Fundamentos de Bases de Datos

Temario y Ejercicios

Ismael Sallami Moreno

Mayo de 2025

Índice general

1 Apuntes FBD

1.1. Capítulo 1: Definición del esquema de una base de datos

En este capítulo se presentan los comandos fundamentales para definir el esquema de una base de datos en SQL:

- **DESCRIBE**: Muestra la estructura de una tabla.
- **CREATE**: Permite crear nuevas tablas y otros objetos de base de datos.

1.1.1. Tipos de datos en SQL

Tipo de dato | Descripción |

|–|| | `INT` / `INTEGER` / `NUMERIC` | Números enteros con signo (el rango depende del sistema). | | `REAL` / `FLOAT` | Números en coma flotante. | | `NUMBER(p,s)` | Número con precisión `p` y escala `s`. La precisión es el número total de dígitos y la escala, los decimales. | | `CHAR(n)` | Cadena de longitud fija de `n` caracteres. | | `VARCHAR(n)` | Cadena de longitud variable de hasta `n` caracteres. | | `VARCHAR2(n)` | Cadena de longitud variable (1 a 4000 caracteres). Implementación eficiente en Oracle. | | `BLOB` | Almacena objetos binarios de hasta 4 GB (específico de Oracle). | | `LONG RAW(size)` | Cadena binaria de longitud variable de hasta 4 GB (específico de Oracle). | | `DATE` / `TIME` / `TIMESTAMP` | Tipos de datos para fechas y tiempos. |

1.1.1.1. Modificación de tablas

- **ALTER TABLE** `nombre_tabla modificador`;
- **ADD** (atributo [tipo] [DEFAULT expresión] [restricción]);
- **ADD CONSTRAINT** `nombre_restriccion` [[UNIQUE | PRIMARY KEY] (columnas) | FOREIGN KEY (columnas) REFERENCES tabla(columnas) | CHECK (condición)];
- **DROP CONSTRAINT** `nombre_restriccion` [CASCADE];

1.1.1.2. Ejemplo de creación de tabla

```
CREATE TABLE proveedor (
    codpro VARCHAR2(3) CONSTRAINT codpro_no_nulo NOT NULL
        CONSTRAINT codpro_clave_primaria PRIMARY KEY,
    nompro VARCHAR2(30) CONSTRAINT nompro_no_nulo NOT NULL,
    status NUMBER CONSTRAINT status_entre_1_y_10
        CHECK (status >= 1 AND status <= 10),
```

```
    ciudad VARCHAR2(15)
);
```

- Para consultar las tablas existentes:

  ```
  SELECT table_name FROM user_tables;
  ```

1.2. Capítulo 2: Mantenimiento de una base de datos

A continuación se muestran los comandos básicos para el mantenimiento de datos en una base de datos SQL:

1.2.1. Inserción de datos

```
INSERT INTO nombre_tabla [(columna1, columna2, ...)]
VALUES (valor1, valor2, ...);

-- Insertar datos a partir de otra tabla
INSERT INTO nombre_tabla [(columna1, columna2, ...)]
SELECT columna1, columna2, ...
FROM otra_tabla;

-- Ejemplo
INSERT INTO prueba2 VALUES ('aa', 1); -- 'aa' en la primera columna, 1 en la segunda
```

1.2.2. Consulta de datos

- Mostrar todo el contenido de una tabla:

  ```
  SELECT * FROM nombre_tabla;
  ```

- Mostrar campos individuales:

  ```
  SELECT campo1, campo2, ... FROM nombre_tabla;
  ```

1.2.3. Modificación de datos

```
UPDATE nombre_tabla
SET nombre_atributo = 'nuevo_valor'
    [, nombre_atributo2 = 'nuevo_valor2', ...]
[WHERE <condición>];
```

1.2.3.1. Ejemplo de actualización de datos

```
UPDATE plantilla
SET estadocivil = 'divorciado'
WHERE nombre = 'Juan';
```

1.2.3.2. Borrado de tuplas

La instrucción `DELETE` elimina tuplas de una tabla según una condición. Su sintaxis es:

```
DELETE [FROM] nombre_tabla [WHERE <condicion>];
```

- Si se omite la cláusula `WHERE`, se eliminan todas las tuplas de la tabla.

Ejemplo:
Borrar todas las tuplas de la tabla `prueba2`:

```
DELETE FROM prueba2;
```

Borrar todas las tuplas de la tabla `plantilla`:

```
DELETE FROM plantilla;
```

> Puede producir un error si existen restricciones de integridad referencial.

Borrar todas las tuplas de la tabla `serjefe`:

```
DELETE FROM serjefe;
```

1.2.4. Particularidades del tipo de dato DATE

El tipo `DATE` almacena fechas y horas, con un rango muy amplio (desde el 1 de enero de 4712 a.C. hasta el 31 de diciembre de 9999). Permite operaciones aritméticas como sumar o restar días:

- `SYSDATE + 1` es mañana.
- `SYSDATE - 7` es hace una semana.
- `SYSDATE + (10/1440)` es dentro de diez minutos.

Ejemplo:
Actualizar la fecha de alta de Juan al día siguiente:

```
UPDATE plantilla
SET fechaalta = fechaalta + 1
WHERE nombre = 'Juan';
```

Aunque se podrían usar otros tipos para fechas, `DATE` ofrece funciones específicas para su manejo.

1.2.4.1. Introducción de fechas con TO_DATE

La función `TO_DATE` convierte una cadena a tipo fecha usando un formato:

```
INSERT INTO plantilla
VALUES ('11223355', 'Miguel', 'casado', TO_DATE('22/10/2005', 'dd/mm/yyyy'), null);
```

1.2.4.2. Mostrar fechas con TO_CHAR

La función `TO_CHAR` permite mostrar fechas en un formato específico.

1.2.5. Inserción de tuplas en las tablas de ejemplo

Recuerda que SQL distingue mayúsculas y minúsculas en cadenas. Ejemplo de contenido de las tablas:

proveedor:

COD	NOMPRO	STATUS	CIUDAD
S1	Jose Fernandez	2	Madrid
S2	Manuel Vidal	1	Londres
S3	Luisa Gomez	3	Lisboa
S4	Pedro Sanchez	4	Paris
S5	Maria Reyes	5	Roma
S6	Jose Perez	6	Bruselas
S7	Luisa Martin	7	Atenas

pieza:

COD | NOMPIE | COLOR | PESO | CIUDAD |

|–|-|||-| | P1 | Tuerca | Gris | 2.5 | Madrid | | P2 | Tornillo | Rojo | 1.25 | Paris | | P3 | Arandela | Blanco | 3 | Londres | | P4 | Clavo | Gris | 5.5 | Lisboa | | P5 | Alcayata | Blanco | 10 | Roma |

proyecto:

COD	NOMPJ	CIUDAD
J1	Proyecto 1	Londres
J2	Proyecto 2	Londres
J3	Proyecto 3	Paris
J4	Proyecto 4	Roma

ventas (ejemplo de consulta a tabla de otro usuario):

```
SELECT * FROM medina.ventas;
```

Para copiar los datos de `medina.ventas` a tu tabla `ventas` (si los esquemas coinciden):

```
INSERT INTO ventas
SELECT * FROM medina.ventas;
```

Después de insertar, ejecuta `COMMIT` para guardar los cambios permanentemente.

1.2.6. Ejercicio 2.7: Preparar archivo de inserción de datos para la base de datos de baloncesto

A continuación se muestra un ejemplo de archivo SQL para insertar datos en las tablas `Equipos`, `Jugadores`, `Encuentros` y `Faltas`, cumpliendo los criterios indicados:

1.2.7. Inserción de equipos

```
INSERT INTO Equipos VALUES ('E1', 'Tigres');
INSERT INTO Equipos VALUES ('E2', 'Leones');
INSERT INTO Equipos VALUES ('E3', 'Águilas');
INSERT INTO Equipos VALUES ('E4', 'Osos');
```

1.2.8. Inserción de jugadores (5 por equipo)

```
-- Jugadores del equipo E1
INSERT INTO Jugadores VALUES ('J1', 'Carlos Ruiz', 'E1');
INSERT INTO Jugadores VALUES ('J2', 'Luis Pérez', 'E1');
INSERT INTO Jugadores VALUES ('J3', 'Miguel Soto', 'E1');
INSERT INTO Jugadores VALUES ('J4', 'David Gil', 'E1');
INSERT INTO Jugadores VALUES ('J5', 'Sergio Lara', 'E1');

-- Jugadores del equipo E2
INSERT INTO Jugadores VALUES ('J6', 'Juan López', 'E2');
INSERT INTO Jugadores VALUES ('J7', 'Pedro Díaz', 'E2');
INSERT INTO Jugadores VALUES ('J8', 'Raúl Torres', 'E2');
INSERT INTO Jugadores VALUES ('J9', 'Iván Gómez', 'E2');
INSERT INTO Jugadores VALUES ('J10', 'Mario Cano', 'E2');

-- Jugadores del equipo E3
INSERT INTO Jugadores VALUES ('J11', 'Adrián Vera', 'E3');
INSERT INTO Jugadores VALUES ('J12', 'Rubén Ríos', 'E3');
INSERT INTO Jugadores VALUES ('J13', 'Óscar León', 'E3');
INSERT INTO Jugadores VALUES ('J14', 'Pablo Mora', 'E3');
INSERT INTO Jugadores VALUES ('J15', 'Tomás Vidal', 'E3');

-- Jugadores del equipo E4
INSERT INTO Jugadores VALUES ('J16', 'Álvaro Peña', 'E4');
INSERT INTO Jugadores VALUES ('J17', 'Jorge Soler', 'E4');
INSERT INTO Jugadores VALUES ('J18', 'Hugo Marín', 'E4');
INSERT INTO Jugadores VALUES ('J19', 'Samuel Cruz', 'E4');
INSERT INTO Jugadores VALUES ('J20', 'Víctor Rey', 'E4');
```

1.2.9. Inserción de encuentros (10 en total, liga no terminada)

```
-- Formato: (ID, EquipoLocal, EquipoVisitante, PuntosLocal, PuntosVisitante, Fecha)
INSERT INTO Encuentros VALUES (1, 'E1', 'E2', 80, 70, TO_DATE('01/03/2024',
↪  'DD/MM/YYYY'));
INSERT INTO Encuentros VALUES (2, 'E3', 'E4', 65, 75, TO_DATE('02/03/2024',
↪  'DD/MM/YYYY'));
INSERT INTO Encuentros VALUES (3, 'E1', 'E3', 90, 60, TO_DATE('05/03/2024',
↪  'DD/MM/YYYY'));
INSERT INTO Encuentros VALUES (4, 'E2', 'E4', 68, 85, TO_DATE('06/03/2024',
↪  'DD/MM/YYYY'));
INSERT INTO Encuentros VALUES (5, 'E1', 'E4', 78, 70, TO_DATE('09/03/2024',
↪  'DD/MM/YYYY'));
INSERT INTO Encuentros VALUES (6, 'E2', 'E3', 72, 80, TO_DATE('10/03/2024',
↪  'DD/MM/YYYY'));
INSERT INTO Encuentros VALUES (7, 'E4', 'E1', 65, 82, TO_DATE('13/03/2024',
↪  'DD/MM/YYYY'));
INSERT INTO Encuentros VALUES (8, 'E3', 'E2', 77, 69, TO_DATE('14/03/2024',
↪  'DD/MM/YYYY'));
INSERT INTO Encuentros VALUES (9, 'E4', 'E2', 88, 60, TO_DATE('17/03/2024',
↪  'DD/MM/YYYY'));
INSERT INTO Encuentros VALUES (10, 'E1', 'E3', 85, 55, TO_DATE('18/03/2024',
↪  'DD/MM/YYYY'));
```

> En este ejemplo, el equipo `E1` (Tigres) gana todos sus partidos y queda invicto.

1.2.10. Inserción de faltas (ejemplo para algunos encuentros y jugadores)

```
-- Formato: (IDFalta, IDEncuentro, IDJugador, Minuto, TipoFalta)
INSERT INTO Faltas VALUES (1, 1, 'J1', 12, 'Personal');
INSERT INTO Faltas VALUES (2, 1, 'J6', 15, 'Personal');
INSERT INTO Faltas VALUES (3, 2, 'J11', 8, 'Técnica');
INSERT INTO Faltas VALUES (4, 3, 'J3', 20, 'Personal');
INSERT INTO Faltas VALUES (5, 4, 'J17', 30, 'Antideportiva');
```

Puedes ampliar la inserción de faltas según sea necesario para realizar consultas significativas.

1.3. Capítulo 3 Realización de consultas a una base de datos

1.3.1. La sentencia de consulta SELECT

La sentencia `SELECT` permite recuperar datos de una o varias tablas, seleccionando columnas y filas según criterios específicos. Su sintaxis general es:

```
SELECT [DISTINCT | ALL]
  expresión [AS alias_columna]
  [, expresión [AS alias_columna] ...]
```

```
FROM [esquema.]tabla_o_vista [AS alias_tabla]
[WHERE condición]
[GROUP BY expresión [, expresión ...]]
[HAVING condición]
[{UNION | UNION ALL | INTERSECT | MINUS} <SELECT instrucción>]
[ORDER BY expresión [ASC | DESC] ...];
```

Principales cláusulas:

- `SELECT`: Indica las columnas o expresiones a mostrar.
- `FROM`: Especifica la tabla o vista origen de los datos.
- `WHERE`: Filtra las filas según una condición.
- `GROUP BY`: Agrupa filas que tienen valores iguales en las columnas indicadas.
- `HAVING`: Filtra los grupos creados por `GROUP BY`.
- `ORDER BY`: Ordena el resultado según una o varias columnas.
- `DISTINCT`: Elimina duplicados en el resultado.
- `UNION`, `INTERSECT`, `MINUS`: Permiten combinar resultados de varias consultas.

A continuación se detallarán y ejemplificarán las principales cláusulas de la sentencia `SELECT`.

1.3.2. Proyección en SQL (Álgebra Relacional)

La proyección en el Álgebra Relacional (AR) se implementa en SQL mediante la lista de columnas especificadas tras la palabra clave `SELECT`. Si se desea proyectar todos los campos, se utiliza el asterisco (`*`). Para eliminar duplicados, se emplea la cláusula `DISTINCT`.

1.3.2.1. Ejemplo 3.1: Mostrar las ciudades donde hay un proyecto

- **Álgebra Relacional:**

$$\pi_{\text{ciudad}}(\text{proyecto})$$

- **SQL:**

```
SELECT ciudad FROM proyecto;
```

- **Ejercicio 3.1:**
  Comprueba el resultado de la proyección. ¿Es éste conforme a lo que se obtiene en el AR?

  Solución:
  Para obtener solo ciudades distintas (sin duplicados), se utiliza `DISTINCT`:

```
SELECT DISTINCT ciudad FROM proyecto;
```

1.3.2.2. Ejemplo 3.2: Mostrar la información disponible acerca de los proveedores

- **SQL (todos los campos):**

```
SELECT * FROM proveedor;
```

- **SQL (campos específicos):**

```
SELECT codpro, nompro, status, ciudad FROM proveedor;
```

1.3.2.3. Ejercicio 3.2: Mostrar los suministros realizados (solo los códigos de los componentes de una venta)

Supongamos que la tabla de suministros se llama `ventas` y el código del componente es `codcomponente`:

```
SELECT codcomponente FROM ventas;
```

¿Es necesario utilizar `DISTINCT`?

Respuesta:
Solo si existen códigos de componentes repetidos en la tabla de ventas y se desea que cada código aparezca una sola vez. En ese caso:

```
SELECT DISTINCT codcomponente FROM ventas;
```

1.3.3. Selección en SQL (Álgebra Relacional)

La selección en el Álgebra Relacional (AR), representada por el operador σ, se implementa en SQL mediante la cláusula `WHERE`, que filtra las filas según una condición booleana sobre los atributos de la tabla.

1.3.3.1. Ejemplo 3.3: Mostrar los códigos de los proveedores que suministran al proyecto 'J1'

- **Álgebra Relacional:**

$$\pi_{\text{codpro}}(\sigma_{\text{codpj}='J1'}(\text{ventas}))$$

- **SQL:**

```
SELECT codpro FROM ventas WHERE codpj = 'J1';
```

1.3.3.2. Ejercicio 3.3: Muestra las piezas de Madrid que son grises o rojas

```
SELECT * FROM pieza
WHERE ciudad = 'Madrid' AND (color = 'Gris' OR color = 'Rojo');
```

1.3.3.3. Ejercicio 3.4: Encontrar todos los suministros cuya cantidad está entre 200 y 300, ambos inclusive

```
SELECT * FROM ventas
WHERE cantidad BETWEEN 200 AND 300;
```

1.3.4. Operadores adicionales específicos de SQL

1.3.4.1. El operador `LIKE` y los caracteres comodín _ y %

El operador `LIKE` permite comparar cadenas de caracteres usando patrones.
- `%` representa cualquier secuencia de 0 o más caracteres. - _ representa un solo carácter.

1.3.4.1.1. Ejemplo 3.4: Mostrar los proveedores cuyo nombre de ciudad empieza por 'L'

```
SELECT nompro, ciudad FROM proveedor
WHERE ciudad LIKE 'L%';
```

1.3.4.2. Ejercicio 3.5: Mostrar las piezas que contengan la palabra 'tornillo' con la 't' en mayúscula o minúscula

```
SELECT * FROM pieza
WHERE nompie LIKE '%tornillo%' OR nompie LIKE '%Tornillo%';
```

1.3.5. Uso de operadores aritméticos y funciones numéricas

SQL permite realizar operaciones aritméticas y aplicar funciones como `ROUND`, `TRUNC`, `FLOOR`, `CEIL`, etc.

1.3.5.0.1. Ejemplo 3.5: Describe la cantidad de cada venta expresada en docenas, docenas redondeadas y truncadas al tercer decimal, y aproximadas por el entero inferior y superior, solo de las ventas cuyo número de piezas es mayor de diez docenas

```
SELECT cantidad/12 AS docenas,
       ROUND(cantidad/12, 3) AS docenas_redondeadas,
       TRUNC(cantidad/12, 3) AS docenas_truncadas,
       FLOOR(cantidad/12) AS docenas_inferior,
       CEIL(cantidad/12) AS docenas_superior
FROM ventas
WHERE (cantidad/12) > 10;
```

1.3.6. Comparación con el valor nulo: operador `IS [NOT] NULL`

Para comprobar si un atributo es nulo o no, se usan los operadores `IS NULL` y `IS NOT NULL`.

1.3.6.0.1. Ejemplo 3.6: Encontrar los proveedores que tienen su status registrado en la base de datos

```
SELECT codpro, nompro FROM proveedor
WHERE status IS NOT NULL;
```

1.3.7. Consultas sobre el catálogo

Ya podemos consultar con más detalle algunas de las vistas del catálogo.

Para listar las tablas propias del usuario:

```
SELECT table_name
FROM USER_TABLES;
```

1.3.7.1. Ejemplo 3.7: Mostrar la información de todas las tablas denominadas `ventas` a las que tienes acceso

```
SELECT table_name
FROM ALL_TABLES
WHERE TABLE_NAME LIKE '%ventas';
```

1.3.7.2. Ejercicio 3.6

Comprueba que la consulta anterior no devuelve ninguna fila. Sin embargo, sí que existen tablas llamadas `ventas`. Prueba a usar la función `UPPER()` comparando con `'VENTAS'` o la función `LOWER()` comparando con `'ventas'`:

```
-- Usando UPPER
SELECT table_name
FROM ALL_TABLES
WHERE UPPER(table_name) = 'VENTAS';

-- Usando LOWER
SELECT table_name
FROM ALL_TABLES
WHERE LOWER(table_name) = 'ventas';
```

Estas consultas permiten encontrar la tabla independientemente de cómo esté almacenado el nombre (mayúsculas/minúsculas).

1.3.8. Operadores AR sobre conjuntos en SQL

Los operadores de conjuntos en SQL implementan las operaciones del Álgebra Relacional (AR):

- **UNION**: Unión de conjuntos, elimina duplicados.
- **UNION ALL**: Unión de conjuntos, incluye duplicados.
- **INTERSECT**: Intersección de conjuntos.
- **MINUS**: Diferencia de conjuntos.

```
<SELECT instruccion>
  UNION | UNION ALL | INTERSECT | MINUS
<SELECT instruccion>
```

1.3.8.1. Restricciones:

- Los esquemas de las tablas resultantes de cada sentencia `SELECT` deben coincidir en número, posición y tipo de atributos, aunque los nombres pueden diferir.
- El esquema del resultado coincide con el esquema del primer operando.

1.3.8.2. Ejemplo 3.8: Ciudades donde viven proveedores con `status > 2` en las que no se fabrica la pieza `'P1'`

Álgebra Relacional:

$$\pi_{\text{ciudad}}(\sigma_{\text{status}>2}(\text{proveedor})) - \pi_{\text{ciudad}}(\sigma_{\text{codpie}='P1'}(\text{pieza}))$$

SQL:

```
(SELECT DISTINCT ciudad FROM proveedor WHERE status > 2)
MINUS
(SELECT DISTINCT ciudad FROM pieza WHERE codpie = 'P1');
```

1.3.8.3. Ejercicio 3.7: Resolver el ejemplo 3.8 utilizando el operador `INTERSECT`

SQL:

```
(SELECT DISTINCT ciudad FROM proveedor WHERE status > 2)
INTERSECT
(SELECT DISTINCT ciudad FROM pieza WHERE codpie = 'P1');
```

1.3.8.4. Ejercicio 3.8: Encontrar los códigos de proyectos abastecidos únicamente por `'S1'`

Álgebra Relacional:

$$\pi_{\text{codpj}}(\sigma_{\text{codpro}='S1'}(\text{ventas})) - \pi_{\text{codpj}}(\sigma_{\text{codpro}\neq'S1'}(\text{ventas}))$$

SQL:

```
(SELECT DISTINCT codpj FROM ventas WHERE codpro = 'S1')
MINUS
(SELECT DISTINCT codpj FROM ventas WHERE codpro != 'S1');
```

1.3.8.5. Ejercicio 3.9: Mostrar todas las ciudades de la base de datos utilizando `UNION`

SQL:

```
(SELECT DISTINCT ciudad FROM proveedor)
UNION
```

```
(SELECT DISTINCT ciudad FROM pieza)
UNION
(SELECT DISTINCT ciudad FROM proyecto);
```

1.3.8.6. Ejercicio 3.10: Mostrar todas las ciudades de la base de datos utilizando `UNION ALL`

SQL:

```
(SELECT ciudad FROM proveedor)
UNION ALL
(SELECT ciudad FROM pieza)
UNION ALL
(SELECT ciudad FROM proyecto);
```

> **Nota:** `UNION ALL` incluye duplicados, mientras que `UNION` los elimina.

1.3.9. Producto cartesiano en SQL

En SQL, el producto cartesiano se realiza cuando en la cláusula `FROM` se incluyen múltiples tablas o subconsultas. El sistema combina todas las tuplas de las tablas especificadas, generando todas las combinaciones posibles. Posteriormente, se aplica la condición de la cláusula `WHERE` (si existe) para filtrar las tuplas resultantes.

1.3.9.1. Ejercicio 3.11: Comprobar cuántas tuplas resultan del producto cartesiano aplicado a `ventas` y `proveedor`

SQL:

```
SELECT COUNT(*)
FROM ventas, proveedor;
```

1.3.9.2. Ejemplo 3.9: Mostrar las posibles ternas `(codpro, codpie, codpj)` tal que todos los implicados sean de la misma ciudad

Álgebra Relacional:

$$\pi_{\text{codpro,codpie,codpj}}(\sigma_{\text{Proveedor.ciudad=Proyecto.ciudad} \wedge \text{Proyecto.ciudad=Pieza.ciudad}}((\text{proveedor} \times \text{proyecto}) \times \text{pieza}))$$

SQL:

```
SELECT codpro, codpie, codpj
FROM proveedor, proyecto, pieza
WHERE proveedor.ciudad = proyecto.ciudad
AND proyecto.ciudad = pieza.ciudad;
```

1.3.9.3. Ejemplo 3.10: Mostrar las ternas (codpro, codpie, codpj) tal que todos los implicados sean de Londres

Álgebra Relacional:

$$\pi_{\text{codpro,codpie,codpj}}((\sigma_{\text{proveedor.ciudad}='Londres'}(\text{proveedor}) \times \sigma_{\text{proyecto.ciudad}='Londres'}(\text{proyecto})) \times \sigma_{\text{pieza.ciudad}='Londres'}(\text{pieza}))$$

SQL:

```
SELECT codpro, codpie, codpj
FROM (SELECT * FROM proveedor WHERE ciudad = 'Londres') proveedor,
     (SELECT * FROM pieza WHERE ciudad = 'Londres') pieza,
     (SELECT * FROM proyecto WHERE ciudad = 'Londres') proyecto;
```

1.3.9.4. Ejercicio 3.12: Mostrar las ternas que son de la misma ciudad pero que hayan realizado alguna venta

SQL:

```
SELECT codpro, codpie, codpj
FROM proveedor, proyecto, pieza, ventas
WHERE proveedor.ciudad = proyecto.ciudad
AND proyecto.ciudad = pieza.ciudad
AND ventas.codpro = proveedor.codpro
AND ventas.codpj = proyecto.codpj
AND ventas.codpie = pieza.codpie;
```

1.3.10. El renombramiento o alias en SQL

El uso de alias en SQL es útil para abreviar texto y eliminar ambigüedades, especialmente cuando:

1. Se realiza un producto cartesiano de una tabla consigo misma.
2. Se hace referencia a los campos de una consulta incluida en la cláusula `FROM`.

Los alias se definen asociándolos a las tablas o consultas presentes en la cláusula `FROM` que se deseen renombrar.

1.3.10.1. Ejemplo 3.11: Mostrar las posibles ternas (codpro, codpie, codpj) tal que todos los implicados sean de la misma ciudad (revisitando el ejemplo 3.9)

Álgebra Relacional:

$$\pi_{\text{codpro,codpie,codpj}}(\sigma_{\text{S.ciudad=Y.ciudad} \wedge \text{Y.ciudad=P.ciudad}}(\rho_{\text{S}}(\text{Proveedor}) \times \rho_{\text{Y}}(\text{Proyecto}) \times \rho_{\text{P}}(\text{Pieza})))$$

> Cuando se usa rho, lo que se hace es renombrar la tabla como la letra que se le indica.

SQL:

```sql
SELECT codpro, codpie, codpj
FROM proveedor S, proyecto Y, pieza P
WHERE S.ciudad = Y.ciudad
AND Y.ciudad = P.ciudad;
```

1.3.10.2. Ejercicio 3.13: Encontrar parejas de proveedores que no viven en la misma ciudad

SQL:

```sql
SELECT A.codpro AS proveedor1, B.codpro AS proveedor2
FROM proveedor A, proveedor B
WHERE A.ciudad != B.ciudad;
```

1.3.10.3. Ejercicio 3.14: Encontrar las piezas con el máximo peso

SQL:

```sql
SELECT *
FROM pieza
WHERE peso = (SELECT MAX(peso) FROM pieza);
```

1.3.11. La equi-reunión y la reunión natural en SQL

En SQL, la reunión natural y la equi-reunión permiten combinar tablas basándose en condiciones específicas:

1. **Reunión natural (`NATURAL JOIN`)**:
   Se utiliza dentro de la cláusula `FROM` y aplica la reunión sobre los campos que tienen el mismo nombre en las tablas o subconsultas involucradas. Si los tipos de los campos no coinciden, se genera un error.
2. **Equi-reunión (`JOIN ... ON`)**:
   Permite especificar explícitamente los campos sobre los que se realiza la reunión, incluso si no tienen el mismo nombre.

1.3.11.1. Ejemplo 3.12: Mostrar los nombres de proveedores y cantidad de aquellos que han realizado alguna venta en cantidad superior a 800 unidades

Álgebra Relacional:

$$\pi_{\text{nompro,cantidad}}(\text{proveedor} \bowtie \sigma_{\text{cantidad}>800}(\text{ventas}))$$

SQL con `NATURAL JOIN`:

```
SELECT nompro, cantidad
FROM proveedor NATURAL JOIN (SELECT * FROM ventas WHERE cantidad > 800);
```

SQL equivalente sin `NATURAL JOIN` (usando alias):

```
SELECT nompro, cantidad
FROM proveedor s, (SELECT * FROM ventas WHERE cantidad > 800) v
WHERE s.codpro = v.codpro;
```

SQL con `JOIN ... ON`:

```
SELECT nompro, cantidad
FROM proveedor s JOIN (SELECT * FROM ventas WHERE cantidad > 800) v
ON s.codpro = v.codpro;
```

1.3.11.2. Ejercicio 3.15: Mostrar las piezas vendidas por los proveedores de Madrid

SQL:

```
SELECT p.nompie
FROM proveedor s JOIN ventas v
ON s.codpro = v.codpro
JOIN pieza p
ON v.codpie = p.codpie
WHERE s.ciudad = 'Madrid';
```

1.3.11.3. Ejercicio 3.16: Encontrar la ciudad y los códigos de las piezas suministradas a cualquier proyecto por un proveedor que está en la misma ciudad donde está el proyecto

SQL:

```
SELECT p.ciudad, p.codpie
FROM proveedor s JOIN ventas v
ON s.codpro = v.codpro
JOIN proyecto j
ON v.codpj = j.codpj
JOIN pieza p
ON v.codpie = p.codpie
WHERE s.ciudad = j.ciudad;
```

1.3.12. Ordenación de resultados en SQL

En el modelo relacional, no existe orden entre las tuplas ni entre los atributos. Sin embargo, en SQL es posible ordenar los resultados mediante la cláusula `ORDER BY`. Por defecto, el orden es creciente (`ASC`), pero también se puede especificar un orden descendente (`DESC`).

```
SELECT [DISTINCT | ALL] expresion [alias_columna_expresion]
  {,expresion [alias_columna_expresion]}
FROM [esquema.]tabla|vista [alias_tabla_vista]
[WHERE <condicion>]
ORDER BY expresion [ASC | DESC]{,expresion [ASC | DESC]}
```

1.3.12.1. Ejemplo 3.13: Encontrar los nombres de proveedores ordenados alfabéticamente

SQL:

```
SELECT nompro
FROM proveedor
ORDER BY nompro;
```

1.3.12.2. Ejercicio 3.17: Comprobar la salida de la consulta anterior sin la cláusula `ORDER BY`

SQL:

```
SELECT nompro
FROM proveedor;
```

> **Nota:** Sin `ORDER BY`, el orden de las tuplas depende de cómo el SGBD almacena los datos internamente.

1.3.12.3. Ejercicio 3.18: Listar las ventas ordenadas por cantidad, y si algunas ventas coinciden en la cantidad, ordenarlas por fecha de manera descendente

SQL:

```
SELECT *
FROM ventas
ORDER BY cantidad ASC, fecha DESC;
```

1.3.13. Subconsultas en SQL

Las subconsultas permiten operar sobre el resultado de una consulta, incorporándolas en la cláusula `WHERE` de la consulta principal. Esto fragmenta la consulta original en varias más sencillas, evitando numerosas reuniones. Las subconsultas pueden anidarse hasta un nivel permitido por el sistema.

1.3.13.1. Sintaxis general:

```
SELECT <expresion>
FROM tabla
WHERE <expresion> OPERADOR <SELECT instruccion>;
```

Donde `OPERADOR` puede ser cualquiera de los siguientes: `IN`, `EXISTS`, `ANY`, `ALL`, entre otros.

1.3.14. Operador `IN` (Pertenencia a un conjunto)

El operador `IN` se utiliza para comprobar si un valor pertenece a un conjunto obtenido mediante una subconsulta.

1.3.14.1. Ejemplo: Encontrar las piezas suministradas por proveedores de Londres (sin usar reunión)

```
SELECT codpie
FROM ventas
WHERE codpro IN (SELECT codpro FROM proveedor WHERE ciudad = 'Londres');
```

1.3.14.2. Ejercicio: Mostrar las piezas vendidas por los proveedores de Madrid (usando `IN`)

```
SELECT codpie
FROM ventas
WHERE codpro IN (SELECT codpro FROM proveedor WHERE ciudad = 'Madrid');
```

1.3.14.3. Ejercicio: Encontrar los proyectos que están en una ciudad donde se fabrica alguna pieza

```
SELECT codpj
FROM proyecto
WHERE ciudad IN (SELECT ciudad FROM pieza);
```

1.3.14.4. Ejercicio: Encontrar los códigos de proyectos que no utilizan ninguna pieza roja suministrada por un proveedor de Londres

```
SELECT codpj
FROM proyecto
WHERE codpj NOT IN (
    SELECT codpj
    FROM ventas
    WHERE codpie IN (SELECT codpie FROM pieza WHERE color = 'Rojo')
    AND codpro IN (SELECT codpro FROM proveedor WHERE ciudad = 'Londres')
);
```

1.3.15. Operador `EXISTS` (Comprobación de existencia)

El operador `EXISTS` devuelve verdadero si existe al menos una tupla en la relación sobre la que se aplica.

1.3.15.1. Ejemplo: Encontrar los proveedores que suministran la pieza 'P1'

```
SELECT codpro
FROM proveedor
WHERE EXISTS (
    SELECT *
    FROM ventas
    WHERE ventas.codpro = proveedor.codpro
    AND ventas.codpie = 'P1'
);
```

1.3.16. Operadores comparadores sobre conjuntos (ANY y ALL)

Los operadores relacionales (<, <=, >, >=, <>) junto con los cuantificadores ANY y ALL conectan una subconsulta con la consulta principal.

1.3.16.1. Ejemplo: Mostrar el código de los proveedores cuyo estatus sea igual al del proveedor 'S3'

```
SELECT codpro
FROM proveedor
WHERE status = (SELECT status FROM proveedor WHERE codpro = 'S3');
```

1.3.16.2. Ejemplo: Mostrar el código de las piezas cuyo peso es mayor que el peso de alguna pieza 'tornillo'

```
SELECT codpie
FROM pieza
WHERE peso > ANY (SELECT peso FROM pieza WHERE nompie LIKE 'Tornillo%');
```

1.3.16.3. Ejercicio: Mostrar el código de las piezas cuyo peso es mayor que el peso de cualquier 'tornillo'

```
SELECT codpie
FROM pieza
WHERE peso > ALL (SELECT peso FROM pieza WHERE nompie LIKE 'Tornillo%');
```

1.3.16.4. Ejercicio: Encontrar las piezas con peso máximo (comparar con otra solución)

```
SELECT codpie
FROM pieza
WHERE peso = (SELECT MAX(peso) FROM pieza);
```

Las subconsultas son herramientas poderosas para realizar consultas dinámicas y específicas, permitiendo fragmentar problemas complejos en partes más manejables.

1.3.17. La división relacional en SQL

La división relacional en SQL permite resolver consultas en las que se busca determinar si un conjunto de elementos satisface una condición para todos los elementos de otro conjunto. Este operador no es primitivo en el Álgebra Relacional, pero se puede expresar mediante combinaciones de otros operadores.

1.3.17.1. Ejemplo: Mostrar el código de los proveedores que suministran todas las piezas

Álgebra Relacional:

$$\pi_{\text{codpro}}(\text{ventas}) - \pi_{\text{codpro}}((\pi_{\text{codpro}}(\text{ventas}) \times \pi_{\text{codpie}}(\text{pieza})) - \pi_{\text{codpro,codpie}}(\text{ventas}))$$

1.3.18. Aproximación usando expresión equivalente en AR

La división relacional se puede expresar en SQL utilizando operadores como `MINUS` y productos cartesianos.

SQL:

```
(SELECT codpro FROM ventas)
MINUS
(SELECT codpro
 FROM (
   (SELECT v.codpro, p.codpie
    FROM (SELECT DISTINCT codpro FROM ventas) v,
         (SELECT codpie FROM pieza) p)
   MINUS
   (SELECT codpro, codpie FROM ventas)
 ));
```

1.3.19. Aproximación basada en el Cálculo Relacional

Este enfoque utiliza subconsultas anidadas con el operador `NOT EXISTS`.

SQL:

```
SELECT codpro
FROM proveedor
WHERE NOT EXISTS (
  SELECT *
  FROM pieza
  WHERE NOT EXISTS (
    SELECT *
    FROM ventas
    WHERE pieza.codpie = ventas.codpie
    AND proveedor.codpro = ventas.codpro
```

```
  )
);
```

1.3.20. Aproximación mixta usando `NOT EXISTS` y la diferencia relacional

Este enfoque combina el operador `NOT EXISTS` con la diferencia relacional.

SQL:

```
SELECT codpro
FROM proveedor
WHERE NOT EXISTS (
  (SELECT DISTINCT codpie FROM pieza)
  MINUS
  (SELECT DISTINCT codpie FROM ventas WHERE proveedor.codpro = ventas.codpro)
);
```

1.3.20.1. Ejercicio: Encontrar los códigos de las piezas suministradas a todos los proyectos localizados en Londres

SQL:

```
SELECT codpie
FROM pieza
WHERE NOT EXISTS (
  (SELECT codpj FROM proyecto WHERE ciudad = 'Londres')
  MINUS
  (SELECT codpj FROM ventas WHERE pieza.codpie = ventas.codpie)
);
```

1.3.20.2. Ejercicio: Encontrar aquellos proveedores que envían piezas procedentes de todas las ciudades donde hay un proyecto

SQL:

```
SELECT codpro
FROM proveedor
WHERE NOT EXISTS (
  (SELECT DISTINCT ciudad FROM proyecto)
  MINUS
  (SELECT DISTINCT ciudad FROM pieza WHERE pieza.codpie IN (
    SELECT codpie FROM ventas WHERE proveedor.codpro = ventas.codpro
  ))
);
```

La división relacional en SQL es una herramienta poderosa para resolver consultas complejas que involucran relaciones entre conjuntos.

1.3.21. Funciones de agregación en SQL

Las funciones de agregación permiten resumir información sobre un conjunto de tuplas. Algunas de las funciones más comunes son:

- **SUM()**: Calcula la suma de los valores.
- **MIN()**: Encuentra el valor mínimo.
- **MAX()**: Encuentra el valor máximo.
- **AVG()**: Calcula la media de los valores.
- **COUNT()**: Cuenta el número de tuplas.
- **STDDEV()**: Calcula la desviación típica.

Cuando se usa la cláusula `DISTINCT` como argumento de la función, solo se consideran los valores distintos.

1.3.21.1. Ejemplo: Mostrar el máximo, el mínimo y el total de unidades vendidas

SQL:

```
SELECT MAX(cantidad), MIN(cantidad), SUM(cantidad)
FROM ventas;
```

Comparar con:

```
SELECT MAX(DISTINCT cantidad), MIN(DISTINCT cantidad), SUM(DISTINCT cantidad)
FROM ventas;
```

1.3.21.2. Ejercicio: Encontrar el número de envíos con más de 1000 unidades

SQL:

```
SELECT COUNT(*)
FROM ventas
WHERE cantidad > 1000;
```

1.3.21.3. Ejercicio: Mostrar el máximo peso

SQL:

```
SELECT MAX(peso)
FROM pieza;
```

1.3.21.4. Ejercicio: Mostrar el código de la pieza de máximo peso (comparar con ejercicios anteriores)

SQL:

```sql
SELECT codpie
FROM pieza
WHERE peso = (SELECT MAX(peso) FROM pieza);
```

1.3.21.5. Ejercicio: Mostrar los códigos de proveedores que han hecho más de 3 envíos diferentes

SQL:

```sql
SELECT codpro
FROM ventas
GROUP BY codpro
HAVING COUNT(*) > 3;
```

1.3.22. Agrupación de datos con `GROUP BY`

La cláusula `GROUP BY` permite agrupar tuplas según los valores de uno o más atributos y aplicar funciones de agregación a cada grupo.

1.3.22.1. Ejemplo: Para cada proveedor, mostrar la cantidad de ventas realizadas y el máximo de unidades suministradas en una venta

SQL:

```sql
SELECT codpro, COUNT(*), MAX(cantidad)
FROM ventas
GROUP BY codpro;
```

1.3.22.2. Ejercicio: Mostrar la media de las cantidades vendidas por cada código de pieza junto con su nombre

SQL:

```sql
SELECT codpie, nompie, AVG(cantidad)
FROM ventas v JOIN pieza p ON v.codpie = p.codpie
GROUP BY codpie, nompie;
```

1.3.22.3. Ejercicio: Encontrar la cantidad media de ventas de la pieza `'P1'` realizadas por cada proveedor

SQL:

```sql
SELECT codpro, AVG(cantidad)
FROM ventas
WHERE codpie = 'P1'
GROUP BY codpro;
```

1.3.22.4. Ejercicio: Encontrar la cantidad total de cada pieza enviada a cada proyecto

SQL:

```
SELECT codpie, codpj, SUM(cantidad)
FROM ventas
GROUP BY codpie, codpj;
```

1.3.23. Selección de grupos con `HAVING`

La cláusula `HAVING` permite establecer condiciones sobre los grupos creados con `GROUP BY`.

1.3.23.1. Ejemplo: Hallar la cantidad media de ventas realizadas por cada proveedor, indicando solo los códigos de proveedores que han hecho más de 3 ventas

SQL:

```
SELECT codpro, AVG(cantidad)
FROM ventas
GROUP BY codpro
HAVING COUNT(*) > 3;
```

1.3.23.2. Ejemplo: Mostrar la media de unidades vendidas de la pieza `'P1'` realizadas por cada proveedor, indicando solo aquellos que han hecho entre 2 y 10 ventas

SQL:

```
SELECT codpro, codpie, AVG(cantidad)
FROM ventas
WHERE codpie = 'P1'
GROUP BY codpro, codpie
HAVING COUNT(*) BETWEEN 2 AND 10;
```

1.3.23.3. Ejemplo: Encontrar los nombres de proyectos y la cantidad media de piezas que recibe por proveedor

SQL:

```
SELECT v.codpro, v.codpj, j.nompj, AVG(v.cantidad)
FROM ventas v JOIN proyecto j ON v.codpj = j.codpj
GROUP BY v.codpj, j.nompj, v.codpro;
```

1.3.24. Subconsultas en la cláusula `HAVING`

Las subconsultas también pueden usarse en la cláusula `HAVING` para realizar consultas más complejas.

1.3.24.1. Ejemplo: Mostrar el proveedor que más ha vendido en total

SQL:

```
SELECT codpro, SUM(cantidad)
FROM ventas
GROUP BY codpro
HAVING SUM(cantidad) = (
  SELECT MAX(SUM(V1.cantidad))
  FROM ventas V1
  GROUP BY V1.codpro
);
```

1.3.24.2. Ejercicio: Mostrar la pieza que más se ha vendido en total

SQL:

```
SELECT codpie
FROM ventas
GROUP BY codpie
HAVING SUM(cantidad) = (
  SELECT MAX(SUM(V1.cantidad))
  FROM ventas V1
  GROUP BY V1.codpie
);
```

Las funciones de agregación y las cláusulas `GROUP BY` y `HAVING` son herramientas esenciales para resumir y analizar datos en SQL.

1.3.25. Consultas adicionales en SQL

SQL permite realizar consultas avanzadas utilizando funciones de fecha, vistas del catálogo, índices y condiciones complejas. A continuación se presentan ejemplos y ejercicios relacionados.

1.3.26. Uso de fechas en SQL

1.3.26.1. Ejemplo: Lista las fechas de las ventas en formato día, mes y año con 4 dígitos

```
SELECT TO_CHAR(fecha, 'DD-MM-YYYY')
FROM ventas;
```

1.3.26.2. Ejemplo: Encontrar las ventas realizadas entre el 1 de enero de 2002 y el 31 de diciembre de 2004

```
SELECT *
FROM ventas
WHERE fecha BETWEEN TO_DATE('01/01/2002', 'DD/MM/YYYY') AND TO_DATE('31/12/2004',
↪  'DD/MM/YYYY');
```

1.3.26.3. Ejemplo: Mostrar las piezas que nunca fueron suministradas después del año 2001

```
SELECT DISTINCT codpie
FROM pieza
MINUS
SELECT DISTINCT codpie
FROM ventas
WHERE TO_NUMBER(TO_CHAR(fecha, 'YYYY')) > 2001;
```

1.3.27. Agrupación de suministros por años

1.3.27.1. Ejemplo: Agrupar los suministros de la tabla de ventas por años y sumar las cantidades totales anuales

```
SELECT TO_CHAR(fecha, 'YYYY'), SUM(cantidad)
FROM ventas
GROUP BY TO_CHAR(fecha, 'YYYY');
```

1.3.27.2. Ejercicio: Encontrar la cantidad media de piezas suministradas cada mes

```
SELECT TO_CHAR(fecha, 'MM-YYYY'), AVG(cantidad)
FROM ventas
GROUP BY TO_CHAR(fecha, 'MM-YYYY');
```

1.3.28. Consultas sobre el catálogo

1.3.28.1. Ejemplo: Mostrar la información de todos los usuarios del sistema

```
SELECT *
FROM ALL_USERS;
```

1.3.28.2. Ejemplo: Consultar las vistas del catálogo relacionadas con índices

```
DESCRIBE DICTIONARY;
SELECT * FROM DICTIONARY
WHERE table_name LIKE '%INDEX%';
```

1.3.28.3. Ejercicio: Mostrar las tablas `ventas` a las que tienes acceso junto con el nombre del propietario y su número de identificación

```
SELECT table_name, owner, object_id
FROM ALL_TABLES
WHERE table_name = 'VENTAS';
```

1.3.28.4. Ejercicio: Mostrar todos tus objetos creados en el sistema

```
SELECT *
FROM USER_OBJECTS;
```

1.3.29. Ejercicios adicionales

1.3.29.1. Ejercicio: Mostrar los códigos de proveedores que hayan superado las ventas totales realizadas por el proveedor 'S1'

```
SELECT codpro
FROM proveedor
WHERE (SELECT SUM(cantidad) FROM ventas WHERE codpro = proveedor.codpro) >
      (SELECT SUM(cantidad) FROM ventas WHERE codpro = 'S1');
```

1.3.29.2. Ejercicio: Mostrar los mejores proveedores (los que tienen mayores cantidades totales)

```
SELECT codpro, SUM(cantidad) AS total
FROM ventas
GROUP BY codpro
ORDER BY total DESC;
```

1.3.29.3. Ejercicio: Mostrar los proveedores que venden piezas a todas las ciudades de los proyectos a los que suministra 'S3', sin incluirlo

```
SELECT codpro
FROM proveedor
WHERE codpro != 'S3'
AND NOT EXISTS (
  (SELECT DISTINCT ciudad FROM proyecto WHERE codpj IN (SELECT codpj FROM ventas WHERE
 ↪  codpro = 'S3'))
  MINUS
  (SELECT DISTINCT ciudad FROM proyecto WHERE codpj IN (SELECT codpj FROM ventas WHERE
 ↪  codpro = proveedor.codpro))
);
```

1.3.29.4. Ejercicio: Encontrar aquellos proveedores que hayan hecho al menos diez pedidos

```
SELECT codpro
FROM ventas
GROUP BY codpro
HAVING COUNT(*) >= 10;
```

1.3.29.5. Ejercicio: Encontrar aquellos proveedores que venden todas las piezas suministradas por 'S1'

```
SELECT codpro
FROM proveedor
WHERE NOT EXISTS (
  (SELECT DISTINCT codpie FROM ventas WHERE codpro = 'S1')
  MINUS
  (SELECT DISTINCT codpie FROM ventas WHERE codpro = proveedor.codpro)
);
```

1.3.29.6. Ejercicio: Encontrar la cantidad total de piezas que ha vendido cada proveedor que cumple la condición de vender todas las piezas suministradas por 'S1'

```
SELECT codpro, SUM(cantidad)
FROM ventas
WHERE codpro IN (
  SELECT codpro
  FROM proveedor
  WHERE NOT EXISTS (
    (SELECT DISTINCT codpie FROM ventas WHERE codpro = 'S1')
    MINUS
    (SELECT DISTINCT codpie FROM ventas WHERE codpro = proveedor.codpro)
  )
)
GROUP BY codpro;
```

1.3.29.7. Ejercicio: Encontrar qué proyectos están suministrados por todos los proveedores que suministran la pieza 'P3'

```
SELECT codpj
FROM proyecto
WHERE NOT EXISTS (
  (SELECT codpro FROM ventas WHERE codpie = 'P3')
  MINUS
  (SELECT codpro FROM ventas WHERE codpj = proyecto.codpj)
);
```

1.3.29.8. Ejercicio: Encontrar la cantidad media de piezas suministradas a aquellos proveedores que venden la pieza 'P3'

```
SELECT codpro, AVG(cantidad)
FROM ventas
WHERE codpro IN (SELECT codpro FROM ventas WHERE codpie = 'P3')
GROUP BY codpro;
```

1.3.29.9. Ejercicio: Mostrar los nombres de tus índices y sobre qué tablas están montados, indicando además su propietario

```
SELECT index_name, table_name, owner
FROM ALL_INDEXES
WHERE owner = USER;
```

1.3.29.10. Ejercicio: Implementar el comando DESCRIBE para tu tabla ventas a través de una consulta a las vistas del catálogo

```
SELECT column_name, data_type, data_length
FROM ALL_TAB_COLUMNS
WHERE table_name = 'VENTAS';
```

1.3.30. Ejercicios sobre condiciones con "todos/as"

1.3.30.1. Ejercicio: Encontrar todos los proveedores que venden una pieza roja

```
SELECT codpro
FROM ventas
WHERE codpie IN (SELECT codpie FROM pieza WHERE color = 'Rojo');
```

1.3.30.2. Ejercicio: Encontrar todos los proveedores que venden todas las piezas rojas

```
SELECT codpro
FROM proveedor
WHERE NOT EXISTS (
  (SELECT codpie FROM pieza WHERE color = 'Rojo')
  MINUS
  (SELECT codpie FROM ventas WHERE codpro = proveedor.codpro)
);
```

1.3.30.3. Ejercicio: Encontrar todos los proveedores tales que todas las piezas que venden son rojas

```
SELECT codpro
FROM proveedor
WHERE NOT EXISTS (
  (SELECT codpie FROM ventas WHERE codpro = proveedor.codpro)
  MINUS
  (SELECT codpie FROM pieza WHERE color = 'Rojo')
);
```

1.3.30.4. Ejercicio: Encontrar el nombre de aquellos proveedores que venden más de una pieza roja

```
SELECT codpro
FROM ventas
WHERE codpie IN (SELECT codpie FROM pieza WHERE color = 'Rojo')
GROUP BY codpro
HAVING COUNT(DISTINCT codpie) > 1;
```

1.3.30.5. Ejercicio: Encontrar todos los proveedores que vendiendo todas las piezas rojas cumplen la condición de que todas sus ventas son de más de 10 unidades

```
SELECT codpro
FROM proveedor
WHERE NOT EXISTS (
  (SELECT codpie FROM pieza WHERE color = 'Rojo')
  MINUS
  (SELECT codpie FROM ventas WHERE codpro = proveedor.codpro AND cantidad > 10)
);
```

1.3.30.6. Ejercicio: Colocar el `status` igual a 1 a aquellos proveedores que solo suministran la pieza `'P1'`

```
UPDATE proveedor
SET status = 1
WHERE codpro IN (
  SELECT codpro
  FROM ventas
  GROUP BY codpro
  HAVING COUNT(DISTINCT codpie) = 1 AND MIN(codpie) = 'P1'
);
```

1.3.30.7. Ejercicio: Encontrar, de entre las piezas que no se han vendido en septiembre de 2009, las ciudades de aquellas que se han vendido en mayor cantidad durante agosto de ese mismo año

```
SELECT ciudad
FROM pieza
WHERE codpie IN (
  SELECT codpie
  FROM ventas
  WHERE TO_CHAR(fecha, 'MM-YYYY') = '08-2009'
  AND cantidad = (SELECT MAX(cantidad) FROM ventas WHERE TO_CHAR(fecha, 'MM-YYYY') =
   ↪  '08-2009')
)
AND codpie NOT IN (
  SELECT codpie
```

```
  FROM ventas
  WHERE TO_CHAR(fecha, 'MM-YYYY') = '09-2009'
);
```

Estas consultas adicionales permiten explorar y analizar datos de manera avanzada en SQL.

1.3.31. Ejercicios adicionales

1.3.31.1. Consultas sin operadores de agregación

1.3.31.1.1. Ejercicio: Mostrar la información disponible acerca de los encuentros de liga

```
SELECT *
FROM Encuentros;
```

1.3.31.1.2. Ejercicio: Mostrar los nombres de los equipos y de los jugadores ordenados alfabéticamente

```
SELECT nombreE AS equipo, nombreJ AS jugador
FROM Equipos JOIN Jugadores ON Equipos.codE = Jugadores.codE
ORDER BY equipo, jugador;
```

1.3.31.1.3. Ejercicio: Mostrar los jugadores que no tienen ninguna falta

```
SELECT nombreJ
FROM Jugadores
WHERE codJ NOT IN (SELECT codJ FROM Alineaciones WHERE Faltas > 0);
```

1.3.31.1.4. Ejercicio: Mostrar los compañeros de equipo del jugador con código 'x'

```
SELECT nombreJ
FROM Jugadores
WHERE codE = (SELECT codE FROM Jugadores WHERE codJ = 'x')
AND codJ != 'x';
```

1.3.31.1.5. Ejercicio: Mostrar los jugadores y la localidad donde juegan

```
SELECT nombreJ, localidad
FROM Jugadores JOIN Equipos ON Jugadores.codE = Equipos.codE;
```

1.3.31.1.6. Ejercicio: Mostrar todos los encuentros posibles de la liga

```
SELECT E1.nombreE AS equipo_local, E2.nombreE AS equipo_visitante
FROM Equipos E1, Equipos E2
WHERE E1.codE != E2.codE;
```

1.3.31.1.7. Ejercicio: Mostrar los equipos que han ganado algún encuentro jugando como local

```
SELECT nombreE
FROM Equipos JOIN Encuentros ON Equipos.codE = Encuentros.ELocal
WHERE PLocal > PVisitante;
```

1.3.31.1.8. Ejercicio: Mostrar los equipos que han ganado algún encuentro

```
SELECT DISTINCT nombreE
FROM Equipos JOIN Encuentros ON Equipos.codE IN (Encuentros.ELocal,
 ↪  Encuentros.EVisitante)
WHERE (codE = ELocal AND PLocal > PVisitante) OR (codE = EVisitante AND PVisitante >
 ↪  PLocal);
```

1.3.31.1.9. Ejercicio: Mostrar los equipos que todos los encuentros que han ganado lo han hecho como equipo local

```
SELECT nombreE
FROM Equipos
WHERE codE NOT IN (
  SELECT EVisitante
  FROM Encuentros
  WHERE PVisitante > PLocal
);
```

1.3.31.1.10. Ejercicio: Mostrar los encuentros que faltan para terminar la liga

```
SELECT E1.nombreE AS equipo_local, E2.nombreE AS equipo_visitante
FROM Equipos E1, Equipos E2
WHERE E1.codE != E2.codE
AND (E1.codE, E2.codE) NOT IN (SELECT ELocal, EVisitante FROM Encuentros);
```

1.3.31.1.11. Ejercicio: Mostrar los encuentros que tienen lugar en la misma localidad

```
SELECT *
FROM Encuentros JOIN Equipos ON Encuentros.ELocal = Equipos.codE
WHERE Equipos.localidad = (SELECT localidad FROM Equipos WHERE codE =
 ↪  Encuentros.EVisitante);
```

1.3.31.2. Consultas con operadores de agregación

1.3.31.2.1. Ejercicio: Para cada equipo, mostrar la cantidad de encuentros que ha disputado como local

```
SELECT ELocal AS equipo, COUNT(*) AS encuentros_local
FROM Encuentros
GROUP BY ELocal;
```

1.3.31.2.2. Ejercicio: Mostrar los encuentros en los que se alcanzó mayor diferencia

```
SELECT *
FROM Encuentros
WHERE ABS(PLocal - PVisitante) = (
  SELECT MAX(ABS(PLocal - PVisitante)) FROM Encuentros
);
```

1.3.31.2.3. Ejercicio: Mostrar los jugadores que no han superado 3 faltas acumuladas

```
SELECT codJ
FROM Alineaciones
GROUP BY codJ
HAVING SUM(Faltas) <= 3;
```

1.3.31.2.4. Ejercicio: Mostrar los equipos con mayores puntuaciones en los partidos jugados fuera de casa

```
SELECT EVisitante AS equipo, MAX(PVisitante) AS puntos
FROM Encuentros
GROUP BY EVisitante;
```

1.3.31.2.5. Ejercicio: Mostrar la cantidad de victorias de cada equipo, jugando como local o como visitante

```
SELECT codE AS equipo,
       SUM(CASE WHEN codE = ELocal AND PLocal > PVisitante THEN 1 ELSE 0 END) AS
        ↪ victorias_local,
       SUM(CASE WHEN codE = EVisitante AND PVisitante > PLocal THEN 1 ELSE 0 END) AS
        ↪ victorias_visitante
FROM Equipos LEFT JOIN Encuentros ON codE IN (ELocal, EVisitante)
GROUP BY codE;
```

1.3.31.2.6. Ejercicio: Mostrar el equipo con mayor número de victorias

```
SELECT equipo, MAX(victorias_local + victorias_visitante) AS total_victorias
FROM (
  SELECT codE AS equipo,
         SUM(CASE WHEN codE = ELocal AND PLocal > PVisitante THEN 1 ELSE 0 END) AS
          ↪ victorias_local,
         SUM(CASE WHEN codE = EVisitante AND PVisitante > PLocal THEN 1 ELSE 0 END) AS
          ↪ victorias_visitante
  FROM Equipos LEFT JOIN Encuentros ON codE IN (ELocal, EVisitante)
  GROUP BY codE
) victorias;
```

1.3.31.2.7. Ejercicio: Mostrar el promedio de puntos por equipo en los encuentros de ida

```
SELECT codE AS equipo, AVG(PLocal) AS promedio_puntos
FROM Equipos JOIN Encuentros ON codE = ELocal
GROUP BY codE;
```

1.3.31.2.8. Ejercicio: Mostrar el equipo con mayor número de puntos en total de los encuentros jugados

```
SELECT codE AS equipo, SUM(PLocal + PVisitante) AS total_puntos
FROM Equipos JOIN Encuentros ON codE IN (ELocal, EVisitante)
GROUP BY codE
ORDER BY total_puntos DESC
LIMIT 1;
```

Estos ejercicios abarcan consultas básicas y avanzadas, utilizando operadores de agregación y condiciones complejas para analizar datos de la liga de baloncesto.

1.4. Capítulo 4: Definición del nivel externo de un DBMS

En esta unidad se introduce el concepto de **vista** como mecanismo para materializar el nivel externo de una Base de Datos y se proporcionan las sentencias SQL necesarias para su creación y manipulación.

1.4.1. 4.1 Creación y manipulación de vistas

- Una **vista** es una presentación de datos procedentes de una o más tablas, hecha a la medida de un usuario.
- Básicamente, consiste en asignar un nombre a la salida de una consulta y utilizarla como si de una tabla almacenada se tratara.
- En general, pueden usarse en lugar de cualquier nombre de tabla en las sentencias del DML (Lenguaje de Manipulación de Datos).
- La vista es la estructura de más alto nivel dentro del nivel lógico y es el mecanismo básico de implementación del nivel externo.

1.4.1.1. Almacenamiento y Reconstrucción:

- Salvo que se especifique lo contrario (como en vistas materializadas), las vistas no contienen datos almacenados físicamente de forma persistente.
- Su definición se almacena en el diccionario (catálogo) de la base de datos.
- Los datos que representan se reconstruyen cada vez que se accede a ellas, ejecutando la consulta almacenada.

1.4.1.2. Vistas Materializadas:

- Existe una variedad de vista que sí contiene datos: la vista materializada, que se usa en entornos distribuidos para replicar datos.

1.4.1.3. Restricciones de Integridad en Vistas:

- En Oracle®, se pueden aplicar restricciones de integridad mediante el uso de disparadores de tipo "INSTEAD OF", que interceptan operaciones DML sobre las vistas para programar sus efectos sobre las tablas base.

1.4.1.4. Beneficios de las Vistas:

- **Seguridad:** Permiten establecer niveles de seguridad adicionales, ocultando cierta información y haciendo visible a los usuarios sólo la parte de la BD que necesiten.
- **Simplificación:** Simplifican el aspecto de la BD y el uso de algunos comandos para los usuarios.
- **Ejemplo en el Catálogo:** El catálogo de la BD usa vistas para mostrar a cada usuario la información que le concierne de la estructura de la BD.

1.4.2. Creación de Vistas (`CREATE VIEW`)

- Se utiliza el comando `CREATE VIEW`.
- En la cláusula `AS` se especifica la consulta `SELECT` que determina qué filas y columnas de la tabla o tablas almacenadas forman parte de la vista.
- La ejecución de esta sentencia, básicamente produce la inserción de una fila en el catálogo.

1.4.2.1. Ejemplo:

```
CREATE VIEW VentasParis (codpro, codpie, codpj, cantidad, fecha) AS
SELECT codpro, codpie, codpj, cantidad, fecha
FROM ventas
WHERE (codpro, codpie, codpj) IN
    (SELECT codpro, codpie, codpj
    FROM proveedor, pieza, proyecto
    WHERE proveedor.ciudad='Paris' AND
          pieza.ciudad='Paris' AND
          proyecto.ciudad='Paris');
```

1.4.2.2. Información de Vistas en el Catálogo:

- La información registrada sobre las vistas puede consultarse a través de la vista del catálogo `ALL_VIEWS`.
- Atributos relevantes de `ALL_VIEWS`: `owner` (propietario), `view_name` (nombre de la vista), y `text` (la sentencia `SELECT` que la define).

1.4.3. Consulta de vistas

- Una vez creada, cualquier usuario autorizado podrá hacer uso de la vista como si de cualquier tabla se tratara.

1.4.3.1. Ejemplo:

```
SELECT DISTINCT codpro
FROM VentasParis
WHERE codpj='J4';
```

1.4.4. Actualización de vistas

- Debido a su naturaleza virtual, existen fuertes restricciones para insertar o actualizar datos en una vista. No siempre cada fila de una vista se corresponde con una fila de una tabla concreta.
- Los comandos `DELETE`, `INSERT`, `UPDATE` sólo se podrán utilizar en determinadas ocasiones.

1.4.4.1. Restricciones Comunes para la Actualización de Vistas:

- La definición de la vista no podrá incluir cláusulas de agrupamiento (`GROUP BY`) o funciones de agregación (`MAX`, `COUNT`, `AVG`, etc.).
- La definición de la vista no podrá incluir la cláusula `DISTINCT`.
- La definición de la vista no podrá incluir operaciones de reunión (`JOIN`) ni de conjuntos (`UNION`, `MINUS`, etc.); deberá construirse sobre una única tabla base.
- Todos los atributos de la tabla base que deban tomar siempre valor (`NOT NULL` y `PRIMARY KEY`) han de estar incluidos necesariamente en la definición de la vista.

1.4.4.2. Ejemplo de Inserción:

```
CREATE VIEW PiezasLondres AS
SELECT codpie, nompie, color, peso FROM Pieza
WHERE pieza.ciudad='Londres';

INSERT INTO PiezasLondres
VALUES('P9', 'Pieza 9', 'rojo', 90);
```

1.4.5. Eliminación de vistas

- El comando para borrar una vista es: `DROP VIEW <nombre_vista>`.

1.4.5.1. Ejemplo:

```
DROP VIEW VentasParis;
```

- Para cambiar la definición de una vista existente, es mejor utilizar `CREATE OR REPLACE VIEW <nombre_vista>`.

1.4.6. Ejercicios de vistas

1.4.6.1. Ejercicio 4.1: Crear una vista con los proveedores de Londres

```
CREATE VIEW ProveedoresLondres AS
SELECT codpro, nompro, ciudad
FROM proveedor
WHERE ciudad = 'Londres';
```

1.4.6.2. Ejercicio 4.2: Crear una vista con nombres y ciudades de proveedores

```
CREATE VIEW NombresCiudadesProveedores AS
SELECT nompro, ciudad
FROM proveedor;
```

1.4.6.3. Ejercicio 4.3: Crear una vista con código de proveedor, nombre de proveedor y código de proyecto para piezas grises

```
CREATE VIEW ProveedoresPiezasGrises AS
SELECT proveedor.codpro, proveedor.nompro, proyecto.codpj
FROM proveedor
JOIN ventas ON proveedor.codpro = ventas.codpro
JOIN pieza ON ventas.codpie = pieza.codpie
JOIN proyecto ON ventas.codpj = proyecto.codpj
WHERE pieza.color = 'Gris';
```

Estas vistas permiten simplificar consultas y mejorar la seguridad y organización de los datos en la base de datos.

1.5. Capítulo 5: Introducción a la administración: el catálogo y gestión de privilegios

Este capítulo se enfoca en dos aspectos importantes de la administración de bases de datos: el **catálogo** (o diccionario de datos) y la **gestión de privilegios** para controlar el acceso y las operaciones de los usuarios.

1.5.1. 5.1 Información acerca de la base de datos: las vistas del catálogo

- El **catálogo** o **diccionario de datos** de un SGBD proporciona toda la información relevante sobre la estructura y el contenido de una base de datos.
- Está formado por una serie de tablas y vistas que almacenan metadatos sobre todos los objetos de la base de datos (tablas, restricciones, usuarios, roles, privilegios, etc.).
- Para ver las vistas del catálogo en su totalidad y para modificarlas se necesitan privilegios especiales. Por ello, existen vistas predefinidas para las consultas más habituales que permiten a los usuarios acceder a *su* información.

1.5.1.1. 5.1.1 Algunas vistas relevantes del catálogo de la base de datos

El catálogo incluye varias vistas importantes, entre ellas:

- `DICTIONARY`: Contiene descripciones de las tablas y vistas del diccionario de datos.
- `USER_CATALOG`: Muestra tablas, vistas, clústeres, índices, sinónimos y secuencias propiedad del usuario actual.
- `USER_TABLES`: Contiene información sobre las tablas del usuario, como `TABLE_NAME` (nombre de la tabla) y `TABLESPACE_NAME` (espacio de tablas donde se almacena).
- `USER_CONSTRAINTS`: Definiciones de restricciones sobre las tablas del usuario.
- `USER_TAB_COLUMNS`: Descripción de las columnas de las tablas, vistas y clústeres pertenecientes al usuario.
- `USER_INDEXES`: Información de los índices del usuario.
- `ALL_TABLES`: Información de aquellas tablas a las que el usuario tiene acceso porque el propietario lo ha permitido.
- `ALL_VIEWS`: Información de aquellas vistas a las que el usuario tiene acceso.
- `ALL_USERS`: Información sobre todos los usuarios del sistema.

1.5.2. 5.2 Gestión de privilegios

Este apartado identifica los tipos de privilegios en Oracle® y cómo otorgarlos y retirarlos. Se distinguen privilegios del sistema y privilegios sobre objetos.

1.5.2.1. 5.2.1 Privilegios del sistema

- Permiten al usuario realizar acciones particulares en la base de datos.
- Existen más de 80 privilegios de este tipo, y su número aumenta con cada versión de Oracle®.
- Solo usuarios con privilegios de administración pueden gestionar estos privilegios.
- Los comandos `GRANT` y `REVOKE` controlan estos privilegios.
- **Concesión de un privilegio de sistema (`GRANT`):** `sql GRANT system_priv TO user [WITH ADMIN OPTION];`
  - `WITH ADMIN OPTION` permite que el usuario autorizado pueda, a su vez, otorgar el privilegio a otros.
- **Derogación de privilegios de sistema (`REVOKE`):** `sql REVOKE system_priv FROM user;`

1.5.2.2. 5.2.2 Privilegios sobre los objetos

- Autorizan la realización de ciertas operaciones sobre objetos concretos de la base de datos.
- Ejemplos de privilegios:

  - **Tabla/Vista:** `DELETE`, `INSERT`, `REFERENCES`, `SELECT`, `UPDATE`.
  - **Secuencia:** `SELECT`.
  - **Procedimiento:** `EXECUTE`.

- **Concesión de privilegios sobre objetos (`GRANT`):** `sql GRANT object_priv ON object TO user [WITH GRANT OPTION];`
  - `WITH GRANT OPTION` autoriza al usuario a conceder a su vez el privilegio a otros.
- **Derogación de privilegios sobre objetos (`REVOKE`):** `sql REVOKE object_priv ON object FROM user;`

1.5.3. Ejercicios de gestión de privilegios

1.5.3.1. Ejercicio 5.1: Crear una tabla y conceder privilegio `SELECT` a otro usuario

```
CREATE TABLE ejemplo (
    id NUMBER PRIMARY KEY,
    nombre VARCHAR2(50)
);

GRANT SELECT ON ejemplo TO usuario2;
```

1.5.3.2. Ejercicio 5.2: Retirar el privilegio `SELECT`

```
REVOKE SELECT ON ejemplo FROM usuario2;
```

1.5.3.3. Ejercicio 5.3: Conceder el privilegio `SELECT` con `WITH GRANT OPTION`

```
GRANT SELECT ON ejemplo TO usuario2 WITH GRANT OPTION;
```

1.5.3.4. Ejercicio 5.4: Comprobar el efecto en cascada al retirar el privilegio original

```
REVOKE SELECT ON ejemplo FROM usuario1;
```

Este capítulo proporciona las herramientas necesarias para administrar el acceso y las operaciones en una base de datos, garantizando la seguridad y el control de los datos.

1.6. Capítulo 6: Nivel interno: Índices, clusters y hashing

Este capítulo aborda la gestión del nivel interno de una base de datos, centrándose en estructuras que optimizan el acceso y almacenamiento de datos.

1.6.1. 6.1 Creación y manipulación de índices

Los **índices** son estructuras asociadas a tablas y clústeres que aceleran la ejecución de sentencias SQL. No afectan la sintaxis de las sentencias SQL, sino que el SGBD los utiliza automáticamente para un acceso rápido a la información.

1.6.1.1. 6.1.1 Selección de índices

- **Beneficios vs. Costo:** Los índices mejoran la eficiencia en accesos, pero ralentizan las actualizaciones y ocupan espacio en disco, ya que deben actualizarse con cada modificación de la tabla.
- **Mantenimiento Automático:** El SGBD se encarga de usarlos y mantenerlos automáticamente una vez creados.
- **Índice de Clave Primaria:** Oracle® crea automáticamente un índice asociado a la llave primaria de una tabla.
- **Recomendación:** Crear solo los índices necesarios y borrarlos si no son útiles frente a las consultas habituales.

1.6.1.2. 6.1.2 Creación de índices

Se utiliza la sentencia `CREATE INDEX`.

- **Sintaxis básica:** `sql CREATE INDEX nombre_del_indice ON tabla (campo [ASC | DESC],...);`
- **Ejemplo:** Para acelerar búsquedas por nombre de proveedor: `sql CREATE INDEX indice_proveedores ON proveedor (nompro);`

1.6.1.3. 6.1.3 Índices compuestos

- Un índice compuesto se crea sobre más de una columna de la tabla.
- Pueden acelerar consultas que referencien a todas las columnas indexadas o solo a las primeras.
- **Ejemplo:** `sql CREATE INDEX indicelibros ON libros (genero, titulo, editorial);`

1.6.1.4. 6.1.4 Estructura de los índices

- Oracle® usa **árboles B* balanceados** para igualar el tiempo de acceso a cualquier fila.

1.6.1.5. 6.1.5 Eliminación de índices

Se puede eliminar un índice por varias razones: ya no se necesita, no mejora la eficiencia, se necesita cambiar los campos indexados o rehacer un índice fragmentado.

- **Sintaxis:** `sql DROP INDEX nombre_del_indice;`

1.6.1.6. 6.1.6 Creación y uso de otros tipos de Índices

- **Índices por clave invertida:**
  - Mejoran el rendimiento en configuraciones paralelizadas de Oracle® (RAC).
  - **Sintaxis:** Se usa la cláusula `REVERSE`. `sql CREATE INDEX nombre_del_indice ON tabla (campo [ASC | DESC] ,...) REVERSE;`
- **Índices BITMAP:**
  - Apropiados cuando los valores de la clave tienen baja cardinalidad (pocos valores distintos).
  - **Ejemplo:** Crear un índice BITMAP para una columna con pocos valores distintos: `sql CREATE BITMAP INDEX indice_color ON pieza (color);`

1.6.2. 6.2 "Clusters"

Un **clúster** es un método alternativo para almacenar información de tablas.

- Está formado por un conjunto de tablas que se almacenan en los mismos bloques de datos porque comparten campos comunes (la **clave del clúster**) y se accede a ellas frecuentemente de forma conjunta (reunidas).
- **Beneficios:**
  - Reduce el acceso a disco y mejora el rendimiento en reuniones (`JOINs`) de las tablas del clúster.

1.6.2.1. 6.2.1 Selección de tablas y clave del "cluster"

- Usar para tablas principalmente consultadas y escasamente modificadas, donde las consultas reúnan las tablas del clúster.
- **Clave del clúster:** Elegir con cuidado. Debe tener un número adecuado de valores distintos para optimizar el rendimiento.

1.6.2.2. 6.2.2 Creación de un "cluster" indizado

Se emplea la sentencia `CREATE CLUSTER`.

- **Sintaxis básica:** `sql CREATE CLUSTER nombre_del_cluster(campo tipo_de_dato);`
- **Ejemplo:** Para el clúster de `proveedor` y `ventas` por `codpro`: `sql CREATE CLUSTER cluster_codpro (codpro VARCHAR2(3));`

1.6.2.3. 6.2.3 Creación de las tablas del "cluster" indizado

Una vez creado el clúster, se crean las tablas que pertenecerán a él usando `CREATE TABLE` con la opción `CLUSTER`.

- **Ejemplo (tabla proveedor2 dentro de cluster_codpro):** `sql CREATE TABLE proveedor2( codpro VARCHAR2(3) PRIMARY KEY, nompro VARCHAR2(30) NOT NULL, status NUMBER (2) CHECK (status>=1 AND status<=10), ciudad VARCHAR2(15) ) CLUSTER cluster_codpro (codpro);`

1.6.2.4. 6.2.4 Creación del índice del "cluster"

Antes de insertar información en las tablas del clúster, es necesario crear un índice para el clúster.

- **Sintaxis:** `sql CREATE INDEX indice_cluster ON CLUSTER nombre_del_cluster;`
- **Ejemplo:** `sql CREATE INDEX indice_cluster ON CLUSTER cluster_codpro;`

1.6.2.5. 6.2.5 Creación de un "cluster hash"

Se usa `CREATE CLUSTER` con las cláusulas `HASH IS`, `SIZE` y `HASHKEYS`.

- **Sintaxis:** `sql CREATE CLUSTER nombre_del_cluster(campo tipo_de_dato) SIZE <tamaño> HASHKEYS <cant_valores_clave>;`
- **Ejemplo:** Crear un clúster hash para `proveedor` y `ventas`: `sql CREATE CLUSTER cluster_codpro_hash(codpro VARCHAR2(3)) SIZE 610 HASHKEYS 50;`

1.6.2.6. 6.2.6 Creación de un "cluster hash" de una sola tabla

Para acceso hash a una sola tabla, se crea un clúster hash con la cláusula `SINGLE TABLE`.

- **Ejemplo:** Crear un clúster hash para la tabla `proveedor_hash`: "'sql CREATE CLUSTER cluster_codpro_single_hash (codpro VARCHAR2(3)) SIZE 50 SINGLE TABLE HASHKEYS 100;

  CREATE TABLE proveedor_hash(codpro VARCHAR2(3) PRIMARY KEY, nompro VARCHAR2(30) NOT NULL, status NUMBER (2) CHECK (status>=1 AND status<=10), ciudad VARCHAR2(15)) CLUSTER cluster_codpro_single_hash(codpro); "'

1.6.2.7. 6.2.7 Eliminación de "clusters"

Se usa `DROP CLUSTER`.

- **Sintaxis:** `sql DROP CLUSTER nombre_del_cluster [INCLUDING TABLES [CASCADE CONSTRAINTS]];`

1.6.2.8. 6.2.8 Eliminación de tablas del "cluster"

Se usa `DROP TABLE`. No afecta al clúster, a otras tablas del clúster ni al índice del clúster.

1.6.2.9. 6.2.9 Eliminación del índice del "cluster"

Se usa `DROP INDEX`. No afecta al clúster ni a sus tablas, pero no se podrá acceder a la información de las tablas hasta que se reconstruya el índice.

Este capítulo proporciona herramientas avanzadas para optimizar el almacenamiento y acceso a datos en una base de datos.

1.7. Comandos Usados

1.7.1. DESCRIBE

Muestra la estructura de una tabla, incluyendo columnas, tipos de datos y restricciones.

1.7.2. CREATE

Permite crear nuevos objetos en la base de datos, como tablas, vistas o índices.

1.7.3. CREATE TABLE

Define una nueva tabla con sus columnas, tipos de datos y restricciones.

1.7.4. ALTER TABLE

Modifica la estructura de una tabla existente, como agregar o eliminar columnas.

1.7.5. ADD

Agrega nuevas columnas o atributos a una tabla.

1.7.6. ADD CONSTRAINT

Define restricciones como claves primarias, únicas o foráneas en una tabla.

1.7.7. DROP CONSTRAINT

Elimina una restricción previamente definida en una tabla.

1.7.8. SELECT

Recupera datos de una o varias tablas según criterios específicos.

1.7.9. INSERT INTO

Inserta nuevas filas en una tabla.

1.7.10. UPDATE

Modifica los valores de las filas existentes en una tabla.

1.7.11. DELETE (o DELETE FROM)

Elimina filas de una tabla según una condición.

1.7.12. COMMIT

Confirma los cambios realizados en la base de datos de forma permanente.

1.7.13. UNION

Combina los resultados de dos consultas, eliminando duplicados.

1.7.14. UNION ALL

Combina los resultados de dos consultas, incluyendo duplicados.

1.7.15. INTERSECT

Devuelve las filas comunes entre dos consultas.

1.7.16. MINUS

Devuelve las filas que están en la primera consulta pero no en la segunda.

1.7.17. JOIN (incluyendo NATURAL JOIN y JOIN … ON)

Combina filas de dos tablas basándose en una condición. Incluye variantes como `NATURAL JOIN` y `JOIN ... ON`.

1.7.18. ORDER BY (usado como cláusula de SELECT)

Ordena los resultados de una consulta según una o varias columnas.

1.7.19. IN (usado como operador en WHERE)

Comprueba si un valor pertenece a un conjunto especificado en una subconsulta.

1.7.20. EXISTS (usado como operador en WHERE)

Devuelve verdadero si una subconsulta tiene al menos una fila.

1.7.21. ANY (usado como operador en WHERE)

Compara un valor con cualquier elemento de un conjunto.

1.7.22. ALL (usado como operador en WHERE)

Compara un valor con todos los elementos de un conjunto.

1.7.23. NOT EXISTS (usado como operador en WHERE)

Devuelve verdadero si una subconsulta no tiene filas.

1.7.24. SUM (función de agregación)

Calcula la suma de los valores de una columna.

1.7.25. MIN (función de agregación)

Encuentra el valor mínimo en una columna.

1.7.26. MAX (función de agregación)

Encuentra el valor máximo en una columna.

1.7.27. AVG (función de agregación)

Calcula el promedio de los valores de una columna.

1.7.28. COUNT (función de agregación)

Cuenta el número de filas en una consulta.

1.7.29. GROUP BY (usado como cláusula de SELECT)

Agrupa filas según los valores de uno o más atributos.

1.7.30. HAVING (usado como cláusula de SELECT)

Filtra los grupos creados por `GROUP BY` según una condición.

1.7.31. TO_CHAR (función)

Convierte valores como fechas o números a cadenas con un formato específico.

1.7.32. TO_DATE (función)

Convierte cadenas a valores de tipo fecha según un formato específico.

1.7.33. CASE (expresión condicional)

Evalúa condiciones y devuelve valores según el resultado de las mismas.

1.7.34. JOIN (incluyendo LEFT JOIN, usado en la cláusula FROM)

Devuelve todas las filas de la tabla izquierda y las filas coincidentes de la tabla derecha.

1.7.35. CREATE VIEW

Crea una vista basada en una consulta SELECT.

1.7.36. SELECT

Recupera datos de una vista como si fuera una tabla.

1.7.37. INSERT INTO

Inserta filas en una vista (si es actualizable).

1.7.38. DROP VIEW

Elimina una vista existente.

1.7.39. CREATE OR REPLACE VIEW

Crea una vista o reemplaza su definición si ya existe.

1.7.40. GRANT

Concede privilegios de acceso a usuarios sobre objetos (tablas, vistas, etc.).

1.7.41. REVOKE

Revoca privilegios previamente concedidos.

1.7.42. CREATE TABLE

Crea una nueva tabla en la base de datos.

1.7.43. CREATE INDEX

Crea un índice para acelerar búsquedas en una tabla.

1.7.44. DROP INDEX

Elimina un índice existente.

1.7.45. CREATE CLUSTER

Crea un clúster para almacenar varias tablas juntas según una clave común.

1.7.46. DROP CLUSTER

Elimina un clúster de la base de datos.

1.7.47. DROP TABLE

Elimina una tabla de la base de datos.

1.8. Resumen de Fundamentos de Bases de Datos (SQL)

1.8.1. Capítulo 1: Definición del Esquema (DDL - Lenguaje de Definición de Datos)

Sirve para crear y modificar la estructura de tu base de datos.

- **Comandos Principales:**
  - `CREATE TABLE nombre_tabla (...);`: Define una nueva tabla.
    - Dentro de los paréntesis van las **columnas**: `nombre_columna TIPO_DE_DATO [RESTRICCIONES]`
    - **Tipos de Datos Comunes:**
      - `INT` / `INTEGER` / `NUMBER(p,s)`: Números.
      - `CHAR(n)`: Texto de longitud fija.

- ◇ `VARCHAR(n)` / `VARCHAR2(n)`: Texto de longitud variable (¡VARCHAR2 es común en Oracle!).
- ◇ `DATE` / `TIMESTAMP`: Fechas y horas.
- ◇ `BLOB`: Datos binarios grandes.

- ◦ **Restricciones (Constraints):**
  - ◇ `NOT NULL`: El campo no puede estar vacío.
  - ◇ `PRIMARY KEY`: Identificador único de la fila (implica `NOT NULL` y `UNIQUE`).
  - ◇ `UNIQUE`: El valor debe ser único en esa columna.
  - ◇ `FOREIGN KEY (col_local) REFERENCES tabla_externa(col_externa)`: Asegura la integridad referencial (relaciona tablas).
  - ◇ `CHECK (condición)`: El valor debe cumplir una condición (ej. `status >= 1 AND status <= 10`).

- `ALTER TABLE nombre_tabla ...;`: Modifica una tabla existente.
  - ◦ `ADD nombre_columna TIPO_DE_DATO [RESTRICCION];`: Añade una columna.
  - ◦ `ADD CONSTRAINT nombre_restriccion TIPO_RESTRICCION (...);`: Añade una restricción.
  - ◦ `DROP CONSTRAINT nombre_restriccion;`: Elimina una restricción.
- `DESCRIBE nombre_tabla;`: Muestra la estructura de una tabla.

- **Consulta de Tablas Existentes:**
  - `SELECT table_name FROM user_tables;` (Tablas del usuario actual).

1.8.2. Capítulo 2: Mantenimiento de Datos (DML - Lenguaje de Manipulación de Datos)

Para añadir, modificar y borrar datos de las tablas.

- **Comandos Principales:**
  - `INSERT INTO nombre_tabla [(col1, col2, ...)] VALUES (val1, val2, ...);`: Inserta nuevas filas (tuplas).
    - ◦ Se puede omitir la lista de columnas si se dan valores para todas en el orden correcto.
    - ◦ También: `INSERT INTO tabla1 SELECT ... FROM tabla2;` (copiar datos).
  - `UPDATE nombre_tabla SET col1 = val1 [, col2 = val2, ...] [WHERE condición];`: Modifica filas existentes.
    - ◦ **¡CUIDADO!** Sin `WHERE`, actualiza TODAS las filas.
  - `DELETE FROM nombre_tabla [WHERE condición];`: Borra filas.
    - ◦ **¡CUIDADO!** Sin `WHERE`, borra TODAS las filas.
- **Manejo de Fechas:**
  - Tipo `DATE`: Almacena fecha y hora.
  - `SYSDATE`: Fecha y hora actual del sistema. Se puede operar con él (ej. `SYSDATE + 1`).

- `TO_DATE('cadena_fecha', 'formato')`: Convierte una cadena a fecha (ej. `TO_DATE('22/10/2025', 'DD/MM/YYYY')`).
- `TO_CHAR(campo_fecha, 'formato')`: Convierte una fecha a cadena con un formato específico.

- **Transacciones:**
  - `COMMIT;`: Guarda permanentemente los cambios hechos por DML.

1.8.3. Capítulo 3: Realización de Consultas (DQL - Lenguaje de Consulta de Datos con `SELECT`)

El comando más potente y versátil para recuperar información.

- **Estructura Básica y Cláusulas Principales (en orden de escritura general):**
  1. `SELECT [DISTINCT] col1_o_expresion [AS alias1], col2_o_expresion [AS alias2], ... | *`: Especifica qué columnas mostrar.
     - `DISTINCT`: Elimina filas duplicadas del resultado.
     - `*`: Todas las columnas.
     - `AS alias`: Renombra una columna en el resultado.
  2. `FROM tabla1 [alias1] [JOIN tabla2 [alias2] ON condicion_join | NATURAL JOIN tabla2 | , tabla2] ...`: Especifica de qué tabla(s) obtener los datos.
     - **Producto Cartesiano (implícito):** `FROM tabla1, tabla2` (generalmente se filtra con `WHERE`).
     - **Reuniones (Joins):**
       - `INNER JOIN` (o solo `JOIN`) … `ON tabla1.col = tabla2.col`: Filas que coinciden en ambas tablas según la condición.
       - `NATURAL JOIN`: Une por columnas con el mismo nombre (¡cuidado con nombres ambiguos!).
       - (También existen `LEFT/RIGHT/FULL OUTER JOIN` no detallados aquí, pero importantes).
     - **Alias de tabla:** `FROM proveedor p` (útil para abreviar y en auto-joins).
  3. `WHERE condición_filtrado_filas;`: Filtra las filas ANTES de cualquier agrupación.
     - **Operadores Comunes:** `=`, `!=` o `<>`, `>`, `<`, `>=`, `<=`, `AND`, `OR`, `NOT`.
     - `BETWEEN valor1 AND valor2`: Rango (inclusivo).
     - `LIKE 'patrón'` (con comodines `%` para cualquier secuencia, `_` para un carácter).
     - `IS NULL` / `IS NOT NULL`: Para valores nulos.
     - `IN (valor1, valor2, ...)`: Pertenencia a un conjunto de valores.
  4. `GROUP BY col1, col2, ...;`: Agrupa filas con los mismos valores en las columnas especificadas para aplicar funciones de agregación.
  5. `HAVING condición_filtrado_grupos;`: Filtra los grupos creados por `GROUP BY` (similar a `WHERE` pero para grupos y usa funciones de agregación).

6. `ORDER BY col1 [ASC | DESC], col2 [ASC | DESC], ...;`: Ordena el resultado final.
   - `ASC`: Ascendente (por defecto).
   - `DESC`: Descendente.

- **Operadores de Conjuntos (unen resultados de dos o más `SELECT`):**
  - `UNION`: Une resultados eliminando duplicados.
  - `UNION ALL`: Une resultados incluyendo duplicados.
  - `INTERSECT`: Filas comunes a ambos resultados.
  - `MINUS` (o `EXCEPT` en algunos SGBD): Filas del primer resultado que no están en el segundo.
  - **Requisito:** Las consultas deben tener el mismo número de columnas y tipos de datos compatibles.
- **Subconsultas (Consultas Anidadas):** Un `SELECT` dentro de otro.
  - **En `WHERE` / `HAVING`:**
    - Con `IN`: `WHERE columna IN (SELECT otra_columna FROM ...)`
    - Con `EXISTS`: `WHERE EXISTS (SELECT * FROM ... WHERE condición_correlacionada)` (devuelve true si la subconsulta tiene filas).
    - Con `NOT EXISTS`: Lo contrario.
    - Con comparadores (`=`, `>`, `<`, etc.):
      - Si la subconsulta devuelve un valor único: `WHERE columna = (SELECT valor_unico FROM ...)`
      - Con `ANY` / `SOME`: `WHERE columna > ANY (SELECT valores FROM ...)` (mayor que alguno).
      - Con `ALL`: `WHERE columna > ALL (SELECT valores FROM ...)` (mayor que todos).
  - **En `FROM`:** La subconsulta actúa como una tabla virtual (debe tener alias). `FROM (SELECT ... ) AS tabla_derivada`.
  - **En `SELECT`:** Si devuelve un valor escalar.
- **Funciones de Agregación (usadas con `SELECT` o `HAVING`, a menudo con `GROUP BY`):**
  - `COUNT(*)`: Número de filas.
  - `COUNT(columna)`: Número de valores no nulos en la columna.
  - `COUNT(DISTINCT columna)`: Número de valores distintos no nulos.
  - `SUM(columna)`: Suma de los valores.
  - `AVG(columna)`: Media de los valores.
  - `MAX(columna)`: Valor máximo.
  - `MIN(columna)`: Valor mínimo.
- **División Relacional ("para todos"):** Consultas complejas que buscan entidades que se relacionan con *todos* los elementos de otro conjunto. No hay un operador directo. Se implementa con:

  - Doble `NOT EXISTS`.
  - `GROUP BY` y `COUNT` comparando con el total del conjunto requerido.
  - Usando `MINUS` para encontrar "lo que falta".
- **Consultas al Catálogo (Diccionario de Datos):** Metadatos sobre la base de datos.
  - `USER_TABLES`: Tablas del usuario.
  - `ALL_TABLES`: Tablas a las que el usuario tiene acceso.
  - `USER_OBJECTS`: Todos los objetos del usuario.
  - `ALL_INDEXES`: Índices.
  - `ALL_TAB_COLUMNS`: Columnas de las tablas.
  - `DICTIONARY`: Lista de vistas del catálogo.

Conceptos Clave Adicionales:

- **Álgebra Relacional:** Base teórica de SQL (Proyección π -> `SELECT lista_cols`, Selección σ -> `WHERE`, Reunión $\bowtie$ -> `JOIN`).
- **Nulos (`NULL`):** Representan valor desconocido o inaplicable. Se comparan con `IS NULL` o `IS NOT NULL`.
- **Funciones:**
  - Numéricas: `ROUND()`, `TRUNC()`, `FLOOR()`, `CEIL()`.
  - De cadena: `UPPER()`, `LOWER()`, `SUBSTR()`, `LENGTH()`.
  - De fecha: `TO_DATE()`, `TO_CHAR()`, `ADD_MONTHS()`, `MONTHS_BETWEEN()`.
- **Alias (`AS`):** Para renombrar columnas o tablas en el resultado, mejora la legibilidad y resuelve ambigüedades.

1.8.4. Capítulo 4: Definición del Nivel Externo (Vistas)

- **Vistas:** Tablas virtuales basadas en consultas SQL almacenadas. No guardan datos físicamente (salvo vistas materializadas), sino su definición en el catálogo. Los datos se reconstruyen al consultarlas.
  - **Vistas Materializadas:** Excepción que sí almacena datos, útil en entornos distribuidos.
- **Propósito:**
  - **Seguridad:** Ocultan información sensible y muestran solo lo relevante a cada usuario.
  - **Simplificación:** Permiten reutilizar consultas complejas bajo un nombre sencillo.
- **Comandos Clave:**
  - `CREATE VIEW nombre_vista AS SELECT ...;`
  - `CREATE OR REPLACE VIEW ...;`
  - `DROP VIEW nombre_vista;`
- **Actualización de Vistas:**

- Restricciones: No se pueden actualizar vistas con `GROUP BY`, agregaciones, `DISTINCT`, uniones de varias tablas, o si faltan columnas `NOT NULL`/`PRIMARY KEY`.
- Se pueden usar disparadores `INSTEAD OF` para definir cómo aplicar operaciones DML.
- La opción `WITH CHECK OPTION` asegura que las modificaciones cumplen la condición de la vista.

1.8.5. Capítulo 5: Introducción a la Administración (Catálogo y Privilegios)

- **Catálogo (Diccionario de Datos):**
  - Conjunto de tablas y vistas del sistema que almacenan metadatos sobre la estructura, objetos, usuarios y privilegios de la base de datos.
  - Se accede mediante vistas como `USER_TABLES`, `ALL_TABLES`, `USER_CONSTRAINTS`, `ALL_VIEWS`, etc.
- **Gestión de Privilegios:** Controla el acceso y operaciones de los usuarios.
  - **Comandos Clave:** `GRANT` (otorga permisos), `REVOKE` (retira permisos).
  - **Tipos de Privilegios:**
    - **Del Sistema:** Permiten acciones globales (ej. `CREATE TABLE`). Se pueden conceder con `WITH ADMIN OPTION`.
    - **Sobre Objetos:** Permiten operaciones sobre objetos concretos (ej. `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `EXECUTE`). Se pueden conceder con `WITH GRANT OPTION`. La opción `CASCADE CONSTRAINTS` en `REVOKE` propaga la retirada a restricciones referenciales.

1.8.6. Capítulo 6: Nivel Interno (Índices, Clústeres y Hashing)

- **Índices:** Estructuras que aceleran la recuperación de datos. El SGBD los usa automáticamente. Oracle crea índices para claves primarias.
  - **Comandos Clave:** `CREATE INDEX ... ON tabla (...);`, `DROP INDEX ...;`
  - **Tipos:** Compuestos (varias columnas), clave invertida (`REVERSE`), bitmap (`BITMAP`), y tablas organizadas por índice (IOT).
- **Clústeres:** Agrupan filas de distintas tablas que comparten una clave común y suelen consultarse juntas, optimizando los `JOIN`. Reducen el acceso a disco.
  - **Comandos Clave:** `CREATE CLUSTER ...`, `CREATE TABLE ... CLUSTER ...`, `CREATE INDEX ... ON CLUSTER ...`, `DROP CLUSTER ...`
  - **Clúster Hash:** Usa una función hash sobre la clave para ubicar los datos (`HASHKEYS`, `SIZE`). También existe el hash cluster de una sola tabla.
- **Hashing:** Se implementa principalmente mediante clústeres hash para optimizar el acceso físico a los datos.

1.9. Resumen Completo: Álgebra Relacional (Seminario 4)

1.9.1. 1. Introducción

- Un **lenguaje de consulta** permite al usuario solicitar información de la base de datos. Son normalmente de más alto nivel que los lenguajes de programación estándar .
- Se clasifican en:
  - **Procedimentales:** El usuario instruye al sistema para realizar una secuencia de operaciones para calcular el resultado (ej. Álgebra Relacional) .
    - El Álgebra Relacional fue introducida por E. F. Codd en 1970 .
  - **Declarativos:** El usuario describe la información deseada sin especificar el procedimiento para obtenerla (ej. Cálculo Relacional) .
    - El Cálculo Relacional también fue propuesto por Codd .
- Las operaciones del álgebra relacional son **internas**: la entrada es una o más relaciones y la salida es siempre una relación .

Operadores Principales y Notación SQL (general):

Operador | Notación AR | SQL (Idea General) |
: | : | :– |
Selección | σ | `WHERE <condición>` |
Proyección | π | `SELECT A1,..,An FROM ...` |
Producto Cartesiano | $\times$ | `FROM R1, ..., Rm` |
Unión | $\cup$ | `(SELECT ...) UNION (SELECT ...)` |
Diferencia | $-$ | `(SELECT ...) MINUS (SELECT ...)` |
Intersección | $\cap$ | `(SELECT ...) INTERSECT (SELECT ...)` |
θ-Reunión | $\bowtie_\Theta$ | `FROM R1 JOIN R2 ON <condición>` |
Reunión Natural | $\bowtie$ | `FROM R1 NATURAL JOIN R2` |
División | $\div$ | Compleja, a menudo con `NOT EXISTS` y `MINUS` o `COUNT` |

1.9.2. Clasificación de los Operadores

- **Por aridad:**
  - **Monarios** (una relación de entrada): Selección (σ), Proyección (π) .
  - **Binarios** (dos relaciones de entrada): Unión ($\cup$), Intersección ($\cap$), Diferencia ($-$), Producto Cartesiano ($\times$), θ-Reunión ($\bowtie_\Theta$), División ($\div$) .
- **Por relación con el modelo relacional:**
  - **Conjuntistas:** Unión, Diferencia, Intersección, Producto .
  - **Relacionales Específicos:** Selección, Proyección, Reunión, División .
- **Por necesidad (primitividad):**

- **Fundamentales (Primitivos):** Selección, Proyección, Unión, Diferencia, Producto Cartesiano .
- **No Fundamentales (Derivados):** Intersección, θ-Reunión, División (se pueden expresar en términos de los fundamentales) .

1.9.3. 2. Selección (σ)

- **Definición:** $\sigma_{\Theta}(R)$. Obtiene las tuplas de la relación R para las que la condición (propiedad) Θ es cierta .
- **Condición Θ:** Se forma usando atributos de R y operadores:
  - Relacionales: $=, \neq$ ($<>$), $<, >, \leq, \geq$.
  - Lógicos: $\wedge$ (AND), $\vee$ (OR), $\neg$ (NOT) .
  - Paréntesis `()` para modificar precedencia .
- **Ejemplo:** $\sigma_{\text{categoria='AS'}}(\text{profesores})$. Selecciona los profesores cuya categoría es 'AS' .
- **Ejemplo Compuesto:** $\sigma_{\text{categoria}\neq\text{'AS'}\wedge(\text{area='COMPUT'}\vee\text{area='ELECTR'})}(\text{profesores})$.

1.9.4. 3. Proyección (π)

- **Definición:** $\pi_{A_i,...,A_j}(R)$. Obtiene las tuplas de la relación R considerando solo el subconjunto de atributos $\{A_i, ..., A_j\}$, eliminando atributos no especificados y suprimiendo tuplas redundantes (duplicadas) en el resultado .
- **Eliminación de Duplicados:** Es crucial. Si la lista de atributos proyectados no incluye una clave candidata de la relación original, pueden aparecer tuplas duplicadas, las cuales deben ser eliminadas para que el resultado sea una relación válida .
- **Ejemplo:** $\pi_{\text{nrp, nom\_prof, categoria}}(\text{profesores})$. Proyecta NRP, nombre y categoría de los profesores .
- **Ejemplo con Duplicados:** Si se proyecta $\pi_{\text{area, cod\_dep}}(\text{profesores})$, y la clave primaria `nrp` no está, las tuplas duplicadas (area, cod_dep) se eliminan en el resultado final .

1.9.5. 4. Composición de Operadores

- El Álgebra Relacional se basa en la aplicación sucesiva de operadores hasta obtener la tabla solución .
- Como el resultado de una operación de álgebra relacional es siempre una relación, este resultado puede usarse como operando (entrada) para otra operación .
- **Ejemplo:** Obtener NRP y nombre de profesores del departamento 'ELEC' .
  1. Selección: $\sigma_{\text{cod\_dep='ELEC'}}(\text{profesores})$.
  2. Proyección sobre el resultado anterior: $\pi_{\text{nrp, nom\_prof}}(\sigma_{\text{cod\_dep='ELEC'}}(\text{profesores}))$.
- **Otros ejemplos:**
  - Nombres de profesores no 'AS' de áreas 'TSEÑAL' o 'ARQUIT': $\pi_{\text{nom\_prof}}(\sigma_{(\text{categoria}\neq\text{'AS'})\wedge(\text{area='TSEÑAL'}\vee\text{area='ARQUIT'})}(\text{profesores}))$

- DNI y nombre de alumnos nacidos antes del '2000-01-01': $\pi_{\text{dni, nomb\_alum}}(\sigma_{\text{fecha\_nac<date('2000-01-01')}}(\text{alumnos}))$

1.9.6. 5. Producto Cartesiano ($\times$)

- **Definición:** $R \times S$. Forma tuplas resultado concatenando cada tupla de la relación R con cada tupla de la relación S .
- **Esquema Resultante:** Si $W = R \times S$, entonces $esquema(W) = esquema(R) \cup esquema(S)$. (Los nombres de atributos deben ser únicos; se usan prefijos si hay colisión) .
- **Cardinalidad Resultante:** $card(W) = card(R) \times card(S)$.
- **Ambigüedad de Atributos:** Si las relaciones R y S tienen atributos con el mismo nombre, se necesita un mecanismo para desambiguar :
  - **Prefijos:** `NombreRelacion.NombreAtributo` (ej. `profesores.nrp`, `grupos.nrp`) .
  - **Operador de Redefinición/Alias (ρ):** $\rho_S(R)$ permite referirse a la relación R con el nombre (alias) S. Útil si una misma relación aparece varias veces en la consulta o para acortar nombres .
- **Ejemplo de uso (simulando una reunión):** Para obtener nombre de departamento y nombre de su director:
  1. Producto: profesores $\times$ departamentos .
  2. Selección (condición de reunión): $\sigma_{\text{departamentos.director=profesores.nrp}}(\text{profesores} \times \text{departamentos})$.
  3. Proyección: $\pi_{\text{profesores.nom\_prof, departamentos.nom\_dep}}(\sigma_{\text{departamentos.director=profesores.nrp}}(\text{profesores} \times \text{departamentos}))$.

1.9.7. 6. Unión ($\cup$) y Diferencia ($-$)

- **Compatibilidad para Unión/Diferencia/Intersección:** Las relaciones deben ser "union-compatibles", es decir, tener el mismo número de atributos y los tipos de datos de los atributos correspondientes deben ser compatibles .
- **Unión ($R \cup S$):**
  - Produce una relación que contiene todas las tuplas que están en R, en S, o en ambas, eliminando duplicados .
  - **Ejemplo:** Asignaturas de curso 4 ó 5: $\sigma_{\text{curso=4}}(\text{asignaturas}) \cup \sigma_{\text{curso=5}}(\text{asignaturas})$.
- **Diferencia ($R - S$):**
  - Produce una relación con todas las tuplas que están en R pero no están en S .
  - **Ejemplo:** Profesores 'TU' que no son del área 'COMPUT': $\sigma_{\text{categoria='TU'}}(\text{profesores}) - \sigma_{\text{area='COMPUT'}}(\text{profesores})$.
  - **Ejemplo:** Asignaturas sin alumnos matriculados: $\pi_{\text{cod\_asig}}(\text{asignaturas}) - \pi_{\text{cod\_asig}}(\text{matriculas})$.

1.9.8. 7. Reunión ($\bowtie$)

- **θ-Reunión ($R \bowtie_\Theta S$):**
  - Es una abreviatura para $\sigma_\Theta(R \times S)$.
  - Θ es una condición de comparación entre atributos de R y S .
  - **Ejemplo:** profesores $\bowtie_{\text{director=nrp}}$ departamentos (equivale al ejemplo de producto cartesiano + selección visto antes) .
- **Reunión Natural ($R \bowtie S$):**
  - Es un caso especial de equi-reunión .
  - Se realiza sobre todos los atributos que tienen el mismo nombre en R y S .
  - En el resultado, las columnas comunes aparecen solo una vez .
  - Formalmente: $\pi_{(\text{esquema(R)}\cup\text{esquema(S)})-\text{atributos\_comunes}}(\sigma_{\text{condiciones\_igualdad\_atributos\_comunes}}(R\times S))$ (la proyección elimina una copia de los atributos comunes) .
  - **Ejemplo:** Nombre de profesor y nombre de su departamento:

$$\pi_{\text{nrp, nom\_prof, nom\_dep}}(\text{departamentos} \bowtie \text{profesores}).$$

1.9.9. 8. Intersección ($\cap$)

- **Definición ($R \cap S$):**
  - Las relaciones R y S deben ser union-compatibles .
  - Produce una relación con las tuplas que están presentes tanto en R como en S .
  - **Ejemplo:** Alumnos becarios Y de Almería: $\sigma_{\text{beca=True}}(\text{alumnos})\cap\sigma_{\text{provincia='Almeria'}}(\text{alumnos})$.
- **Operador Derivado:** Se puede expresar usando la diferencia: $R \cap S = R - (R - S)$.