Lenguaje SQL

> ⚠ EN DESARROLLO ⚠

Introducción

SQL, que significa ***"Structured Query Language"*** (Lenguaje de Consulta Estructurado), es un lenguaje de programación utilizado para gestionar y manipular bases de datos relacionales. Fue desarrollado en la década de 1970 en IBM basándose en el trabajo de Edgard Codd y se ha convertido en el estándar de facto para interactuar con sistemas de gestión de bases de datos relacionales (SGBDR).

La primera versión de SQL normalizada por ANSI data de 1986. La norma SQL2 o SQL92 es la más importante y la mayoría de los SGBDR existentes implementan esta versión.

Características clave de SQL:

- **Declarativo**: SQL es un lenguaje de programación declarativo, lo que significa que describe el resultado deseado sin especificar el método para alcanzarlo. Esto permite a los usuarios centrarse en lo que quieren obtener, en lugar de cómo lograrlo.
- **Gestión de datos relacional**: SQL se utiliza principalmente en entornos de bases de datos relacionales, donde la información se organiza en tablas con relaciones entre ellas. Esto facilita la gestión y recuperación eficiente de datos.
- **Manipulación de datos**: SQL permite realizar operaciones fundamentales sobre datos, como la inserción (INSERT), la actualización (UPDATE), la eliminación (DELETE), y la recuperación (SELECT) de información de la base de datos.
- **Creación y modificación de estructuras de datos**: SQL también se utiliza para definir y modificar la estructura de las bases de datos, mediante la creación de tablas (CREATE TABLE), la alteración de tablas (ALTER TABLE), y la eliminación de tablas (DROP TABLE).
- **Consulta de datos**: La sentencia SELECT es esencial en SQL y se utiliza para recuperar datos de una o varias tablas. Permite filtrar, ordenar y agrupar información según las necesidades del usuario.
- **Integridad de datos**: SQL garantiza la integridad de los datos mediante restricciones como claves primarias, claves foráneas y otros mecanismos que aseguran la consistencia y fiabilidad de la información almacenada.
- **Transacciones**: SQL ofrece soporte para transacciones, permitiendo agrupar varias operaciones en una unidad atómica. Esto asegura que todas las operaciones se realicen con éxito o ninguna de ellas.

SQL se ha convertido en una herramienta esencial para cualquier persona involucrada en el desarrollo de software, administración de bases de datos, análisis de datos y otras disciplinas relacionadas con el manejo de información estructurada. Con su amplia adopción, el conocimiento de SQL es valioso en diversos contextos profesionales.

El lenguaje SQL se divide en cuatro subconjuntos:

- **DDL** (*Data Definition Language*), que agrupa todos los comandos utilizados para crear, modificar o eliminar las estructuras de la base de datos (tablas, índices, vistas, etc.). Se trata principalmente de los comandos:
  - `CREATE`
  - `ALTER`
  - `DROP`

- **DML** (*Data Manipulation Language*), que agrupa los comandos utilizados para manipular los datos contenidos en la base de datos. Se trata principalmente de los comandos:
  - `SELECT`
  - `INSERT`
  - `DELETE`
  - `UPDATE`
- **DCL** (*Data Control Language*), que agrupa los comandos utilizados para administrar la seguridad de acceso a los datos. Se trata principalmente de los comandos:
  - `GRANT`
  - `REVOKE`
- **TCL** (*Transaction Control Language*), que agrupa los comandos utilizados para administrar la confirmación o no de actualizaciones realizadas sobre la base de datos. Se trata principalmente de los comandos:
  - `COMMIT`
  - `ROLLBACK`

Las palabras clave SQL **no distinguen entre mayúsculas y minúsculas**. Por ejemplo, `SELECT` es lo mismo que `select`.

Algunos sistemas de bases de datos **requieren un punto y coma** (`;`) al final de cada instrucción SQL. Es la forma estándar de separar cada sentencia SQL en sistemas de bases de datos que permiten ejecutar **más de una sentencia SQL** en la misma llamada al servidor.

> ⚠️ **Sección introductoria generada por ChatGPT**

Modelo relacional

El modelo relacional fue creado por **Edgar Codd**, un investigador que trabajaba en IBM a principios de los años 70. Su trabajo pionero, "*A Relational Model of Data for Large Shared Data Banks*", sentó las bases de este modelo.

El modelo relacional se basa en el concepto de **conjunto**. Esquemáticamente, el modelo relacional se puede representar a través de una **tabla**, que en términos relacionales es una representación de una relación. Cada fila en la tabla corresponde a una **tupla (o registro)**, y cada columna representa un **atributo**.

> En el modelo relacional, el término **relación** se utiliza para referirse a una tabla, mientras que el término **tupla** se utiliza para referise a una fila. De forma similar, el término **atributo** se refiere a una columna de la tabla.

Este modelo presenta los datos de forma lógica, independientemente del modelo físico. El proveedor de la base de datos decide el modo de almacenamiento físico de las tablas. Esta independencia entre lo lógico y lo físico es una de las mayores ventajas de las bases de datos relacionales.

Una vez definidas las tablas, es necesario un lenguaje para manipularlas, lo cual se logra a través del **álgebra relacional**. Este formalismo matemático permite realizar operaciones sobre conjuntos de datos, como selecciones, proyecciones y uniones.

SQL implementa el álgebra relacional, y los sistemas de gestión de bases de datos relacionales (SGBDR) son los encargados de implementar el modelo relacional en la práctica.

El objeto principal gestionado por el modelo relacional es la **relación**, que está asociada a los conceptos de **dominio** y de **atributo**.

A esta relación se le aplican **reglas**:

- **Coherencia**: los datos en cualquier columna deben pertenecer al tipo definido por el dominio de ese atributo. La coherencia se asegura cuando cada valor pertenece al dominio sobre el que está definido el atributo.
- **Unicidad de las filas**: no pueden existir dos tuplas (filas) idénticas dentro de una misma relación (tabla). Cada tupla debe ser única.
- **Aceptación de valores nulos**: el concepto de valor nulo es válido y representa datos desconocidos o inaplicables en una columna.
    - **Restricción de entidad**: cualquier valor que forme parte de una clave primaria debe ser diferente de NULL, ya que las claves primarias deben identificar de manera única cada fila.
- **Atomicidad de los valores**: Los datos almacenados en cada columna de una tabla deben ser atómicos (monovalorados). Es decir, cada valor no puede ser dividido en valores más pequeños. Esto garantiza que la relación esté normalizada.
- **Independencia del orden**
    - El orden de los atributos (columnas) en una relación no tiene importancia.
    - El orden de las tuplas (filas) tampoco importa; las tuplas no están ordenadas.
- **Nombre único**:
    - Cada columna de una relación debe tener un nombre único dentro de esa tabla.
    - Cada relación (tabla) debe tener un nombre específico y diferente al resto de las relaciones en la base de datos.
- **Identificador**: atributo o conjunto de atributos que permiten caracterizar de forma unívoca cada elemento de la relación.
    - **Clave primaria**: identificador mínimo de una relación, que asegura la unicidad de cada tupla. Por ejemplo, en una tabla `clientes`, el campo `id_cliente` puede ser la clave primaria.
    - **Clave secundaria**: otros atributos que también pueden identificar de manera única una tupla, pero no son el identificador principal. Un ejemplo podría ser el `nif` de un cliente en la misma tabla `clientes`.
- **Integridad referencial**: esta regla impone que un atributo o conjunto de atributos de una relación aparezca como clave primaria en otra relación. Esto garantiza que las relaciones entre tablas sean válidas y consistentes.
    - **Clave externa**: atributo o conjunto de atributos que referencia la clave primaria de otra tabla, asegurando la integridad referencial. Por ejemplo, en una tabla `pedidos`, el campo `id_cliente` puede ser una clave externa que referencia a la clave primaria `id_cliente` en la tabla `clientes`.

Dominio

El dominio es el **conjunto de posibles valores** que un atributo puede tomar, representado por un nombre. Por ejemplo, el atributo `edad` en una tabla de personas puede tener como dominio el conjunto de números enteros positivos (edad ≥ 0). Si otro atributo es `nombre`, el dominio puede ser el conjunto de todas las cadenas de texto válidas.

Si tenemos un atributo `estado civil`, su dominio podría ser {"soltero", "casado", "divorciado", "viudo"}.

Cardinalidad

El número de elementos en un dominio se denomina **cardinalidad**. En el ejemplo anterior, la cardinalidad del dominio `estado civil` es 4.

Relación

Una relación definida sobre los dominios D1, D2, …, Dn es un subconjunto del producto cartesiano de estos dominios, caracterizado por un nombre. En términos prácticos, una **relación se representa como una tabla** en una base de datos relacional.

Una tabla `personas` puede ser una relación sobre los dominios `nombre`, `edad` y `estado civil`, donde cada fila (tupla) representa una persona.

Atributo

Un **atributo es una columna de la relación**, caracterizada por un nombre. Cada atributo está asociado con un dominio que define los valores posibles para esa columna.

En la relación `personas`, los atributos podrían ser `nombre`, `edad` y `estado civil`.

Grado

En una relación, el **grado es el número de atributos** que tiene. Dicho de otro modo, el número de columnas de una tabla.

Si la tabla `personas` tiene los atributos `nombre`, `edad` y `estado civil`, entonces el grado de la relación es 3.

Álgebra relacional

El **álgebra relacional** es un conjunto de operaciones que permite extraer y manipular datos de bases de datos relacionales. Se basa en **la teoría de conjuntos** y permite combinar, filtrar y transformar tablas (relaciones) existentes para generar nuevas tablas de resultados. Estos resultados pueden ser usados de **manera inmediata** o como base para otras operaciones.

Los operadores fundamentales basados en la teoría de conjuntos:

- **Selección (σ)**: este operador se usa para filtrar filas de una tabla que cumplen una condición específica. El resultado es un subconjunto de la tabla original.
- **Proyección (π)**: este operador selecciona columnas específicas de una tabla, eliminando las columnas que no son necesarias en la nueva tabla de resultados.
- **Unión (∪)**: combina las filas de dos tablas que tienen la misma estructura (mismo número de columnas y dominios compatibles). La tabla de resultado contiene todas las filas de ambas tablas, eliminando duplicados.
- **Intersección (∩)**: devuelve las filas que son comunes a dos tablas que tienen la misma estructura.
- **Diferencia (−)**: devuelve las filas que están en la primera tabla, pero no en la segunda, para tablas con la misma estructura.
- **Producto cartesiano (×)**: combina todas las filas de dos tablas, generando todas las posibles combinaciones de sus tuplas.
- **Combinación (⋈)**: combina las filas de dos tablas basándose en una condición que involucra una o más columnas de cada tabla. Existen diferentes tipos de *join*:
  - **Theta Join (θ)**: combina tablas usando una condición cualquiera.
  - **Equijoin**: un tipo específico de *join* donde las columnas relacionadas deben tener valores iguales.

- **Natural Join**: Similar al *equijoin*, pero elimina las columnas repetidas en las tablas relacionadas.

Selección (σ)

La **selección o restricción (σ)** es una operación del álgebra relacional que permite **extraer filas (tuplas) de una relación (tabla)** que cumplen con una **condición específica**. La tabla resultante tendrá la misma estructura (número de columnas y dominios) que la relación original, pero solo contendrá las filas que satisfacen el criterio de selección.

La notación es `Rx = σ(condición)(R)`, donde `R` es la relación de la cual se realiza la selección y `Rx` es la nueva relación resultante.

Las condiciones se pueden concatenar utilizando los operadores lógicos `AND` y `OR`, además de utilizar operadores de comparación como `=, >, <, >=, <=,` entre otros.

> El término ***"selección"*** del álgebra relacional tiene un **significado diferente** del que se utiliza en SQL, que resulta de un hecho histórico desafortunado. En álgebra relacional, el término ***"selección"*** corresponde a lo que en SQL se corresponde con la cláusula ***where***.

Como ejemplo de una selección, si tenemos una tabla `CLIENTES` y deseamos seleccionar todos los clientes que pertenecen a la región "Centro":

- **Tabla** `CLIENTES`

ID	Nombre	Región
1	Juan Pérez	Centro
2	Ana Gómez	Oeste
3	Luis Rodríguez	Centro
4	Marta López	Este
5	Carlos Ruiz	Oeste
6	Laura Sánchez	Centro

Esta operación se expresa como `CLIENTES_CENTRO = σ(región = "Centro")(CLIENTES)`. Esto generará una nueva tabla `CLIENTES_CENTRO` que incluirá únicamente las filas de `CLIENTES` donde la columna región tiene el valor "Centro".

- **Resultado:** `CLIENTES_CENTRO = σ(región = "Centro")(CLIENTES)`

ID	Nombre	Región
1	Juan Pérez	Centro
3	Luis Rodríguez	Centro
6	Laura Sánchez	Centro

Otras condiciones podrían ser:

- `σ(región = 'Centro' AND nombre = 'Juan Pérez')(CLIENTES)`
- `σ(región = 'Centro' OR región = 'Oeste')(CLIENTES)`
- `σ(id > 2)(CLIENTES)`

- `σ(id <= 3)(CLIENTES)`

Traducción a SQL

La **selección o restricción (σ)** en álgebra relacional permite extraer únicamente las filas que cumplen una condición específica.

La operación `σ s(condición)` se traduce de la siguiente manera:

```
SELECT * FROM s WHERE condición;
```

Por ejemplo, la restricción que selecciona los pedidos con el número de pedido igual a 100, representada en álgebra relacional como `PED = σ PEDIDOS(NUMPED = 100)` se traduce a SQL de la siguiente manera:

```
SELECT * FROM PEDIDOS WHERE NUMPED = 100;
```

Proyección (π)

La **proyección (π)** es una operación del álgebra relacional que permite **extraer columnas específicas de una relación**, generando así una nueva relación que contiene únicamente los atributos seleccionados. Este operador es útil cuando se desea obtener una vista reducida de los datos, centrándose solo en los atributos relevantes.

La notación es `Rx = π(A1, A2, ..., An)(R)`, donde `A1, A2, ..., An` son los atributos a proyectar, `R` es la relación de la cual se realiza la proyección y `Rx` es la relación resultante.

La operación de proyección no solo reduce la cantidad de datos visualizados, sino que también elimina duplicados en los resultados, asegurando que cada combinación de valores de las columnas proyectadas sea única.

Por ejemplo, dada la tabla `CLIENTES`, si queremos extraer únicamente los nombres y las regiones de los clientes, aplicaríamos la proyección de la siguiente manera:

- **Tabla** `CLIENTES`

ID	Nombre	Región
1	Juan Pérez	Centro
2	Ana Gómez	Oeste
3	Luis Rodríguez	Centro
4	Marta López	Este
5	Carlos Ruiz	Oeste
6	Laura Sánchez	Centro

- **Resultado:** `CLIENTES_NOMBRES_REGIONES = π(Nombre, Región)(CLIENTES)`

Nombre	Región
Juan Pérez	Centro
Ana Gómez	Oeste

Nombre	Región
Luis Rodríguez	Centro
Marta López	Este
Carlos Ruiz	Oeste
Laura Sánchez	Centro

Traducción a SQL

La **proyección (π)** en álgebra relacional tiene por objetivo eliminar las columnas no necesarias. En SQL, esto se realiza enumerando únicamente aquellas columnas requeridas en la instrucción `SELECT`.

Por ejemplo, la proyección sobre una tabla de clientes: `π CLIENTES(NUM, NOMBRE, PROVINCIA)` se traduce a SQL de la siguiente manera:

```
SELECT NUM, NOMBRE, PROVINCIA FROM CLIENTES;
```

Las **proyecciones de agrupación** se pueden efectuar con la ayuda de dos posibles sintaxis:

- `SELECT DISTINCT {* | lista de columnas} FROM tabla;`
- `SELECT lista de columnas FROM tabla GROUP BY lista de columnas;`

La primera sintaxis (`DISTINCT`) permite mostrar solo una fila en caso de que la consulta devuelva varias filas idénticas, mostrando así solo valores únicos.

La segunda sintaxis (`GROUP BY`) se utiliza cuando se desea realizar una proyección de grupo, calculando valores agregados sobre las filas agrupadas. Agrupar primero las filas permite aplicar funciones agregadas, aunque generalmente implica un mayor uso de recursos del servidor.

Unión (∪)

La **unión (∪)** es una operación del álgebra relacional que permite **combinar dos relaciones** para obtener una nueva relación que contenga **todos los elementos distintos de ambas**, eliminando duplicados. Esta operación solo es válida para relaciones que tienen la misma estructura, es decir, deben tener el mismo número de columnas (mismo grado) y los mismos dominios para cada columna.

La notación es `Rx = R1 ∪ R2`, donde `Rx` es la nueva relación resultante.

La operación de unión es útil en la manipulación de bases de datos para consolidar información de diferentes fuentes o categorías. Por ejemplo, puede ser utilizada para combinar listas de clientes de diferentes regiones o periodos de tiempo en una sola lista.

Por ejemplo, dadas dos tablas diferentes `CLIENTES_OESTE` y `CLIENTES_CENTRO`, la unión puede representarse como `CLIENTES = CLIENTES_OESTE ∪ CLIENTES_CENTRO` y contendrá todas las filas de ambas tablas, eliminando los duplicados y proporcionando un conjunto de clientes que pertenecen a cualquiera de las dos regiones.

- **Tabla** `CLIENTES_OESTE`

ID	Nombre	Región
1	Juan	Oeste
2	María	Oeste
3	Pedro	Oeste

- **Tabla** `CLIENTES_CENTRO`

ID	Nombre	Región
3	Pedro	Centro
4	Laura	Centro
5	Elena	Centro

- **Resultado:** `CLIENTES = CLIENTES_OESTE ∪ CLIENTES_CENTRO`

ID	Nombre	Región
1	Juan	Oeste
2	María	Oeste
3	Pedro	Oeste
3	Pedro	Centro
4	Laura	Centro
5	Elena	Centro

Otra aplicación sería, por ejemplo, utilizar la selección sobre una misma tabla. Como ya se ha comentado, aplicar un operador de álgebra relacional genera una tabla resultante que puede ser utilizada. Por lo tanto, aplicar dos selecciones sobre una tabla genera dos tablas resultantes.

Dada la tabla `CLIENTES` queremos seleccionar los clientes de la región "Centro" y "Oeste":

- **Tabla** `CLIENTES`

ID	Nombre	Región
1	Juan Pérez	Centro
2	Ana Gómez	Oeste
3	Luis Rodríguez	Centro
4	Marta López	Este
5	Carlos Ruiz	Oeste
6	Laura Sánchez	Centro

- **Resultado:** `CLIENTES = σ(región = "Centro")(CLIENTES) ∪ σ(región = "Oeste")(CLIENTES)`

ID	Nombre	Región
1	Juan Pérez	Centro
2	Ana Gómez	Oeste
3	Luis Rodríguez	Centro
5	Carlos Ruiz	Oeste
6	Laura Sánchez	Centro

Traducción a SQL

Esta operación permite combinar los resultados de dos o más consultas, obteniendo un conjunto de filas que cumplen con el mismo formato (mismo nombre de columnas, mismo tipo y en el mismo orden).

La operación `S ∪ T` se expresa de la siguiente manera:

```
SELECT lista de columnas FROM S {UNION / UNION ALL} SELECT lista de columnas FROM T;
```

El operador `UNION` elimina las filas duplicadas del resultado, mientras que `UNION ALL` permite extraer todas las filas procedentes de las consultas, incluyendo las duplicadas.

Supongamos que tenemos dos tablas `CLIENTES_2022` y `CLIENTES_2023`, y queremos obtener la lista de todos los clientes de ambos años:

```
SELECT nombre FROM CLIENTES_2022
UNION
SELECT nombre FROM CLIENTES_2023;
```

Si usamos `UNION ALL`, obtendríamos todos los nombres, incluyendo los que están duplicados en ambas tablas:

```
SELECT nombre FROM CLIENTES_2022
UNION ALL
SELECT nombre FROM CLIENTES_2023;
```

Intersección (∩)

La **intersección (∩)** es una operación del álgebra relacional que permite obtener el conjunto de **elementos que son comunes a dos relaciones**. Esta operación solo es válida para relaciones que tienen la misma estructura, es decir, deben tener el mismo número de columnas (mismo grado) y los mismos dominios para cada columna.

La notación es `Rx = R1 ∩ R2`, donde `Rx` es la nueva relación resultante.

La operación de intersección es útil en la manipulación de bases de datos para encontrar elementos que comparten características específicas en dos conjuntos diferentes. Por ejemplo, puede ser utilizada para identificar clientes que han realizado compras en dos diferentes periodos o categorías.

Es importante notar que, aunque la intersección es una operación válida y útil en álgebra relacional, algunos textos mencionan que su función se puede expresar **usando el operador diferencia (-)**. Es decir, si queremos encontrar los elementos comunes

entre dos relaciones, podemos calcularlo utilizando la diferencia de cada relación con la unión de ambas, lo que puede hacer que la intersección sea conceptualmente redundante en algunos contextos.

La fórmula para expresar esto sería: `R1 ∩ R2 = R1 - (R1 - R2)`

Como ejemplo de una intersección (⋂), dadas las tablas `CLIENTES_OESTE` y `CLIENTES_CENTRO` la intersección puede representarse como `CLIENTES = CLIENTES_OESTE ∩ CLIENTES_CENTRO` y contendrá todas las filas de ambas tablas, eliminando los duplicados y proporcionando un conjunto de clientes que pertenecen a las dos regiones.

- **Tabla** `CLIENTES_OESTE`

ID	Nombre	Región
1	Juan	Gijón
2	María	Zamora
3	Pedro	Mérida

- **Tabla** `CLIENTES_CENTRO`

ID	Nombre	Región
3	Pedro	Mérida
4	Laura	Madrid
5	Elena	Toledo

- **Resultado:** `CLIENTES = CLIENTES_OESTE ∩ CLIENTES_CENTRO`

ID	Nombre	Región
3	Pedro	Mérida

Traducción a SQL

Esta operación permite obtener filas que están presentes en ambas consultas, siempre que estas tengan el mismo formato (mismo nombre de columnas, mismo tipo y en el mismo orden).

La operación `S ∩ T` se expresa de la siguiente manera:

```
SELECT lista de columnas FROM S INTERSECT SELECT lista de columnas FROM T;
```

Por ejemplo, si tenemos dos tablas `CLIENTES_2022` y `CLIENTES_2023`, y queremos obtener los clientes que han estado en ambas tablas, podríamos usar:

```
SELECT nombre FROM CLIENTES_2022 INTERSECT SELECT nombre FROM CLIENTES_2023;
```

Diferencia (−)

La **diferencia (−)** es una operación del álgebra relacional que permite obtener una nueva relación **al eliminar de una relación (R1) los elementos que también están presentes en otra relación (R2)**. Esta operación solo es válida para relaciones que tienen la misma estructura, es decir, deben tener el mismo número de columnas (mismo grado) y los mismos dominios para cada columna.

La notación es `Rx = R1 – R2`, donde `Rx` es la nueva relación resultante.

La operación de diferencia es útil para identificar elementos que son exclusivos de una relación y que no están presentes en otra. Por ejemplo, puede ser utilizada para encontrar clientes que no han realizado compras en un periodo determinado.

Por ejemplo, dada la tabla `CLIENTES`, queremos seleccionar aquellos clientes existentes en el año 2022 pero que ya no son clientes en 2023:

- **Tabla** `CLIENTES`

ID	Nombre	Región	Año
1	Juan Pérez	Centro	2022
2	Ana Gómez	Oeste	2022
3	Luis Rodríguez	Centro	2022
4	Marta López	Este	2022
3	Luis Rodríguez	Oeste	2023
6	Laura Sánchez	Centro	2023
7	Elena Torres	Oeste	2023

- **Resultado:** `CLIENTES = σ(año = 2022)(CLIENTES) - σ(año = 2023)(CLIENTES)`

ID	Nombre	Región	Año
1	Juan Pérez	Centro	2022
2	Ana Gómez	Oeste	2022
4	Marta López	Este	2022

En 2022, los clientes con ID 1, 2 y 4 están registrados, mientras que en 2023 no están presentes, lo que los convierte en clientes exclusivos de 2022.

Traducción a SQL

Esta operación permite obtener filas que están en una consulta pero no en otra, siempre que ambas consultas usen el mismo formato (mismo nombre de columnas, mismo tipo y en el mismo orden).

La operación `S – T` se expresa de la siguiente manera:

```
SELECT lista de columnas FROM S MINUS SELECT lista de columnas FROM T;
```

Por ejemplo, si tenemos dos tablas `CLIENTES_2022` y `CLIENTES_2023`, y queremos obtener los clientes que solo estuvieron en 2022, podríamos usar:

```
SELECT nombre FROM CLIENTES_2022 MINUS SELECT nombre FROM CLIENTES_2023;
```

Producto cartesiano (×)

El **producto cartesiano (×)** es una operación del álgebra relacional que permite **combinar todas las tuplas de una relación (R1) con todas las tuplas de otra relación (R2)**, generando una nueva relación con todas las combinaciones posibles de tuplas de ambas relaciones. El resultado es una relación que tiene el número total de columnas de R1 más las columnas de R2.

La notación es `Rx = R1 × R2`, donde `Rx` es la nueva relación resultante de combinar las dos relaciones.

El producto cartesiano es útil cuando queremos combinar todas las posibles combinaciones de dos relaciones, pero en la práctica, suele ir acompañado de una operación selección para reducir el conjunto de resultados. Es una de las operaciones fundamentales que puede llevar a un `join` si luego se aplica una condición de filtrado.

Por ejemplo, dada la tabla `CLIENTES` y la tabla `PEDIDOS`, podemos generar todas las combinaciones posibles de clientes con sus pedidos:

- **Tabla** `CLIENTES`

ID	Nombre	Región
1	Juan Pérez	Centro
2	Ana Gómez	Oeste
3	Luis Rodríguez	Este

- **Tabla** `PEDIDOS`

Pedido_ID	Producto
101	Laptop
102	Smartphone

- **Resultado:** `PEDIDOS_CLIENTES = CLIENTES × PEDIDOS`

ID	Nombre	Región	Pedido_ID	Producto
1	Juan Pérez	Centro	101	Laptop
1	Juan Pérez	Centro	102	Smartphone
2	Ana Gómez	Oeste	101	Laptop
2	Ana Gómez	Oeste	102	Smartphone
3	Luis Rodríguez	Este	101	Laptop
3	Luis Rodríguez	Este	102	Smartphone

Traducción a SQL

El producto cartesiano permite **asociar cada fila de una tabla con cada fila de otras tablas**, generando todas las combinaciones posibles de filas. Esto puede resultar en un número de filas igual al producto del número de filas de las tablas

involucradas.

La operación `R × T` se expresa en SQL de la siguiente manera:

```
SELECT lista de columnas FROM S, T;
```

Si las columnas tienen el mismo nombre en ambas tablas, el nombre de la columna va precedido por el nombre de la tabla. Por lo tanto, el nombre completo de la columna es `nombretabla.nombrecolumna`.

Si tenemos dos tablas, `AULAS` y `PROFESORES`, el producto cartesiano podría verse así:

```
SELECT AULAS.nombre_aula, PROFESORES.nombre_profesor FROM AULAS, PROFESORES;
```

Esto devolvería una lista que combina cada aula con cada profesor, generando así todas las posibles combinaciones.

Combinaciones

La **combinación (JOIN)** es una operación del álgebra relacional que permite **combinar filas de dos relaciones basándose en una condición**. Esta operación une las filas de las dos tablas donde los valores de las columnas especificadas cumplen una condición determinada.

Existen varios tipos de combinaciones, siendo las más comunes: **combinación natural**, **combinación interna (inner join)** y **combinación externa (outer join)**.

- **Combinación interna (`INNER JOIN`)**: Devuelve las filas donde existe una coincidencia en ambas tablas, es decir, solo las filas que cumplen la condición especificada.

La notación para una combinación interna es `R1 ⋈ condición R2`.

- **Combinación externa (`OUTER JOIN`)**: Devuelve todas las filas de una tabla, y cuando no hay coincidencia en la otra tabla, los valores de esta se rellenan con NULL. Hay tres tipos principales:
  - **LEFT JOIN**: Devuelve todas las filas de la primera tabla y las coincidencias de la segunda, si las hay.
  - **RIGHT JOIN**: Devuelve todas las filas de la segunda tabla y las coincidencias de la primera.
  - **FULL JOIN**: Devuelve todas las filas de ambas tablas, con NULL donde no haya coincidencias.
- **Combinación natural (`NATURAL JOIN`)**: Une dos tablas automáticamente basándose en las columnas con el mismo nombre y tipo de dato.

Traducción a SQL

La combinación es una restricción sobre el producto cartesiano que vincula cada fila de una tabla con filas de otra tabla, de acuerdo a una condición dada. Existen varios tipos de combinaciones que se utilizan para extraer datos relacionados de diferentes tablas:

- **Combinación interna (`INNER JOIN`)**: retorna solo las filas que tienen coincidencias en ambas tablas.

  La operación `S INNER JOIN T ON (condición)` se expresa en SQL de la siguiente manera:

```
SELECT lista de columnas FROM S INNER JOIN T ON condición;
```

- **Combinación externa (`OUTER JOIN`)**: devuelve todas las filas de una tabla y las filas coincidentes de otra tabla. Si no hay coincidencias, se llenan con valores nulos (NULL).
    - **Combinación externa izquierda (`LEFT OUTER JOIN`)**: retorna todas las filas de la tabla izquierda (S) y las filas coincidentes de la tabla derecha (T).

```
SELECT lista de columnas FROM S LEFT OUTER JOIN T ON condición;
```

- **Combinación externa derecha (`RIGHT OUTER JOIN`)**: retorna todas las filas de la tabla derecha (T) y las filas coincidentes de la tabla izquierda (S).

```
SELECT lista de columnas FROM S RIGHT OUTER JOIN T ON condición;
```

- **Combinación externa completa (`FULL OUTER JOIN`)**: retorna todas las filas de ambas tablas, con coincidencias donde existan, y valores nulos donde no haya coincidencias.

```
SELECT lista de columnas FROM S FULL OUTER JOIN T ON condición;
```

- **Combinación natural (`NATURAL JOIN`)**: se basa en columnas con el mismo nombre en ambas tablas y combina automáticamente las filas donde estas columnas coinciden.

  La operación `S NATURAL JOIN T` se expresa en SQL de la siguiente manera:

```
SELECT lista de columnas FROM S NATURAL JOIN T;
```

Campos calculados elementales

Proyección sobre una relación asociada a un **cálculo** que se realiza sobre cada línea para **crear uno o varios atributos nuevos**.

La notación es `Rx = π S (A1, ..., N1 = expresión calculada...)`, donde π (pi) indica la proyección y se aplica un cálculo a cada fila para generar nuevas columnas.

La expresión calculada puede ser:

- una operación aritmética
- una función matemática
- una función de cadena

Por ejemplo, podemos calcular el total de un pedido multiplicando el precio unitario por la cantidad pedida: `TOTAL_PEDIDO = π PEDIDO (PEDIDONUM, REF, LINEA = PRECIO * CANTIDAD)`

Traducción a SQL

Los campos calculados elementales permiten obtener columnas calculadas para cada fila.

En SQL, los campos calculados se generan utilizando la cláusula `SELECT` con operaciones. La traducción de la operación `π S (A1, ..., N1 = expresión calculada...)` se traduce como:

```
SELECT col, ..., expresión FROM S;
```

Por ejemplo, podemos calcular el valor de los productos de un almacén:

`ALMACEN = π ARTICULOS(REF, DESCRIPCION, COSTO = (PRECIO * VALOR))`

Esto se puede traducir a SQL como:

```
SELECT REF, DESCRIPCION, (PRECIO * VALOR) FROM ARTICULOS;
```

Para asignar un nombre más claro a la columna calculada `(PRECIO * VALOR)`, se utiliza el alias `AS`:

```
SELECT REF, DESCRIPCION, (PRECIO * VALOR) AS VALOR_TOTAL FROM ARTICULOS;
```

Campos de valores agregados

Proyección sobre una relación asociada con uno o varios valores agregados que se calculan sobre un atributo para todos los elementos de la relación o de la agrupación vinculada a la proyección, con el fin de **crear uno o varios atributos nuevos**.

La notación es `Rx = π S (A1, ..., N1 = función estadística(Ax), ...)`

Las funciones estadísticas son:

- **COUNT(*)**
- **COUNT(atributo)**
- **SUM(atributo)**
- **MAX(atributo)**
- **MIN(atributo)**

Por ejemplo, podemos calcular el número de clientes que hay en una tabla `NUMCLIENTES = π CLIENTES(N = COUNT(*))`. También podemos calcular el precio más alto, el más bajo y el precio medio por categoría de artículos `STATS = π ARTICULOS(CATEGORIA, CARO=MAX(PRECIO), BARATO=(PRECIO), MEDIO=(PRECIO))`.

Traducción a SQL

Las **proyecciones del cálculo de valores agregados** permiten realizar cálculos estadísticos sobre los grupos especificados mediante `GROUP BY`.

La operación `π S (col, ..., nvcol = cálculo estadístico)` se traduce a SQL de la siguiente manera:

```
SELECT lista de columnas, función_agregada FROM S GROUP BY lista de columnas;
```

La lista de columnas proyectadas debe ser **idéntica** a la lista de columnas agrupadas con `GROUP BY`.

Si en la consulta se utilizan expresiones calculadas en las columnas del `SELECT`, la agrupación debe realizarse sobre las mismas expresiones. Por ejemplo, si se calcula el valor total de ventas con `precio * cantidad`, la agrupación también debe incluir

`precio * cantidad` en lugar de solo precio o cantidad, para que los resultados reflejen correctamente los cálculos proyectados.

```
SELECT producto, precio * cantidad AS total_venta
FROM ventas
GROUP BY producto, precio * cantidad;
```

Referencias a las funciones de agregación en diferentes SGBDR:

- Aggregate Functions - MySQL
- Aggregate Functions - PostgreSQL
- Aggregate Functions - Oracle Database

SQL

Select

La sentencia `SELECT` se utiliza para seleccionar datos de una base de datos.

```
SELECT column1, column2, ... FROM table_name;
```

Para devolver todas las columnas sin especificar cada nombre de columna, se utiliza el asterisco (`*`) con el `SELECT` :

```
SELECT * FROM table_name;
```

La sentencia `SELECT DISTINCT` se utiliza para devolver solo valores distintos (diferentes). Dentro de una tabla, una columna a menudo contiene muchos valores duplicados y en ocasiones sólo se necesita listar los valores diferentes (distintos).

```
SELECT DISTINCT column1, column2, ... FROM table_name;
```

Al utilizar la palabra clave `DISTINCT` en una función `COUNT()` , podemos devolver el número de registros diferentes:

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

Esta sentencia no funciona en Microsoft Acces. Una solución alternativa sería:

```
SELECT COUNT(*) AS column_name FROM (SELECT DISTINCT column_name FROM table_name);
```

Where

La cláusula `WHERE` se utiliza para filtrar registros, extrayendo aquellos registros que cumplen una condición específica.

```
SELECT column1, column2, ... FROM table_name WHERE condition;
```

La cláusula `WHERE` no solo se utiliza en sentencias `SELECT` sino que también se emplea en sentencias `UPDATE`, `DELETE`, etc.

Text Fields vs. Numeric Fields

SQL requiere **comillas simples alrededor de los valores de texto** (la mayoría de los sistemas de bases de datos también permiten comillas dobles).

Sin embargo, **los campos numéricos no deben ir entre comillas**.

```
-- Usando comillas simples
SELECT * FROM clientes WHERE nombre = 'Carlos';

-- Usando comillas dobles (si el sistema lo permite)
SELECT * FROM clientes WHERE nombre = "Carlos";

-- Valor numérico sin comillas
SELECT * FROM productos WHERE precio = 100;

-- Incorrecto: el valor numérico entre comillas puede causar errores inesperados
SELECT * FROM productos WHERE precio = '100';
```

Operators

Los operadores se utilizan para filtrar la búsqueda.

- **Igual que (`=`)**

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

- **Mayor o menor que (`>`, `<`)**

```
SELECT * FROM productos WHERE precio > 50;
```

- **Mayor o igual que (`>=`), menor o igual que (`<=`)**

```
SELECT * FROM empleados WHERE antiguedad >= 5;
```

- **Distinto de (`<>` o `!=`)**

```
SELECT * FROM pedidos WHERE estado <> 'completado';
```

- **Operador `LIKE` (para búsquedas de patrones)**

```
SELECT * FROM clientes WHERE nombre LIKE 'An%';
```

- **Operador `BETWEEN` (para rangos)**

```
SELECT * FROM productos WHERE precio BETWEEN 100 AND 200;
```

- **Operador `IN` (para listas de valores)**

```
SELECT * FROM empleados WHERE departamento_id IN (1, 2, 3);
```

Order BY

La palabra clave `ORDER BY` se utiliza para ordenar el conjunto de resultados en orden ascendente o descendente.

```
SELECT column1, column2, ... FROM table_name ORDER BY column1, column2, ... ASC|DESC;
```

En una cláusula de ejemplo como `ORDER BY column1, column2 ASC;`, el orden de los resultados funcionaría de la siguiente manera:

1. Primero, se ordena por `column1`. Los registros se ordenan en orden ascendente (de menor a mayor) según los valores en `column1`.

2. Luego, se ordena por `column2` en caso de empate. Si hay registros que tienen el mismo valor en `column1`, entonces esos registros se ordenarán según los valores de `column2`, también en orden ascendente (de menor a mayor).

Otro cláusula de ejemplo como `ORDER BY column1 DESC, column2 ASC;` funcionaría de la misma manera, cambiando únicamente si es ascendente o descendente.

La palabra clave `ORDER BY` ordena los registros **en orden ascendente por defecto**. Para ordenar los registros en orden descendente, se debe usar la palabra clave `DESC`.

Para los valores de texto, la palabra clave `ORDER BY` ordenará **alfabéticamente**.

And

El operador `AND` se utiliza para filtrar registros en función de dos o más condiciones. Se mostrará el registro **si todas las condiciones son verdaderas**.

```
SELECT column1, column2, ... FROM table_name WHERE condition1 AND condition2 AND condition3 ...;
```

Los operadores `AND` y `OR` se pueden combinar en la misma consulta. En ese caso se puede usar paréntesis para agrupar distintas cláusulas.

Or

La cláusula `WHERE` puede contener uno o más operadores `OR`.

Este operador se utiliza para filtrar registros en función de dos o más condiciones. Se mostrará el registro **si alguna de las condiciones es verdadera**.

```
SELECT column1, column2, ... FROM table_name WHERE condition1 OR condition2 OR condition3 ...;
```

Los operadores `AND` y `OR` se pueden combinar en la misma consulta. En ese caso se puede usar paréntesis para agrupar distintas cláusulas.

Not

El operador `NOT` se utiliza en combinación con otros operadores para obtener el resultado opuesto, también llamado resultado negativo.

```
SELECT column1, column2, ... FROM table_name WHERE NOT condition;
```

- `NOT LIKE`

```
-- Select customers that does not start with the letter 'A':
SELECT * FROM Customers WHERE CustomerName NOT LIKE 'A%';
```

- `NOT BETWEEN`

```
-- Select customers with a customerID not between 10 and 60:
SELECT * FROM Customers WHERE CustomerID NOT BETWEEN 10 AND 60;
```

- `NOT IN`

```
-- Select customers that are not from Paris or London:
SELECT * FROM Customers WHERE City NOT IN ('Paris', 'London');
```

- `NOT Greater Than`

```
-- Select customers with a CustomerId not greater than 50:
SELECT * FROM Customers WHERE NOT CustomerID > 50;
```

- `NOT Less Than`

```
-- Select customers with a CustomerID not less than 50:
SELECT * FROM Customers WHERE NOT CustomerId < 50;
```

Insert Into

La instrucción `INSERT INTO` se utiliza para insertar nuevos registros en una tabla. Es posible escribir la instrucción `INSERT INTO` de varias maneras:

- De **forma declarativa**, especificando tanto los nombres de las columnas como los valores a insertar. Es la forma menos propensa a errores:

```
INSERT INTO table_name (column1, column2, column3, ...) VALUES (value1, value2, value3, ...);
```

- **De forma implícita**. ya que si se están agregando valores para **todas las columnas** de la tabla, no es necesario especificar los nombres de las columnas en la consulta SQL. Sin embargo, hay que asegurarse de que el orden de los valores sea el mismo que el de las columnas en la tabla. En este caso, la sintaxis de `INSERT INTO` sería la siguiente:

```
INSERT INTO table_name VALUES (value1, value2, value3, ...);
```

- **Mediante una subconsulta** `SELECT`, para insertar registros desde otra tabla o consulta. Esto permite copiar datos de una fuente existente o generar datos dinámicamente. En este caso, la sintaxis de `INSERT INTO` sería la siguiente:

```
NSERT INTO table_name (column1, column2, column3, ...)
SELECT value1, value2, value3, ... FROM other_table WHERE condition;
```

También es posible insertar **varias filas en una sola instrucción**. Para insertar múltiples filas de datos, se utiliza la misma instrucción `INSERT INTO`, pero con varios conjuntos de valores:

```
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1_1, value2_1, value3_1, ...),
       (value1_2, value2_2, value3_2, ...),
       (value1_3, value2_3, value3_3, ...);
```

Hay que asegurarse de separar cada conjunto de valores con una coma (`,`).

Null Values

Un campo con un valor `NULL` es un **campo sin valor**.

Si un campo en una tabla es opcional, es posible insertar un nuevo registro o actualizar un registro sin agregar un valor a este campo. En ese caso, el campo se guardará con un valor `NULL`.

> Un valor `NULL` es diferente de un valor cero o de un campo que contiene espacios. Un campo con un valor `NULL` es aquel que ha sido dejado en blanco durante la creación del registro.

No es posible probar los valores `NULL` con operadores de comparación, como `=`, `<` o `<>`. En su lugar, se debe usar los operadores `IS NULL` e `IS NOT NULL`.

```
-- IS NULL Syntax
SELECT column_names FROM table_name WHERE column_name IS NULL;

-- IS NOT NULL Syntax
SELECT column_names FROM table_name WHERE column_name IS NOT NULL;
```

Update

La instrucción `UPDATE` se utiliza para modificar los registros existentes en una tabla.

```
UPDATE table_name SET column1 = value1, column2 = value2, ... WHERE condition;
```

> Hay que tener **especial atención** al actualizar registros en una tabla. La cláusula `WHERE` especifica qué registro(s) deben ser actualizados. Si se omite la cláusula `WHERE`, ¡todos los registros en la tabla serán actualizados!

Delete

La instrucción `DELETE` se utiliza para eliminar registros existentes en una tabla.

```
DELETE FROM table_name WHERE condition;
```

> Hay que tener **especial atención** al eliminar registros en una tabla. La cláusula `WHERE` especifica qué registro(s) deben ser eliminados. Si se omite la cláusula `WHERE`, ¡todos los registros en la tabla serán eliminados!

Filter

Cuando trabajamos con bases de datos, a menudo necesitamos **limitar el número de registros** que una consulta `SELECT` devuelve. Esto es útil cuando solo queremos obtener una muestra de los datos o cuando estamos trabajando con tablas grandes y queremos optimizar el rendimiento.

En SQL, se pueden utilizar diferentes cláusulas para controlar la cantidad de registros devueltos, como `LIMIT` en MySQL o `SELECT TOP` en SQL Server. A continuación, veremos cómo se aplica en los principales sistemas de gestión de bases de datos:

- ***SQL Server / MS Access Syntax:***

```
SELECT TOP number|percent column_name(s) FROM table_name WHERE condition;

-- With ORDER BY
SELECT TOP number column_name(s) FROM table_name WHERE condition ORDER BY column_name ASC|DESC;
```

- ***MySQL | PostgreSQL Syntax:***

```
SELECT column_name(s) FROM table_name WHERE condition LIMIT number;

-- With ORDER BY
SELECT column_name(s) FROM table_name WHERE condition ORDER BY column_name(s) ASC|DESC LIMIT number;
```

- ***Oracle 12 Syntax:***

```
SELECT column_name(s) FROM table_name ORDER BY column_name(s) FETCH FIRST number ROWS ONLY;
```

- ***Older Oracle Syntax:***

```
SELECT column_name(s) FROM table_name WHERE ROWNUM <= number;
```

- ***Older Oracle Syntax (with ORDER BY):***

```
SELECT * FROM (SELECT column_name(s) FROM table_name ORDER BY column_name(s)) WHERE ROWNUM <= number;
```

Aggregate Functions

Una función de agregación es una función que realiza un **cálculo sobre un conjunto de valores y devuelve un único valor**.

Las funciones de agregación se utilizan a menudo junto con la cláusula `GROUP BY` en la instrucción `SELECT`. La cláusula `GROUP BY` divide el conjunto de resultados en grupos de valores y la función de agregación puede usarse para devolver un solo valor para cada grupo.

Las funciones de agregación ignoran los valores `NULL` (excepto `COUNT()`).

MIN() and MAX()

- `MIN()`: devuelve el valor más pequeño dentro de la columna seleccionada

```
SELECT MIN(column_name) FROM table_name WHERE condition;
```

- `MAX()`: devuelve el valor más grande dentro de la columna seleccionada

```
SELECT MAX(column_name) FROM table_name WHERE condition;
```

Cuando se utiliza `MIN()` o `MAX()`, la columna devuelta no tendrá un nombre descriptivo. Para darle un nombre descriptivo a la columna, se utiliza la palabra clave `AS`:

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

COUNT()

La función `COUNT()` devuelve el número de filas que coinciden con un criterio especificado.

```
SELECT COUNT(column_name) FROM table_name WHERE condition;
```

Se puede especificar un nombre de columna en lugar del símbolo asterisco (`*`).

- `COUNT(*)` cuenta todas las filas, incluyendo aquellas con valores `NULL` en cualquiera de sus columnas.
- `COUNT(column_name)` cuenta solo las filas donde *'column_name'* no es `NULL`.

```
-- Cuenta todas las filas en la tabla, incluyendo las que tienen valores NULL
SELECT COUNT(*) AS total_employees FROM employees;

-- Cuenta solo las filas donde 'salary' no es NULL
SELECT COUNT(salary) AS employees_with_salary FROM employees;

-- Cuenta solo las filas donde 'department' no es NULL
SELECT COUNT(department) AS employees_with_department FROM employees;

-- Cuenta las filas donde 'salary' es NULL (filtradas por la cláusula WHERE)
SELECT COUNT(*) AS employees_without_salary FROM employees WHERE salary IS NULL;
```

Se pueden ignorar duplicados utilizando la palabra clave `DISTINCT` en la función `COUNT()`.

Si se especifica `DISTINCT`, las filas con el mismo valor en la columna especificada se contarán como una sola.

Es importante destacar que, al usar `DISTINCT`, se debe especificar un nombre de columna, ya que no se puede utilizar `DISTINCT` con el asterisco (`*`).

```
SELECT COUNT(DISTINCT column_name) FROM table_name WHERE condition;
```

Para darle un nombre a la columna contada, se puede usar la palabra clave `AS` para asignar un alias:

```
-- Cuenta filas únicas en la columna 'department' y le da un nombre a la columna con 'AS'
SELECT COUNT(DISTINCT department) AS unique_departments FROM employees;
```

SUM()

La función `SUM()` devuelve la suma total de una columna numérica.

```
SELECT SUM(column_name) FROM table_name WHERE condition;
```

Para darle un nombre a la columna utilizada, se puede usar la palabra clave `AS` para asignar un alias:

```
SELECT SUM(Quantity) AS total FROM OrderDetails;
```

El parámetro dentro de la función `SUM()` también puede ser una expresión. Esto permite realizar cálculos sobre los valores antes de sumarlos, como podrían ser sumas, restas, multiplicaciones o divisiones.

```
-- Suma de los salarios después de aplicar un descuento
SELECT SUM(salary * 0.9) AS total_salary_after_discount FROM employees;
```

También se puede usar funciones matemáticas provistas por los distintos SGBD como `ROUND()` o `ABS()`:

```
SELECT SUM(ROUND(salary, 2)) AS total_rounded_salary FROM employees;
```

AVG()

La función `AVG()` devuelve el valor promedio de una columna numérica, dónde los valores `NULL` son ignorados.

```
SELECT AVG(column_name) FROM table_name WHERE condition;
```

Para darle un nombre a la columna utilizada, se puede usar la palabra clave `AS` para asignar un alias:

```
SELECT AVG(Price) AS AveragePrice FROM Products;
```

Para listar todos los registros con un precio superior al promedio, podemos utilizar la función `AVG()` en una subconsulta:

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

Like

El operador `LIKE` se utiliza en cláusulas `WHERE` para buscar un patrón específico en una columna.

```
SELECT column1, column2, ... FROM table_name WHERE columnN LIKE pattern;
```

Existen dos **comodines o *wildcards*** que suelen utilizarse junto con el operador `LIKE`:

- El signo de porcentaje (`'%'`) representa cero, uno o múltiples caracteres.

```
-- Encuentra todos los registros donde el nombre empieza con 'A'
SELECT * FROM table_name WHERE column_name LIKE 'A%';

-- Encuentra todos los registros donde el nombre termina con 'Z'
SELECT * FROM table_name WHERE column_name LIKE '%Z';

-- Encuentra todos los registros donde el nombre contiene 'abc'
SELECT * FROM table_name WHERE column_name LIKE '%abc%';

-- Encuentra todos los registros donde el nombre empieza con 'A' y termina con 'Z'
SELECT * FROM table_name WHERE column_name LIKE 'A%Z';
```

- El guion bajo (`'_'`) representa un solo carácter individual, lo que incluye tanto letras como números y símbolos.

```
-- Encuentra todos los registros donde el nombre tiene exactamente tres caracteres y empieza con 'A'
SELECT * FROM table_name WHERE column_name LIKE 'A__';

-- Encuentra todos los registros donde el nombre:
-- - Empieza con 'L'
-- - Tiene cualquier carácter en la segunda posición
-- - Continúa con 'nd'
-- - Tiene cualquier carácter en la quinta y sexta posición
SELECT * FROM table_name WHERE column_name LIKE 'L_nd__';
```

Cualquier comodín, como `'%'` y `'_'`, se puede combinar entre sí:

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

Si no se especifica ningún comodín, la frase debe coincidir exactamente para que se devuelva un resultado:

```
SELECT * FROM Customers WHERE Country LIKE 'Spain';
```

Wildcards

Un **carácter comodín o *wildcard*** se utiliza para sustituir uno o más caracteres en una cadena.

Los caracteres comodín se utilizan con el operador `LIKE`. El operador `LIKE` se usa en cláusulas `WHERE` para buscar un patrón específico en una columna.

- `%` - Representa cero o más carácteres
- `_` - Representa un carácter individual
- `[]` - Representa cualquier carácter individual dentro de los corchetes (*)
- `^` - Representa cualquier carácter que no esté dentro de los corchetes (*)

- `-` - Representa cualquier carácter individual dentro del rango especificado (*)
- `{}` - Representa cualquier carácter escapado (**)

(*) No soportado en PostgreSQL y MySQL

(**) *Soportado sólo en Oracle*

In

El operador `IN` permite especificar múltiples valores en una cláusula `WHERE`.

El operador `IN` es una **forma abreviada** de múltiples condiciones `OR`.

```
SELECT column_name(s) FROM table_name WHERE column_name IN (value1, value2, ...);
```

Al utilizar la palabra clave `NOT` delante del operador `IN`, se devuelven todos los registros que **NO** coinciden con ninguno de los valores en la lista.

```
SELECT column_name(s) FROM table_name WHERE column_name NOT IN (value1, value2, ...);
```

También se puede usar `IN` con una subconsulta en la cláusula `WHERE`.

Con una subconsulta, se pueden devolver todos los registros de la consulta principal que estén presentes en el resultado de la subconsulta utilizando el operador `IN`, o bien los registros que no estén en el resultado de la subconsulta si se utiliza el operador `NOT IN`:

```
-- IN
SELECT * FROM Customers WHERE CustomerID IN (SELECT CustomerID FROM Orders);

-- NOT IN
SELECT * FROM Customers WHERE CustomerID NOT IN (SELECT CustomerID FROM Orders);
```

Between

El operador `BETWEEN` selecciona valores dentro de un rango dado. Los valores pueden ser números, texto o fechas.

El operador `BETWEEN` es **inclusivo**: los valores inicial y final están incluidos.

```
SELECT column_name(s) FROM table_name WHERE column_name BETWEEN value1 AND value2;
```

Para mostrar los productos fuera del rango del ejemplo anterior, utiliza `NOT BETWEEN`:

```
SELECT column_name(s) FROM table_name WHERE column_name NOT BETWEEN value1 AND value2;
```

Algunos ejemplos con `BETWEEN`:

```
-- MySQL, PostgreSQL y SQL Server
SELECT * FROM Orders WHERE OrderDate BETWEEN '1996-07-01' AND '1996-07-31';
```

```sql
-- Microsoft Access
SELECT * FROM Orders WHERE OrderDate BETWEEN #07/01/1996# AND #07/31/1996#;

-- Selects all products with a ProductName alphabetically between Carnarvon and Mozzarella:
SELECT * FROM Products WHERE ProductName BETWEEN 'Carnarvon' AND 'Mozzarella' ORDER BY ProductName;

-- NOT BETWEEN
SELECT * FROM Products WHERE Price NOT BETWEEN 10 AND 20;

-- BETWEEN with IN
SELECT * FROM Products WHERE Price BETWEEN 10 AND 20 AND CategoryID IN (1,2,3);
```

Aliases

Los alias en SQL se utilizan para dar **un nombre temporal a una tabla o columna**, mejorando la legibilidad.

Un alias solo existe durante la ejecución de la consulta y se crea con la palabra clave `AS`. En la mayoría de los lenguajes de bases de datos, puedes omitir la palabra clave `AS` ya que es **opcional** y obtener el mismo resultado.

```sql
-- Alias used on COLUMN
SELECT column_name AS alias_name FROM table_name;

-- Alias used on TABLE
SELECT column_name(s) FROM table_name AS alias_name;
```

Se pueden utilizar múltiples alias en una misma consulta, como en el siguiente ejemplo:

```sql
SELECT CustomerID AS ID, CustomerName AS Customer FROM Customers;
```

Si un alias contiene **espacios en blanco**, es necesario envolverlo entre corchetes (`[]`) o comillas dobles (`""`) para que sea válido. Sin embargo, no todos los sistemas de gestión de bases de datos permiten ambos métodos:

```sql
-- SQL Server
SELECT ProductName AS [My Great Products] FROM Products;

-- PostgreSQL
SELECT ProductName AS "My Great Products" FROM Products;

-- MySQL
SELECT ProductName AS `My Great Products` FROM Products;
```

Se pueden concatencar columnas cuando se crea un alias con `AS`:

```sql
-- SQL Server
SELECT CustomerName, Address + ', ' + PostalCode + ' ' + City + ', ' + Country AS Address FROM Customers;

-- Oracle y PostgreSQL (||)
SELECT CustomerName, (Address || ', ' || PostalCode || ' ' || City || ', ' || Country) AS Address FROM Customers;

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

Las mismas reglas se aplican cuando se usan los alias para una tabla. Puede parecer innecesario usar alias en las tablas, pero cuando se utilizan más de una tabla en ciertas consultas, puede hacer que las sentencias SQL sean más cortas.

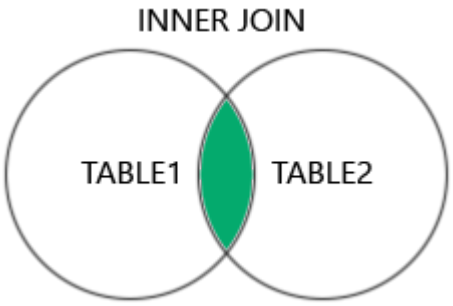
```sql
SELECT o.OrderID, o.OrderDate, c.CustomerName
FROM Customers AS c, Orders AS o
WHERE c.CustomerName='Around the Horn' AND c.CustomerID=o.CustomerID;
```

Joins

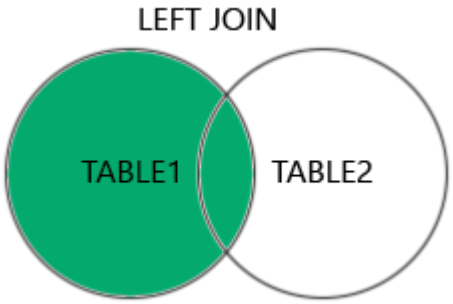
Una cláusula `JOIN` se utiliza para combinar filas de dos o más tablas, basándose en una columna relacionada entre ellas.

Aquí están los diferentes tipos de `JOIN` en SQL:

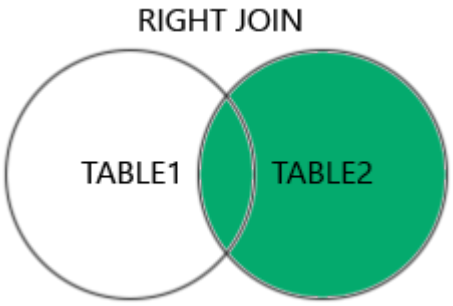
- `(INNER) JOIN`: Devuelve los registros que tienen valores coincidentes en ambas tablas.



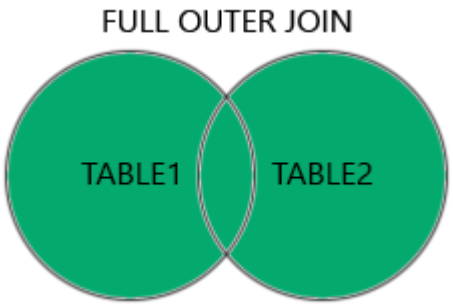
- `LEFT (OUTER) JOIN`: Devuelve todos los registros de la tabla izquierda y los registros coincidentes de la tabla derecha.



- `RIGHT (OUTER) JOIN`: Devuelve todos los registros de la tabla derecha y los registros coincidentes de la tabla izquierda.



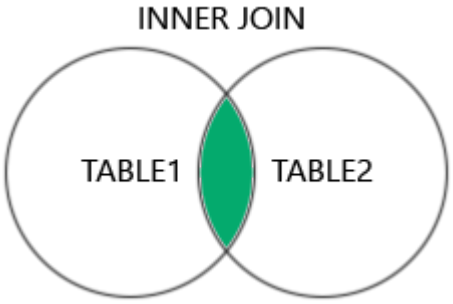
- `FULL (OUTER) JOIN`: Devuelve todos los registros cuando hay una coincidencia en la tabla izquierda o en la tabla derecha.



Inner Join

La palabra clave `INNER JOIN` selecciona los registros que tienen **valores coincidentes en ambas tablas**.

```
SELECT column_name(s)
FROM table1
INNER JOIN table2
ON table1.column_name = table2.column_name;
```



Dadas las tablas `Products` y `Categories`, la siguiente consulta utiliza un `INNER JOIN` para **combinar los datos de ambas tablas**. La consulta selecciona los productos que tienen categoría, es decir, aquellos que tienen una categoría asociada a la tabla `Categories`.

Si un producto no tiene un `CategoryID` o si el `CategoryID` no coincide con ningún valor en la tabla `Categories`, ese producto no será incluido en el resultado.

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

Es una buena práctica incluir el nombre de la tabla al especificar las columnas en la instrucción SQL, salvo cuando un campo tiene **el mismo nombre** en ambas tablas y además se incluye en la consulta. En ese caso, para evitar ambigüedades, se debe especificar de forma explícita el nombre de la tabla o el alias de la tabla para cada columna, utilizando la notación `Tabla.Columna`:

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

`JOIN` e `INNER JOIN` devolverán el mismo resultado ya que `INNER` es el **tipo por defecto** de `JOIN`.

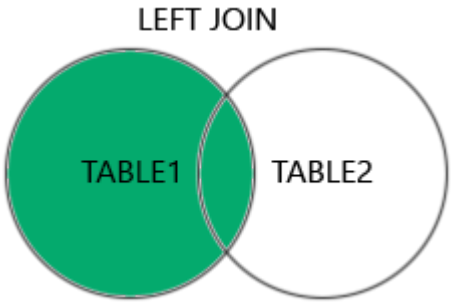
Es posible utilizar un `INNER JOIN` para trabajar con **tres o más tablas**. Esto se logra combinando cada tabla adicional con la anterior mediante las condiciones de unión necesarias.

```
-- Selecciona las órdenes junto con los nombres de los clientes y los nombres de los productos
SELECT Orders.OrderID, Customers.CustomerName, Products.ProductName
FROM Orders
INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID
INNER JOIN Products ON Orders.ProductID = Products.ProductID;
```

Left Join

La palabra clave `LEFT JOIN` devuelve todos los registros de la tabla izquierda y los registros coincidentes de la tabla derecha. Si no hay coincidencia, el resultado incluirá 0 registros del lado derecho.

```
SELECT column_name(s)
FROM table1
LEFT JOIN table2
ON table1.column_name = table2.column_name;
```



En algunas bases de datos, `LEFT JOIN` se denomina `LEFT OUTER JOIN`.

En los cuatro principales SGBD (*MySQL, PostgreSQL, SQL Server y Oracle*), se puede usar tanto `LEFT JOIN` como `LEFT OUTER JOIN`, ya que ambas formas producen exactamente el **mismo resultado**. La palabra `OUTER` es opcional, ya que no cambia el comportamiento del comando. Estos SGBD admiten ambas formas como parte del estándar SQL.

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

Dadas las tablas `Orders` y `Customers`, con la relación garantizada por una clave foránea (`Orders.CustomerID` referencia a `Customers.CustomerID`), la siguiente consulta selecciona todas las órdenes realizadas, junto con la información del cliente asociado. Como la clave foránea asegura que siempre hay un cliente asociado a cada pedido, todos los campos de la tabla `Customers` estarán presentes.

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

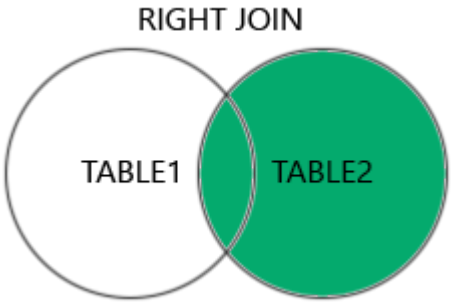
En la práctica, `LEFT JOIN` suele utilizarse con más frecuencia que `RIGHT JOIN`. Esto se debe a que la tabla que colocamos a la izquierda (tabla 1) suele ser la que centra el análisis o aquello que queremos averiguar.

Al priorizar esa tabla, `LEFT JOIN` resulta más intuitivo y directo para resolver la mayoría de las preguntas. Aunque `RIGHT JOIN` puede producir los mismos resultados invirtiendo las tablas, `LEFT JOIN` suele ser la elección más común en diseños y consultas reales.

Right Join

La palabra clave `RIGHT JOIN` devuelve todos los registros de la tabla derecha y los registros coincidentes de la tabla izquierda. Si no hay coincidencia, el resultado incluirá 0 registros del lado izquierdo.

```
SELECT column_name(s)
FROM table1
RIGHT JOIN table2
ON table1.column_name = table2.column_name;
```



Al igual que antes, en los cuatro principales SGBD (*MySQL, PostgreSQL, SQL Server y Oracle*), se puede usar tanto `RIGHT JOIN` como `RIGHT OUTER JOIN`, ya que ambas formas producen exactamente el **mismo resultado**. La palabra `OUTER` es opcional, ya que no cambia el comportamiento del comando. Estos SGBD admiten ambas formas como parte del estándar SQL.

Dadas las mismas tablas, la siguiente consulta selecciona todos los clientes, incluyendo aquellos que no hayan realizado ningún pedido. Para los clientes sin órdenes, los campos correspondientes a los pedidos aparecerán como `NULL`. Este caso es útil si necesitas ver todos los clientes, destacando quiénes no han realizado compras todavía.

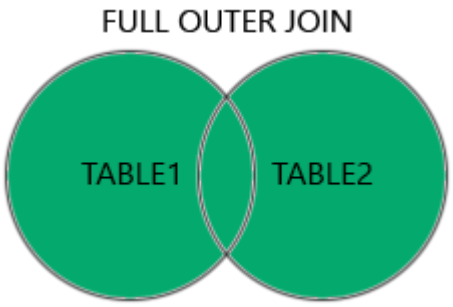
```
SELECT Customers.CustomerName, Orders.OrderID, Orders.OrderDate
FROM Orders
RIGHT JOIN Customers ON Orders.CustomerID = Customers.CustomerID;
```

Full Join

La palabra clave `FULL JOIN` devuelve todos los registros cuando hay una coincidencia en los registros de la tabla izquierda o de la tabla derecha.

En los cuatro principales SGBD (*MySQL, PostgreSQL, SQL Server y Oracle*), se puede usar tanto `FULL JOIN` como `FULL OUTER JOIN`, ya que ambas formas producen exactamente el **mismo resultado**. La palabra `OUTER` es opcional, ya que no cambia el comportamiento del comando. Estos SGBD admiten ambas formas como parte del estándar SQL.

```
SELECT column_name(s)
FROM table1
FULL JOIN table2
ON table1.column_name = table2.column_name
WHERE condition;
```



Un `FULL JOIN` es útil cuando se necesita analizar todos los datos posibles de ambas tablas, incluyendo los registros que no tienen coincidencias en la otra tabla.

Por ejemplo, dada una relación entre clientes y pedidos:

```
SELECT Customers.CustomerName, Orders.OrderID, Orders.OrderDate
FROM Customers
FULL JOIN Orders ON Customers.CustomerID = Orders.CustomerID;
```

- Los clientes sin pedidos tendrán sus columnas de pedidos (`OrderID` , `OrderDate`) en NULL.
- Los pedidos sin cliente asociado (si los hubiera) tendrán sus columnas de clientes (`CustomerName`) en NULL.

> Un `FULL JOIN` combina el resultado de un `LEFT JOIN` y un `RIGHT JOIN` , pero sin duplicar los registros que coinciden en ambas tablas.

Self Join

Un `SELF JOIN` es un tipo de unión regular en la que una tabla se une **consigo misma**.

Esto puede parecer inusual, pero es útil en situaciones donde los datos de una fila están relacionados con los datos de otra fila en la misma tabla.

Para realizar un `SELF JOIN` , se utilizan alias de tabla para diferenciar entre la "versión izquierda" y la "versión derecha" de la tabla. Sin los alias, la consulta generaría un error porque el SGBD no sabría distinguir entre las dos "copias" de la tabla. Los alias son, por tanto, **indispensables** en cualquier `SELF JOIN` .

```
SELECT column_name(s)
FROM table1 AS T1, table1 AS T2
WHERE condition;
```

Por ejemplo, considera una tabla de empleados donde cada empleado tiene un `EmployeeID` y un `ManagerID` que indica quién es su gerente. Si quieres obtener una lista de empleados junto con el nombre de su gerente, se puede usar un `SELF JOIN` :

```
SELECT A.EmployeeName AS Employee, B.EmployeeName AS Manager
FROM Employees A
LEFT JOIN Employees B ON A.ManagerID = B.EmployeeID;
```

Un `SELF JOIN` es útil en casos como:

- Jerarquías, como empleados y gerentes, o categorías principales y subcategorías.

- Relaciones reflexivas, como conexiones entre nodos en un grafo o una red.

> Aunque un `SELF JOIN` utiliza únicamente una tabla, técnicamente sigue siendo un `JOIN`, ya que el SGBD **combina datos de diferentes 'instancias'** de esa misma tabla.

Union

El operador `UNION` se utiliza para **combinar los resultados** de dos o más sentencias `SELECT`.

- Cada sentencia `SELECT` dentro de `UNION` debe tener el mismo número de columnas.
- Las columnas deben tener tipos de datos similares.
- Las columnas en cada sentencia `SELECT` también deben estar en el mismo orden.

```
SELECT column_name(s) FROM table1 WHERE condition
UNION
SELECT column_name(s) FROM table2 WHERE condition;
```

El operador `UNION` selecciona únicamente valores distintos de forma predeterminada. Para permitir valores duplicados, utiliza `UNION ALL`:

```
SELECT column_name(s) FROM table1 WHERE condition
UNION ALL
SELECT column_name(s) FROM table2 WHERE condition;
```

Los nombres de las columnas en el conjunto de resultados suelen ser iguales a los nombres de las columnas en la primera sentencia `SELECT`.

Group By

La sentencia `GROUP BY` agrupa filas que comparten los mismos valores en categorías, como "encontrar el número de clientes en cada país".

Se utiliza a menudo con funciones de agregación (`COUNT()`, `MAX()`, `MIN()`, `SUM()`, `AVG()`) para agrupar el conjunto de resultados por una o más columnas.

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
ORDER BY column_name(s);
```

Es decir, el concepto clave del `GROUP BY` es que permite agrupar los registros en conjuntos basados en el valor de una o más columnas, y una vez formados esos conjuntos, se pueden aplicar funciones de agregación para obtener información resumida sobre ellos, como contar cuántos registros hay (`COUNT()`), calcular promedios (`AVG()`), sumar valores (`SUM()`), o encontrar valores máximos y mínimos (`MAX()`, `MIN()`).

Por ejemplo, si agrupamos los registros por el país de un cliente, cada grupo contendrá todos los registros correspondientes a un país específico, y luego podríamos contar cuántos clientes hay en cada país o calcular el gasto promedio por país.

```sql
-- Agrupa los registros por país y calcula el número total de registros en cada grupo
SELECT pais, COUNT(*) AS total
FROM tabla
GROUP BY pais;
```

La sintaxis de `GROUP BY` efectivamente utiliza comas para separar las columnas, y cuando agrupas por dos o más columnas, se crean conjuntos que coinciden con todas las combinaciones posibles de valores únicos en esas columnas.

```sql
-- Agrupa los registros por país y ciudad y calcula el número total de registros en cada grupo
SELECT pais, ciudad, COUNT(*) AS total
FROM tabla
GROUP BY pais, ciudad;
```

Having

La cláusula `HAVING` se añadió a SQL porque la palabra clave `WHERE` no se puede usar con funciones de agregación.

```sql
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
HAVING condition
ORDER BY column_name(s);
```

La cláusula `HAVING` se utiliza para filtrar los resultados después de aplicar una función de agregación (como `COUNT()`, `SUM()`, `AVG()`, etc.). A diferencia de `WHERE`, que filtra las filas antes de aplicar las funciones de agregación y por tanto antes del `GROUP BY`, la cláusula `HAVING` permite aplicar condiciones sobre los resultados ya agrupados y agregados.

Dicho de otro modo, la cláusula `WHERE` se utiliza antes del `GROUP BY` para filtrar los registros y reducir el conjunto de datos con los que se va a trabajar. Por tanto, la función `COUNT()` o cualquier otra función de agregación solo se aplicará sobre los registros que pasen el filtro del `WHERE`.

Por ejemplo, si queremos contar cuántos registros hay en cada país y ciudad, y luego mostrar solo aquellos países con más de 5 registros, podemos usar `HAVING` para filtrar el resultado de la agregación:

```sql
-- Agrupa los registros por país y ciudad, calcula el número total de registros por cada grupo
-- y muestra solo aquellos países donde el número total de registros es mayor que 5
SELECT pais, ciudad, COUNT(*) AS total
FROM tabla
GROUP BY pais, ciudad
HAVING COUNT(*) > 5; -- Filtra los grupos con más de 5 registros
```

En este ejemplo, `HAVING COUNT(*) > 5` filtra los resultados de la agregación, mostrando solo aquellos países y ciudades que tienen más de 5 registros en total.

Para contar solo las filas donde tanto el país como la ciudad no son `NULL`, se puede usar una condición en la cláusula `WHERE` antes de hacer el `GROUP BY` para restringir los registros:

```sql
-- Cuenta solo las filas donde tanto pais como ciudad no sean NULL
SELECT pais, ciudad, COUNT(*) AS total
FROM tabla
```

```
WHERE pais IS NOT NULL AND ciudad IS NOT NULL
GROUP BY pais, ciudad;
```

Exists

El operador `EXISTS` se utiliza para comprobar la existencia de algún registro en una subconsulta. Retorna `TRUE` si la subconsulta devuelve uno o más registros.

```
SELECT column_name(s)
FROM table_name
WHERE EXISTS
(SELECT column_name FROM table_name WHERE condition);
```

El operador `EXISTS` y el operador `IN` son similares en algunos aspectos, pero tienen diferencias clave en su funcionamiento.

El operador `IN` se utiliza para verificar si **un valor dado coincide con cualquiera de los valores** de una lista o subconsulta. `IN` evalúa una lista de valores y puede devolver un conjunto de resultados basado en los valores exactos que se comparan.

```
SELECT * FROM Customers WHERE CustomerID IN (SELECT CustomerID FROM Orders);
```

En cambio, el operador `EXISTS` se utiliza para comprobar si **la subconsulta devuelve al menos un registro**. `EXISTS` no se preocupa por los valores específicos que devuelve la subconsulta, solo se interesa en si hay registros o no.

```
SELECT * FROM Customers c WHERE EXISTS (SELECT 1 FROM Orders o WHERE c.CustomerID = o.CustomerID);
```

El operador `IN` puede ser más lento si la subconsulta devuelve muchos resultados, ya que tiene que comprobar cada valor. Por su parte, el operador `EXISTS` generalmente es más eficiente, ya que solo necesita saber si existen resultados, sin preocuparse por el número de registros.

Any, All

Los operadores `ANY` y `ALL` permiten realizar una comparación entre un solo valor de columna y un conjunto de valores (un rango). Estos operadores son útiles cuando se necesita comparar un valor con los resultados de una subconsulta.

El operador `ANY` devuelve un valor booleano que será `TRUE` si **CUALQUIERA de los valores** de la subconsulta cumple con la condición.

Por tanto, `ANY` significa que la condición será verdadera si la operación es verdadera para **al menos uno** de los valores en el rango.

```
SELECT column_name(s)
FROM table_name
WHERE column_name operator ANY
  (SELECT column_name FROM table_name WHERE condition);
```

El operador debe ser un **operador de comparación estándar** (=, <>, !=, >, >=, < o <=).

```
-- Muestra los nombres de productos que tienen un ProductID en la lista de OrderDetails con cantidad 10
SELECT ProductName
SELECT ProductName
```

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

-- Muestra todas las órdenes con un monto mayor a alguna de las órdenes de un cliente específico
SELECT *
FROM Orders
WHERE OrderAmount > ANY (SELECT OrderAmount FROM Orders WHERE CustomerID = 'ALFKI');
```

El operador `ALL` también devuelve un valor booleano. En este caso, devuelve `TRUE` si **TODOS los valores** de la subconsulta cumplen con la condición.

Por tanto, `ALL` significa que la condición será verdadera solo si la operación es verdadera para **todos** los valores en el rango.

```
-- Syntax con SELECT
SELECT ALL column_name(s)
FROM table_name
WHERE condition;

-- Syntax con WHERE o HAVING
SELECT column_name(s)
FROM table_name
WHERE column_name operator ALL
  (SELECT column_name FROM table_name WHERE condition);
```

El operador debe ser un **operador de comparación estándar** (=, <>, !=, >, >=, < o <=).

```
-- Muestra los nombres de productos cuyo ProductID coincide con todos los ProductID de OrderDetails con cantidad 10
SELECT ProductName
FROM Products
WHERE ProductID = ALL (SELECT ProductID FROM OrderDetails WHERE Quantity = 10);
```

Select Into

La sentencia `SELECT INTO` copia datos de una tabla a una **nueva tabla**.

La nueva tabla se creará con los nombres de columna y tipos tal como están definidos en la tabla original. Se puede crear nuevos nombres de columna usando la cláusula `AS`.

```
-- Copy all columns into a new table
SELECT *
INTO newtable [IN externaldb]
FROM oldtable
WHERE condition;

-- Copy only some columns into a new table
SELECT column1, column2, column3, ...
INTO newtable [IN externaldb]
FROM oldtable
WHERE condition;
```

> En **MySQL**, el comando `SELECT INTO` no es compatible para crear nuevas tablas. En su lugar, se debe usar la sintaxis `CREATE TABLE ... AS SELECT ...`. **PostgreSQL**, **Microsoft SQL Server** y **SQLite** son compatibles con `SELECT INTO` para crear tablas nuevas. En **Oracle**, `SELECT INTO` se usa en el contexto de **PL/SQL** para almacenar resultados en variables, mientras que para crear tablas también se utiliza `CREATE TABLE ... AS SELECT ...`.

Aunque no se utiliza normalemente, se puede usar `SELECT INTO` para crear una nueva tabla vacía basada en la estructura de otra tabla. Solo se necesita añadir una cláusula `WHERE` que haga que la consulta no devuelva datos:

```sql
SELECT *
INTO nueva_tabla
FROM tabla_original
WHERE FALSE; -- No devuelve datos, pero crea la estructura de la tabla.
```

Insert Into Select

La declaración `INSERT INTO SELECT` se utiliza para **copiar datos de una tabla e insertarlos en otra tabla**.

Es importante destacar que esta declaración requiere que los **tipos de datos coincidan** en las tablas de origen y destino.

Los registros existentes en la tabla de destino no se ven afectados por la operación de inserción, ya que los datos se agregan como nuevos registros, no se reemplazan ni se actualizan los existentes.

```sql
-- Copy all columns from one table to another table
INSERT INTO tabla_destino
SELECT * FROM tabla_origen
WHERE condition;

-- Copy only some columns from one table into another table
INSERT INTO tabla_destino (column1, column2, column3, ...)
SELECT column1, column2, column3, ...
FROM tabla_origen
WHERE condition;
```

Case

La expresión `CASE` evalúa una serie de condiciones y devuelve un valor cuando se cumple la primera condición (similar a una sentencia `if-then-else`).

Una vez que una condición es verdadera, deja de evaluar y devuelve el resultado correspondiente. Si ninguna de las condiciones es verdadera, devuelve el valor de la cláusula `ELSE`.

Si no hay una parte `ELSE` y ninguna de las condiciones es verdadera, devuelve `NULL`.

```sql
SELECT
  CASE
    WHEN condition1 THEN result1
    WHEN condition2 THEN result2
    WHEN conditionN THEN resultN
    ELSE result
  END
FROM table_name;
```

En este ejemplo, la expresión `CASE` revisa el valor de *Quantity* y devuelve el estado del inventario según corresponda:

```sql
SELECT
  ProductName,
  CASE
    WHEN Quantity > 50 THEN 'In stock'
    WHEN Quantity <= 50 AND Quantity > 0 THEN 'Low stock'
    ELSE 'Out of stock'
  END AS StockStatus
FROM Products;
```

Null Functions

COALESCE(...)

La función `COALESCE(expression1, expression2, ..., expressionN)` devuelve el primer valor no `NULL` de una lista de expresiones que se evaluarán en orden:

```
-- Si 'Discount' es 'NULL', se evalúa 'Tax'. Si también es 'NULL', se devuelve 0
SELECT ProductName, COALESCE(Discount, Tax, 0) AS DiscountOrTax FROM Products;
```

La función `COALESCE(...)` es parte del **estándar SQL**, por lo que está disponible en la mayoría de los sistemas de gestión de bases de datos, incluidos **MySQL, Microsoft SQL Server, PostgreSQL y Oracle**. Por tanto, suele ser la mejor opción.

IFNULL(...)

La función `IFNULL(expression, alt_value)` de MySQL permite devolver un valor alternativo si una expresión es `NULL`:

```
-- Si 'Discount' es 'NULL', se devolverá 0
SELECT ProductName, IFNULL(Discount, 0) AS Discount FROM Products;
```

> `IFNULL(...)` es específica de **MySQL**, por lo que si se trabaja en otros SGBD, es recomendable considerar usar `COALESCE(...)` para mayor compatibilidad.

ISNULL(...)

La función `ISNULL(expression, alt_value)` de Microsoft SQL Server permite devolver un valor alternativo si una expresión es `NULL`:

```
-- Si 'Discount' es 'NULL', se devolverá 0
SELECT ProductName, ISNULL(Discount, 0) AS Discount FROM Products;
```

> `ISNULL(...)` es específica de **Microsoft SQL Server**, por lo que si se trabaja en otros SGBD, es recomendable considerar usar `COALESCE(...)` para mayor compatibilidad.

NVL(...)

La función `NVL(expression, alt_value)` de Oracle SQL permite devolver un valor alternativo si una expresión es `NULL`:

```
-- Si 'Discount' es 'NULL', se devolverá 0
SELECT ProductName, NVL(Discount, 0) AS Discount FROM Products;
```

> `NVL(...)` es específica de **Oracle SQL**, por lo que si se trabaja en otros SGBD, es recomendable considerar usar `COALESCE(...)` para mayor compatibilidad.

Stored Procedures

Un **procedimiento almacenado o *Stored Procedure*** es un código SQL preparado que se puede guardar para que se pueda reutilizar una y otra vez.

Cuando se tiene una consulta SQL que se escribe repetidamente, se guarda como un procedimiento almacenado y luego solo se tiene que invocar para ser ejecutado.

También se pueden pasar parámetros a un procedimiento almacenado, de modo que el procedimiento pueda actuar en función de los valores de los parámetros que se le pasen.

Microsoft SQL Server

En Microsoft SQL Server, la sintaxis para crear un procedimiento almacenado es la siguiente:

```
CREATE PROCEDURE procedure_name
AS
sql_statement
GO;
```

Para invocar un procedimiento almacenado, usamos el comando `EXEC` :

```
EXEC procedure_name;
```

MySQL

En MySQL, la sintaxis es diferente, y se usa `DELIMITER` para cambiar el delimitador cuando se crea un procedimiento:

```
DELIMITER //
CREATE PROCEDURE procedure_name()
BEGIN
    sql_statement;
END //
DELIMITER ;
```

Para invocar el procedimiento en MySQL, se usa `CALL` simplemente:

```
CALL procedure_name();
```

PostgreSQL

En PostgreSQL, los procedimientos almacenados se definen como funciones, utilizando el lenguaje ***PL/pgSQL (Procedural Language/PostgreSQL SQL)***:

```
CREATE OR REPLACE FUNCTION procedure_name()
RETURNS void AS $$
BEGIN
    sql_statement;
END;
$$ LANGUAGE plpgsql;
```

Para invocar el procedimiento se utiliza `SELECT` :

```
SELECT procedure_name();
```

Oracle

En Oracle se utiliza ***PL/SQL (Procedural Language/Structured Query Language)*** cuya sintaxis para la creación de un procedimiento es la siguiente:

```
CREATE PROCEDURE procedure_name
IS
BEGIN
    sql_statement;
END procedure_name;
```

Para invocar el procedimiento en Oracle se utiliza `EXEC`:

```
EXEC procedure_name;
```

Comments

Los comentarios de una sola línea comienzan con `--`.

Cualquier texto entre `--` y el final de la línea será ignorado (no se ejecutará).

```
-- Single line comment
SELECT * FROM Customers;

-- Single-line comment to ignore the end of a line
SELECT * FROM Customers -- WHERE City='Berlin';
```

Los comentarios de varias líneas comienzan con `/*` y terminan con `*/`.

Cualquier texto entre `/*` y `*/` será ignorado.

```
/*Select all the columns
of all the records
in the Customers table:*/
SELECT * FROM Customers;

-- To ignore just a part of a statement, also use the /* */ comment
SELECT CustomerName, /*City,*/ Country FROM Customers;
```

Data Types

- MySQL
- PostgreSQL
- Oracle Database

Functions

- MySQL
- PostgreSQL

- Oracle Database

SQL Database

Create DB

La sentencia `CREATE DATABASE` se utiliza para crear una nueva base de datos SQL.

```
CREATE DATABASE databasename;
```

Una vez creada, puedes verificarla en la lista de bases de datos con el siguiente comando SQL:

```
SHOW DATABASES;
```

Drop DB

La sentencia `DROP DATABASE` se utiliza para eliminar una base de datos SQL existente.

```
DROP DATABASE databasename;
```

Create Table

La instrucción `CREATE TABLE` se utiliza para crear una nueva tabla en una base de datos.

```
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    column3 datatype,
   ....
);
```

Los parámetros de columna especifican los nombres de las columnas de la tabla.

El parámetro de tipo de dato especifica el tipo de datos que la columna puede contener (por ejemplo, `varchar`, `integer`, `date`, etc.).

También se puede crear una copia de una tabla existente utilizando `CREATE TABLE`. La nueva tabla tendrá las mismas definiciones de columnas. Se pueden seleccionar todas las columnas o columnas específicas.

Si se crea una nueva tabla a partir de una tabla existente, la nueva tabla se llenará con los valores existentes de la tabla original.

```
CREATE TABLE new_table_name AS
    SELECT column1, column2,...
    FROM existing_table_name
    WHERE ....;
```

Drop Table

La instrucción `DROP TABLE` se utiliza para eliminar una tabla existente en una base de datos. Eliminar una tabla resultará en la pérdida completa de toda la información almacenada en ella.

```
DROP TABLE table_name;
```

La instrucción `TRUNCATE TABLE` se utiliza para eliminar los datos dentro de una tabla, pero no la tabla en sí.

```
TRUNCATE TABLE table_name;
```

Alter Table

La instrucción `ALTER TABLE` se utiliza para agregar, eliminar o modificar columnas en una tabla existente.

La instrucción `ALTER TABLE` también se usa para añadir y eliminar diversas restricciones en una tabla existente.

Add Column

Para agregar una columna en una tabla, se utiliza la siguiente sintaxis:

```
ALTER TABLE table_name
ADD column_name datatype;
```

> Cada SGBD tiene diferentes tipos de datos.

Drop Column

Para eliminar una columna en una tabla, se utiliza la siguiente sintaxis:

```
ALTER TABLE table_name
DROP COLUMN column_name;
```

En **Oracle**, existe la opción de marcar una columna como ***'unused'*** en lugar de eliminarla inmediatamente. Esto significa que la columna permanece **físicamente** en la tabla pero ya **no es accesible ni visible** en las consultas.

```
-- Marca la columna como no utilizada.
ALTER TABLE table_name SET UNUSED COLUMN column_name;
```

Más adelante, se pueden eliminar todas las columans marcadas como ***unused*** de forma definitiva con el comando:

```
-- Elimina físicamente todas las columnas marcadas como no utilizadas.
ALTER TABLE table_name DROP UNUSED COLUMNS
```

Esta característica es exclusiva de Oracle y resulta especialmente útil en entornos donde el tiempo de inactividad o el impacto en el rendimiento son críticos.

Rename Column

Para cambiar el nombre de una columna de una tabla, se utiliza la siguiente sintaxis:

```
ALTER TABLE table_name
RENAME COLUMN old_name to new_name;
```

Alter/modify DataType

No existe una única sintaxis estándar en SQL para cambiar el tipo de datos de una columna. Aunque muy similar, cada SGBD tiene una sintaxis particular:

- ***MySQL Syntax:***

```
ALTER TABLE table_name
MODIFY COLUMN column_name new_data_type;
```

- ***PostgreSQL Syntax:***

```
ALTER TABLE table_name
ALTER COLUMN column_name SET DATA TYPE new_data_type;
```

- ***SQL Server Syntax:***

```
ALTER TABLE table_name
ALTER COLUMN column_name new_data_type;
```

- ***Oracle Syntax:***

```
ALTER TABLE table_name
MODIFY column_name new_data_type;
```

> Cambiar el tipo de datos de una columna puede implicar que se **pierdan datos** si el tipo no es compatible o si los valores actuales no se ajustan al nuevo tipo

Constraints

Las ***"constraints"*** o restricciones en SQL son **reglas** que se aplican a las columnas de una tabla para asegurar la integridad de los datos.

- `PRIMARY KEY` : Asegura unicidad y no permite nulos.
- `FOREIGN KEY` : Garantiza la integridad referencial entre tablas.
- `UNIQUE` : Asegura que los valores sean únicos.
- `CHECK` : Valida que los valores cumplan con una condición.
- `NOT NULL` : Asegura que la columna no contenga valores nulos.
- `DEFAULT` : Define un valor predeterminado para la columna.

PRIMARY KEY

- Define una columna (o combinación de columnas) que identifica de **manera única** cada fila de la tabla.
- No permite valores nulos y debe ser única.
- Solo puede haber una `PRIMARY KEY` por tabla.

```
CREATE TABLE Users (
  UserID INT PRIMARY KEY,
  UserName VARCHAR(100)
);
```

FOREIGN KEY

- Define una columna (o combinación de columnas) que se refiere a la `PRIMARY KEY` de otra tabla.
- Garantiza la integridad referencial, es decir, que los valores en la columna de clave externa coincidan con los valores de la tabla referenciada.

```
CREATE TABLE Orders (
  OrderID INT PRIMARY KEY,
  UserID INT,
  FOREIGN KEY (UserID) REFERENCES Users(UserID)
);
```

UNIQUE

- Garantiza que todos los valores en una columna (o combinación de columnas) sean únicos, es decir, no se repitan.
- A diferencia de `PRIMARY KEY`, los valores pueden ser nulos (si se permiten nulos en la columna).
- Una `PRIMARY KEY` automáticamente implica una restricción `UNIQUE`.

```
CREATE TABLE Employees (
  EmployeeID INT,
  Email VARCHAR(100) UNIQUE
);
```

CHECK

- Permite definir una condición para que los valores de una columna cumplan con un criterio específico.
- Asegura que los valores insertados o actualizados en una columna sean válidos según la condición especificada.

```
CREATE TABLE Products (
  ProductID INT PRIMARY KEY,
  Price DECIMAL(10, 2),
  CHECK (Price >= 0)  -- La condición de que el precio no puede ser negativo
);
```

NOT NULL

- Define que una columna no puede contener valores nulos, ya que por defecto una columna puede contener valores nulos.
- Es comúnmente utilizada en combinación con `PRIMARY KEY` o `UNIQUE` para garantizar que cada registro tenga un valor válido.

```
CREATE TABLE Customers (
  CustomerID INT NOT NULL,
  CustomerName VARCHAR(100) NOT NULL
);
```

DEFAULT

Establece un valor predeterminado para una columna cuando no se proporciona un valor explícito al insertar un registro.

```
CREATE TABLE Orders (
  OrderID INT PRIMARY KEY,
  OrderDate DATE DEFAULT CURRENT_DATE
);
```

Create Index

TODO

Auto Increment

TODO

Práctica

MySQL en Docker con persistencia

Creación del contenedor

```
# Crear el contenedor
docker run --name mysql-container -e MYSQL_ROOT_PASSWORD=tu_contraseña -e MYSQL_DATABASE=mi_base_de_datos -v mysql_data:

# Comprobar que el contenedor está corriendo
docker ps
```

- `--name mysql-container` : Nombre del contenedor.
- `-e MYSQL_ROOT_PASSWORD=tu_contraseña` : Establece la contraseña para el usuario *root*.
- `-e MYSQL_DATABASE=mi_base_de_datos` : Crea una base de datos inicial llamada *mi_base_de_datos*.
- `-v mysql_data:/var/lib/mysql` : Crea un volumen llamado *mysql_data* para persistir los datos.
- `-p 3306:3306` : Expone el puerto 3306 (puerto predeterminado de MySQL) para conectarte desde tu máquina.
- `-d mysql:latest` : Utiliza la imagen más reciente de MySQL y ejecuta el contenedor en segundo plano.

Conectarse al contenedor

1. Conectarse desde cualquier cliente MySQL usando `localhost:3306`, el usuario `root` y la contraseña configurada.
2. Conectarse al contenedor de MySQL desde la terminal:

```
docker exec -it mysql-container mysql -uroot -p
```

- `docker exec`: Ejecuta un comando en un contenedor en ejecución.
- `-it`: Permite la interacción (interactiva) y asigna un pseudo-TTY.
- `mysql-container`: Nombre del contenedor que especificaste al crear el contenedor.
- `mysql`: Llama al cliente de MySQL.
- `-uroot`: Especifica que te conectas con el usuario *root*.
- `-p`: Te pedirá que ingreses la contraseña del usuario *root* (la que configuraste al crear el contenedor).

Reiniciar y parar el contenedor

```
# Reiniciar el contenedor
docker start mysql-container

# Parar el contenedor
docker stop mysql-container
```

PostgreSQL en Docker con persistencia

Creación del contenedor

```
docker run --name postgres-container -e POSTGRES_PASSWORD=tu_contraseña -e POSTGRES_DB=mi_base_de_datos -v postgres_data
```

- `--name postgres-container`: Nombre del contenedor.
- `-e POSTGRES_PASSWORD=tu_contraseña`: Establece la contraseña para el usuario *postgres*.
- `-e POSTGRES_DB=mi_base_de_datos`: Crea una base de datos inicial llamada *mi_base_de_datos*.
- `-v postgres_data:/var/lib/postgresql/data`: Crea un volumen llamado *postgres_data* para persistir los datos.
- `-p 5432:5432`: Expone el puerto 5432 (puerto predeterminado de PostgreSQL) para conectarte desde tu máquina.
- `-d postgres:latest`: Utiliza la imagen más reciente de PostgreSQL y ejecuta el contenedor en segundo plano.

Conectarse al contenedor

Es posible conectarse al contenedor vía el cliente `psql` de PostgreSQL:

```
docker exec -it postgres-container psql -U postgres
```

Reiniciar y parar el contenedor

```
# Reiniciar el contenedor
docker start postgres-container

# Parar el contenedor
docker stop postgres-container
```

Referencias

MySQL

- https://dev.mysql.com/doc/
- https://cheatsheets.zip/mysql

PostgreSQL

- https://www.postgresql.org/docs/
- https://wiki.postgresql.org/
- https://cheatsheets.zip/postgres

Microsoft SQL

- https://learn.microsoft.com/es-es/sql

Oracle SQL

- https://docs.oracle.com/en/database/oracle/oracle-database/index.html

General

- https://roadmap.sh/sql
- https://www.w3schools.com/sql/default.asp
- https://www.sqltutorial.org/
- https://www.sqlitetutorial.net/
- https://datalemur.com/sql-tutorial
- https://github.com/XD-DENG/SQL-exercise
- https://www.sqlzoo.net/wiki/SQL_Tutorial
- https://sqlbolt.com/
- https://bookdown.org/paranedagarcia/database/sql.html
- https://gestionbasesdatos.readthedocs.io/es/latest/index.html

Licencia