizará los resultados priorizando la primera columna especificada en la lista de ordenación. En caso de que existan valores iguales en esta primera columna, SQL utilizará la segunda columna para desempatar, y así sucesivamente.

```
SELECT *
FROM Customers
ORDER BY Country, CustomerName;
```

Number of Records: 91

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
12	Cactus Comidas para llevar	Patricio Simpson	Cerrito 333	Buenos Aires	1010	Argentina
54	Océano Atlántico Ltda.	Yvonne Moncada	Ing. Gustavo Moncada 8585 Piso 20-A	Buenos Aires	1010	Argentina
64	Rancho grande	Sergio Gutiérrez	Av. del Libertador 900	Buenos Aires	1010	Argentina

Nota: Puedes usar ASC y DESC de forma independiente para cada columna en la cláusula.

```
SELECT *
FROM Customers
ORDER BY Country ASC, CustomerName DESC;
```

Number of Records: 91

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
64	Rancho grande	Sergio Gutiérrez	Av. del Libertador 900	Buenos Aires	1010	Argentina
54	Océano Atlántico Ltda.	Yvonne Moncada	Ing. Gustavo Moncada 8585 Piso 20-A	Buenos Aires	1010	Argentina
12	Cactus Comidas para llevar	Patricio Simpson	Cerrito 333	Buenos Aires	1010	Argentina
59	Piccolo und mehr	Georg Pipps	Geislweg 14	Salzburg	5020	Austria

Nota: Cuando no se especifica ni ASC ni DESC en una cláusula ORDER BY, por defecto se asume que el orden es ascendente.

AND: Se utiliza para combinar múltiples condiciones en una cláusula WHERE, especificando que todas las condiciones deben ser verdaderas para que se cumpla la condición general.

```
SELECT *
FROM Customers
WHERE Country = 'Spain' AND CustomerName LIKE 'G%';
```

Number of Records: 2

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
29	Galería del gastrónomo	Eduardo Saavedra	Rambla de Cataluña, 23	Barcelona	08022	Spain
30	Godos Cocina Típica	José Pedro Freyre	C/ Romero, 33	Sevilla	41101	Spain

OR: Se utiliza para combinar múltiples condiciones en una cláusula WHERE, especificando que al menos una de las condiciones debe ser verdadera para que se cumpla la condición general.

```
SELECT *
FROM Customers
WHERE Country = 'Germany' OR Country 'Spain';
```

Number of Records: 16

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
6	Blauer See Delikatessen	Hanna Moos	Forsterstr. 57	Mannheim	68306	Germany
8	Bólido Comidas preparadas	Martín Sommer	C/ Araquil, 67	Madrid	28023	Spain

Nota: Podemos combinar los operadores AND y OR en una misma condición para construir condiciones más complejas y flexibles.

```
SELECT *
FROM Customers
```

```
WHERE Country = 'Spain'
AND (CustomerName LIKE 'G%' OR CustomerName LIKE 'R%');
```

Number of Records: 3

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
29	Galería del gastrónomo	Eduardo Saavedra	Rambla de Cataluña, 23	Barcelona	08022	Spain
30	Godos Cocina Típica	José Pedro Freyre	C/ Romero, 33	Sevilla	41101	Spain
69	Romero y tomillo	Alejandra Camino	Gran Vía, 1	Madrid	28001	Spain

NOT: Se utiliza para negar una condición en una cláusula WHERE, HAVING o CHECK. Básicamente, invierte el resultado de una condición booleana, es decir, convierte las afirmaciones verdaderas en falsas y las afirmaciones falsas en verdaderas.

```
SELECT *
FROM Customers
WHERE NOT Country = 'Spain';
```

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico

Nota: Esta palabra clave se puede combinar con otras instrucciones, como por ejemplo

```
SELECT *
FROM Customers
WHERE CustomerName NOT LIKE 'A%';
```

```
SELECT *
FROM Customers
WHERE City NOT IN ('Paris','London');
```

INSERT INTO: Se utiliza para insertar nuevos registros en una tabla específica de una base de datos.

```
INSERT INTO Customers (CustomerName, City, Country)
VALUES ('Cardinal','Stavanger','Norway');
```

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
89	White Clover Markets	Karl Jablonski	305 - 14th Ave. S. Suite 3B	Seattle	98128	USA
90	Wilman Kala	Matti Karttunen	Keskuskatu 45	Helsinki	21240	Finland
91	Wolski	Zbyszek	ul. Filtrowa 68	Walla	01-012	Poland
92	Cardinal	null	null	Stavanger	null	Norway

Nota: Si insertamos múltiples registros en una tabla, podemos separarlos con una coma al final.

Nota: Es posible escribir la declaración INSERT INTO de dos maneras diferentes

1-. Especificando todas las columnas y sus valores correspondientes:

```
INSERT INTO nombre_de_la_tabla (columna1, columna2, ...)
VALUES (valor1, valor2, ...);
```

En esta forma, especificas explícitamente las columnas en las que deseas insertar los valores y proporcionas los valores correspondientes en el mismo orden.

2-. Especificando solo los valores y dejando que la base de datos asigne los valores predeterminados a las columnas no especificadas:

```
INSERT INTO nombre_de_la_tabla
VALUES (valor1, valor2, ...);
```

En esta forma, no especificas las columnas en las que deseas insertar los valores. En su lugar, proporcionas los valores directamente y la base de datos los insertará en las columnas correspondientes en el mismo orden en que fueron definidas en la tabla. Esto asume que las columnas que no se mencionan tendrán valores predeterminados o permiten valores nulos.

Valor NULL: Representa la ausencia de un valor en una columna de una tabla. No es lo mismo que un valor cero, una cadena vacía o cualquier otro valor específico.

Nota: No podemos utilizar operadores de comparación como =, < o <> para comprobar directamente si un valor es NULL o no. En su lugar, debemos usar los siguientes operadores

IS NULL (Comprueba si un valor es NULL)

```
SELECT CustomerName, ContactName, Address
FROM Customers
WHERE Address IS NULL;
```

Number of Records: 0

CustomerName	ContactName	Address

IS NOT NULL (Comprueba si un valor no es NULL)

```
SELECT CustomerName, ContactName, Address
FROM Customers
WHERE Address IS NOT NULL;
```

Number of Records: 91

CustomerName	ContactName	Address
Alfreds Futterkiste	Maria Anders	Obere Str. 57
Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222
Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312

UPDATE: Se utiliza para modificar los datos existentes en una tabla. Básicamente, te permite actualizar uno o más campos de uno o varios registros en una tabla específica.

```
UPDATE Customers
SET ContactName = 'Alfred Schmidt', City = 'Frankfurt'
WHERE CustomerID = 1;
```

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Alfred Schmidt	Obere Str. 57	Frankfurt	12209	Germany

Nota: Es posible actualizar múltiples registros, un ejemplo es el siguiente

```
UPDATE Customers
SET ContactName = 'Juan'
WHERE Country = 'Mexico';
```

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Alfred Schmidt	Obere Str. 57	Frankfurt	12209	Germany
2	Ana Trujillo Emparedados y helados	Juan	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Juan	Mataderos 2312	México D.F.	05023	Mexico

DELETE: Se utiliza para eliminar registros de una tabla en una base de datos. Cuando ejecutas la instrucción, especificas una condición que determina qué registros deben eliminarse.

```
DELETE FROM Customers
WHERE CustomerName = 'Alfreds Futterkiste';
```

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden

Nota: Es posible eliminar todas las filas de una tabla sin eliminar la estructura de la tabla, los atributos y los índices asociados.

```
DELETE FROM Customers;
```

Nota: La siguiente instrucción elimina completamente una tabla. Esta acción elimina tanto la estructura de la tabla como todos los datos almacenados en ella, por lo que debe utilizarse con precaución.

```
DROP TABLE Customers;
```

TOP: Se utiliza para limitar el número de filas que se devuelven en el resultado de una consulta. Esto es útil en tablas grandes con miles o incluso millones de registros. En estos casos, devolver un gran número de registros puede afectar significativamente al rendimiento de la consulta y al sistema en general.

```
SELECT TOP 3 *
FROM Customers
WHERE Country = 'Germany';
```

Number of Records: 3

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
6	Blauer See Delikatessen	Hanna Moos	Forsterstr. 57	Mannheim	68306	Germany
17	Drachenblut Delikatessend	Sven Ottlieb	Walserweg 21	Aachen	52066	Germany

Nota: Es importante tener en cuenta que la cláusula TOP puede variar en su implementación dependiendo del sistema de gestión de bases de datos que estés utilizando.

MySQL

```
SELECT *
FROM Customers
WHERE Country = 'Germany'
LIMIT 3;
```

Oracle / PostgreSQL

```
SELECT *
FROM Customers
WHERE Country = 'Germany'
FETCH FIRST 3 ROWS ONLY;
```

Nota: TOP PERCENT devuelve un porcentaje específico de registros del conjunto de resultados, no un número fijo de registro.

SQL SERVER / MS Access

```
SELECT TOP 50 PERCENT *
FROM Customers;
```

Oracle / PostgreSQL

```
SELECT *
FROM Customers
FETCH FIRST 50 PERCENT ROWS ONLY;
```

¿Qué son las funciones de agregado?

Las **funciones de agregado** se utilizan para realizar operaciones en conjuntos de valores y devolver un único valor resumido como resultado. Estas funciones operan sobre conjuntos de filas y pueden utilizarse en combinación con la instrucción SELECT para realizar cálculos en columnas de una tabla.

Nota: Las funciones agregadas de SQL más utilizadas son

MIN (Encuentra el valor mínimo en una columna)

```
SELECT MIN(Price)
FROM Products;
```

Expr1000
2.5

```
SELECT MAX(Price)
FROM Products;
```

Expr1000
263.5

Nota: Cuando se utilizan funciones de agregado, la columna resultante no tendrá un nombre descriptivo por defecto. Para asignar un nombre significativo a esta columna (o, en caso necesario, a una tabla), se utiliza la palabra clave AS. Sin embargo, un alias solo existe mientras dure la consulta.

Nota: Podemos usar guiones bajos para separar palabras al asignar alias largos a columnas o tablas en SQL.

```
SELECT MIN(Price) AS SmallestPrice
FROM Products;
```

SmallestPrice
2.5

Nota: GROUP BY se utiliza para agrupar filas que tienen los mismos valores en una o más columnas y aplicar funciones de agregado a los grupos resultantes.

```
SELECT MIN(Price) AS SmallestPrice, CategoryID
FROM Products
GROUP BY CategoryID;
```

Number of Records: 8

SmallestPrice	CategoryID
4.5	1
10	2
9.2	3

Nota: Esta consulta cuenta el número de clientes por país y los ordena de manera descendente según la cantidad de clientes.

```
SELECT COUNT(CustomerID), Country
FROM Customers
GROUP BY Country;
GROUP BY COUNT (CustomerID) DESC;
```

Number of Records: 21

Expr1000	Country
13	USA
11	France
11	Germany

COUNT: Se utiliza para contar el número de filas en un conjunto de resultados o el número de elementos en una columna específica que cumple con ciertas condiciones.

```
SELECT COUNT(*)
FROM Products
```

Expr1000
77

Nota: Los valores NULL no se tienen en cuenta para el conteo.

Nota: Algunas combinaciones que se pueden hacer con esta instrucción son

```
SELECT COUNT(DISTINCT Price)
FROM Products;
```

Number of Records: 1

COUNT(DISTINCT Price)
62

```
SELECT COUNT(*) AS Number of records, CategoryID
FROM Products
GROUP BY CategoryID;
```

Number of Records: 8

Number of records	CategoryID
12	1
12	2
13	3

SUM: Se utiliza para calcular la suma total de los valores en una columna numérica de una tabla.

```
SELECT SUM(Quantity) AS total
FROM OrdenDetails
WHERE ProductID = 11;
```

Number of Records: 1

Expr1000
182

Nota: Dentro de la función, el parámetro también puede ser una expresión en lugar de solo una columna. Esto significa que puedes realizar cálculos o usar funciones dentro de la función.

```
SELECT SUM(Quantity * 10)
FROM OrdenDetails;
```

Expr1000
127430

AVG: Se utiliza para calcular el promedio de los valores en una columna numérica de una tabla.

```
SELECT AVG(Price)
FROM Products
WHERE CategoryID = 1;
```

Number of Records: 1

Expr1000
37.9792

Nota: Con esta instrucción podemos seleccionar todos los productos de una tabla cuyo precio sea mayor que el precio promedio de todos los productos.

```
SELECT *
FROM Products
WHERE price > (SELECT AVG(price) FROM Products);
```

Number of Records: 25

ProductID	ProductName	SupplierID	CategoryID	Unit	Price
7	Uncle Bob's Organic Dried Pears	3	7	12 - 1 lb pkgs.	30
8	Northwoods Cranberry Sauce	3	2	12 - 12 oz jars	40
9	Mishi Kobe Niku	4	6	18 - 500 g pkgs.	97

¿Qué son los comodines?

Los **comodines** son caracteres especiales que se utilizan en combinación con los operadores LIKE y NOT LIKE para realizar búsquedas de valores que coincidan con ciertos criterios sin necesidad de especificar el patrón completo.

Nota: NOT LIKE se utiliza para seleccionar filas de una tabla donde el valor de una columna no coincide con un patrón especificado. Es esencialmente la negación de la cláusula LIKE.

```
SELECT *
FROM Customers
WHERE City NOT LIKE 'a%';
```

Number of Records: 88

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico

Nota: Los comodines más comunes son

1-. % (porcentaje): Este comodín se utiliza para representar cero, uno o múltiples caracteres en una cadena. Por ejemplo, si quieres encontrar todas las palabras que comiencen con "a", puedes usar 'a%'. Esto coincidirá con cualquier cadena que comience con "a", seguido de cero o más caracteres adicionales.

```
SELECT *
FROM Customers
WHERE CustomerName LIKE 'a%';
```

Number of Records: 4

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico

2-. _ (guión bajo): Este comodín se utiliza para representar un solo carácter en una cadena. Por ejemplo, si quieres encontrar todas las palabras de cinco letras, puedes usar '_____'. Esto coincidirá con cualquier cadena que tenga exactamente cinco caracteres.

```
SELECT *
FROM Customers
WHERE City LIKE 'L___on';
```

Number of Records: 6

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
11	B's Beverages	Victoria Ashworth	Fauntleroy Circus	London	EC2 5NT	UK
16	Consolidated Holdings	Elizabeth Brown	Berkeley Gardens 12 Brewery	London	WX1 6LT	UK

3-. [] (corchetes): Se utilizan para especificar un conjunto de caracteres posibles en una posición determinada en una cadena. Por ejemplo, si quieres encontrar todas las palabras que comiencen con una vocal, puedes usar '[aeiou]%'.

```
SELECT *
FROM Customers
WHERE CustomerName LIKE '[bsp]%';
```

Number of Records: 18

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden
6	Blauer See Delikatessen	Hanna Moos	Forsterstr. 57	Mannheim	68306	Germany
7	Blondel père et fils	Frédérique Citeaux	24, place Kléber	Strasbourg	67000	France

Nota: El guión (-) dentro de los corchetes se utiliza para especificar un rango de caracteres dentro de un patrón de búsqueda.

```
SELECT *
FROM Customers
WHERE CustomerName LIKE '[a-f]%';
```

Number of Records: 29

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico

Nota: Dentro de corchetes, el ! indica negación de un conjunto de caracteres.

```
SELECT *
FROM Customers
WHERE City LIKE '[!acf]%';
```

Number of Records: 81

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico

Concatenar columnas en SQL significa combinar los valores de dos o más columnas en una sola cadena de texto.

Nota: Esto se logra habitualmente mediante la función de concatenación ofrecida por la mayoría de los sistemas de gestión de bases de datos. Algunas de las opciones más comunes incluyen la función **CONCAT**, el operador || y **CONCAT_WS**.

```
SELECT CustomerName, CONCAT(Address,', ',PostalCode,',
',City,', ',Country) AS Address
FROM Customers;
```

```
SELECT CustomerName, (Address || ', ' || PostalCode || ' ' ||
City || ', ' || Country) AS Address
FROM Customers;
```

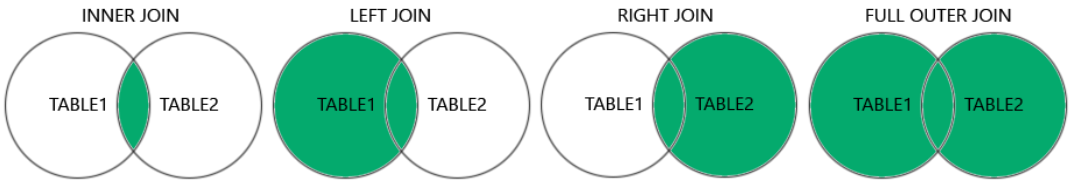
```
SELECT CustomerName, CONCAT_WS(', ', CustomerName, Address,
City) AS Address
FROM Customers;
```

Number of Records: 91

CustomerName	Address
Alfreds Futterkiste	Obere Str. 57, 12209 Berlin, Germany
Ana Trujillo Emparedados y helados	Avda. de la Constitución 2222, 05021 México D.F., Mexico
Antonio Moreno Taquería	Mataderos 2312, 05023 México D.F., Mexico

¿Qué son los JOINS?

Los **JOINS** se utilizan para combinar filas de dos o más tablas basándose en una condición relacionada entre ellas. Permiten recuperar datos de múltiples tablas en una sola consulta, lo que facilita la recuperación de información relacionada almacenada en diferentes tablas de una base de datos relacional.



1-. INNER JOIN: Devuelve únicamente las filas que tienen una coincidencia en ambas tablas basándose en la condición de unión especificada.

```
SELECT ProductID, ProductName, CategoryName
FROM Products
INNER JOIN Categories ON Products.CategoryID = Categories.CategoryID
ORDER BY ProductID ASC;
```

Number of Records: 77

ProductID	ProductName	CategoryName
1	Chais	Beverages
2	Chang	Beverages
3	Aniseed Syrup	Condiments

Nota: Se recomienda incluir el nombre de la tabla al especificar las columnas en la instrucción.

```
SELECT Products.ProductID, Products.ProductName, Categories.CategoryName
FROM Products
INNER JOIN Categories ON Products.CategoryID = Categories.CategoryID
ORDER BY ProductID ASC;
```

Nota: JOIN y INNER JOIN se utilizan indistintamente para realizar la misma operación.

Nota: Para unir tres tablas en una consulta, podemos seguir una estructura similar a la de unir dos tablas, solo que necesitaremos agregar más cláusulas JOIN.

```
SELECT Orders.OrderID, Customers.CustomerName, Shippers.ShipperName
FROM ((Orders
INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID)
INNER JOIN Shippers ON Orders.ShipperID = Shippers.ShipperID)
ORDER BY OrderID;
```

Number of Records: 196

OrderID	CustomerName	ShipperName
10248	Wilman Kala	Federal Shipping
10249	Tradição Hipermercados	Speedy Express
10250	Hanari Carnes	United Package

2-. LEFT JOIN (o LEFT OUTER JOIN): Devuelve todas las filas de la tabla izquierda (la primera tabla en la cláusula JOIN) y las filas coincidentes de la tabla derecha (la segunda tabla en la cláusula JOIN). Si no hay coincidencias en la tabla derecha, se devuelven valores nulos.

```
SELECT Customers.CustomerName, Orders.OrderID
FROM Customers
INNER JOIN Orders ON Customers.CustomerID = Orders.CustomerID)
ORDER BY Customers.CustomerName;
```

Number of Records: 213

CustomerName	OrderID
Alfreds Futterkiste	
Ana Trujillo Emparedados y helados	10308
Antonio Moreno Taquería	10365

3-. RIGHT JOIN (o RIGHT OUTER JOIN): Devuelve todas las filas de la tabla derecha y las filas coincidentes de la tabla izquierda. Si no hay coincidencias en la tabla izquierda, se devuelven valores nulos.

```
SELECT Orders.OrderID, Employees.LastName, Employees.FirstName
FROM Orders
RIGHT JOIN Employees ON Orders.EmployeeID = Employees.EmployeeID
ORDER BY Orders.OrderID;
```

Number of Records: 197

OrderID	LastName	FirstName
	West	Adam
10248	Buchanan	Steven
10249	Suyama	Michael

4-. FULL JOIN (o FULL OUTER JOIN): Devuelve todas las filas de ambas tablas, coincidentes o no. Si no hay coincidencias, se devuelven valores nulos en las columnas de la tabla sin coincidencias.

```
SELECT Customers.CustomerName, Orders.OrderID
FROM Customers
FULL OUTER JOIN Orders ON Customers.CustomerID = Orders.CustomerID
ORDER BY Customers.CustomerName;
```

CustomerName	OrderID
Null	10309
Null	10310
Alfreds Futterkiste	*Null*

¿Qué es una autocombinación?

Una **autocombinación**, también conocida como autojoin o autounión, ocurre cuando una tabla se une consigo misma en una consulta. Esto puede ser útil cuando necesitas comparar filas dentro de la misma tabla, por ejemplo, para encontrar relaciones entre datos en la misma tabla**.**

Nota: En una autocombinación, se utilizan alias para diferenciar las instancias de la misma tabla en la consulta. Por lo general, se utiliza una condición de unión que compara columnas de la misma tabla.

```
SELECT A.CustomerName AS CustomerName1, B.CustomerName AS CustomerName2,
A.City
FROM Customers A, Customers B
WHERE A.CustomerID <> B.CustomerID AND A.City = B.City
ORDER BY A.City;
```

Number of Records: 88

CustomerName1	CustomerName2	City
Océano Atlántico Ltda.	Cactus Comidas para llevar	Buenos Aires
Cactus Comidas para llevar	Océano Atlántico Ltda.	Buenos Aires
Rancho grande	Océano Atlántico Ltda.	Buenos Aires

Nota: La condición A.CustomerID <> B.CustomerID asegura que no estamos comparando un cliente consigo mismo, evitando duplicados.

UNION: Se utiliza para combinar los resultados de dos o más consultas en una sola lista de resultados. Es importante destacar que UNION elimina automáticamente duplicados del conjunto de resultados combinados, a menos que se use UNION ALL, que incluye todas las filas, incluidas las duplicadas.

```
SELECT City, Country
FROM Customers
WHERE Country = 'Germany'
UNION
SELECT City, Country
FROM Suppliers
WHERE Country = 'Germany'
ORDER BY City;
```

Number of Records: 13

City
Aachen
Berlin
Brandenburg

Nota: Algunas características clave son

1-. Las consultas en ambos lados de UNION deben tener el mismo número de columnas.

2-. Las columnas en las consultas deben ser del mismo tipo de datos o compatible, de lo contrario, se producirá un error.

Nota: En este ejemplo, la primera columna en el conjunto de resultados combinados se llamará Type y contendrá los valores 'Customer' o 'Supplier', dependiendo de la fila provenga de la primera o segunda consulta. No es necesario especificar un alias adicional para la segunda consulta porque no es necesario cambiar el nombre de esa columna.

```
SELECT 'Supplier' AS Type, ContactName, City, Country
FROM Customers
UNION
SELECT 'Supplier', ContactName, City, Country
FROM Suppliers
```

Number of Records: 120

Type	ContactName	City	Country
Customer	Alejandra Camino	Madrid	Spain
Customer	Alexander Feuer	Leipzig	Germany
Customer	Ana Trujillo	México D.F.	Mexico

HAVING: Se para filtrar resultados de grupos específicos en una consulta que utiliza la cláusula GROUP BY. Mientras que la cláusula WHERE se utiliza para filtrar filas antes de que se agrupen, la cláusula HAVING se utiliza para aplicar condiciones de filtrado a los grupos resultantes después de que se han agrupado.

```
SELECT COUNT(CustomerID), Country
FROM Customers
GROUP BY Country
HAVING COUNT(CustomerID) > 5;
ORDER BY COUNT(CustomerID) DESC;
```

Number of Records: 5

Expr1000	Country
13	USA
11	Germany
11	France

EXISTS: Se utiliza para verificar la existencia de registros en una subconsulta. Devuelve verdadero si la subconsulta especificada devuelve uno o más registros, y falso si la subconsulta no devuelve ningún registro.

```
SELECT SupplierName
FROM Suppliers
WHERE EXISTS (SELECT ProductName FROM Products WHERE Products.SupplierID
= Suppliers.supplierID AND Price < 20);
```

Number of Records: 24

SupplierName
Exotic Liquid
New Orleans Cajun Delights
Tokyo Traders

Nota: Esta consulta selecciona el nombre de los proveedores que tienen al menos un producto con un precio inferior a 20. La subconsulta dentro de la cláusula EXISTS busca cualquier producto con un precio menor a 20 asociado con el proveedor actual. Si la subconsulta devuelve algún resultado para un proveedor específico, la condición EXISTS se evalúa como verdadera y el nombre del proveedor se selecciona en el resultado final.

ANY: Compara un valor con cualquier valor en el conjunto resultante de la subconsulta. Devuelve verdadero si el valor dado coincide con al menos uno de los valores devueltos por la subconsulta.

```
SELECT ProductName
FROM Products
```

```
WHERE ProductID = ANY (SELECT ProductID FROM OrderDetails WHERE Quantity
= 10);
```

Number of Records: 31

ProductName
Chais
Chang
Chef Anton's Cajun Seasoning

Nota: La condición ProductID = ANY (...) significa que la consulta principal seleccionará los ProductName de la tabla Products donde el ProductID coincida con al menos uno de los ProductID devueltos por la subconsulta. Entonces, si en la tabla Products hay un registro con un ProductID de 1 y en la tabla OrderDetails hay uno o más registros con un ProductID de 1 y una cantidad de 10, entonces ese ProductName será seleccionado en el resultado de la consulta principal.

ALL: Compara un valor con todos los valores en el conjunto resultante de la subconsulta. Devuelve verdadero si el valor dado coincide con todos los valores devueltos por la subconsulta.

```
SELECT ProductName
FROM Products
WHERE ProductID = ALL (SELECT ProductID FROM OrderDetails WHERE Quantity
= 10);
```

Number of Records: 0

ProductName

Nota: En este ejercicio, estamos esperando que todos los ProductID devueltos por la subconsulta (es decir, todos los registros en OrderDetails donde la cantidad es igual a 10) coincidan exactamente con el ProductID en la tabla Products. Esto significaría que para cada registro en OrderDetails con una cantidad igual a 10, hay un registro correspondiente en Products con el mismo ProductID.

SELECT INTO: Se utiliza para copiar datos de una tabla existente en una nueva tabla. Se utiliza comúnmente en SQL para crear una tabla nueva basada en los resultados de una consulta SELECT y luego insertar esos resultados en la nueva tabla.

```
SELECT CustomerName, ContactName
INTO Customers_backup
FROM Customers
WHERE Country = 'Germany';
```

Nota: Esta consulta crea una copia de seguridad de la tabla Customers en la base de datos Backup.mdb con el nombre CustomersBackup2017.

```
SELECT *
INTO CustomersBackup2017 IN 'Backup.mdb'
FROM Customers;
```

Nota: Esta consulta también tiene como objetivo crear una copia de seguridad que incluya los nombres de los clientes y los IDs de los pedidos en una tabla llamada CustomersOrderBackup2017. Los datos se obtienen de las tablas Customers y Orders, y se unen utilizando el ID del cliente como clave de unión.

```
SELECT Customers.CustomerName, Orders.OrderID
INTO CustomersOrderBackup2017
FROM Customers
LEFT JOIN Orders ON Customers.CustomerID = Orders.CustomerID;
```

Nota: También es posible crear una tabla vacía utilizando el esquema de otra tabla. Esto se logra mediante una consulta que no arroje ningún resultado. Para ello, basta con incluir una cláusula que garantice que la consulta no devuelva datos.

```
SELECT *
INTO newtable
FROM oldtable
WHERE 1 = 0;
```

INSERT INTO SELECT: Permite insertar filas en una tabla utilizando los resultados de una consulta SELECT. En lugar de especificar valores directamente, esta sentencia toma los resultados de una consulta SELECT y los inserta en la tabla especificada.

```
INSERT INTO Customers (CustomerName, City, Country)
SELECT SupplierName, City, Country
FROM Suppliers;
```

Nota: Esta consulta SQL insertará filas en la tabla "Customers" utilizando los valores de las columnas "SupplierName", "City" y "Country" de la tabla "Suppliers".

Nota: Mientras los tipos de datos sean compatibles entre las tablas de origen y destino, la inserción de datos debería realizarse sin problemas.

CASE: Se utiliza para realizar evaluaciones condicionales. Funciona de manera similar a un "switch" o una estructura condicional "if-else" en otros lenguajes de programación.

```
SELECT OrderID, Quantity,
CASE
WHEN Quantity > 30 THEN 'The quantity is greater than 30'
WHEN Quantity = 30 THEN 'The quantity is 30'
ELSE 'The quantity is under 30'
END AS QuantityText
FROM OrderDetails;
```

Number of Records: 2155

OrderID	Quantity	QuantityText
10248	12	The quantity is under 30
10248	10	The quantity is under 30
10248	5	The quantity is under 30

Nota: La consulta utiliza la expresión CASE para generar un texto descriptivo llamado QuantityText basado en el valor de la columna Quantity en la tabla OrderDetails. Dependiendo de si la cantidad es mayor que 30, igual a 30 o menor que 30, se devuelve un mensaje específico indicando el estado de la cantidad en relación con el valor 30.

```
SELECT CustomerName, City, Country
FROM Customers
ORDER BY (CASE
WHEN City IS NULL THEN Country
ELSE City
END);
```

Number of Records: 91

CustomerName	City	Country
Drachenblut Delikatessend	Aachen	Germany
Rattlesnake Canyon Grocery	Albuquerque	USA
Old World Delicatessen	Anchorage	USA

Nota: Esta consulta selecciona los nombres de los clientes, ciudades y países de la tabla "Customers" y los ordena según una expresión CASE. Si la ciudad es NULL para un cliente, se ordena por país. De lo contrario, se ordena por ciudad. Esto significa que los clientes se ordenarán primero por ciudad si tienen una ciudad especificada, y si no, se ordenarán por país.

Nota: Si ninguna de las condiciones especificadas se cumple y no se proporciona una cláusula ELSE, la expresión CASE devolverá NULL por defecto.

Las funciones **IFNULL**, **ISNULL**, **COALESCE** y **NVL** son utilizadas para manejar valores nulos de manera diferente. Aunque todas ellas tienen un propósito similar, hay diferencias en su implementación y disponibilidad según el DBMS que se esté utilizando.

Nota: IFNULL → (MySQL y SQLite), ISNULL → (SQL Server), NVL → (Oracle) y COALESCE → (MySQL, SQL Server, PostgreSQL, Oracle)

```
SELECT ProductName, function(UnitPrice * (function(UnitsInStock, 0) +
function(UnitsOnOrder, 0)), 0) AS TotalValue
FROM Products;
```

Nota: En cada caso, la función se utiliza para manejar valores nulos y devolver un valor predeterminado de 0 si alguna de las columnas (UnitPrice, UnitsInStock o UnitsOnOrder) es nula. Esto garantiza que el cálculo del valor total del inventario se realice correctamente sin errores debido a valores nulos.

¿Qué es un procedimiento almacenado?

Un **procedimiento almacenado** es un conjunto de instrucciones SQL guardadas en el servidor de la base de datos para realizar tareas específicas, como consultas, manipulación de datos o lógica empresarial.

Estos procedimientos se crean y almacenan en el servidor de la base de datos y se pueden llamar desde aplicaciones o consultas SQL. Ayudan a encapsular la lógica de la aplicación en el

servidor, mejorando la eficiencia y seguridad. También suelen tener parámetros y pueden incluir sentencias de control de flujo para manejar tareas complejas.

```
CREATE PROCEDURE SelectAllCustomers
AS
SELECT *
FROM Customers
GO;
```

Para ejecutar el procedimiento almacenado mencionado anteriormente, se utilizará:

```
EXEC SelectAllCustomers;
```

Nota: El procedimiento almacenado "SelectAllCustomers" selecciona todos los clientes de una ciudad específica (en este caso, "London") cuando se ejecuta con el parámetro @City establecido en ese valor.

```
CREATE PROCEDURE SelectAllCustomers @City nvarchar(30)
AS
SELECT *
FROM Customers
WHERE City = @City
GO;
```

Para ejecutar el procedimiento almacenado mencionado anteriormente, se utilizará:

```
EXEC SelectAllCustomers @City = 'London';
```

Nota: La lógica dentro del procedimiento es seleccionar todos los clientes de la tabla "Customers" que coincidan tanto con la ciudad especificada (@City) como con el código postal especificado (@PostalCode).

```
CREATE PROCEDURE SelectAllCustomers @City nvarchar(30),
@PostalCode nvarchar(10)
AS
SELECT *
FROM Customers
WHERE City = @City AND PostalCode = @PostalCode
GO;
```

Para ejecutar el procedimiento almacenado mencionado anteriormente, se utilizará:

```
EXEC SelectAllCustomers @City = 'London', @PostalCode = 'WA1
1DP';
```

¿Qué son los comentarios?

Los **comentarios** se utilizan para documentar el código y hacerlo más comprensible para otros desarrolladores o para tu propio uso futuro.

Nota: Hay dos formas comunes de agregar comentarios en SQL

1-. Comentarios de una línea: Se utilizan para añadir explicaciones breves o aclaraciones en una sola línea de código.

```
SELECT *
FROM Customers; -- Aqui iria un comentario
```

2-. Comentarios de varias líneas: Se utilizan para agregar explicaciones más detalladas o comentarios extensos que abarcan varias líneas de código. Estos comentarios se delimitan con /* al inicio y */ al final.

```
/* Aqui va
otro
comentario */
SELECT *
FROM Customers;
```

Nota: Para ignorar solo una parte de una instrucción, use también el comentario /* */.

```
SELECT CustomerName, /*City,*/ Country
FROM Customers;
```

¿Qué son los operadores?

Los **operadores** son símbolos o palabras clave que se utilizan para realizar operaciones en los datos almacenados en una base de datos. Estas operaciones pueden ser de comparación, lógicas, aritméticas o de concatenación, entre otras.

Puede parecer inútil usar alias en tablas, pero cuando se usa más de una tabla en las consultas, puede hacer que las instrucciones SQL sean más cortas.

Capítulo 3: Sentencias para un DBMS

CREATE DATABASE: Se utiliza para crear una nueva base de datos en un sistema de gestión de bases de datos (DBMS).

```
CREATE DATABASE testDB;
```

Nota: Antes de crear una base de datos en un DBMS, es importante asegurarse de tener los privilegios adecuados, como los privilegios de administrador o privilegios específicos de creación de base de datos.

SHOW DATABASES: Es utilizada para mostrar una lista de todas las bases de datos disponibles en el servidor.

Nota: Esto incluye tanto las bases de datos que has creado como las preexistentes en el servidor de la base de datos.

DROP DATABASE: Se utiliza para eliminar una base de datos y todos sus objetos asociados.

```
DROP DATABASE testDB;
```

Nota: Es importante tener cuidado al utilizar esta sentencia, ya que eliminará permanentemente la base de datos. Se recomienda hacer una copia de seguridad de la base de datos antes de ejecutar esta sentencia, especialmente si contiene datos importantes. Además, es necesario tener los permisos adecuados para ejecutar esta operación, ya que puede tener un impacto significativo en el sistema.

BACKUP DATABASE: No es una sentencia estándar en SQL. Sin embargo, en algunos DBMS como SQL Server de Microsoft, se utiliza para realizar copias de seguridad de bases de datos completas.

```
BACKUP DATABASE testDB
TO DISK = 'D:\backups\testDB.bak';
```

Nota: En la ruta 'D:\backups\testDB.bak', la letra "D" indica que el archivo de copia de seguridad se guardará en la unidad de disco D:. La carpeta "backups" es un directorio en esa unidad, y "testDB.bak" es el nombre del archivo de copia de seguridad que se creará en esa ubicación.

Nota: En MySQL, tienes la opción de emplear utilidades externas o comandos de línea de comandos para crear copias de seguridad de tus bases de datos. Por otro lado, en PostgreSQL, dispones de la utilidad pg_dump para llevar a cabo esta tarea

```
mysqldump -u username -p testDB > D:\backups\testDB.sql
```

```
pg_dump -u username -p testDB > D:\backups\testDB.sql
```

Nota: WITH DIFFERENTIAL realiza una copia de seguridad diferencial de una base de datos. Una copia de seguridad diferencial incluye solo los datos que han cambiado desde la última copia de seguridad completa.

```
BACKUP DATABASE testDB
TO DISK = 'D:\backups\testDB.bak'
WITH DIFFERENTIAL;
```

Nota: Cuando necesitas restaurar la base de datos, primero restaurarías la copia de seguridad completa más reciente. Luego, aplicarías la copia de seguridad diferencial más reciente sobre esa copia de seguridad completa. El proceso de restauración automáticamente combina los datos de la copia de seguridad completa con los datos adicionales de la copia de seguridad diferencial, lo que te proporciona una versión actualizada y completa de la base de datos.

CREATE TABLE: Se utiliza para crear una nueva tabla en una base de datos. Permite definir la estructura de la tabla, incluyendo los nombres de las columnas, los tipos de datos de las columnas, las restricciones de integridad y otras propiedades.

```
CREATE TABLE Persons (
PersonID int,
LastName varchar(255),
FirstName varchar(255),
Address varchar(255),
City varchar(255));
```

PersonID	LastName	FirstName	Address	City

Nota: Al usar la sentencia CREATE TABLE junto con AS SELECT, puedes crear una nueva tabla que sea una copia de una tabla existente.

```
CREATE TABLE TestTable AS
SELECT customername, contactname
FROM customers;
```

Nota: Esta consulta crea una nueva tabla llamada " CustomersBackup" que tiene la misma estructura que la tabla existente " Customers", pero sin ningún dato, ya que la condición "1=0" asegura que no se seleccionen filas de la tabla existente.

```
CREATE TABLE CustomersBackup AS
SELECT *
FROM Customers
WHERE 1=0;
```

Nota: Algo similar, pero utilizando la instrucción LIKE sería

```
CREATE TABLE CustomersBackup
LIKE Customers;
```

DROP TABLE: Se utiliza para eliminar una tabla existente de una base de datos.

```
DROP TABLE Shippers;
```

TRUNCATE TABLE: Se utiliza para eliminar todos los registros de una tabla, pero sin eliminar la definición de la tabla en sí misma.

```
TRUNCATE TABLE Shippers;
```

ALTER TABLE: Se utiliza para realizar cambios en una tabla existente. Estos cambios pueden incluir agregar, modificar o eliminar columnas, cambiar el nombre de la tabla o sus columnas, agregar o eliminar restricciones, y más.

Nota: Agregar una columna en una tabla

```
ALTER TABLE Customers
ADD Email varchar(255);
```

Nota: Eliminar una columna en una tabla.

```
ALTER TABLE Customers
DROP COLUMN Email;
```

Nota: Algunos sistemas de gestión de bases de datos (DBMS) pueden tener restricciones para eliminar columnas de tablas debido a factores como la integridad referencial, dependencias de aplicaciones existentes o complejidades técnicas. En ciertas circunstancias, la eliminación puede ser posible si no hay dependencias o si se cumplen condiciones específicas. Sin embargo, en otros casos, podría requerir procedimientos más complejos, como la creación de una nueva tabla sin la columna no deseada y transferir los datos a ella.

Nota: Modificar una columna (Esta instrucción te permite cambiar el tipo de datos o la restricción de nulabilidad de una columna existente en una tabla).

1-. Cambiar el nombre de la columna

```
ALTER TABLE Customers
RENAME COLUMN Email TO Gmail;
```

2-. Cambiar el tipo de dato de una columna

```
ALTER TABLE Products
RENAME COLUMN Price decimal(10,2);
```

Nota: En SQL Server / MS Access (ALTER), en MySQL (MODIFY) y en Oracle (MODIFY sin COLUMN).

3-. Cambiar el nombre de una tabla

```
ALTER TABLE Old_table
RENAME TO New_table;
```

¿Qué son las restricciones?

Las **restricciones** son reglas que se aplican a las columnas de una tabla para garantizar la integridad de los datos y mantener la coherencia en la base de datos, es decir, se utilizan para imponer ciertas reglas o condiciones sobre los datos que se pueden insertar, actualizar o eliminar en una tabla.

Nota: Algunos ejemplos comunes de restricciones son

1-. NOT NULL: Se utiliza para especificar que un campo de una tabla no puede contener valores nulos. Esto significa que el campo debe tener un valor en cada fila de la tabla y no puede permanecer vacío.

```
CREATE TABLE Persons (
ID int NOT NULL,
LastName varchar(255) NOT NULL,
FirstName varchar(255) NOT NULL,
Age int);
```

2-. UNIQUE: Se utiliza para garantizar que los valores en una columna o combinación de columnas sean únicos en una tabla. Esto significa que no puede haber duplicados en los valores de la columna o combinación de columnas especificadas.

```
CREATE TABLE Persons (
ID int NOT NULL UNIQUE,
LastName varchar(255) NOT NULL,
FirstName varchar(255) NOT NULL,
Age int);
```

Nota: En MySQL tendríamos que especificar la restricción en otra fila UNIQUE(ID).

Nota: Otra forma de especificar una restriccion UNIQUE es mediante la siguiente instrucción (en donde UC_Person es el nombre de la restricción).

```
CONSTRAINT UC_Person UNIQUE(ID, LastName);
```

3-. PRIMARY KEY: Se utiliza para identificar de manera única cada fila en una tabla. En resumen, su función principal es garantizar la unicidad y la integridad de los datos en una tabla, ya que no puede haber dos filas con el mismo valor en la columna definida como clave primaria.

```
CREATE TABLE Persons (
ID int NOT NULL PRIMARY KEY,
LastName varchar(255) NOT NULL,
FirstName varchar(255) NOT NULL,
Age int);
```

Nota: En MySQL tendríamos que especificar la restricción en otra fila PRIMARY KEY(ID).

Nota: Otra forma de especificar una PRIMARY KEY es mediante la siguiente instrucción

```
CONSTRAINT Pk_Person PRIMARY KEY(ID, LastName);
```

Nota: Una tabla en una base de datos debe tener una y solo una llave primaria (conjunto de una o más columnas cuyos valores identifican de manera única cada fila en la tabla).

Nota: Supongamos que tenemos una tabla llamada "Estudiantes" que contiene información sobre estudiantes.

ID_Estudiante	Nombre	Apellido	Edad
1	Juan	Pérez	20
2	María	Gómez	22
3	Luis	Martínez	21

En este caso, ID_Estudiante es la llave primaria. Esto significa que:

1-. Unicidad: Cada ID_Estudiante es único. No puede haber dos estudiantes con el mismo ID_Estudiante.

2-. No nulo: Cada estudiante debe tener un valor de ID_Estudiante, no puede ser nulo.

Nota: Supongamos que tenemos una tabla llamada "Inscripciones" que almacena la información sobre los cursos en los que los estudiantes están inscritos.

ID_Estudiante	ID_Curso	Fecha_Inscripcion
1	101	2024-01-15
2	102	2024-01-16
1	103	2024-01-17
3	101	2024-01-18

En este caso, la llave primaria está compuesta por las columnas ID_Estudiante e ID_Curso. Esto significa que la combinación de estas 2 columnas debe ser única para cada registro ya que un estudiante no puede estar inscrito en el mismo curso más de una vez.

Aquí está la definición SQL para crear esta tabla con la llave primaria compuesta:

```
CREATE TABLE Inscripciones (
ID_Estudiante INT,
ID_Curso INT,
Fecha_Inscripcion DATE,
PRIMARY KEY (ID_Estudiante, ID_Curso);
```

Esto garantiza que no haya duplicados en las inscripciones para un mismo estudiante y curso.

4-. FOREIGN KEY: Se utiliza para establecer una relación entre dos tablas. La clave foránea en una tabla apunta a la clave primaria en otra tabla, asegurando que los valores en la columna de la clave foránea correspondan a valores válidos en la columna de la clave primaria de la tabla relacionada. Esto garantiza la integridad referencial en la base de datos.

```
CREATE TABLE Orders (
OrderID int NOT NULL,
OrderNumber int NOT NULL,
PersonID int,
PRIMARY KEY (OrderID),
FOREIGN KEY (PersonID) REFERENCES Persons(PersonID)
);
```

Nota: A qui también podemos darle un nombre a la restricción de la FOREIGN KEY.

Nota: Otra forma de poner una FOREIGN KEY es mediante la siguiente instrucción

```
CONSTRAINT Fk_PersonOrder FOREIGN KEY(PersonID)
REFERENCES Persons(PersonID;
```

5-. CHECK: Se utiliza para asegurar que los valores en una columna o un conjunto de columnas cumplan con una condición específica.

```
CREATE TABLE Persons (
ID int NOT NULL,
LastName varchar(255) NOT NULL,
FirstName varchar(255),
```

```
Age int,
CHECK (Age >= 18)
);
```

Nota: Una forma de especificar 2 o más columnas en una restricción CHECK es mediante la siguiente forma

```
CONSTRAINT CHK_Person CHECK(Age >= 18 AND City = 'Tokio')
```

6-. DEFAULT: Se utiliza para asignar un valor predeterminado a una columna cuando no se especifica ningún valor durante la inserción de un registro. Esto es útil para garantizar que una columna tenga siempre un valor válido, incluso si no se proporciona explícitamente uno durante la inserción de datos.

```
CREATE TABLE Persons (
ID int NOT NULL,
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Age int,
City varchar(255) DEFAULT ('Tokio')
);
```

Nota: Las restricciones de integridad se pueden agregar o modificar después de la creación de la tabla utilizando las instrucciones ALTER TABLE y DROP (la sintaxis dependerá del DBMS que se esté usando).

¿Qué son los índices?

En SQL, un **índice** es una estructura de datos especial que mejora la velocidad de recuperación de datos en una tabla de una base de datos. Es similar a un índice en un libro, que permite encontrar rápidamente una información específica sin tener que leer todo el contenido.

Propósito de un Índice

El propósito principal de un índice es mejorar el rendimiento de las consultas. Sin índices, una búsqueda de datos en una tabla requiere que el sistema de base de datos escanee todas las filas de la tabla (una operación conocida como **escaneo de tabla completa**). Con un índice, la base de datos puede localizar rápidamente las filas que cumplen con los criterios de la consulta.

Funcionamiento de un Índice

Un índice se crea en una o más columnas de una tabla. La base de datos mantiene una estructura de datos ordenada (generalmente un árbol B o un árbol B+), lo que permite búsquedas rápidas, inserciones, actualizaciones y eliminaciones eficientes.

Desventadas de los índices

1-. Espacio Adicional: Los índices requieren espacio de almacenamiento adicional en la base de datos.

2-. Coste de Mantenimiento: Cada vez que se insertan, actualizan o eliminan datos, los índices deben ser actualizados, lo que puede añadir un coste de rendimiento.

3-. Complejidad Adicional: La creación y el mantenimiento de índices puede añadir complejidad al diseño de la base de datos.

Consideraciones

1-. Frecuencia de Consultas: Crea índices en columnas que se utilizan frecuentemente en cláusulas WHERE, JOIN, ORDER BY, y GROUP BY.

2-. Tipo de Datos: Considera los tipos de datos y el tamaño de las columnas. Índices en columnas con tipos de datos más pequeños pueden ser más eficientes.

3-. Cardinalidad: Crea índices en columnas con alta cardinalidad (muchos valores únicos) ya que proporcionan más beneficios que en columnas con baja cardinalidad.

4-. Costo de Mantenimiento: Ten en cuenta que cada índice adicional implica un costo de mantenimiento. Evalúa si el beneficio en la velocidad de consulta justifica este costo.

La actualización de los índices en una base de datos depende de las operaciones que se realizan en la tabla indexada. Aquí hay algunas situaciones comunes en las que los índices pueden necesitar ser actualizados:

1-. Inserciones de Datos: Cuando se inserta una nueva fila en una tabla indexada, los índices relevantes deben actualizarse para incluir la nueva fila. Esto implica agregar una nueva entrada al índice que corresponde a la nueva fila.

2-. Actualizaciones de Datos: Cuando se actualiza una fila existente en una tabla indexada, los índices relevantes deben actualizarse para reflejar los cambios en los valores indexados. Esto puede implicar eliminar la entrada anterior del índice y agregar una nueva entrada con los valores actualizados.

3-. Eliminaciones de Datos: Cuando se elimina una fila de una tabla indexada, las entradas correspondientes en los índices relevantes deben eliminarse para mantener la integridad del índice.

4-. Operaciones de Mantenimiento: Algunos sistemas de gestión de bases de datos pueden realizar automáticamente operaciones de mantenimiento en los índices para optimizar su rendimiento. Estas operaciones pueden incluir reorganizaciones de índices para mejorar la eficiencia de las búsquedas y reducir el espacio utilizado por los índices.

En general, los índices se actualizan automáticamente en tiempo real a medida que se realizan operaciones en la tabla indexada. Sin embargo, en algunos casos, es posible que necesites programar operaciones de mantenimiento periódicas para optimizar el rendimiento de los índices.

Es importante tener en cuenta que la actualización de los índices puede tener un costo en términos de rendimiento y recursos del sistema, especialmente en bases de datos con un alto volumen de operaciones. Por lo tanto, es importante diseñar cuidadosamente los índices y

monitorear su rendimiento para asegurarte de que estén proporcionando el máximo beneficio con el menor impacto en el rendimiento del sistema.

Tipos de Índices

1-. Índice Simple: Un índice en una sola columna.

```
CREATE INDEX idx_lastname
ON Persons(LastName);
```

Nota: Este índice acelera las consultas que buscan datos en la columna LastName de la tabla Persons

2-. Índice Compuesto: Un índice en dos o más columnas.

```
CREATE INDEX idx_lastname
ON Persons(LastName, FirstName);
```

Nota: Este índice es útil para consultas que buscan datos en las columnas LastName y FirstName conjuntamente.

3-. Índice Único (UNIQUE INDEX): Un índice que asegura que todos los valores de la columna o combinación de columnas sean únicos.

```
CREATE UNIQUE INDEX idx_unique_order_number
ON Orders(OrderNumber);
```

Nota: Este índice asegura que cada valor en la columna OrderNumber sea único en la tabla Orders.

4-. Índice de Texto Completo (FULLTEXT INDEX): Utilizado para búsquedas de texto completo en columnas de texto (disponible en algunos DBMS como MySQL).

5-. Índice Clustered: Reorganiza la tabla físicamente para que coincida con el orden del índice (normalmente sólo uno permitido por tabla).

6-. Índice No Clustered: No afecta el orden físico de la tabla y puede haber varios por tabla.

Nota: Para eliminar un índice, utilizaremos la instrucción **DROP INDEX**

```
DROP INDEX nombre_índice
ON nombre_tabla;
```

Nota: En SQL Server (DROP INDEX nombre tabla.nombre_índice), en MySQL (ALTER TABLE nombre_tabla DROP INDEX nombre_índice) y en Oracle (DROP INDEX nombre_índice).

AUTO INCREMENT: Es una característica utilizada en SQL para generar automáticamente valores únicos para una columna, generalmente para una columna que es clave primaria. Esta característica es particularmente útil cuando necesitas crear una columna que contenga valores únicos y secuenciales, como un identificador único para cada fila de una tabla.

Cuando se utiliza AUTO_INCREMENT, el valor de la columna se incrementa automáticamente por uno cada vez que se inserta una nueva fila en la tabla. El primer valor insertado suele ser 1, y cada fila posterior recibe un valor que es uno más grande que el valor de la fila anterior.

```
CREATE TABLE Persons (
PersonID int AUTO_INCREMENT PRIMARY KEY,
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Age int
);
```

Nota: No necesitas especificar explícitamente un valor para la columna ID al realizar una inserción; el DBMS se encargará de asignar automáticamente el siguiente valor disponible.

Nota: Para especificar que la columna "PersonID" debe comenzar en el valor 10, utilice la propiedad `AUTO_INCREMENT = 10;` después de la definición de la columna. Para que los valores se incrementen de 5 en 5, configure el incremento (ya sea a nivel global o de sesión) con la instrucción `SET SESSION auto_increment_increment = 5;`

En MySQL, los ajustes de AUTO_INCREMENT pueden aplicarse a nivel global o a nivel de sesión. Aquí te explico lo que significa cada uno:

1-. **Nivel Global:** Cuando cambias un ajuste a nivel global, este ajuste afecta a todas las conexiones y sesiones en la base de datos. Es decir, cualquier operación que utilice AUTO_INCREMENT se verá afectada por este ajuste hasta que se cambie nuevamente o se reinicie el servidor.

```
SET GLOBAL auto_increment_increment = 5;
```

Nota: Todas las tablas que utilicen AUTO_INCREMENT en toda la base de datos ahora incrementarán sus valores en 5.

2-. Nivel de Sesión: Cuando cambias un ajuste a nivel de sesión, este ajuste afecta solo a la sesión actual de la conexión. Una sesión en MySQL es una única conexión desde un cliente a la base de datos. Una vez que se cierra la sesión (es decir, se cierra la conexión), el ajuste se pierde.

```
SET SESSION auto_increment_increment = 5;
```

Nota: Solo las tablas que utilicen AUTO_INCREMENT en la sesión actual incrementarán sus valores en 5.

Nota: En SQL Server, el comportamiento de AUTO_INCREMENT se logra utilizando la propiedad **IDENTITY**. Para especificar un incremento diferente al valor por defecto (que es 1), puedes definir tanto el valor inicial como el incremento en la columna IDENTITY.

```
CREATE TABLE Persons (
PersonID int AUTO_IDENTITY(1,1) PRIMARY KEY,
```

```
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Age int
);
```

Nota: Para especificar que la columna "PersonID" debe comenzar en el valor 10 y aumentar en 5, utilice la propiedad IDENTITY con los parámetros (10, 5).

Consideraciones importantes:

1-. Sólo una columna por tabla: Normalmente, solo se puede tener una columna AUTO_INCREMENT por tabla.

2-. Inicio y Salto de la Secuencia: En algunos sistemas de bases de datos, como MySQL, puedes especificar el valor inicial de la secuencia y el incremento (o salto) entre los valores generados.

Fechas en SQL:

Las fechas en SQL son datos que representan momentos específicos en el tiempo. Los sistemas de gestión de bases de datos suelen tener tipos de datos específicos para trabajar con fechas y horas.

Nota: Algunos de los tipos de datos comunes para manejar fechas son

1-. DATE: Se utiliza para almacenar fechas sin incluir la hora del día. `'2022-06-14'`

2-. TIME: Se utiliza para almacenar la hora del día sin incluir la fecha. `'14:30:00'`

3-. DATETIME o **TIMESTAMP**: Se utiliza para almacenar tanto la fecha como la hora del día. `'2022-06-14 14:30:00'`

4-. YEAR: Se utiliza para extraer el componente del año de una fecha. `'2022'`

¿Qué son las vistas?

En SQL, una **vista** es una consulta predefinida que se almacena en la base de datos y se trata como una tabla virtual. Una vista es como una tabla virtual porque no almacena datos físicos por sí misma; en cambio, es una consulta que se ejecuta cada vez que se accede a ella. Las vistas permiten a los usuarios o aplicaciones consultar y manipular datos de una manera estructurada y conveniente, sin necesidad de repetir la misma consulta compleja una y otra vez.

Características de las Vistas:

1-. Seguridad: Las vistas pueden usarse para limitar el acceso a ciertos datos. Puedes restringir las columnas disponibles o aplicar filtros en la consulta de la vista para mostrar solo los datos relevantes para ciertos usuarios.

2-. Modularidad y Simplificación: Las vistas pueden ayudar a modularizar y simplificar las consultas complejas al dividirlas en partes más pequeñas y manejables.

```sql
CREATE VIEW Brazil_Customers AS
SELECT CustomerName, ContactName
FROM Customers
WHERE Country = 'Brazil';
```

Podemos consultar la vista de la siguiente manera:

```sql
SELECT *
FROM Brazil_Customers
```

Para actualizar una vista, simplemente debes volver a definirla con la nueva consulta o modificar su definición utilizando la instrucción CREATE OR REPLACE VIEW.

```sql
CREATE OR REPLACE VIEW Brazil_Customers AS
SELECT CustomerName, ContactName, City
FROM Customers
WHERE Country = 'Brazil';
```

Para eliminar una vista, puedes utilizar la instrucción DROP VIEW.

```sql
DROP VIEW Brazil_Customers;
```

Nota: Actualizar hace hincapié a volver a crear la vista o a modificar la vista existente.

¿Qué es una inyección SQL?

La **inyección** SQL es un tipo de ataque informático que se produce cuando los datos de entrada de un usuario no son correctamente validados o escapados antes de ser utilizados en una consulta SQL. Este tipo de vulnerabilidad permite a un atacante ejecutar comandos SQL no deseados en una base de datos, lo que puede llevar a la revelación de información confidencial, modificación o eliminación de datos, y en algunos casos, control total sobre el sistema de gestión de bases de datos (DBMS) subyacente.

Por lo general, estos ataques se llevan a cabo mediante formularios web, parámetros de URL u otras formas de entrada de datos donde un usuario puede introducir información. Si estos datos no se validan o escapan adecuadamente antes de ser utilizados en una consulta SQL, un atacante puede insertar instrucciones SQL maliciosas que se ejecutan cuando la consulta es procesada por el DBMS.

Inyección SQL basado en "boolean-based": Este tipo de ataque se basa en la explotación de las reglas de evaluación de expresiones lógicas. Al inyectar una expresión que siempre evalúa como verdadera, como '' = '', los hackers pueden manipular la lógica de la consulta para obtener acceso no autorizado a la base de datos.

Por ejemplo, considera un formulario de inicio de sesión que utiliza una consulta SQL para verificar las credenciales del usuario:

El fragmento de código perteneciente a esos dos íconos fue creado con el propósito de autenticar a un usuario, comparando su nombre de usuario y contraseña con los valores almacenados en una tabla de usuarios en una base de datos.

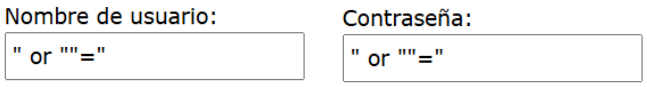
```
uName = getRequestString("username");
uPass = getRequestString("userpassword");

sql = 'SELECT * FROM Users WHERE Name ="' + uName + '" AND Pass ="' + uPass + '"'
```

Por ende, la consulta resultante de ese código es la siguiente:

```
SELECT * FROM Users WHERE Name="John Doe" AND Pass="myPass"
```

Si un atacante introduce una entrada como ' ' OR '1'='1' en el formulario de inicio de sesión, el código en el servidor creará una instrucción SQL válida.




```
SELECT * FROM Users WHERE Name = '' OR '1' = '1'
AND Pass = '' OR '1'='1';
```

Esto devolvería todas las filas de la tabla Usuarios, ya que '1'='1' es una expresión lógica que siempre es verdadera. Con acceso a estas filas, el atacante podría eludir la autenticación y acceder a la cuenta de cualquier usuario.

Inyección SQL basada en sentencias SQL por lotes: Es un tipo de ataque informático en el que un atacante aprovecha una vulnerabilidad en una aplicación web para ejecutar múltiples sentencias SQL en una sola solicitud.

Imagina que tienes un sitio web donde los usuarios pueden buscar información en una base de datos. Para hacer esto, el sitio web ejecuta consultas en la base de datos.

Ahora, un atacante malintencionado podría intentar engañar al sitio web para que ejecute más de una consulta SQL al mismo tiempo. Esto se hace aprovechando el hecho de que la mayoría de las bases de datos permiten ejecutar varias consultas en una sola solicitud.

Por ejemplo, supongamos que el sitio web tiene una consulta SQL que busca información sobre un usuario específico.

```
txtUserId = getRequestString("UserId");
txtSQL = "SELECT * FROM Users WHERE UserId = " + txtUserId;
```

Por ende, la consulta resultante de ese código es la siguiente:

```
SELECT * FROM Users WHERE ID = '$id'
```

El atacante podría intentar ingresar un valor especial en el campo de ID, como 105; DROP TABLE Suppliers.

Esto haría que la consulta SQL resultante se vea así:

```
SELECT * FROM Users WHERE ID = '105'; DROP TABLE Suppliers;
```

Aquí, la base de datos ejecutaría ambas consultas al mismo tiempo. Primero, devolvería la información sobre el usuario con ID 105, y luego eliminaría toda la tabla de proveedores de la base de datos.

Uso de parámetros SQL para la protección: Cuando construyes una consulta SQL dinámicamente en tu código, normalmente concatenas valores de variables directamente en la cadena de consulta.

```
txtSQL = "SELECT * FROM Users WHERE UserId = " + txtUserId;
```

El problema con esta forma de construir consultas es que, si el valor de txtUserId es proporcionado por un usuario malicioso, podría contener código SQL malicioso que alteraría la lógica de la consulta y podría ser utilizado para realizar ataques de inyección SQL.

Para evitar esto, puedes utilizar parámetros SQL. Los parámetros son valores que se agregan a la consulta SQL en tiempo de ejecución, de forma controlada. En lugar de concatenar los valores directamente en la consulta, utilizas marcadores de posición en la consulta y luego proporcionas los valores de los parámetros por separado.

Por ejemplo, en ASP.NET Razor, podrías tener una consulta como esta:

```
txtSQL = "SELECT * FROM Users WHERE UserId = @0";
```

Aquí, @0 es un marcador de posición para el primer parámetro. Luego, proporcionas el valor de txtUserId como un parámetro separado cuando ejecutas la consulta:

```
db.Execute(txtSQL, txtUserId);
```

El motor de SQL verifica cada parámetro para asegurarse de que sea válido para su columna y se trata de forma literal, no como parte del SQL que se va a ejecutar. Esto ayuda a prevenir la inyección SQL porque los valores de los parámetros no pueden alterar la estructura de la consulta.

En conclusión, para prevenir la inyección SQL, es esencial validar y escapar correctamente los datos de entrada del usuario antes de utilizarlos en consultas SQL. Esto puede hacerse utilizando parámetros de consulta parametrizados, funciones de escape proporcionadas por el DBMS, o mediante el uso de marcos y bibliotecas seguras que manejen la entrada de datos de manera segura. Además, es importante practicar la mínima exposición de información en mensajes de error y evitar revelar detalles sobre la estructura interna de la base de datos.