ChatGPT

Consultas SQL en una o varias tablas

Las bases de datos relacionales utilizan **SQL (Structured Query Language)** o Lenguaje Estructurado de Consultas para **recuperar datos** de una o múltiples tablas de manera eficiente. SQL se considera un lenguaje de manipulación de datos (DML) que permite comunicarse con la base de datos mediante **sentencias de consulta** (principalmente la sentencia `SELECT`) [1] [2]. A continuación, presentaremos de forma **completa y estructurada** cómo realizar consultas SQL para obtener información de una sola tabla y de varias tablas relacionadas, abordando condiciones de filtrado, uso de funciones y agrupamientos, así como **consultas con varias tablas** (joins, subconsultas y la integridad referencial que las respalda).

Consultando información de una tabla

Cuando trabajamos con una sola tabla, podemos **recuperar información** usando la sentencia básica `SELECT`. En su forma más simple, un `SELECT` especifica las columnas deseadas y la tabla objetivo. Por ejemplo, `SELECT * FROM Empleados;` retorna todas las columnas de la tabla *Empleados*. También es posible seleccionar columnas específicas: por ejemplo, `SELECT nombre, salario FROM Empleados;` devolvería solo el nombre y salario de cada empleado. La sintaxis general de una consulta de selección básica incluye cláusulas opcionales como `WHERE` para filtrar filas, `ORDER BY` para ordenar resultados, etc. [3]. Es importante notar que **el resultado de una consulta SQL es otra tabla** (un conjunto de filas y columnas que cumplen los criterios) [4], la cual puede tener desde cero filas hasta muchas, dependiendo de la consulta.

Consultas utilizando la llave primaria

En las bases de datos relacionales, cada tabla suele tener una **llave primaria** (primary key) que identifica de forma única a cada fila (registro) de esa tabla. Esto permite hacer consultas muy precisas recuperando, por ejemplo, **un solo registro específico** mediante su valor de llave primaria. Una consulta típica usando la clave primaria sería, por ejemplo:

```
SELECT *
FROM Empleados
WHERE id_empleado = 105;
```

En este caso, `id_empleado` es la clave primaria de la tabla *Empleados*, por lo que la cláusula `WHERE` filtra exactamente la fila cuyo identificador es 105. Dado que la clave primaria es única por definición, la consulta retornará a lo sumo una fila (el empleado con id 105, si existe). Consultar por la llave primaria es muy eficiente y garantiza la **integridad** de que estamos obteniendo el registro correcto. En general, gracias a la integridad referencial de la base de datos, las claves primarias y foráneas mantienen las relaciones entre tablas sincronizadas; es decir, una clave foránea en otra tabla siempre debe apuntar a una clave primaria válida en la tabla referenciada [5]. Esto asegura que al utilizar una clave primaria para buscar datos (o al enlazar tablas, como veremos más adelante), los datos relacionados existen y son consistentes.

Consultas con condiciones de selección (cláusula WHERE)

Con frecuencia no necesitaremos **todos** los registros de una tabla, sino solo aquellos que cumplan ciertas condiciones. Para filtrar resultados usamos la cláusula `WHERE` en la consulta `SELECT`. La cláusula `WHERE` permite especificar **condiciones de selección** que cada fila debe cumplir para ser incluida en el resultado [6]. SQL ofrece operadores de comparación comunes (`=`, `<>` o `!=`, `>`, `<`, `>=`, `<=`) para construir estas condiciones, así como operadores lógicos (`AND`, `OR`, `NOT`) para combinar múltiples criterios.

Por ejemplo, si quisiéramos obtener todos los empleados del departamento "Ventas" cuyo salario supera 3000, la consulta podría ser:

```
SELECT nombre, departamento, salario
FROM Empleados
WHERE departamento = 'Ventas'
  AND salario > 3000;
```

En esta sentencia, solo aparecerán en el resultado aquellos empleados que **simultáneamente** pertenezcan al departamento *Ventas* y tengan salario mayor a 3000 (gracias al operador lógico `AND`). Si quisiéramos una condición alternativa, podríamos usar `OR` (por ejemplo, salario por encima de 3000 *o* departamento Ventas, etc.). La cláusula `WHERE` recorre la tabla y **devuelve solo los datos que satisfacen la condición especificada** [6]. También es posible utilizar **patrones** con el operador `LIKE` para búsquedas por coincidencia parcial (por ejemplo, `WHERE nombre LIKE 'Mar%'` encontraría nombres que comienzan por "Mar"). En resumen, `WHERE` es fundamental para realizar **consultas condicionales** que resuelvan preguntas específicas sobre los datos, filtrando la información según los criterios que plantee el problema.

Utilización de funciones en las consultas SQL

SQL proporciona numerosas **funciones incorporadas** (built-in) que podemos aprovechar dentro de las consultas para transformar o calcular valores sobre la marcha. Estas funciones pueden ser de **texto**, matemáticas, de fecha/hora, entre otras [7] [8]. Por ejemplo, las funciones de cadena como `LOWER()` y `UPPER()` convierten texto a minúsculas o mayúsculas, `CONCAT()` permite concatenar (unir) cadenas, las funciones numéricas como `ABS()` (valor absoluto), `ROUND()` (redondeo) o `SQRT()` (raíz cuadrada) realizan operaciones matemáticas, y funciones de fecha como `NOW()` obtienen la fecha/hora actual, etc. Estas funciones se pueden utilizar tanto en la lista de selección (`SELECT`) como en la cláusula `WHERE` u otras partes de la consulta para ajustar los datos a nuestras necesidades [9] [10].

Por ejemplo, si quisiéramos obtener en una sola columna el nombre completo de un alumno uniendo su nombre y apellidos, podríamos usar la función de concatenación:

```
SELECT CONCAT(nombre, ' ', apellido1, ' ', apellido2) AS nombre_completo
FROM Alumnos;
```

En este caso usamos `CONCAT` para unir el nombre y apellidos separados por espacios, y `AS nombre_completo` para asignar un alias descriptivo a la columna resultante [11] [12]. Del mismo modo, podríamos querer listar todos los productos con su precio elevado al cuadrado (usando `POW(precio,`

`2)`), o convertir todos los nombres de clientes a mayúsculas para una comparación sin distinguir mayúsculas/minúsculas (`WHERE UPPER(cliente) = 'JUAN PEREZ'`). Las **funciones escalar** aplicadas en consultas nos permiten realizar **cálculos y transformaciones en tiempo real** sobre los datos recuperados, haciendo las consultas más expresivas y poderosas.

Consultas de selección con funciones de agrupación (GROUP BY y funciones agregadas)

A veces las preguntas que queremos responder con SQL implican **agrupar los datos** por alguna categoría y obtener **estadísticas resumidas** de cada grupo. Para estos casos, SQL proporciona las **funciones de agregación** (o de agrupación) junto con la cláusula `GROUP BY`. Las funciones agregadas típicas son `COUNT()` (recuento de filas), `SUM()` (suma de valores), `AVG()` (promedio), `MIN()` (mínimo) y `MAX()` (máximo) [13]. Estas funciones procesan múltiples filas y **devuelven un único valor resumen por cada grupo** de filas considerado [13].

Por ejemplo, si en una tabla *Empleados* quisiéramos saber cuántos empleados hay en cada departamento, podríamos escribir:

```
SELECT departamento, COUNT(*) AS num_empleados
FROM Empleados
GROUP BY departamento;
```

Aquí la cláusula `GROUP BY departamento` indica que las filas se agruparán por el valor de la columna *departamento*. Luego `COUNT(*)` cuenta cuántas filas hay en cada grupo, devolviendo el número de empleados por departamento. Del mismo modo, podríamos calcular el **salario promedio por departamento** usando `AVG(salario)` o la **suma total de salarios por departamento** con `SUM(salario)` agrupados por esa columna. La importancia de la agregación radica en poder obtener **métricas resumidas** sin tener que inspeccionar cada registro individual [13].

Es clave mencionar que cuando se usan funciones de agrupación, *solo* se pueden seleccionar las columnas por las que se agrupa (o funciones agregadas sobre otras columnas). Cualquier columna en la parte `SELECT` que no esté agregada debe aparecer en el `GROUP BY`, de lo contrario SQL lanzará un error [14]. Adicionalmente, SQL ofrece la cláusula `HAVING` para filtrar los resultados **después de agrupar**. `HAVING` funciona de forma similar a `WHERE` pero aplicado sobre grupos: por ejemplo, podríamos agregar `HAVING COUNT(*) > 5` a la consulta anterior para mostrar solo los departamentos con más de 5 empleados. De esta manera, las consultas con agrupación nos permiten responder preguntas del tipo "*¿cuál es el total/promedio/mínimo/máximo de X por cada Y?*", extrayendo **información resumida** muy útil de los datos.

Consultando información relacionada en varias tablas

En una base de datos bien diseñada, los datos suelen estar **distribuidos en varias tablas relacionadas** por claves primarias y foráneas, en lugar de estar todos en una sola tabla (esto evita redundancias y asegura integridad). Para **consultar información que reside en tablas distintas** de forma combinada, SQL proporciona varias técnicas. Las más comunes son las **uniones (JOINs)** entre tablas y las **subconsultas (consultas anidadas)**. Antes de abordar estas técnicas, es importante entender el **modelo de datos relacional** subyacente: cómo están vinculadas las tablas y qué es la integridad referencial.

Modelo de datos relacional e integridad referencial

Un **modelo de datos relacional** describe la estructura de la base de datos, indicando qué tablas existen, qué columnas tiene cada tabla y cómo se **relacionan** entre sí. Estas relaciones normalmente se representan mediante **claves foráneas**: una columna en una tabla que referencia (apunta) a la clave primaria de otra tabla. Por ejemplo, si tenemos una tabla *Clientes* y una tabla *Pedidos*, el modelo de datos podría indicar que *Pedidos* tiene una columna `cliente_id` que es clave foránea hacia `id` (clave primaria) en *Clientes*. Esto significa que cada pedido está asociado a un cliente existente.

La **integridad referencial** es la propiedad que garantiza que las relaciones entre tablas sean válidas en todo momento. En otras palabras, cada valor de clave foránea en una tabla debe existir previamente como valor de clave primaria en la tabla referenciada [5]. El propio motor de la base de datos hace cumplir esta regla: por ejemplo, no se podría insertar un pedido con `cliente_id` que no exista en la tabla de *Clientes*, porque violaría la integridad referencial. Asimismo, no se debería poder borrar un cliente que tenga pedidos asociados sin antes tratar esos pedidos (ya sea borrándolos o reasignándolos), ya que quedarían *referencias huérfanas*. Gracias a la integridad referencial, cuando relacionamos o combinamos datos de varias tablas, podemos confiar en que las correspondencias entre ellas son correctas y consistentes (un pedido siempre "alude" a un cliente válido, una matrícula a un estudiante válido, etc.).

En la práctica, **leer un modelo de datos** implica identificar estas relaciones: suele verse en diagramas entidad-relación o esquemas lógicos donde las tablas se dibujan con sus columnas y se trazan líneas desde las claves foráneas a las claves primarias correspondientes de otras tablas. Al interpretar un modelo, buscamos columnas comunes o lógicamente equivalentes. Por ejemplo, si en el modelo se observa una columna *CustomerID* tanto en la tabla *Orders* como en *Customers*, es indicativo de que podemos relacionar ambas tablas a través de ese campo [15]. El nombre de la columna no siempre tiene que ser idéntico, pero los datos deben corresponder (p.ej., *Orders.CustomerID* puede unirse con *Customers.IdCliente* si almacena los mismos identificadores aunque se llamen distinto). Conociendo esto, pasemos a cómo realizar consultas que unan datos de múltiples tablas.

Consultas de selección con tablas relacionadas (JOIN)

La forma más directa de **consultar datos de dos o más tablas a la vez** es usando la cláusula **JOIN** de SQL. Un *JOIN* nos permite **combinar las filas de dos tablas** basándonos en una columna común entre ellas (generalmente una relación de clave primaria a foránea) [16] [17]. En esencia, el motor de la base de datos compara los valores de la columna en común de ambas tablas y, por defecto, **empareja las filas que tienen valores coincidentes** para producir las filas del resultado combinado [17]. Esto es fundamental para explotar la naturaleza relacional de la base de datos y obtener información integrada de varias fuentes.

Por ejemplo, supongamos una tabla *Empleados* y otra tabla *Departamentos*, donde *Empleados* tiene una columna `departamento_id` que referencia al campo `id` de *Departamentos*. Si queremos obtener una lista de empleados junto con el nombre de su departamento, podríamos hacer lo siguiente:

```
SELECT E.nombre AS Empleado,
       D.nombre AS Departamento
FROM Empleados E
JOIN Departamentos D
  ON E.departamento_id = D.id;
```

En esta consulta usamos `JOIN ... ON ...` para indicar cómo se relacionan las tablas: estamos uniendo cada empleado `E` con el departamento `D` donde **el campo clave coincide** (`E.departamento_id = D.id`). El resultado de esta operación será una tabla donde en cada fila aparece un empleado y el nombre de su departamento correspondiente. Internamente, SQL recorrió ambas tablas y **asoció solamente aquellas filas cuyo identificador de departamento coinciden** en ambas [17].

Es importante destacar que, por defecto, `JOIN` en SQL significa **INNER JOIN**, que muestra solo las filas donde hay coincidencia en ambas tablas. Las filas de una tabla que no tengan correspondencia en la otra **no aparecerán** en el resultado. En nuestro ejemplo, si algún empleado tiene un `departamento_id` que no apunta a ningún departamento existente, ese empleado no saldrá en el listado; de igual modo, un departamento sin empleados no aparecerá listado tampoco [18]. Esta es precisamente la característica de la unión interna (inner join): retorna únicamente la **intersección** de ambas tablas, ignorando elementos no relacionados en una u otra.

A continuación, profundizaremos en los **diferentes tipos de JOIN** que SQL ofrece, ya que además del inner join existen las llamadas uniones externas (outer joins) que permiten incluir también los registros no coincidentes de alguna tabla.

Subconsultas (consultas anidadas)

Otra técnica para consultar varias tablas es usar **subconsultas**, también conocidas como *subqueries* o consultas anidadas. Una **subconsulta** es simplemente una consulta SQL completa que se inserta *dentro de otra* consulta más grande [19]. Generalmente, la subconsulta se ejecuta primero y sus resultados se utilizan por la consulta externa para filtrar o calcular algo. Las subconsultas pueden aparecer en distintas partes: típicamente en la cláusula `WHERE` (para filtrar según resultados de otra consulta), en la cláusula `FROM` (como si fuera una tabla derivada o temporal), o incluso en la lista `SELECT` (subconsultas escalares que retornan un solo valor) [20] [21].

Por ejemplo, imaginemos que queremos listar los empleados cuyo salario es mayor que el salario promedio de todos los empleados. Esto se puede resolver con una subconsulta que calcule el salario promedio, y usar ese resultado para filtrar en la consulta principal:

```
SELECT nombre, salario
FROM Empleados
WHERE salario > (
    SELECT AVG(salario)
    FROM Empleados
);
```

Aquí la subconsulta `(SELECT AVG(salario) FROM Empleados)` obtiene el promedio salarial de la tabla *Empleados*. Luego, la consulta externa selecciona aquellos empleados cuyo salario es superior a ese valor promedio. La subconsulta actúa como un *valor* calculado dinámicamente para la cláusula `WHERE` de la consulta externa. De forma similar, podríamos usar subconsultas para: obtener clientes que tienen pedidos (por ejemplo usando `WHERE cliente_id IN (SELECT cliente_id FROM Pedidos)`), productos cuyo precio está por encima del precio medio de su categoría, etc.

Las subconsultas son muy útiles cuando una consulta depende de resultados que provienen de otra. En algunos casos, un mismo problema puede resolverse tanto con subconsultas como con JOINs, y la

elección depende de la claridad o eficiencia deseada [22]. Es importante mencionar que existen subconsultas de diferentes tipos: subconsultas que devuelven un solo valor (escalares), una fila completa, una tabla completa, etc., y SQL tiene reglas sobre cómo integrarlas en la consulta principal (por ejemplo, si una subconsulta devuelve varias filas, suele usarse con operadores como `IN`, `EXISTS` o con operadores *ANY/ALL* según el caso [23] [24]). En síntesis, las subconsultas nos permiten **anidar preguntas** dentro de otras preguntas, logrando consultas muy expresivas que combinan múltiples pasos lógicos de obtención de datos.

Tipos de JOIN: INNER, LEFT, RIGHT, FULL

Cuando combinamos tablas con JOIN, existen variantes según qué filas queremos conservar en el resultado final. Los **tipos principales de JOIN** en SQL son: **INNER JOIN**, **LEFT JOIN** (o LEFT OUTER JOIN), **RIGHT JOIN** (RIGHT OUTER) y **FULL OUTER JOIN** [25]. A continuación resumimos cada uno:

- **INNER JOIN**: es el tipo por defecto de `JOIN`. Devuelve solo las filas **coincidentes** entre las tablas, es decir, la intersección. Si una fila de la tabla izquierda no tiene correspondencia en la derecha, se excluye (y viceversa) [18]. En el ejemplo de *Empleados* y *Departamentos*, un inner join mostraría solo empleados asignados a departamentos existentes, omitiendo empleados sin departamento y departamentos sin empleados.

- **LEFT JOIN (JOIN externo izquierdo)**: en un left join, se devuelven **todas las filas de la tabla izquierda** (la que aparece primero en el `FROM`), aunque no tengan coincidencia en la tabla derecha [26]. Para las filas de la izquierda que *no* encuentren pareja en la derecha, el resultado incluirá esas filas igualmente, con valores `NULL` en las columnas provenientes de la tabla derecha [27] [28]. En cambio, las filas de la derecha que no tengan correspondencia *sí* se omiten. Siguiendo el ejemplo, un left join de *Empleados (izq)* con *Departamentos (der)* listar\u00eda **todos los empleados**, mostrando el nombre del departamento si existe uno coincidente, o `NULL` si el empleado no está asignado a ningún departamento. (Los departamentos vacíos no aparecerían, porque son tabla derecha en este caso).

- **RIGHT JOIN (JOIN externo derecho)**: funciona de manera análoga al left join pero invertido: preserva **todas las filas de la tabla derecha**, agregando `NULL` en las columnas de la izquierda cuando no hay coincidencia [29] [30]. Usando el join de Empleados-Departamentos pero ahora considerando *Departamentos* como tabla derecha prioritaria, un right join listaría todos los departamentos, incluyendo aquellos sin empleados (mostrando NULL para los campos del empleado). Los empleados sin departamento quedarían fuera en este escenario (porque son filas sin match en la tabla izquierda).

- **FULL JOIN (JOIN externo completo)**: este tipo de join combina el efecto de ambos left y right join. **Devuelve todas las filas de ambas tablas**, emparejando las que coinciden y rellenando con `NULL` las columnas faltantes cuando alguna fila no tiene par en la otra tabla [31]. En otras palabras, conserva **absolutamente todos** los registros de la tabla izquierda y de la derecha. En el resultado de un full outer join, cualquier fila que no encontró pareja en la otra tabla igualmente aparece: las filas no coincidentes de la izquierda tendrán NULLs en las columnas derechas, y las no coincidentes de la derecha tendrán NULLs en las columnas izquierdas. Este join es útil para, por ejemplo, combinar dos listas donde queremos ver tanto las correspondencias como los elementos "sobrantes" de cada lado.

Cada uno de estos JOINs puede ayudarnos a resolver diferentes requerimientos. Por ejemplo, si deseamos un reporte de todos los empleados con sus departamentos, incluyendo empleados sin

departamento (pero no necesitamos departamentos vacíos), usaríamos LEFT JOIN. Si quisiéramos listar todos los departamentos con sus empleados, incluyendo departamentos sin empleados, usaríamos RIGHT JOIN (o invertir el orden de las tablas y hacer left join). Y si quisiéramos un listado maestro de **todas** las entidades de ambas tablas con sus relaciones, usaríamos FULL JOIN. Cabe mencionar que SQL también define otros joins especiales no mencionados en el enunciado, como el **CROSS JOIN** (producto cartesiano, combina todas las filas de ambas tablas sin condición) o el **SELF JOIN** (una tabla unida consigo misma), pero los casos más comunes han sido cubiertos arriba. En la práctica, entender y dominar los tipos de JOIN es fundamental, pues nos permiten **navegar por las relaciones** del modelo de datos y obtener la información integrada que responde a las necesidades más complejas de consulta en una base de datos relacional.

Figura: A continuación se ilustra gráficamente la diferencia entre un *inner join* y un *left join*. El diagrama de Venn sombreado en azul representa las filas incluidas en el resultado para cada caso, considerando un conjunto izquierdo (tabla A) y uno derecho (tabla B):

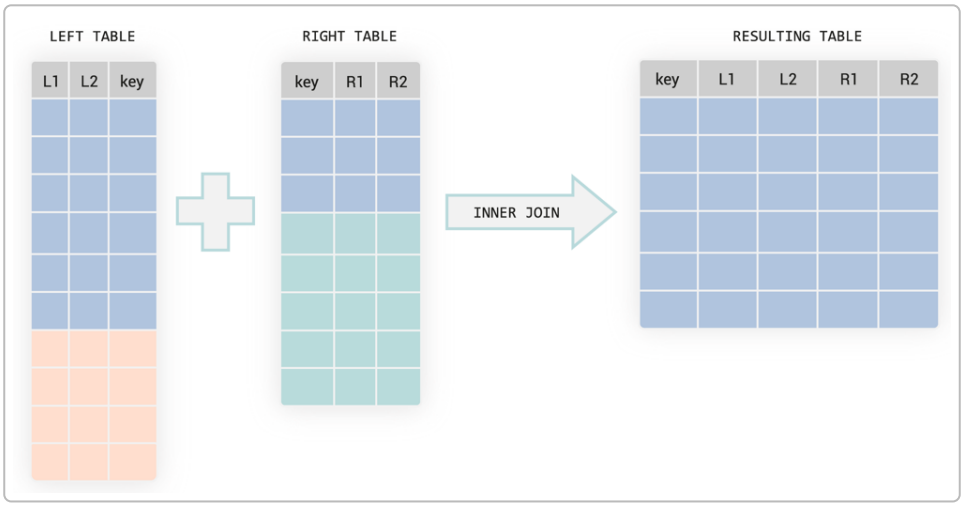

Diagrama de Venn que representa un INNER JOIN: solo se devuelven las filas comunes a ambas tablas (intersección). Las porciones no superpuestas de cada conjunto no aparecen en el resultado [18] .

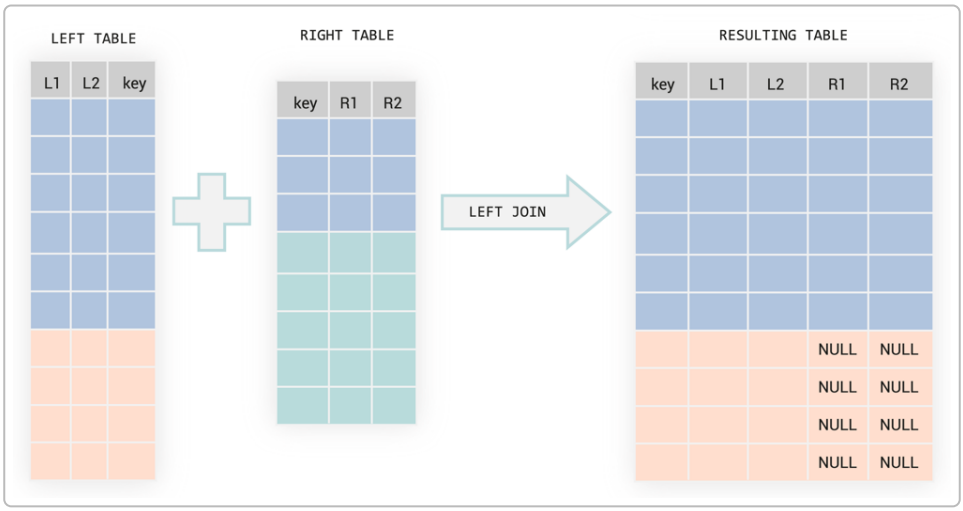

Diagrama de Venn que representa un LEFT JOIN: se devuelven todas las filas de la tabla izquierda (A), incluyendo aquellas sin coincidencia en la tabla derecha. Las filas sin match en B aparecen con valores nulos para las columnas de B [26] *. (El RIGHT JOIN sería simétrico preservando todas las filas de la tabla derecha). El*

FULL JOIN retornaría todas las zonas sombreadas de ambos diagramas, incluyendo tanto la intersección como las partes no comunes de A y B [31].

Como hemos visto, SQL nos ofrece un abanico amplio de herramientas para consultar datos: desde la extracción básica de una tabla con filtros (`SELECT ... FROM ... WHERE ...`), pasando por la generación de resúmenes con agregación (`GROUP BY`, funciones agregadas), hasta llegar a la combinación de datos de múltiples tablas vía `JOIN` o subconsultas. El dominio de estas técnicas nos permite satisfacer prácticamente cualquier requerimiento de información a partir de un modelo de datos dado, garantizando resultados correctos y completos respaldados por la integridad referencial de la base de datos.

Fuentes utilizadas: Referencias de documentación y tutoriales en SQL [1] [6] [7] [18] [26] [5] [17], entre otros, para fundamentar las definiciones y ejemplos presentados. Cada concepto clave ha sido contrastado con fuentes reconocidas a fin de asegurar la claridad y exactitud de la explicación.

---

[1] [6] [25] 20 ejemplos de consultas SQL básicas para principiantes: Una visión completa | LearnSQL.es
https://learnsql.es/blog/20-ejemplos-de-consultas-sql-basicas-para-principiantes-una-vision-completa/

[2] [3] [4] [7] [8] [9] [10] [11] [12] Unidad Didáctica 5. Consultas SQL sobre una tabla
https://josejuansanchez.org/bd/unidad-05-teoria/index.html

[5] Integridad referencial - Wikipedia, la enciclopedia libre
https://es.wikipedia.org/wiki/Integridad_referencial

[13] [14] Explicación de la función de recuento de SQL con 7 ejemplos | LearnSQL.es
https://learnsql.es/blog/explicacion-de-la-funcion-de-recuento-de-sql-con-7-ejemplos/

[15] Relaciones entre tablas en un modelo de datos - Soporte técnico de Microsoft
https://support.microsoft.com/es-es/office/relaciones-entre-tablas-en-un-modelo-de-datos-533dc2b6-9288-4363-9538-8ea6e469112b

[16] [18] [26] [27] [28] [30] [31] ¿Cómo funciona INNER JOIN, LEFT JOIN, RIGHT JOIN y FULL JOIN?
https://programacionymas.com/blog/como-funciona-inner-left-right-full-join

[17] Explicación de los tipos de JOIN en SQL | LearnSQL.es
https://learnsql.es/blog/explicacion-de-los-tipos-de-join-en-sql/

[19] [20] [21] [22] [23] [24] Unidad Didáctica 9. Subconsultas
https://josejuansanchez.org/bd/unidad-09-teoria/index.html

[29] Todos los tipos de JOIN en SQL - Guía de referencia rápida
https://ingenieriadesoftware.es/tipos-sql-join-guia-referencia/