



JORGE LÓPEZ BLASCO

ÍNDICE

1. Módulo 1. Procesamiento lógico de consultas

- 1.1. Orden en el procesamiento lógico de consultas
- 1.2. Ejemplo de procesamiento lógico de consultas
- 1.3. Detalles de las fases en el procesamiento lógico

2. Módulo 2. Ajuste de consultas

- 2.1. Aspectos internos y ajuste de índices
- 2.2. Estimaciones de cardinalidad
- 2.3. Tablas temporales
- 2.4. Conjuntos vs. Cursos

ÍNDICE

3. Módulo 3. Consultas de varias tablas

- 3.1. Subconsultas, expresiones de tabla y consultas recursivas
- 3.2. Operador APPLY
- 3.3. Joins
- 3.4. Operadores de conjuntos

4. Módulo 4. Modificación de datos

- 4.1. Inserción de datos
- 4.2. Eliminación de datos
- 4.3. Actualización de datos
- 4.4. Combinación de datos
- 4.5. La cláusula OUTPUT

5. Módulo 5. Objetos programables

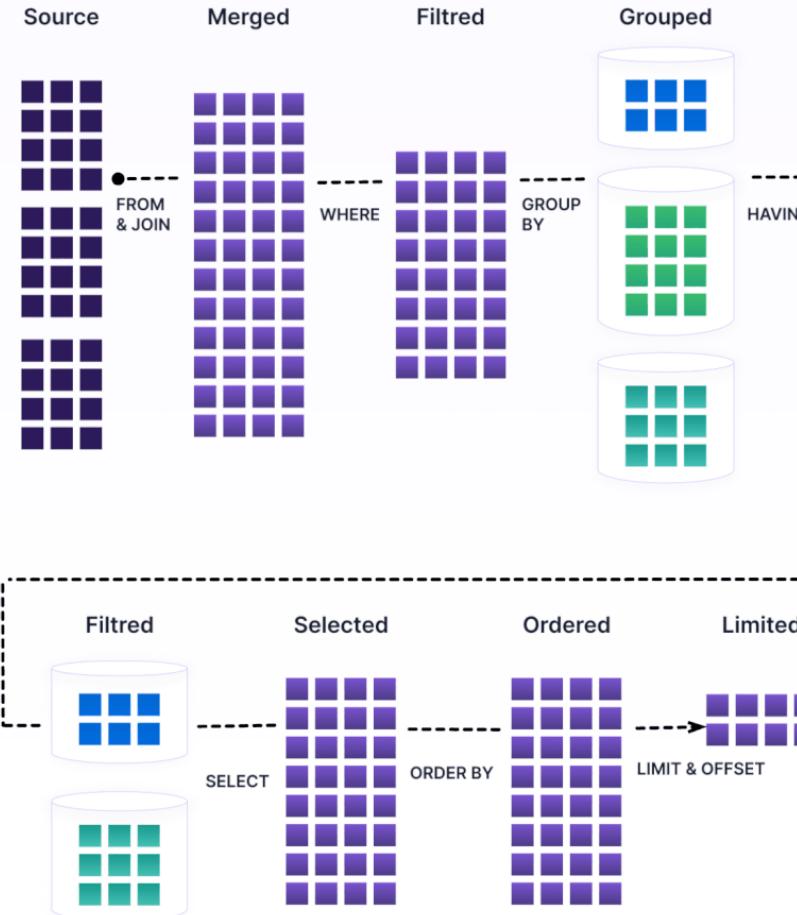
- 5.1. SQL dinámico
- 5.2. Funciones definidas por el usuario
- 5.3. Procedimientos almacenados
- 5.4. Triggers
- 5.5. Control de errores y excepciones

MÓDULO 1. PROCESAMIENTO LÓGICO DE CONSULTAS

ORDEN EN EL PROCESAMIENTO LÓGICO DE CONSULTAS

Cómo se ve la consulta	Como se ejecuta	Por qué funciona de esta manera
SELECT	▶ FROM	▶ SQL comienza con la tabla de la que su consulta está tomando datos
FROM	▶ WHERE	▶ Así es como SQL filtra en filas
WHERE	▶ GROUP BY	▶ Aquí es donde su consulta SQL verifica si tiene una agregación
GROUP BY	▶ HAVING	▶ HAVING requiere una declaración GROUP BY
HAVING	▶ SELECT	▶ Solo después de que se hayan realizado todos estos cálculos, SQL "SELECCIONARÁ" qué columnas desea que se devuelvan.
ORDER BY	▶ ORDER BY	▶ Esto ordena los datos devueltos
LIMIT	▶ LIMIT	▶ Por último, puede limitar el número de filas devueltas.

SQL QUERY EXECUTION ORDER





EJEMPLO

- -- #6+7 SELECT DISTINCT department_id
 - -- #1 FROM employees
 - -- #2 JOIN orders ON customers.customer_id =
orders.customer_id
 - -- #3 WHERE salary > 3000
 - -- #4 GROUP BY department
 - -- #5 HAVING AVG(salary) > 5000
 - -- #8 ORDER BY department
 - -- #9 LIMIT 10 OFFSET 5
 - -- #10 OFFSET 5 ROWS FETCH NEXT 10 ROWS ONLY;
-

ERRORES



Utilizar alias de columna en la cláusula WHERE: La cláusula WHERE se ejecuta antes que la cláusula SELECT, por lo que se producirá un error si utilizas un alias en la cláusula WHERE. Para evitar este error, utiliza siempre la expresión original en lugar de un alias en la cláusula WHERE.



Uso de alias de columna en la cláusula WHERE: Como la cláusula WHERE se ejecuta antes que la cláusula SELECT, intentar utilizar un alias en WHERE dará lugar a un error. Comprender que SQL evalúa WHERE antes de la cláusula SELECT te enseña que debes repetir la expresión completa en lugar de confiar en un alias.



Utilizar HAVING para filtrar filas en lugar de WHERE: La cláusula HAVING se ejecuta después de GROUP BY y está pensada para filtrar datos agregados. Si estás filtrando datos no agregados, pertenece a la cláusula WHERE. Conocer [la diferencia en el orden de ejecución entre WHERE y HAVING](#) te ayuda a determinar dónde debe colocarse cada condición.

ERRORES

- **Uso incorrecto de agregados en SELECT sin GROUP BY:** Puesto que GROUP BY se ejecuta antes que HAVING o SELECT, si no agrupas tus datos antes de aplicar una función de agregado, se producirán resultados incorrectos o errores. Comprender el orden de ejecución aclara por qué estas dos cláusulas deben ir juntas.
- **No utilizar correctamente los alias en la cláusula ORDER BY:** A diferencia de la cláusula WHERE, la cláusula ORDER BY se evalúa después de SELECT. Esto te permite utilizar los alias creados en SELECT para la clasificación, ayudándote a evitar confusiones al saber cuándo están disponibles los alias para su uso.

BUENAS PRÁCTICAS

Filtra antes con WHERE: Como la cláusula WHERE se ejecuta antes que GROUP BY y JOIN, aplicar los filtros antes reduce el número de filas procesadas por las cláusulas posteriores, lo que mejora el rendimiento de la consulta. Al filtrar los datos no agrupados lo antes posible, limitas los datos que hay que agrupar o unir, ahorrando tiempo de procesamiento.

Preagregar datos antes de unirlos: Sabiendo que FROM y JOIN son las primeras cláusulas que se ejecutan, preagrupar los datos mediante subconsultas o expresiones comunes de tabla (CTE) te permite reducir el conjunto de datos antes del proceso de unión. Esto garantiza que se procesen menos filas durante la unión.



BUENAS PRÁCTICAS

- **Optimizar ORDER BY con índices:** Puesto que la cláusula ORDER BY es uno de los últimos pasos ejecutados, asegurarse de que las columnas ordenadas están indexadas acelerará el rendimiento de la consulta al ayudar a la base de datos a gestionar las operaciones de ordenación con mayor eficacia.
- **Evita SELECT * en las consultas de producción:** La cláusula SELECT se ejecuta después de filtrar, agrupar y agregar, por lo que especificar sólo las columnas necesarias minimiza la cantidad de datos recuperados, reduciendo la sobrecarga innecesaria.

DONDE VAMOS A TRABAJAR

into the sheet ... →

The screenshot shows the Oracle Live SQL interface. The top navigation bar is red with the text "ORACLE® Live SQL". Below it is a toolbar with icons for Home, SQL Worksheet, Session search, and other session-related functions. The main area is titled "SQL Worksheet". On the left, there's a sidebar with icons for Home, SQL Worksheet, Session search, and other session-related functions. The central workspace contains two parts: a code editor on the left and a results grid on the right. The code editor displays the following SQL statements:

```
56  7566, 'JONES', 'MANAGER', 7839,
57  to_date('2-4-1981','dd-mm-yyyy'),
58  2975, null, 20
59 )
60
61 insert into emp
62 values(
63  7788, 'SCOTT', 'ANALYST', 7566,
64  to_date('13-JUL-87','dd-mm-rr') - 85,
65  3000, null, 20
66 )
67
68 insert into emp
69 values(
70  7902, 'FORD', 'ANALYST', 7566,
71  to_date('3-12-1981','dd-mm-yyyy'),
72  3000, null, 20
73 )
```

The results grid on the right shows the following data:

ENAME	DNAME	JOB	EMPNO	HIREDATE	LOC
ADAMS	RESEARCH	CLERK	7876	23-MAY-87	DALLAS
ALLEN	SALES	SALESMAN	7499	20-FEB-81	CHICAGO
BLAKE	SALES	MANAGER	7698	01-MAY-81	CHICAGO
CLARK	ACCOUNTING	MANAGER	7782	09-JUN-81	NEW YORK
FORD	RESEARCH	ANALYST	7902	03-DEC-81	DALLAS
JAMES	SALES	CLERK	7900	03-DEC-81	CHICAGO
JONES	RESEARCH	MANAGER	7566	02-APR-81	DALLAS
KING	ACCOUNTING	PRESIDENT	7839	17-NOV-81	NEW YORK

DETALLES DE LAS FASES EN EL PROCESAMIENTO LÓGICO DE CONSULTAS



FASE DE ANÁLISIS Y VALIDACIÓN

- **Análisis sintáctico**

- El sistema verifica que la consulta esté **bien escrita** según las reglas del lenguaje SQL.

Se comprueba la estructura: si después de SELECT vienen columnas, si la cláusula WHERE tiene expresiones válidas, etc.

Si algo está mal escrito, aquí es donde se producen errores como:

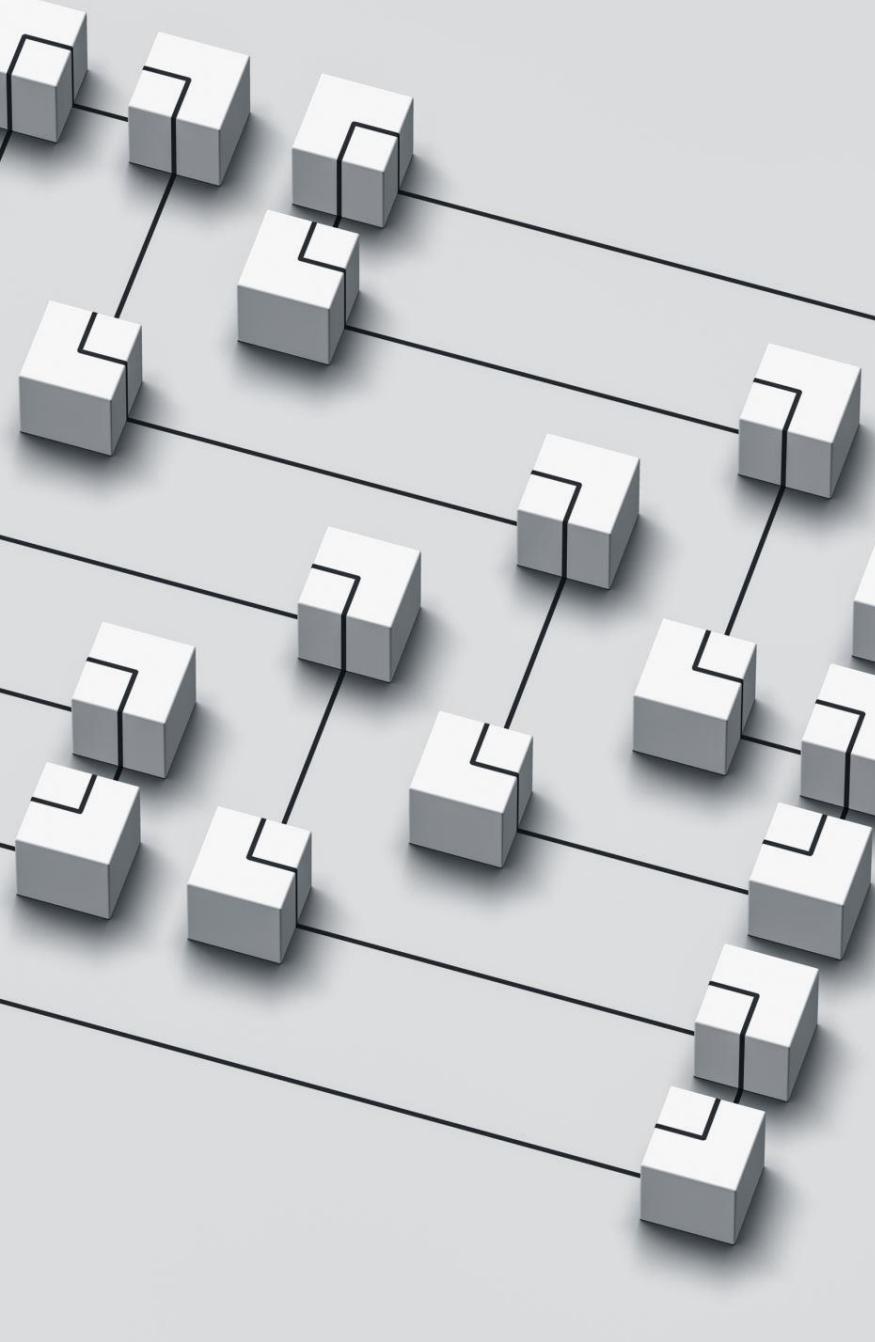
- “*Error de sintaxis cerca de...*”
- “*Falta palabra clave...*”

- **Análisis semántico**

- Aquí la base de datos verifica si lo que pedimos **tiene sentido** en su contenido. Por ejemplo:
 - ¿Las tablas existen?
 - ¿Las columnas están dentro de las tablas que mencionamos?
 - ¿El tipo de dato permite la operación que solicitamos?
 - ¿El usuario tiene permisos para acceder a esa información?

FASE DE REESCRITURA DE LA CONSULTA

- **Eliminación de redundancias**
 - Por ejemplo:
 - Quitar condiciones duplicadas.
 - Simplificar expresiones complejas.
 - Eliminar subconsultas innecesarias.
- **Expansión de vistas:** Si el usuario consulta una **vista**, la base de datos sustituye automáticamente esa vista por la consulta real que la define.
- **Aplicación de reglas algebraicas**
 - El sistema aplica leyes del álgebra relacional para optimizar la estructura lógica, como:
 - mover los filtros lo más cerca posible de las tablas,
 - descomponer condiciones complejas,
 - reorganizar operaciones de unión y selección.
- **Reescritura basada en integridad referencial:** Algunos sistemas usan llaves foráneas, unicidad y otras reglas para optimizar aún más la consulta.



FASE DE OPTIMIZACIÓN LÓGICA

- **Orden de las operaciones**
 - Por ejemplo, si tenemos varias tablas en un JOIN, ¿en qué orden conviene unirlas?
- **Selección de métodos de acceso a datos**
 - Puede elegir entre:
 - un escaneo completo de la tabla (*full table scan*),
 - usar un índice,
 - usar índices compuestos,
 - realizar particionamiento.

FASE DE OPTIMIZACIÓN LÓGICA

- **Elección de algoritmos**

- Como:
- Hash Join,
- Nested Loop Join,
- Merge Join,
- ordenamiento interno versus ordenamiento externo.

- **Estimación de costos**

- El optimizador trabaja con un **modelo de costos**, donde analiza:
- número de filas,
- distribución de valores,
- cardinalidades,
- estimaciones estadísticas,
- tamaño de las tablas,
- selectividad de los filtros.

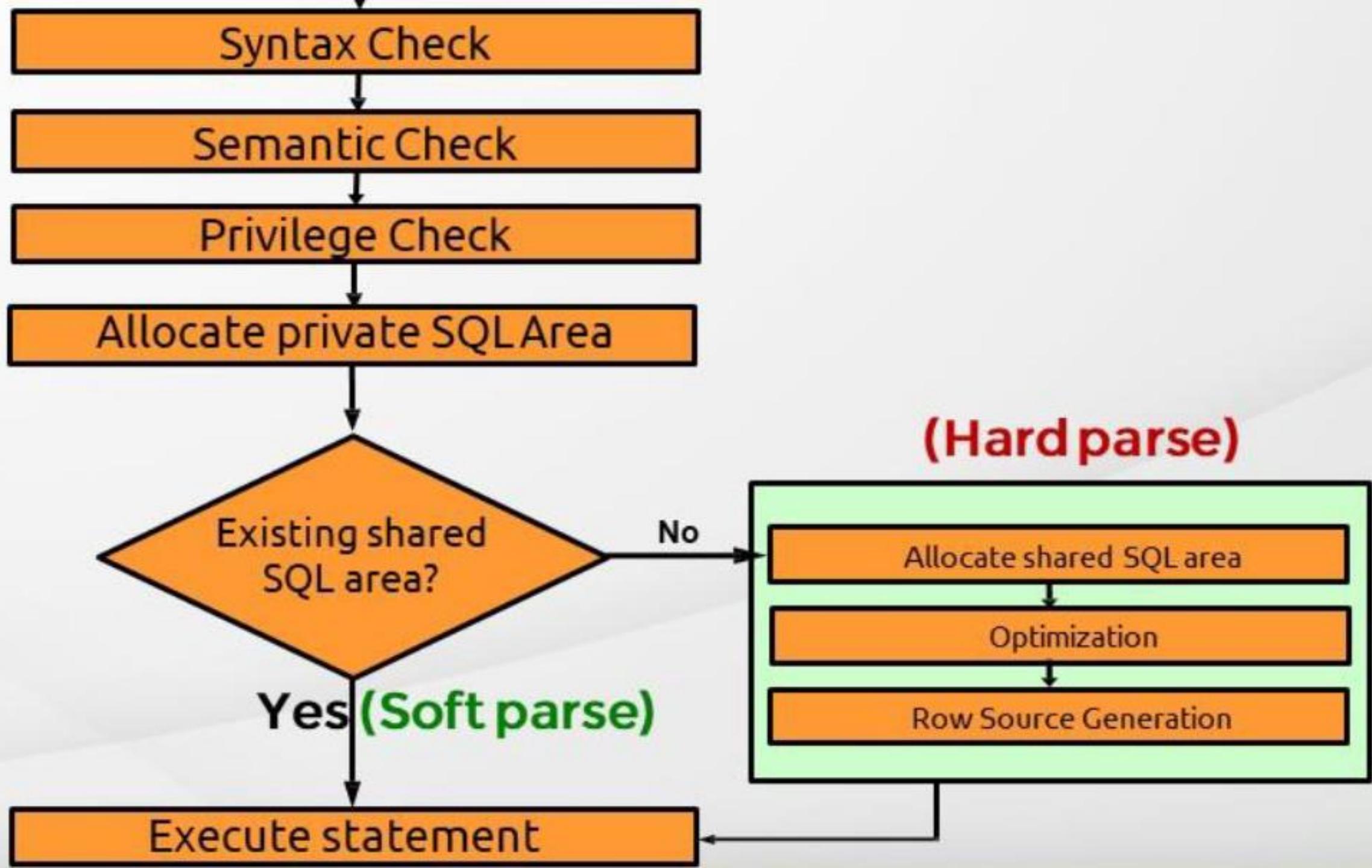
GENERACIÓN DEL PLAN DE EJECUCIÓN

Este plan es una secuencia de pasos que el motor seguirá al momento de ejecutar la consulta, indicando:

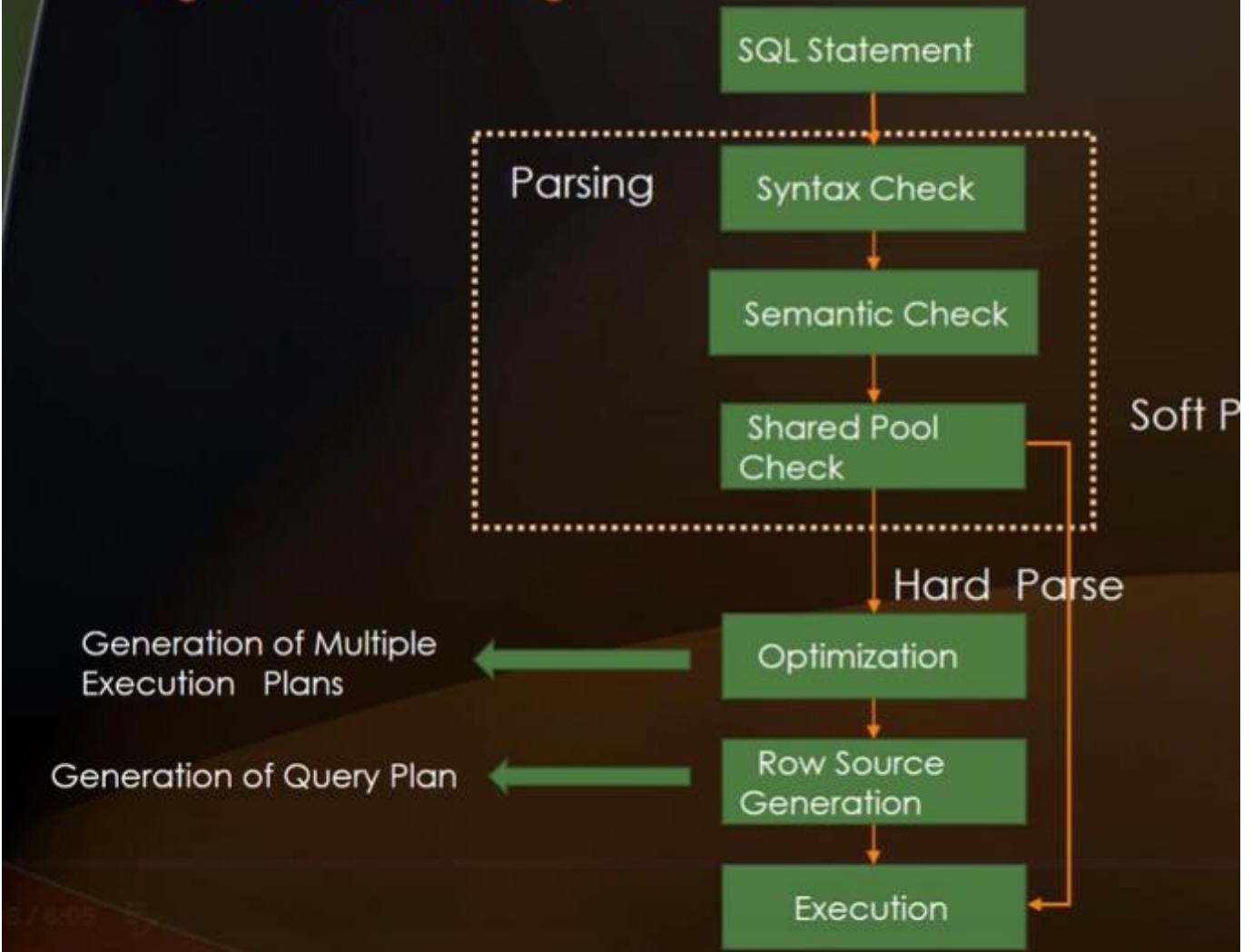
- qué índices utilizar,
- en qué orden acceder a las tablas,
- qué algoritmos aplicar para combinar datos,
- cómo recuperar, ordenar y presentar la información.

En esta etapa se produce lo que conocemos como **execution plan**, que puede visualizarse con comandos como:

- *EXPLAIN*,
 - *EXPLAIN ANALYZE*,
 - *DESCRIBE*,
- dependiendo del sistema gestor de bases de datos.



Stages of SQL Processing



```
SELECT * FROM products WHERE prod_category = 'Electronics';
```

Check Schema Information

Find Possible Access Paths

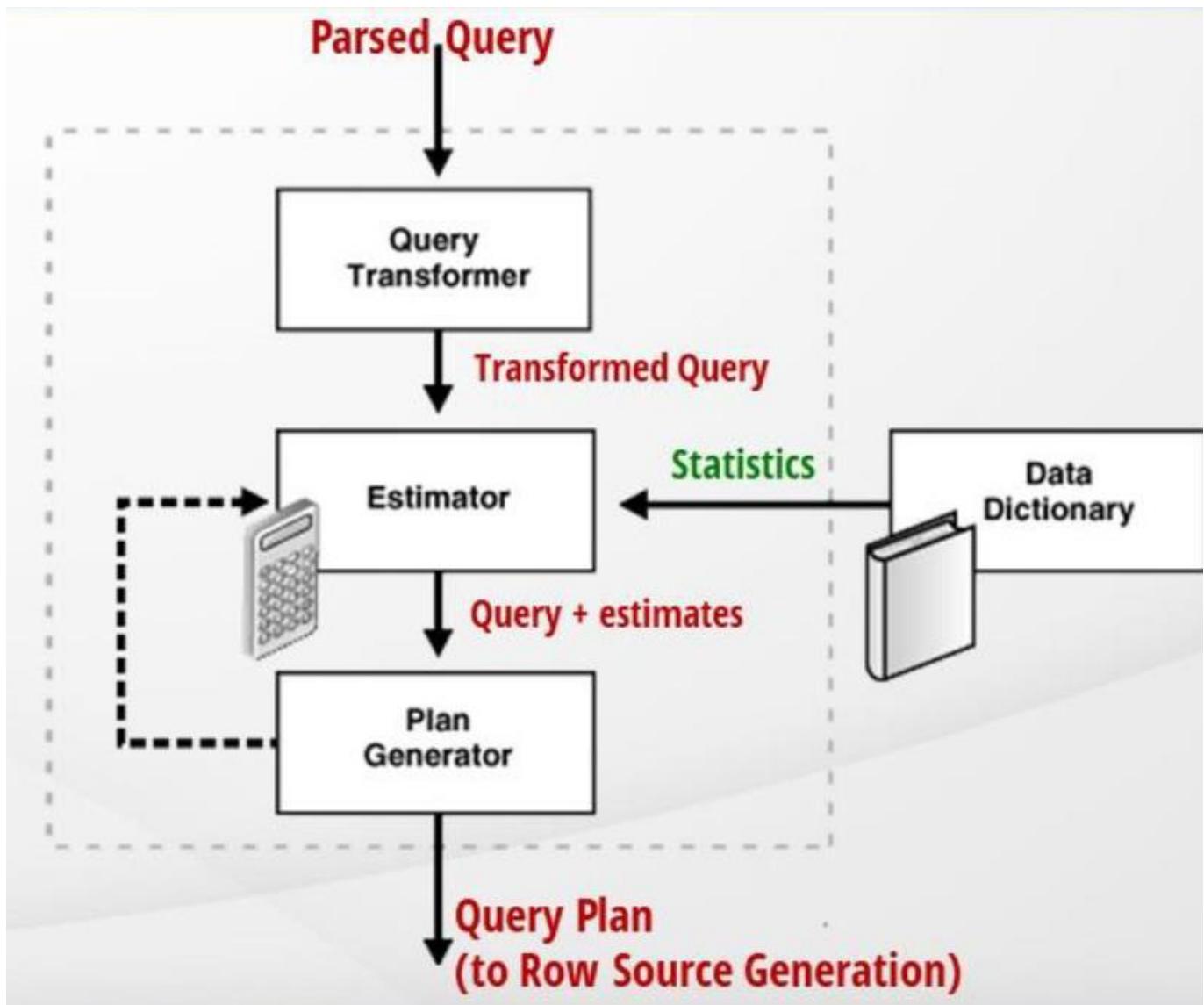
Use Index

Read Whole Table

Check Statistics

Result is 1% of the whole table.

Result is 25% of the whole table.



```
SELECT e.first_name, e.last_name, e.department_name FROM employees e, departments d  
WHERE e.department_id = d.department_id;
```

```
Join order[1]: DEPARTMENTS[D]#0 EMPLOYEES[E]#1  
NL Join Cost: 42.25  
SM Join cost: 8.48  
HA Join cost: 5.20  
Best:: JoinMethod: Hash  
Cost: 5.2  
Join order[2]: EMPLOYEES[E]#1 DEPARTMENTS[D]#0  
NL Join Join: 88.28  
SM Join cost: 7.57  
HA Join cost: 5.50  
Join order aborted  
Final cost for query block SEL$1 (#0)  
All Rows Plan:
```

Best join order: 1

Id	Operation	Name	Rows	Bytes	Cost
0	SELECT STATEMENT				5
1	HASH JOIN		106	6042	5
2	TABLE ACCESS FULL	DEPARTMENTS	27	810	3
3	TABLE ACCESS FULL	EMPLOYEES	107	2889	3

```

SELECT prod_name, time_id, max(amount_sold) FROM sales, products
WHERE sales.prod_id = products.prod_id AND promo_id = 33
GROUP BY prod_name, time_id;

```

OPERATION	OBJECT_NAME	CARDINALITY	COST	PARTITION_START	PARTITION_STOP
SELECT STATEMENT			2074	290	
HASH (GROUP BY)			2074	290	
HASH JOIN			2074	289	
Access Predicates					
SALES.PROD_ID=PRODUCTS.PROD_ID					
NESTED LOOPS			2074	289	
NESTED LOOPS					
STATISTICS COLLECTOR					
TABLE ACCESS (FULL)	PRODUCTS	72	3		
PARTITION RANGE (ALL)				1	28
BITMAP CONVERSION (TO ROWIDS)					
BITMAP AND					
BITMAP INDEX (SINGLE VALUE)	SALES_PROMO_BIX			1	28
Access Predicates					
PROMO_ID=33					
BITMAP INDEX (SINGLE VALUE)	SALES_PROD_BIX			1	28
Access Predicates					
SALES.PROD_ID=PRODUCTS.					
TABLE ACCESS (BY LOCAL INDEX ROWID)	SALES	29	286	1	1
PARTITION RANGE (ALL)			2074	286	28
TABLE ACCESS (BY LOCAL INDEX ROWID BATCHED)	SALES	2074	286	1	28
BITMAP CONVERSION (TO ROWIDS)					
BITMAP INDEX (SINGLE VALUE)	SALES_PROMO_BIX			1	28
Access Predicates					
PROMO_ID=33					

OPERATION	OBJECT_NAME	CARDINALITY	COST	PARTITION_START	PARTITION_STOP
SELECT STATEMENT			2074	290	
HASH (GROUP BY)			2074	290	
HASH JOIN			2074	289	
Access Predicates					
SALES.PROD_ID=PRODUCTS.PROD_ID					
NESTED LOOPS		2074	289		
NESTED LOOPS					
STATISTICS COLLECTOR					
TABLE ACCESS (FULL)	PRODUCTS	72	3		
PARTITION RANGE (ALL)				1	28
BITMAP CONVERSION (TO ROWIDS)					
BITMAP AND					
BITMAP INDEX (SINGLE VALUE)	SALES_PROMO_BIX			1	28
Access Predicates					
PROMO_ID=33					
BITMAP INDEX (SINGLE VALUE)	SALES_PROD_BIX			1	28
Access Predicates					
SALES.PROD_ID=PRODUCTS.					
TABLE ACCESS (BY LOCAL INDEX ROWID)	SALES	29	286	1	1
PARTITION RANGE (ALL)		2074	286	1	28
TABLE ACCESS (BY LOCAL INDEX ROWID BATCHED)	SALES	2074	286	1	28
BITMAP CONVERSION (TO ROWIDS)					
BITMAP INDEX (SINGLE VALUE)	SALES_PROMO_BIX			1	28
Access Predicates					
PROMO_ID=33					

EXECUTION PLAN



LEER LOS PLANES DE EJECUCION

▪ Where to look?

- Cost
- Access Methods
- Cardinality
- Join Methods&Join Types
- Partition Pruning
- Others

```
SELECT p.prod_id,p.prod_name, s.amount_sold, s.quantity_sold
FROM sales s, products p, customers c
WHERE s.prod_id = p.prod_id
AND s.cust_id = c.cust_id
AND s.cust_id IN (2,3,4,5);
```

Id Operation	Name	Rows	Bytes	Cost (%CPU)	Time	Pstart	Pstop
0 SELECT STATEMENT		521	27092	211 (0)	00:00:01		
1 NESTED LOOPS		521	27092	211 (0)	00:00:01		
* 2 HASH JOIN		521	24487	211 (0)	00:00:01		
3 TABLE ACCESS FULL	PRODUCTS	72	2160	3 (0)	00:00:01		
4 PARTITION RANGE ALL		521	8857	208 (0)	00:00:01	1	28
5 INLIST ITERATOR							
6 TABLE ACCESS BY LOCAL INDEX ROWID BATCHED	SALES	521	8857	208 (0)	00:00:01	1	28
7 BITMAP CONVERSION TO ROWIDS							
* 8 BITMAP INDEX SINGLE VALUE	SALES CUST BIX					1	28
* 9 INDEX UNIQUE SCAN	CUSTOMERS PK	1	5	0 (0)	00:00:01		

Predicate Information (identified by operation id):

```
2 - access("S"."PROD ID"="P"."PROD ID")
8 - access("S"."CUST ID"=2 OR "S"."CUST ID"=3 OR "S"."CUST ID"=4 OR "S"."CUST ID"=5)
9 - access("S"."CUST ID"="C"."CUST ID")
      filter("C"."CUST ID"=2 OR "C"."CUST ID"=3 OR "C"."CUST ID"=4 OR "C"."CUST ID"=5)
```

ANALIZAR LOS PLANES DE EJECUCION

LABORATORIO 1





—

¿Que es el ajuste SQL y por que lo necesitamos?

¿Qué es la optimización de SQL?

Oracle es una base de datos muy robusta.

Mejor rendimiento, menor costo de hardware.

La optimización de SQL implica la combinación de varias técnicas.

¿Quién se encargará de la optimización?

¿Cuándo ajustar?

1. Proceso Continuo de Ajuste

Ajusta consultas al crearlas y revisa las de alto consumo regularmente.

2. Razones para Ajustar

Cambios estructurales, volumen de datos, actualizaciones de aplicaciones, entre otros.

3. Foco en Consultas de Alto Consumo

<5% de consultas consume ~80% de recursos, ajustarlas es crucial.

4. Retroalimentación de Usuarios

Usuarios pueden reportar problemas de rendimiento debido a cambios imperceptibles.

5. Conclusión

El ajuste de consultas es vital para mantener eficiencia y rendimiento. Próxima conferencia: Identificar y Ajustar Consultas de Alto Consumo.



SQL Incorrecto: ¿Qué es?

SQL que funciona, pero con tasas de rendimiento ineficientes.

Varias razones afectan el rendimiento de las consultas.

Tiempo de análisis, E/S, tiempo de CPU y tiempo de espera.

Análisis: Verificar validez, asignación de área SQL privada, generación de plan de ejecución.

1. E/S Excesivas

1. Operaciones de lectura de datos desde discos.
2. Consultas que leen demasiados bloques de datos pueden ser ineficientes.

2. Tiempo de CPU Elevado

1. Operaciones de combinación, ordenación, cálculos, etc.
2. Consultas ineficientes consumen ciclos de CPU innecesarios.

3. Tiempo de Espera Excesivo

1. Esperas en la CPU o la red mientras otras consultas se ejecutan.
2. DB TIME: Tiempo total de espera y ejecución.



DISEÑO DEL SCHEMA

Selección de Tipos de Datos

Coincidencia de Tipos de Datos en Claves

Definición de Restricciones

Normalización

Selección de Tablas Apropiadas

Uso de Clústeres

Uso de Índices

Tablas Organizadas por Índices



CARDINALIDAD

La cardinalidad se refiere al número de valores únicos de una columna del conjunto de datos. Una columna con cardinalidad alta contiene un gran número de valores únicos, mientras que las columnas con cardinalidad baja contienen menos entradas únicas.

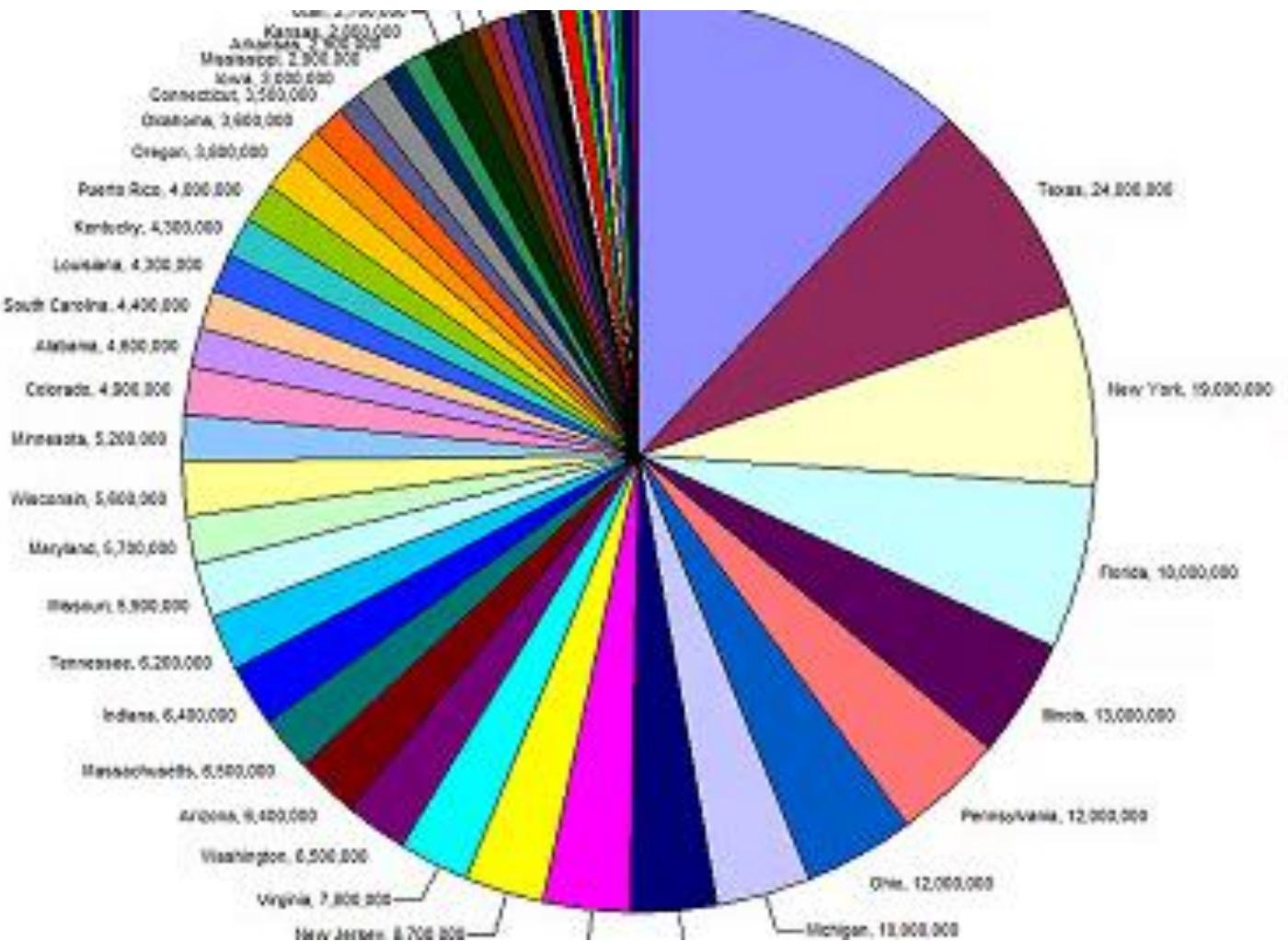
La cardinalidad alta es especialmente común en conjuntos de datos que implican identificadores únicos, como:

Marcas de tiempo: Casi todos los valores son únicos cuando los datos se registran a nivel de milisegundos o nanosegundos.

ID de cliente: Cada cliente tiene un identificador distinto para seguir su actividad.

ID de transacción: Los conjuntos de datos financieros suelen incluir códigos únicos para cada transacción.

RETOS



ESTRATEGIAS



Agrupamiento (o binning): Agrupar los datos de alta cardinalidad en categorías más amplias para reducir el número de valores únicos. Por ejemplo, agrupar los códigos postales por regiones en lugar de almacenar cada uno individualmente.

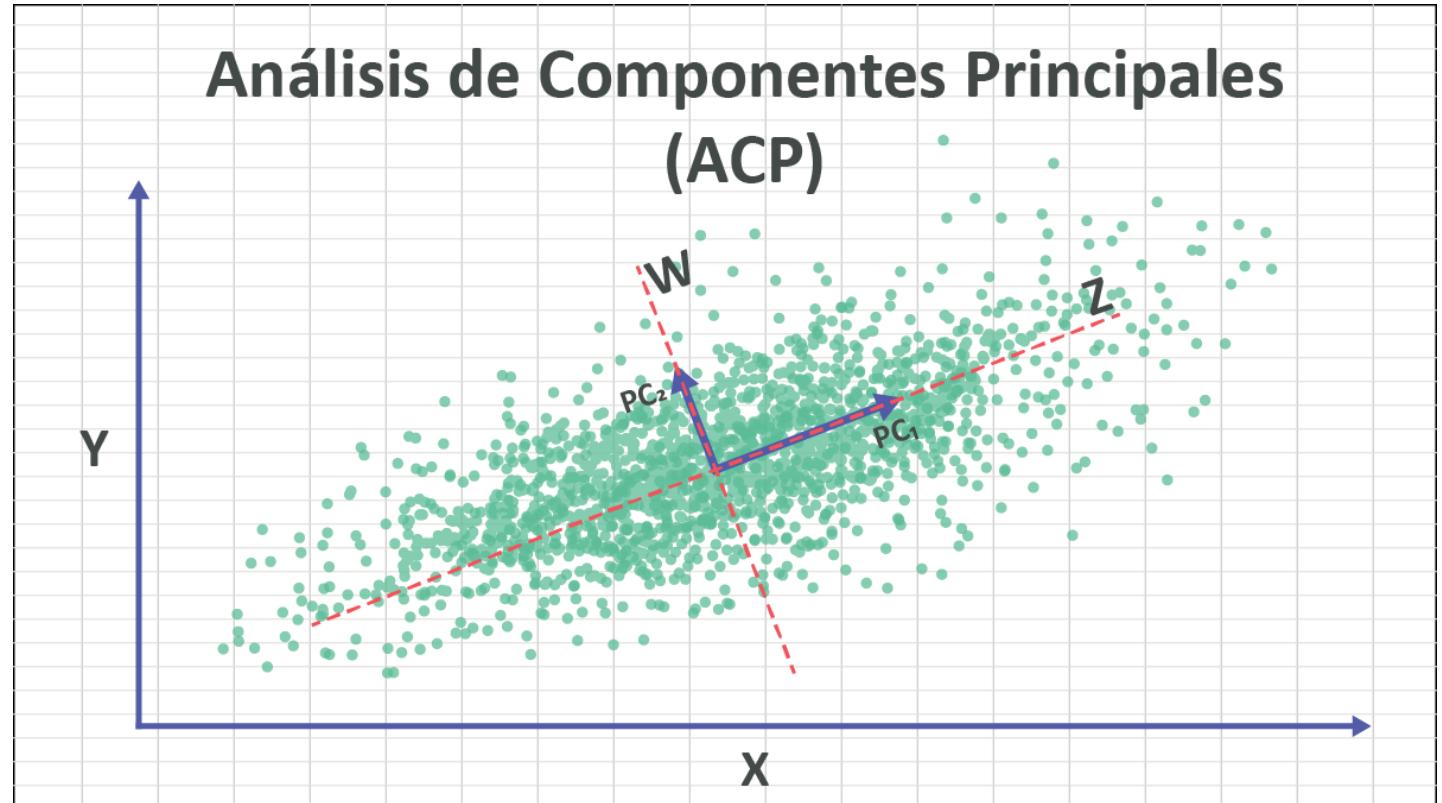


Partición: Estructurar tablas por particiones basadas en campos de consulta frecuente. Por ejemplo, particionar los datos de los clientes por países puede mejorar el rendimiento de las consultas en sistemas distribuidos geográficamente.



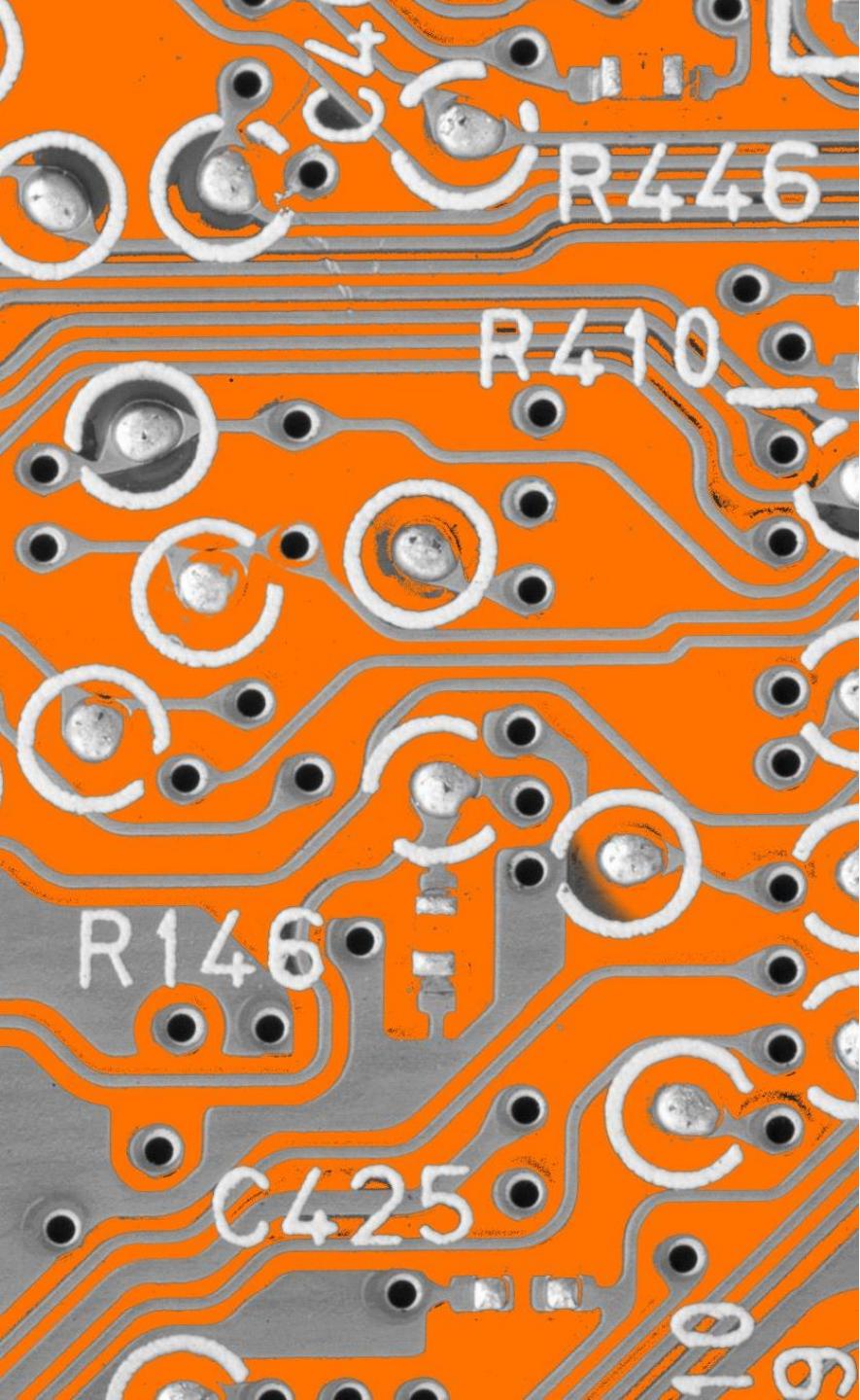
Indexación: El uso de índices diseñados específicamente para campos de alta cardinalidad, como los índices de mapa de bits o compuestos, puede mejorar la velocidad de consulta, equilibrando al mismo tiempo la eficiencia del almacenamiento.

REDUCCION
DIMENSIONALIDAD



LABORATORIO 2



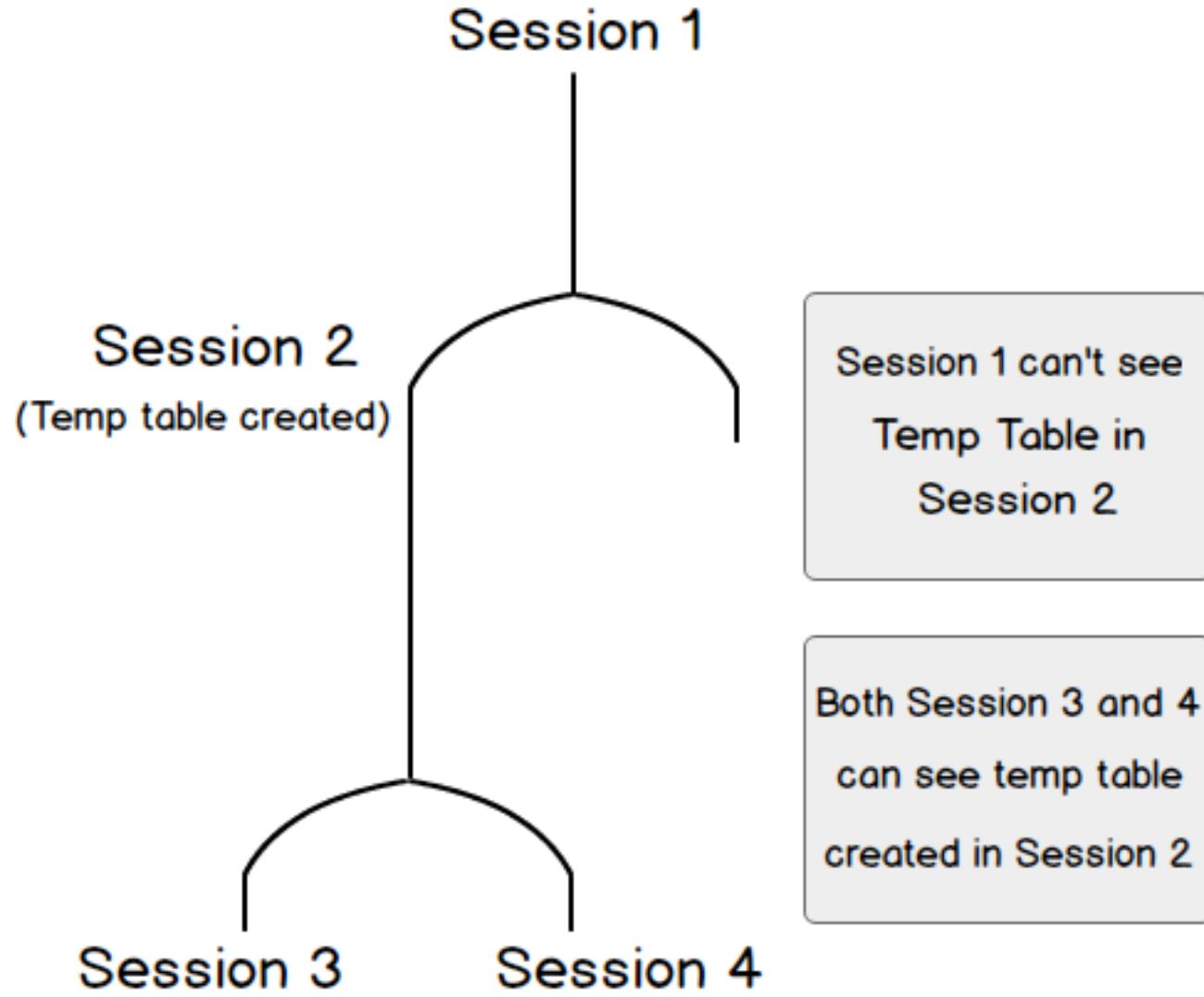


TABLAS TEMPORALES

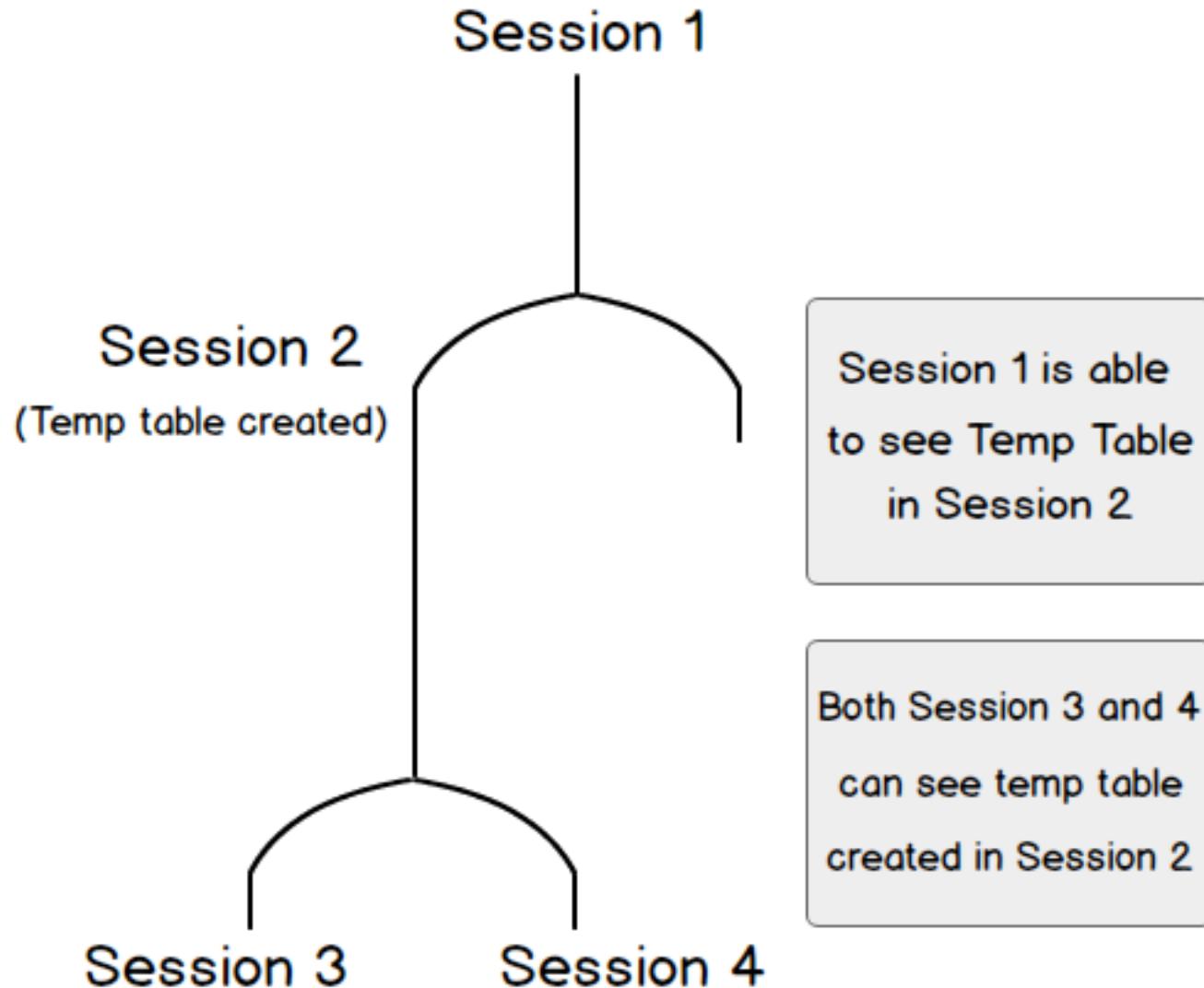
- Una tabla temporal es un objeto almacenado en la base de datos que:
 - Se crea en *tempdb* (SQL Server) o en memoria/disco temporal (MySQL/PostgreSQL).
 - Solo existe mientras dure **la sesión, la conexión, o la transacción**.
 - Permite almacenar datos intermedios sin afectar las tablas permanentes.
 - Mejora rendimiento en consultas complejas o en ETL
-

SINTAXIS

```
/*Insert Databases names into SQL Temp Table*/
BEGIN TRY
    DROP TABLE #DBRecovery
END TRY
BEGIN CATCH SELECT 1 END CATCH
SELECT ROWNUM = ROW_NUMBER() OVER (ORDER BY sys.[databases]),
       DBName = [name],
       RecoveryModel = [recovery_model_desc]
INTO #DBRecovery
FROM sys.[databases]
WHERE [recovery_model_desc] NOT IN ('Simple')
```



LOCALES



GLOBALES

VARIABLES DE TABLA

```
DECLARE @TotalProduct AS TABLE  
(ProductID INT NOT NULL PRIMARY KEY,  
Quantity INT NOT NULL)  
  
INSERT INTO @TotalProduct  
    ( [ProductID], [Quantity] )  
SELECT  
    A.[ProductID],  
    [Quantity] = SUM(B.Quantity)  
FROM dbo.Product AS A  
INNER JOIN dbo.SalesDetails AS B ON A.ProductID = B.  
ProductID
```



CUANDO UTILIZARLAS

- ✓ Procesamiento de ETL (Transformación de datos)
 - Limpieza de datos
 - Cálculos intermedios
 - ✓ Mejorar rendimiento en consultas complejas
 - En vez de subconsultas pesadas:
 - SELECT AVG(Cantidad)
 - FROM #VentasTemp;
 - ✓ Dividir procedimientos complejos
 - SPs que requieren múltiples pasos.
 - ✓ Ordenaciones, joins y agrupaciones temporales
-

CUANDO NO



Cuando la consulta es simple.



Cuando una vista o CTE es suficiente.



En sistemas con poca RAM bajo alta carga.

Característica	Tabla temporal	CTE
Persistencia	Sí (durante sesión)	No
Se puede indexar	Sí	No
Se puede reutilizar	Sí	No
Rendimiento	Mejor en procesos grandes	Mejor en consultas simples

TABLAS TEMPORALES VS CTE

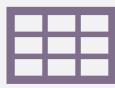
QUE ES CTE



Hacer consultas complejas *más legibles*.



Reutilizar una subconsulta sin repetirla.



Crear consultas recursivas (por ejemplo, jerarquías, árboles, niveles).



Organizar pasos lógicos en consultas grandes.



¿QUÉ SON “CONJUNTOS” EN SQL?

- El **enfoque basado en conjuntos** significa que SQL procesa **filas en bloque**, todas juntas, como un conjunto de datos.
 - Es el *paradigma natural* de SQL.
 - **Ejemplo (UPDATE basado en conjuntos)**
UPDATE Empleados
SET Salario = Salario * 1.10
WHERE Departamento = 'Ventas';
 - → SQL actualiza *todas las filas* que cumplen la condición **al mismo tiempo**.
-

¿QUÉ ES UN CURSOR?

- Un **cursor** procesa **fila por fila** (*row-by-row processing* o *RBAR: Row-By-Agonizing-Row*).
- Es como un bucle en programación tradicional → SQL procesa **una fila por vez**.



EJEMPLO (CURSOR EN SQL SERVER)

```
DECLARE @Id INT, @Salario DECIMAL(10,2);

DECLARE Cur CURSOR FOR

    SELECT Id, Salario FROM Empleados;

OPEN Cur;

FETCH NEXT FROM Cur INTO @Id, @Salario;

WHILE @@FETCH_STATUS = 0

BEGIN

    UPDATE Empleados SET Salario = @Salario * 1.10 WHERE Id = @Id;

    FETCH NEXT FROM Cur INTO @Id, @Salario;

END;

CLOSE Cur;

DEALLOCATE Cur;
```

DIFERENCIAS

Tema	Conjuntos (SET)	Cursos
Lógica	Basada en SQL declarativo	Basado en imperativo
Procesamiento	Por lote	Fila × fila
Rendimiento	Muy alto	Bajo en grandes volúmenes
Memoria	Baja	Alta, reserva buffers
Complejidad	Sencillo	Complejo
Reutilizable	Sí	No
Casos ideales	Agregaciones, actualizaciones masivas	Procesos fila-a-fila, lógica secuencial

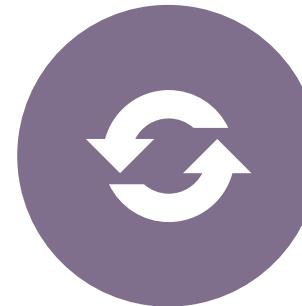
¿POR QUÉ LOS CONJUNTOS SON MEJORES?



SQL ESTÁ OPTIMIZADO
PARA OPERACIONES
MASIVAS.



EL OPTIMIZADOR
PUEDE USAR ÍNDICES Y
PLANES EFICIENTES.



REDUCEN LLAMADAS Y
CICLOS.

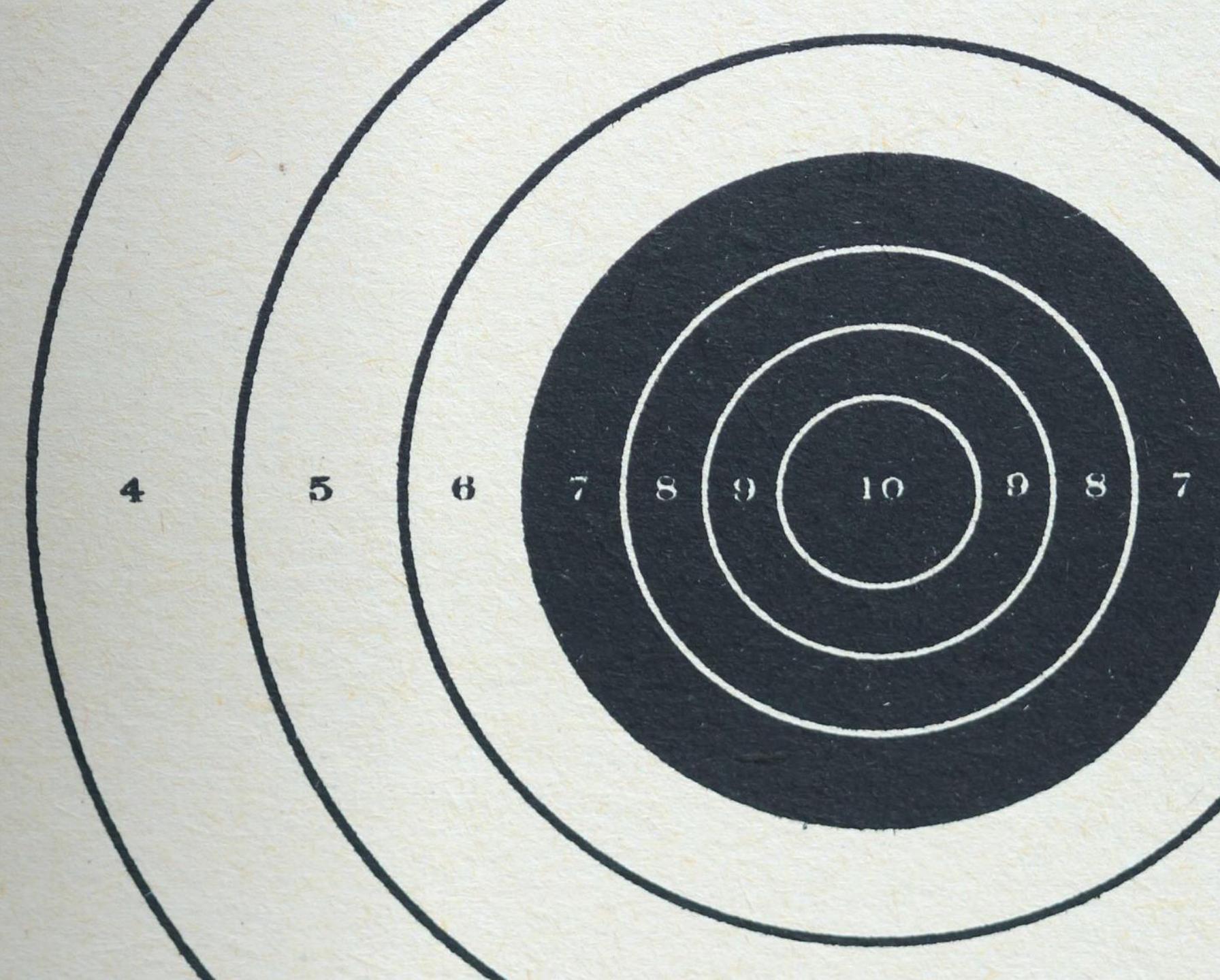


PERMITEN
PARALELISMO.

¿CUÁNDO USAR UN CURSOR? (CASOS REALES)

- Aunque no se recomiendan para todo, **sí tienen usos válidos**:
- ✓ Cuando necesitas **procesamiento secuencial**
 - ✓ Cuando un paso depende del resultado anterior
 - ✓ Cuando no existe una solución basada en conjuntos
 - ✓ Cuando debes llamar procedimientos, API o procesos externos por cada fila
 - ✓ Recorridos complejos de jerarquías (aunque a veces se puede usar CTE recursivo)
- Ejemplo real:
Enviar un correo por cada cliente → requiere proceso fila por fila.

DEMO



SUBCONSULTAS, EXPRESIONES DE TABLA Y CONSULTAS RECURSIVAS

SUBCONSULTAS (SUBQUERIES)

- Una **subconsulta** es una consulta dentro de otra consulta. Se ejecuta primero y su resultado se usa en la consulta principal.

```
SELECT l.nombre, poblacion
FROM localidades l
WHERE poblacion>=ALL(
    SELECT poblacion
    FROM localidades l2
    WHERE l2.n_provincia=l.n_provincia
) a
```

SUBCONSULTA ESCALAR (DEVUELVE UN VALOR)

SELECT Nombre,

(SELECT AVG(Salario) FROM Empleados) AS
SalarioPromedio

FROM Empleados;



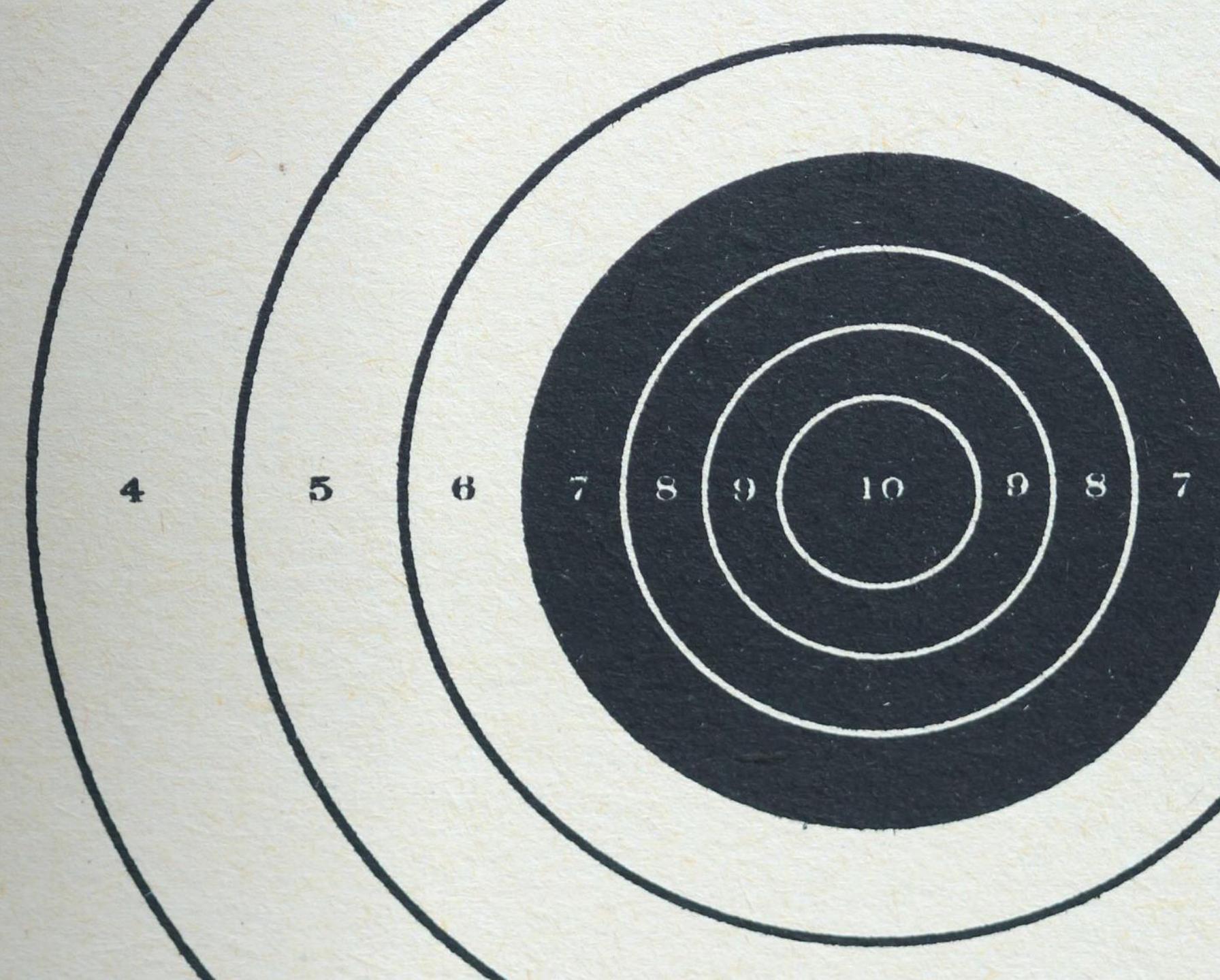
SUBCONSULTA DE MÚLTIPLES FILAS (IN, ANY, ALL)

```
SELECT *  
FROM Clientes  
WHERE Ciudad IN (  
    SELECT Ciudad FROM Sucursales  
);
```

SUBCONSULTA CORRELACIONADA (SE EJECUTA POR CADA FILA)

```
SELECT e.Nombre
FROM Empleados e
WHERE Salario > (
    SELECT AVG(Salario)
    FROM Empleados
    WHERE Departamento = e.Departamento
);
```

DEMO





EXPRESIONES DE TABLA (CTE – COMMON TABLE EXPRESSIONS)

WITH EmpleadosActivos AS (

 SELECT Id, Nombre, Salario

 FROM Empleados

 WHERE Activo = 1

)

SELECT *

FROM EmpleadosActivos

ORDER BY Salario DESC;



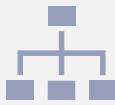
VARIOS CTE EN CADENA

```
WITH A AS (
    SELECT * FROM Ventas WHERE Año = 2024
),
B AS (
    SELECT ClienteId, SUM(Monto) AS Total
    FROM A
    GROUP BY ClienteId
)
SELECT * FROM B WHERE Total > 5000;
```

DIFERENCIAS ENTRE SUBCONSULTAS Y CTE

Tema	Subconsulta	CTE
Legibilidad	Baja	Alta
Reutilización	No	Sí
Mantenimiento	Difícil	Fácil
Complejidad	Buena para simple	Excelente para compleja
Recursividad	No	Sí

CONSULTAS RECURSIVAS



Se usan para trabajar con **jerarquías, árboles, niveles, relaciones padre-hijo**.



Ejemplo clásico: empleados y jefes.



Sintaxis de un CTE recursivo



Tiene 3 partes:

Anchor (nivel inicial)

Recursion (unión recursiva)

Select final

EJEMPLO

```
WITH RECURSIVE Jerarquia AS (
    -- Parte 1: Empleados sin jefe (nivel 1)
    SELECT Id, Nombre, JefeId, 1 AS Nivel
    FROM Empleados
    WHERE JefeId IS NULL
    UNION ALL
    -- Parte 2: Empleados subordinados
    SELECT e.Id, e.Nombre, e.JefeId, j.Nivel + 1
    FROM Empleados e
    JOIN Jerarquia j ON e.JefeId = j.Id
)
SELECT *
FROM Jerarquia
ORDER BY Nivel, Nombre;
```

CUANDO USAR CADA UNA

- **Subconsultas**
 - Consultas rápidas y cortas
 - Comparaciones sencillas
 - Validaciones tipo EXISTS, IN, NOT EXISTS
- **CTE**
 - Consultas largas que necesitan claridad
 - Agrupación por etapas
 - Reutilizar los mismos datos en la misma consulta
 - Alternativa a tablas temporales para cálculos medianos
- **CTE Recursivos**
 - Estructuras jerárquicas
 - Árboles o niveles
 - Caminos en grafos
 - Procesos que necesitan repetición estructurada



¿QUÉ ES EL OPERADOR APPLY?

- APPLY permite ejecutar una **subconsulta por cada fila** de una tabla externa.
Se usa para aplicar **funciones tabla-valor (TVF)** o subconsultas que dependen de cada fila.
 - Es similar a un *lateral join* (como LATERAL en PostgreSQL o CROSS JOIN LATERAL).
-



TIPOS

1. CROSS APPLY

- Solo devuelve las filas donde la subconsulta interna **sí devuelve resultados**.
- Equivale a un **INNER JOIN** lateral.

2. OUTER APPLY

- Devuelve **todas** las filas de la tabla externa, aunque la subconsulta no genere datos.
- Equivale a un **LEFT JOIN** lateral.
-

EJEMPLO

```
CREATE FUNCTION dbo.GetTopVentas(@ClienteId INT)
RETURNS TABLE
AS
RETURN (
    SELECT TOP 3 *
    FROM Ventas
    WHERE ClienteId = @ClienteId
    ORDER BY Monto DESC
);
```



EJEMPLO

CROSS APPLY

- Solo clientes que tienen ventas:
- SELECT c.Nombre, v.*
- FROM Clientes c
- CROSS APPLY dbo.GetTopVentas(c.Id) v;

OUTER APPLY

- Incluye clientes sin ventas:
- SELECT c.Nombre, v.*
- FROM Clientes c
- OUTER APPLY dbo.GetTopVentas(c.Id) v;

DIFERENCIAS CON JOIN

Característica	APPLY	JOIN
Puede usar columnas externas dentro de la subconsulta	✓ Sí	✗ No
Funciona por fila (row-wise)	✓ Sí	✗ No
Reemplaza TVF	✓ Sí	✗ No
JOIN lateral	✓ Sí	✗ No



CASOS DE USO

- El operador **APPLY** es una de las herramientas más potentes de SQL Server cuando necesitas:
 - Subconsultas dependientes de cada fila
 - Top N por grupo
 - Uso avanzado de funciones tabla-valor
 - Alternativa a cursosres
 - Procesamiento basado en filas pero eficiente
-

LABORATORIO 3



OPERADORES DE CONJUNTOS

Los **operadores de conjuntos** permiten **combinar resultados de dos o más consultas (SELECT)** siempre que:

- Las consultas tengan **el mismo número de columnas**
 - Las columnas correspondan en **tipo de dato compatible**
 - El orden de las columnas sea el **mismo**
-

UNION

Combina los resultados de dos consultas **eliminando filas duplicadas**.

```
SELECT ciudad FROM clientes
```

```
UNION
```

```
SELECT ciudad FROM proveedores;
```

 *Resultado: listado de ciudades sin duplicados.*

UNION ALL

Combina los resultados de dos consultas **incluyendo duplicados**.

```
SELECT ciudad FROM clientes
```

```
UNION ALL
```

```
SELECT ciudad FROM proveedores;
```

📌 *Resultado: listado de todas las ciudades, incluso repetidas.*

INTERSECT

Devuelve solo las filas que son **comunes a ambas consultas**.

```
SELECT ciudad FROM clientes
```

```
INTERSECT
```

```
SELECT ciudad FROM proveedores;
```

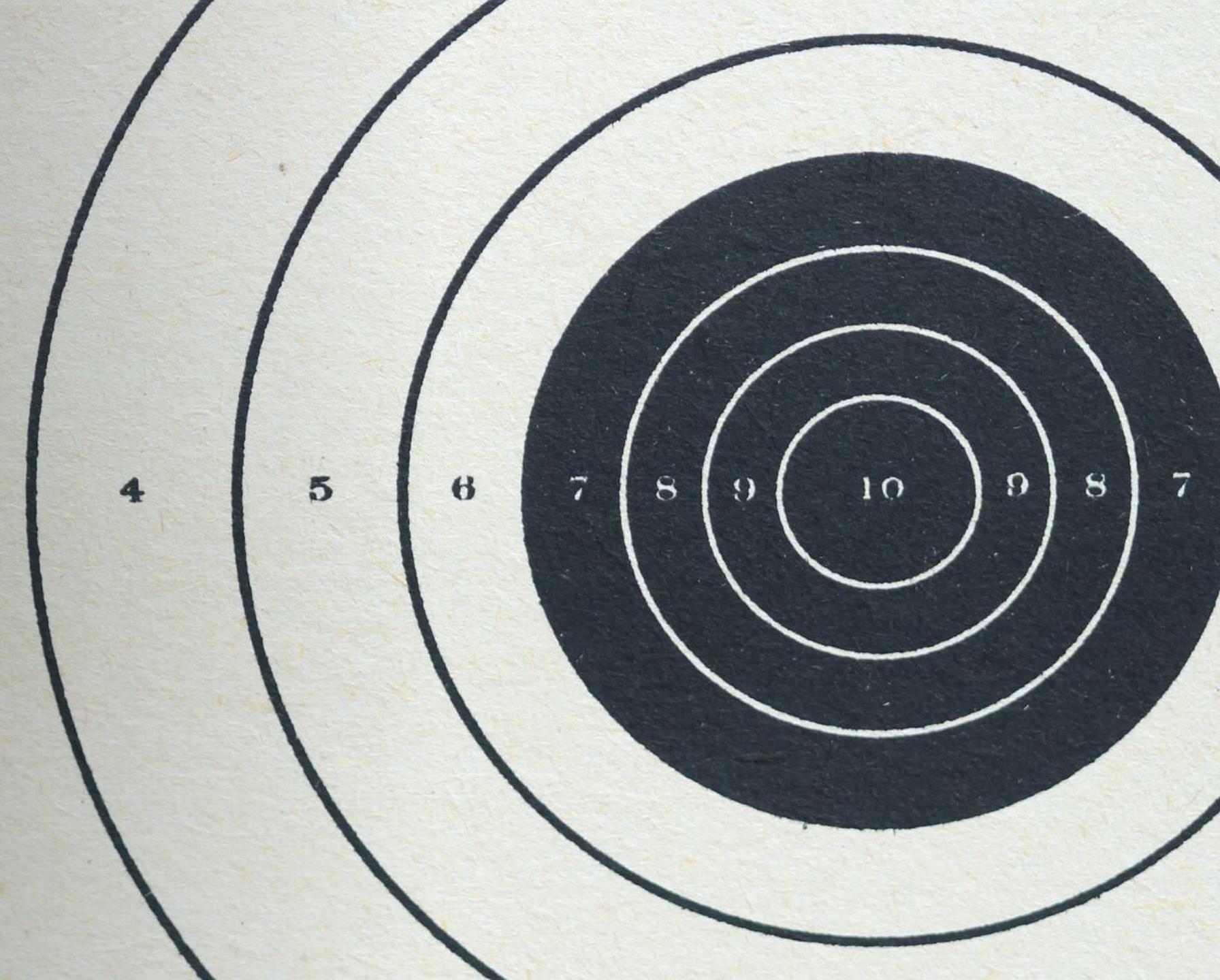
📌 *Resultado: ciudades que aparecen tanto en clientes como en proveedores.*

⚠ *INTERSECT no está disponible en MySQL (solo en SQL Server, Oracle y PostgreSQL).*

EXCEPT/MINUS

- Devuelve las filas de la primera consulta que **no aparecen en la segunda**.
- **EXCEPT (SQL Server, PostgreSQL)**
 - SELECT ciudad FROM clientes
 - EXCEPT
 - SELECT ciudad FROM proveedores;
- **MINUS (Oracle)**
 - SELECT ciudad FROM clientes
 - MINUS
 - SELECT ciudad FROM proveedores;
-  *Resultado: ciudades que están en clientes pero no en proveedores.*

DEMO



VARIABLES

- Las **variables** en SQL permiten almacenar valores temporales para utilizarlos dentro de scripts, procedimientos almacenados o consultas.
- El manejo depende del motor SQL (MySQL, SQL Server, Oracle, PostgreSQL).



VARIABLES EN ORACLE (PL/SQL)

DECLARE

```
    salario_min NUMBER := 1000;
```

BEGIN

```
    SELECT nombre INTO nombre_emp
    FROM empleados WHERE salario >
    salario_min;
```

END;



TIPOS DE VARIABLES

- **Numéricas:** INT, DECIMAL, FLOAT
- **Texto:** VARCHAR, CHAR, NVARCHAR
- **Fecha y hora:** DATE, TIME, DATETIME
- **Booleanas** (dependen del motor)
- **Variables de sistema:** valores internos del servidor (ej. @@VERSION, @@ROWCOUNT en SQL Server)

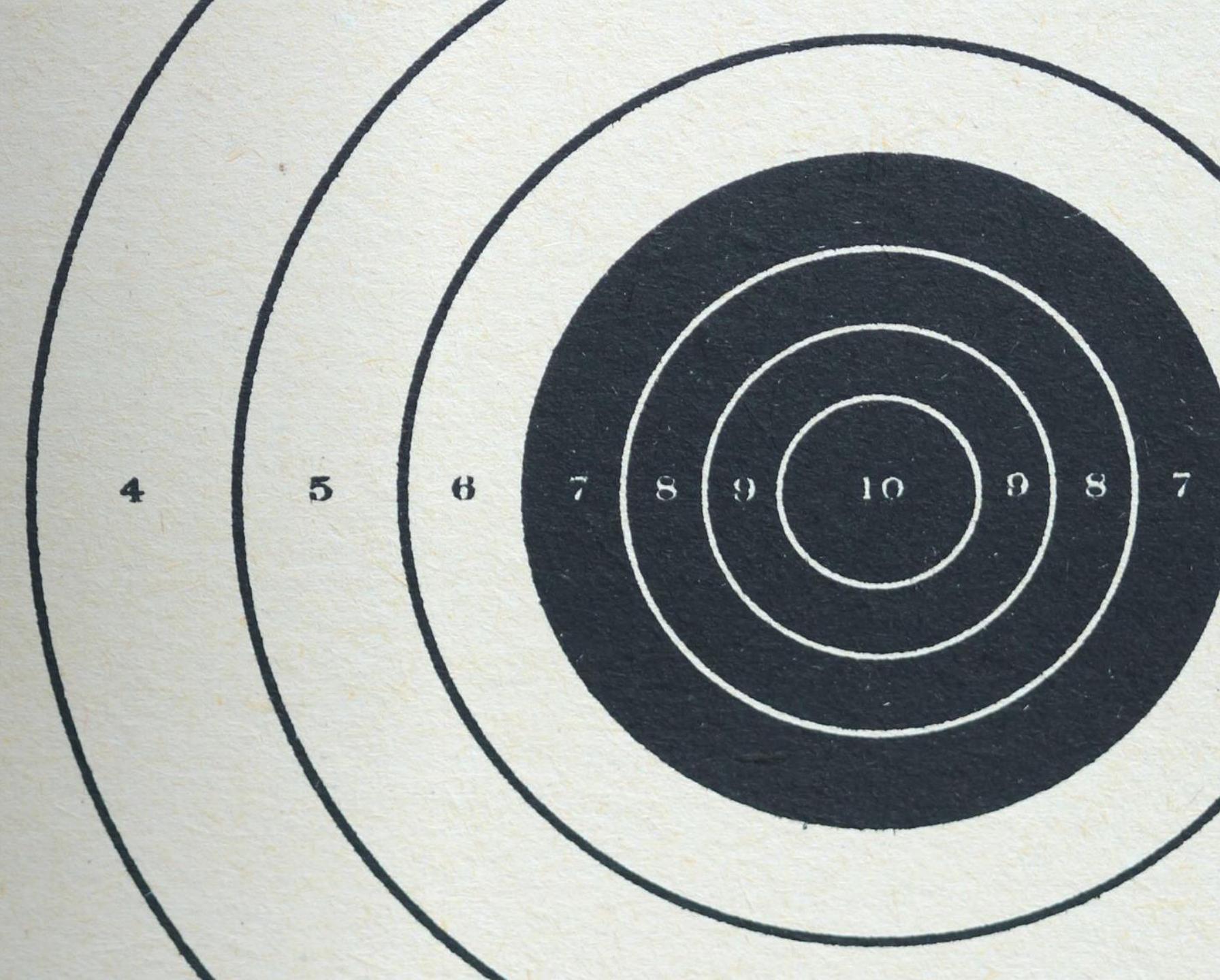


¿PARA QUÉ SE USAN?

- Almacenar valores temporales
- Parametrizar consultas
- Crear procedimientos almacenados
- Controlar flujo (loops, condiciones)
- Crear cálculos intermedios

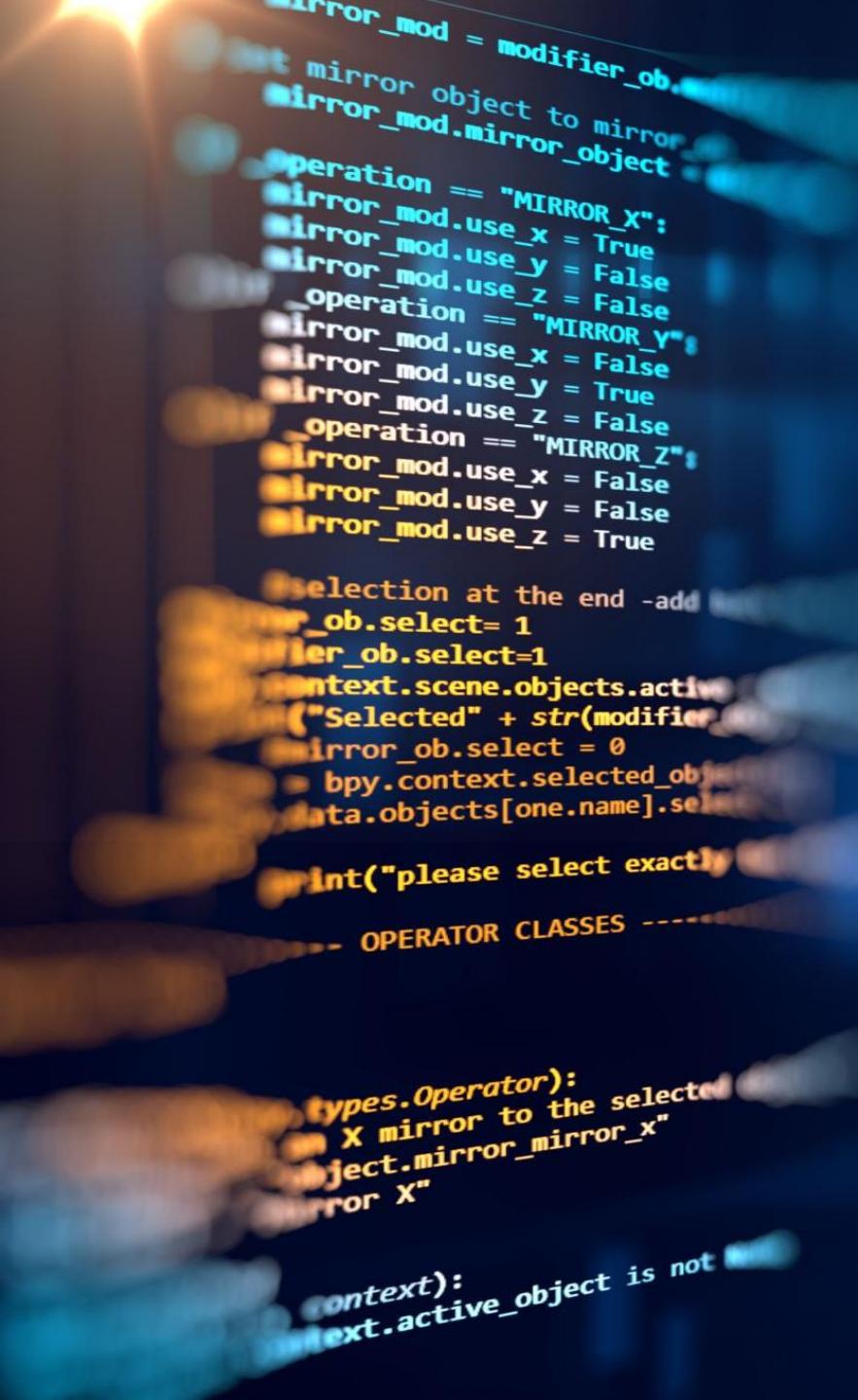


DEMO



FUNCTION DEFINIDA POR EL USUARIO

- Una **función en PL/SQL** es un bloque de código almacenado en la base de datos que **siempre devuelve un valor**.
La escribe el programador para reutilizar lógica, hacer cálculos o procesar datos de forma más ordenada.
- Se guardan en Oracle y pueden usarse en muchos lugares del sistema: consultas, otros bloques PL/SQL, triggers, paquetes, etc.





¿PARA QUÉ SIRVEN?

- Sirven para:
- Centralizar reglas de negocio
- Evitar escribir la misma lógica en varios programas
- Validar datos antes de insertarlos o procesarlos
- Realizar cálculos complejos
- Transformar valores, por ejemplo: convertir, formatear, limpiar texto
- Crear funciones matemáticas, financieras, lógicas, etc.

En resumen, permiten **ordenar el código** y hacerlo más fácil de **mantener**.

CARACTERISTICAS

- **Devuelven un valor obligatorio**

Puede ser número, texto, fecha, etc.

- **Pueden recibir parámetros**

Los parámetros más comunes son IN (de entrada).

- **Se integran con SQL**

Puedes usar una función dentro de un SELECT, WHERE, GROUP BY...

Siempre que la función no haga cosas prohibidas dentro de SQL (como DML).

- **Se pueden llamar desde PL/SQL o SQL**

Desde SQL solo si no usan BOOLEAN ni hacen acciones “no permitidas”.

- **Tienen una estructura fija**

- Nombre

- Parámetros

- Tipo de dato que devuelve

- Inicio/fin del bloque

- Al menos un RETURN



EJEMPLO

```
CREATE FUNCTION check_sal RETURN Boolean IS
    v_dept_id employees.department_id%TYPE; v_empno      employees.employee_id%TYPE;
    v_sal       employees.salary%TYPE; v_avg_sal employees.salary%TYPE;

    BEGIN
        v_empno:=205;

        SELECT salary,department_id INTO v_sal,v_dept_id FROM employees
        WHERE employee_id= v_empno;

        SELECT avg(salary) INTO v_avg_sal FROM employees WHERE department_id=v_dept_id;
        IF v_sal > v_avg_sal THEN RETURN TRUE;
        ELSE
            RETURN FALSE; END IF;
        EXCEPTION
            WHEN NO_DATA_FOUND THEN RETURN NULL;
        END;
```



PROCEDIMIENTO ALMACENADO

- Un **procedimiento almacenado (stored procedure)** es un bloque de código PL/SQL que se guarda dentro de Oracle y que **no devuelve un valor obligatorio**. Se usa para ejecutar acciones dentro de la base de datos.
- Es parecido a una función, pero **no tiene RETURN obligatorio**.

¿PARA QUÉ SIRVEN?

Los procedimientos sirven para:

- Insertar datos
- Actualizar o eliminar registros
- Validar información antes de grabarla
- Automatizar procesos
- Controlar transacciones
- Agrupar lógica de negocio compleja
- Evitar escribir las mismas sentencias muchas veces

Son esenciales en sistemas reales porque **permiten controlar la lógica desde la base de datos.**



CARACTERÍSTICAS PRINCIPALES

- **No devuelven un valor obligatorio**

Aunque pueden devolver valores usando parámetros OUT o IN OUT.

- **Pueden tener muchos parámetros**

De entrada (IN), salida (OUT) o mixtos (IN OUT).

- **Pueden ejecutar DML sin restricciones**

A diferencia de las funciones, aquí sí se puede usar:

- INSERT

- UPDATE

- DELETE

- MERGE

- **Pueden manejar excepciones**

Es decir, controlar errores de forma personalizada.

- **Pueden llamar a otros procedimientos y funciones**

Permiten modularizar el código.



COMPARATIVA

Característica	Procedimiento	Función
¿Devuelve un valor?	No obligatorio	Sí, siempre
¿Se usa en SELECT?	No	Sí, si está permitida
¿Puede hacer DML?	Sí	Limitado
¿Se usa para cálculos?	No principalmente	Sí
¿Se usa para acciones?	Sí	No principalmente

VENTAJAS

- Mejoran la seguridad (evitas exponer SQL directo).
- Reducen tráfico en la red (la lógica está en el servidor).
- Hacen el sistema más rápido cuando se usan bien.
- Organizan el código en bloques reutilizables.
- Facilitan mantenimiento y escalabilidad.



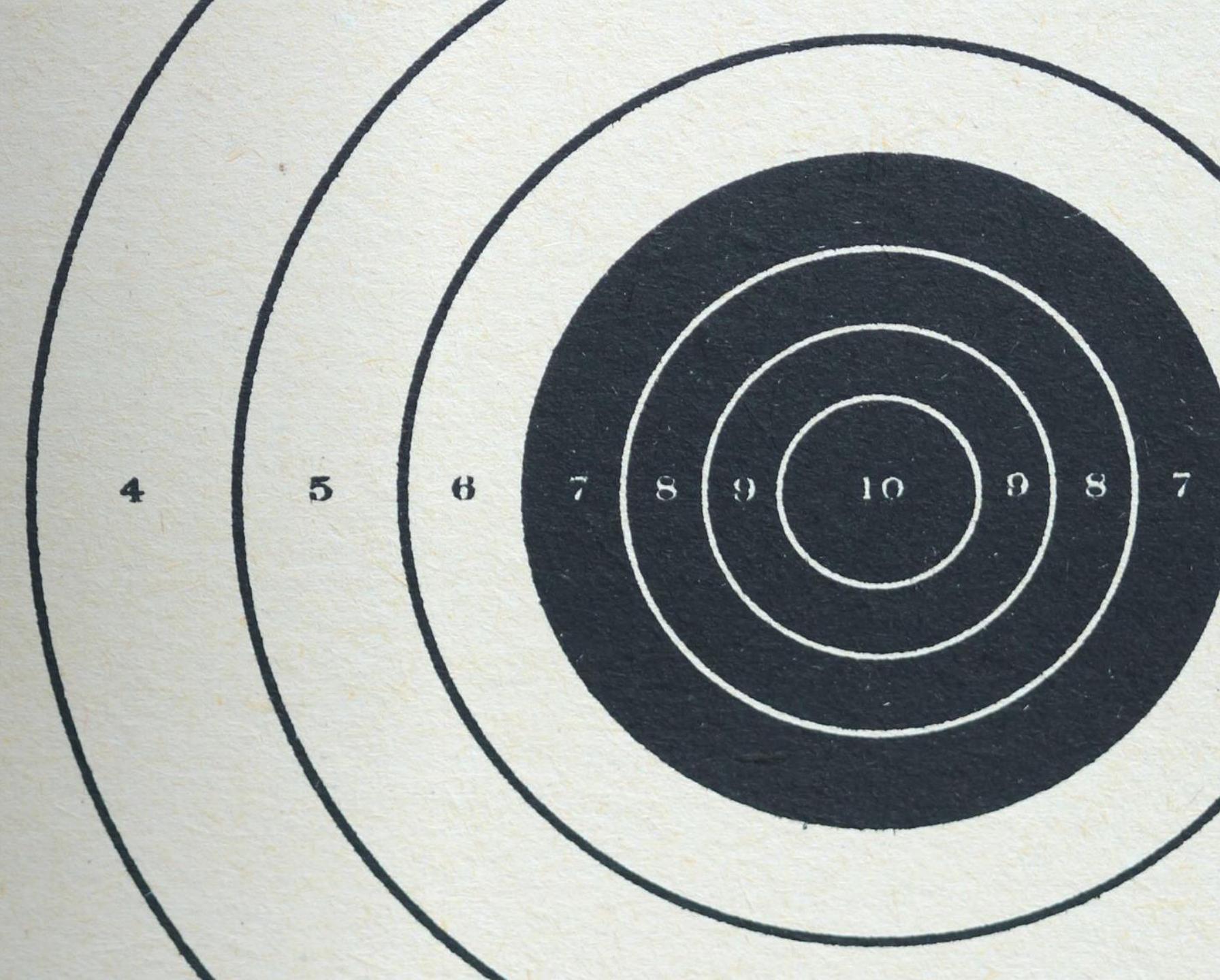
EJEMPLO

```
CREATE TABLE dept AS SELECT * FROM departments;  
CREATE PROCEDURE add_dept IS v_dept_id  
dept.department_id%TYPE;  
v_dept_name dept.department_name%TYPE; BEGIN  
v_dept_id:=280; v_dept_name:='ST-Curriculum';  
INSERT INTO dept(department_id,department_name)  
VALUES(v_dept_id,v_dept_name); DBMS_OUTPUT.PUT_LINE('  
Inserted '|| SQL%ROWCOUNT  
||' row '); END;
```

LABORATORIO 4



DEMO



Do not use **select *** for all the queries!

- ▶ The optimizer may select a worse plan if you query for unnecessary columns
- ▶ While joining multiple tables or querying from views, selecting less columns might affect the performance
- ▶ If you use **select ***, the database needs to check the data dictionary to get the table structure
- ▶ **select *** will make the database perform more I/O operations
- ▶ **select *** may decrease the performance significantly if the table has LOBs
- ▶ **select *** will have a higher overload on the network. So there might be more network waits
- ▶ **select *** tends to have problems on maintenance

BAD

```
SELECT first_name, last_name, department_name FROM employees  
WHERE first_name||last_name = 'StevenKING';
```



GOOD

```
SELECT first_name, last_name, department_name FROM employees  
WHERE first_name = 'Steven' AND last_name = 'KING';
```



BAD

```
SELECT prod_id, cust_id, time_id FROM sales  
WHERE time_id + 10 = '20-JAN-98';
```



GOOD

```
SELECT prod_id, cust_id, time_id FROM sales  
WHERE time_id = '10-JAN-98';
```



GOOD

```
SELECT prod_id, cust_id, time_id FROM sales  
WHERE time_id = to_date('20-JAN-98', 'DD-MON-RR')-10;
```



BAD

```
SELECT employee_id, first_name, last_name, salary FROM employees  
WHERE last_name LIKE '%on';
```



GOOD

```
SELECT employee_id, first_name, last_name, salary FROM employees  
WHERE last_name LIKE 'Ba%';
```



GOOD

```
SELECT employee_id, first_name, last_name, reverse(last_name)  
FROM employees WHERE reverse(last_name) LIKE 'rahh%';
```



BAD

```
SELECT employee_id, first_name, last_name, salary FROM employees  
WHERE TRUNC(hire_date, 'YEAR') = '01-JAN-2002';
```



GOOD

```
SELECT employee_id, first_name, last_name, salary FROM employees  
WHERE hire_date BETWEEN '01-JAN-2002' AND '01-JAN-2002';
```



BAD

```
SELECT cust_id, cust_first_name, cust_last_name FROM customers  
WHERE cust_postal_code = 60332;
```



GOOD

```
SELECT cust_id, cust_first_name, cust_last_name FROM customers  
WHERE cust_postal_code = '60332';
```



BAD

```
SELECT prod_id,cust_id,time_id,amount_sold,channel_id FROM sales  
WHERE channel_id = 3;  
UNION  
SELECT prod_id,cust_id,time_id,amount_sold,channel_id FROM sales  
WHERE channel_id = 4;
```



GOOD

```
SELECT prod_id,cust_id,time_id,amount_sold,channel_id FROM sales  
WHERE channel_id = 3;  
UNION ALL  
SELECT prod_id,cust_id,time_id,amount_sold,channel_id FROM sales  
WHERE channel_id = 4;
```



BAD

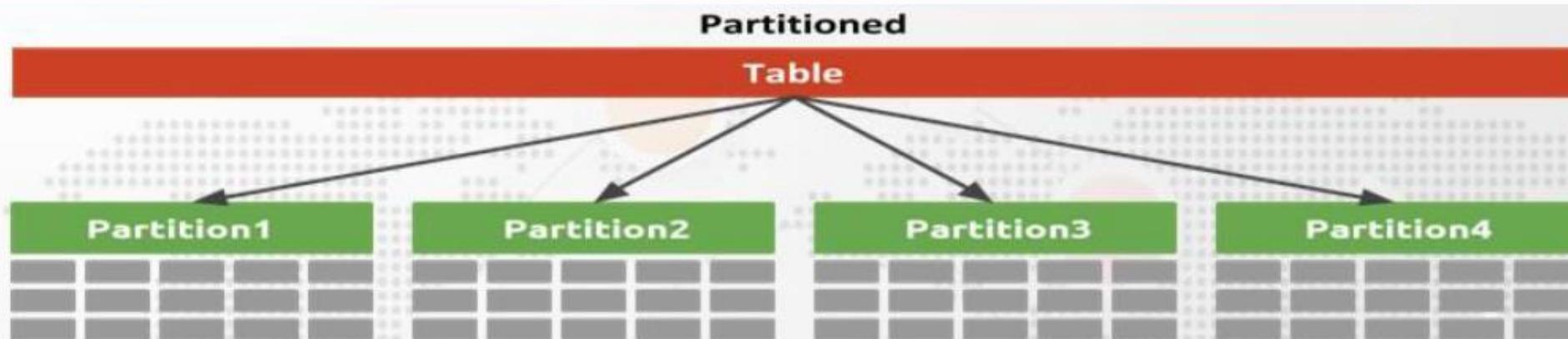
```
SELECT prod_id, SUM(amount_sold) FROM sales  
GROUP BY prod_id  
HAVING prod_id = 136;
```



GOOD

```
SELECT prod_id, SUM(amount_sold) FROM sales  
WHERE prod_id = 136  
GROUP BY prod_id;
```





If the query has low selectivity, the optimizer mostly prefers performing full-table scans.

Creating partitioned tables increases the cost for the queries having low selectivity.

Selecting from specific partitions is called as partition pruning

How can we prune the partitions?

- ▶ Selecting directly from the partition by using the partition name
- ▶ Adding predicates to the where clause including the partition key (partitioned columns)

Dim

Dim

Fact

Dim

Dim

1

*

1

*

1

*

1

*

*

1

Using the BIND Variables may increase the performance by decreasing the parse counts

```
SELECT AVG(salary) FROM employees WHERE department_id = 30;
SELECT AVG(salary) FROM employees WHERE department_id = 40;
SELECT AVG(salary) FROM employees WHERE department_id = 50;

SELECT sql_id,executions,parse_calls,first_load_time,last_load_time,sql_text
FROM v$sql
WHERE sql_text LIKE '%avg(salary) from employees%'
ORDER BY first_load_time DESC;

SELECT AVG(salary) FROM employees WHERE department_id = :b;
```

Common reasons for a Bad SQL:

- ✗ Poorly written query
- ✗ Index used or not used
- ✗ There is no index
- ✗ Predicates are not used
- ✗ Wrong types in predicates
- ✗ Wrong join order
- ✗ Other

➤ Common possible solutions:

- ✓ Make the statistics up to date
- ✓ Use dynamic statistics
- ✓ Create or modify indexes
- ✓ Rewrite the query to use an index
- ✓ Use hints
- ✓ Remove wrong hints
- ✓ Change the hints
- ✓ Eliminate implicit data type conversion
- ✓ Create function-based indexes
- ✓ Use index-organized tables
- ✓ Change the optimizer mode
- ✓ Use parallel execution
- ✓ Use materialized views
- ✓ Modify or disable triggers and constraints
- ✓ Other



¿QUÉ ES UN TRIGGER?

- Un trigger permite **automatizar acciones** cuando ocurre un cambio en la tabla.
Por ejemplo:
- Registrar cambios en una tabla de auditoría
- Validar datos antes de insertar
- Evitar que se eliminen registros
- Actualizar automáticamente otra tabla

TIPOS

Tipo	Se ejecuta cuando
BEFORE	antes de la operación
AFTER	después de la operación
INSTEAD OF	(SQL Server) reemplaza la operación

EJEMPLO AFTER

```
CREATE OR REPLACE TRIGGER tr_insert_empleado
AFTER INSERT ON empleados
FOR EACH ROW
BEGIN
    INSERT INTO auditoria_empleados(id_empleado, accion,
fecha)
    VALUES (:NEW.id_empleado, 'INSERT', SYSDATE);
END;
/
```

BEFORE UPDATE

```
CREATE OR REPLACE TRIGGER tr_check_salario
BEFORE UPDATE ON empleados
FOR EACH ROW
BEGIN
    IF :NEW.salario < 1000 THEN
        RAISE_APPLICATION_ERROR(-20001, 'El salario no puede
ser menor a 1000');
    END IF;
END;
/
```

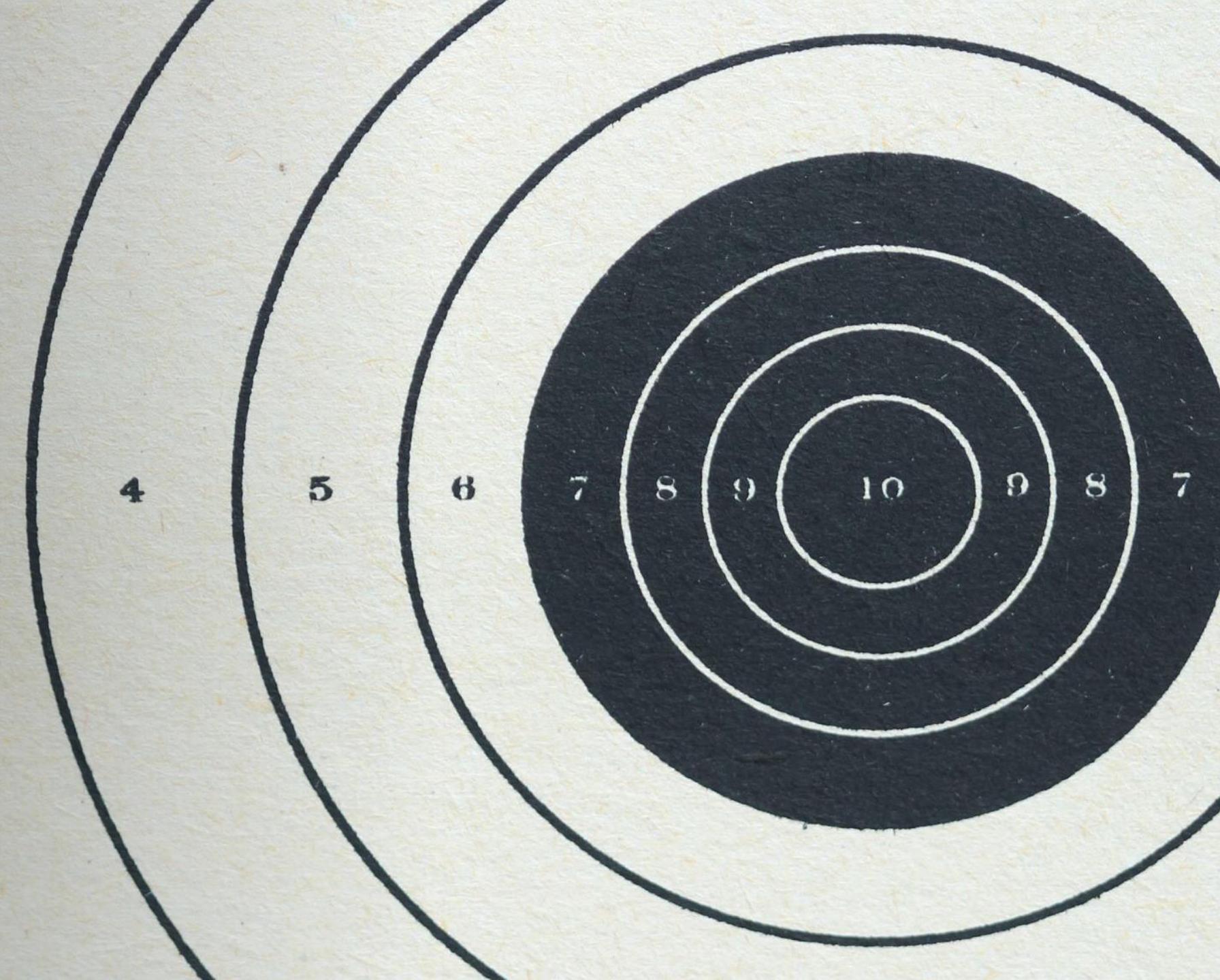


EVITAR DELETE

```
CREATE OR REPLACE TRIGGER tr_instead_insert
INSTEAD OF INSERT ON vista_empleados
FOR EACH ROW
BEGIN
    INSERT INTO empleados(id, nombre, salario)
    VALUES (:NEW.id, :NEW.nombre, :NEW.salario);
END;
/
```



DEMO



¿QUÉ ES LA GESTIÓN DE EXCEPCIONES?

- La **gestión de excepciones** en PL/SQL permite **controlar errores** que ocurren durante la ejecución de un bloque. En lugar de que el programa falle, puedes **atrapar** el error y actuar (mostrar un mensaje, registrar en una tabla, ignorar, etc.).

ESTRUCTURA BÁSICA

BEGIN

-- Código principal

EXCEPTION

WHEN tipo_de_excepcion THEN

-- Acción o mensaje de error

END;

/

EXCEPCIONES PREDEFINIDAS

Excepción

NO_DATA_FOUND

TOO_MANY_ROWS

ZERO_DIVIDE

DUP_VAL_ON_INDEX

INVALID_NUMBER

VALUE_ERROR

Ocurre cuando...

una consulta SELECT INTO
no devuelve filas

SELECT INTO devuelve más
de una fila

división entre cero

clave duplicada en índice
UNIQUE

conversión de número fallida

conversión o tamaño inválido

EXCEPCIONES DEFINIDAS POR EL USUARIO

```
DECLARE
    ex_salario_bajo EXCEPTION;
BEGIN
    IF v_salario < 1000 THEN
        RAISE ex_salario_bajo;
    END IF;
EXCEPTION
    WHEN ex_salario_bajo THEN
        DBMS_OUTPUT.PUT_LINE('El salario es demasiado bajo.');
END;
/
```

LANZAR ERRORES PERSONALIZADOS CON



```
IF :NEW.sueldo < 0 THEN
```

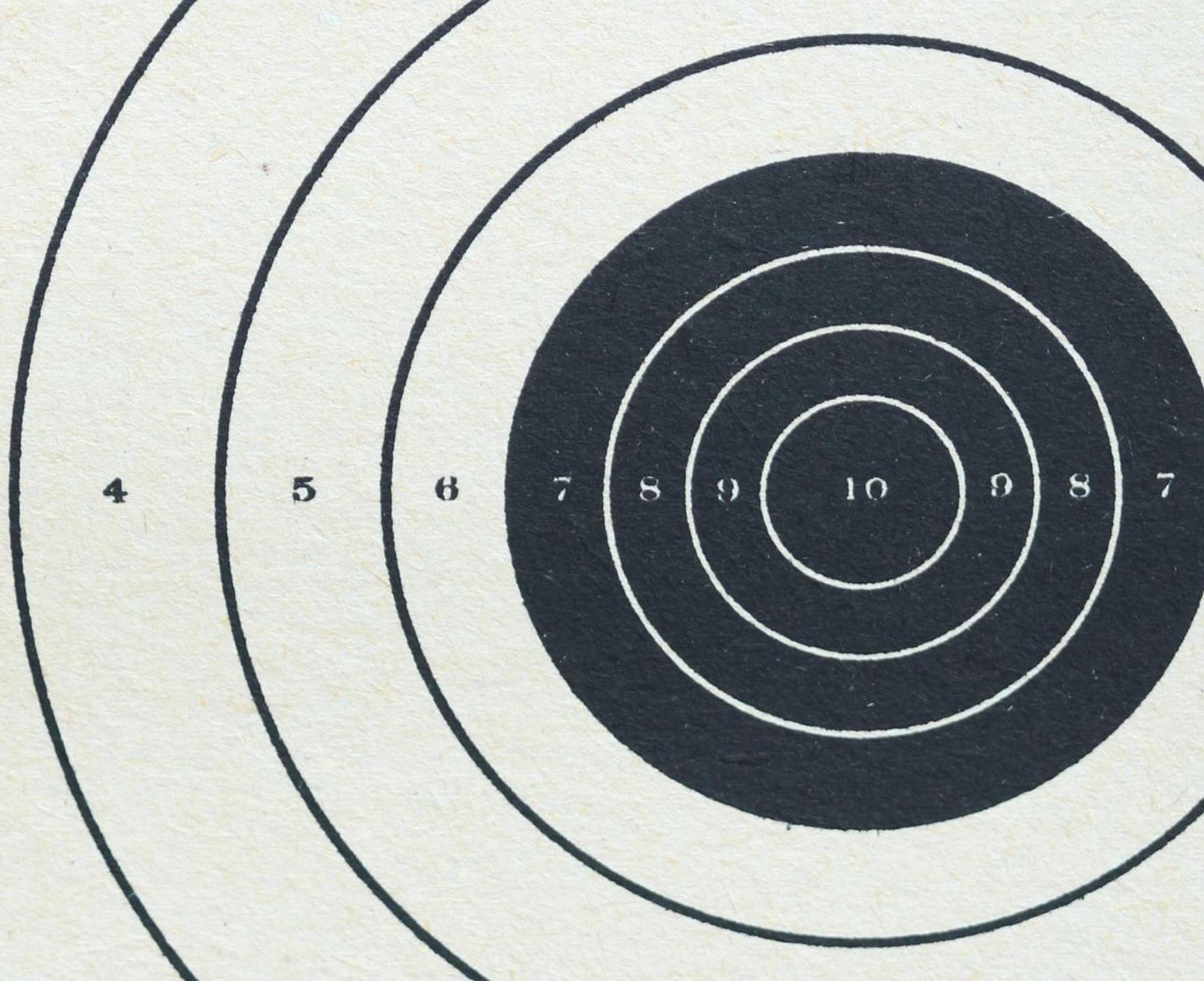


```
RAISE_APPLICATION_ERROR(-20002, 'El sueldo no puede ser negativo.');
```



```
END IF;
```

DEMO



LABORATORIO 5





INNER JOIN (JOIN INTERNO)

- Devuelve **solo las filas que tienen coincidencias en ambas tablas**.
- Si no hay coincidencia, no se muestra la fila.
- **Ejemplo:**

```
SELECT empleados.nombre,  
departamentos.nombre_departamento  
FROM empleados e  
INNER JOIN departamentos d  
ON e.id_departamento = d.id;
```



LEFT JOIN (LEFT OUTER JOIN)

- Devuelve **todas las filas de la tabla de la izquierda**, aunque no tengan coincidencia en la tabla de la derecha.
 - Las columnas de la tabla derecha que no coincidan aparecerán como NULL.
 - **Ejemplo:**

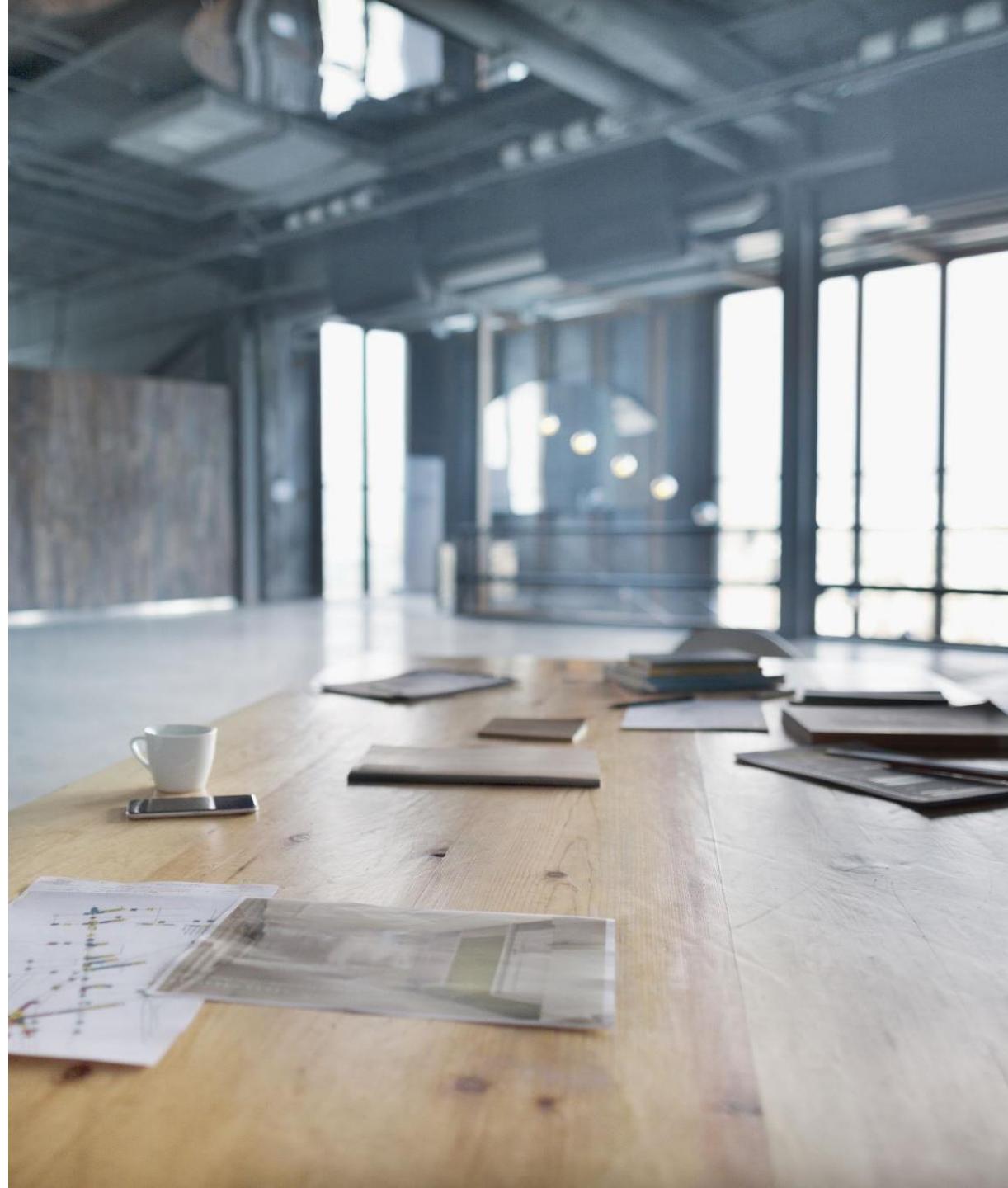
```
SELECT empleados.nombre, departamentos.nombre_departamento
FROM empleados e
LEFT JOIN departamentos d
ON e.id_departamento = d.id;
```
 - Muestra todos los empleados, incluso si no tienen departamento asignado.
-

RIGHT JOIN (RIGHT OUTER JOIN)

- Devuelve **todas las filas de la tabla de la derecha**, aunque no tengan coincidencia en la tabla de la izquierda.
- **Ejemplo:**

```
SELECT empleados.nombre,  
departamentos.nombre_departamento  
FROM empleados e  
RIGHT JOIN departamentos d  
ON e.id_departamento = d.id;
```

- Muestra todos los departamentos, incluso si no tienen empleados.

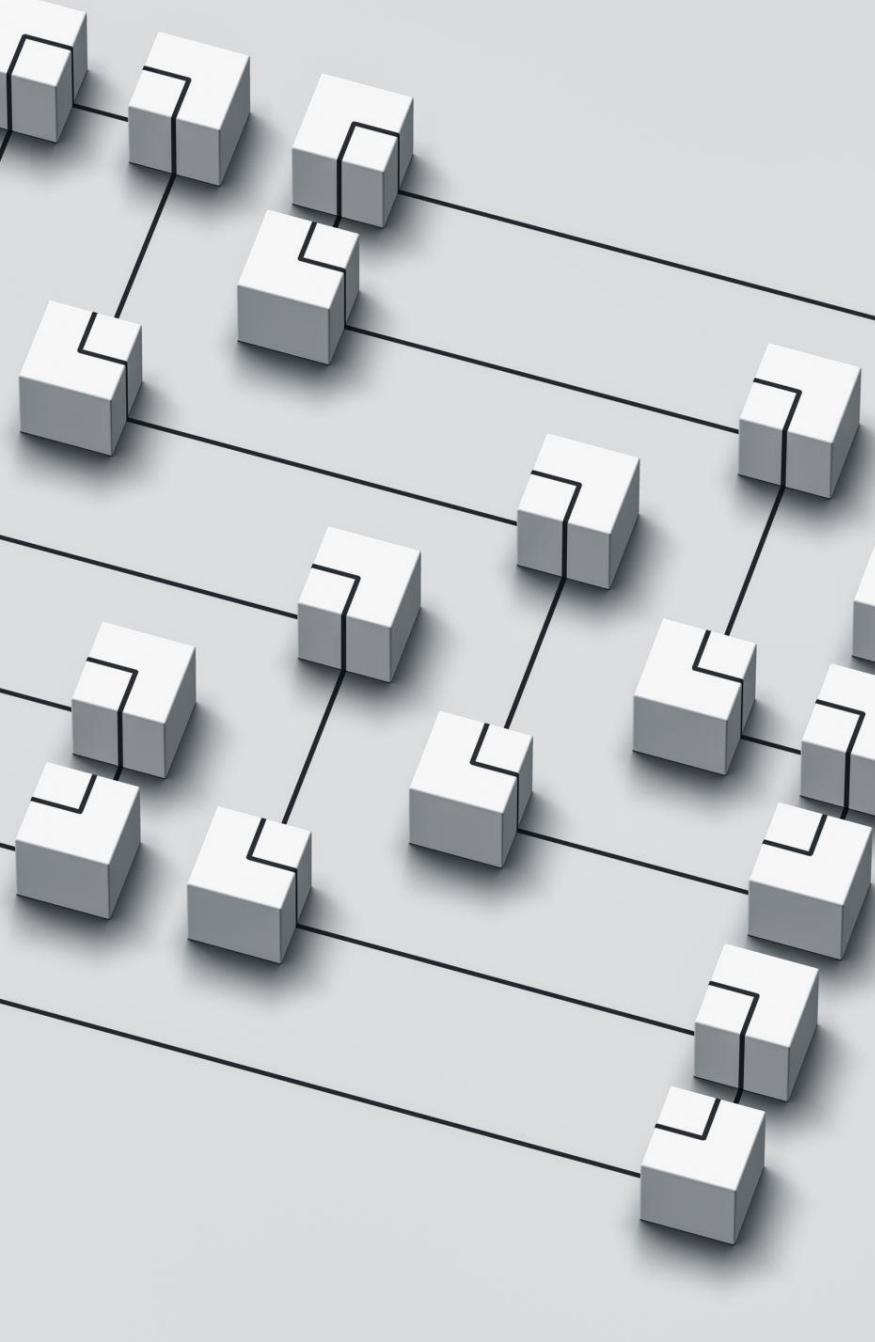


FULL JOIN (FULL OUTER JOIN)

- Devuelve **todas las filas de ambas tablas**.
- Si no hay coincidencia, se muestra NULL en la tabla que no tenga datos.
- **Ejemplo:**

```
SELECT e.nombre, d.nombre_departamento  
FROM empleados e  
FULL OUTER JOIN departamentos d  
ON e.id_departamento = d.id;
```

- Muestra todos los empleados y todos los departamentos, aunque algunos no tengan coincidencias.
-



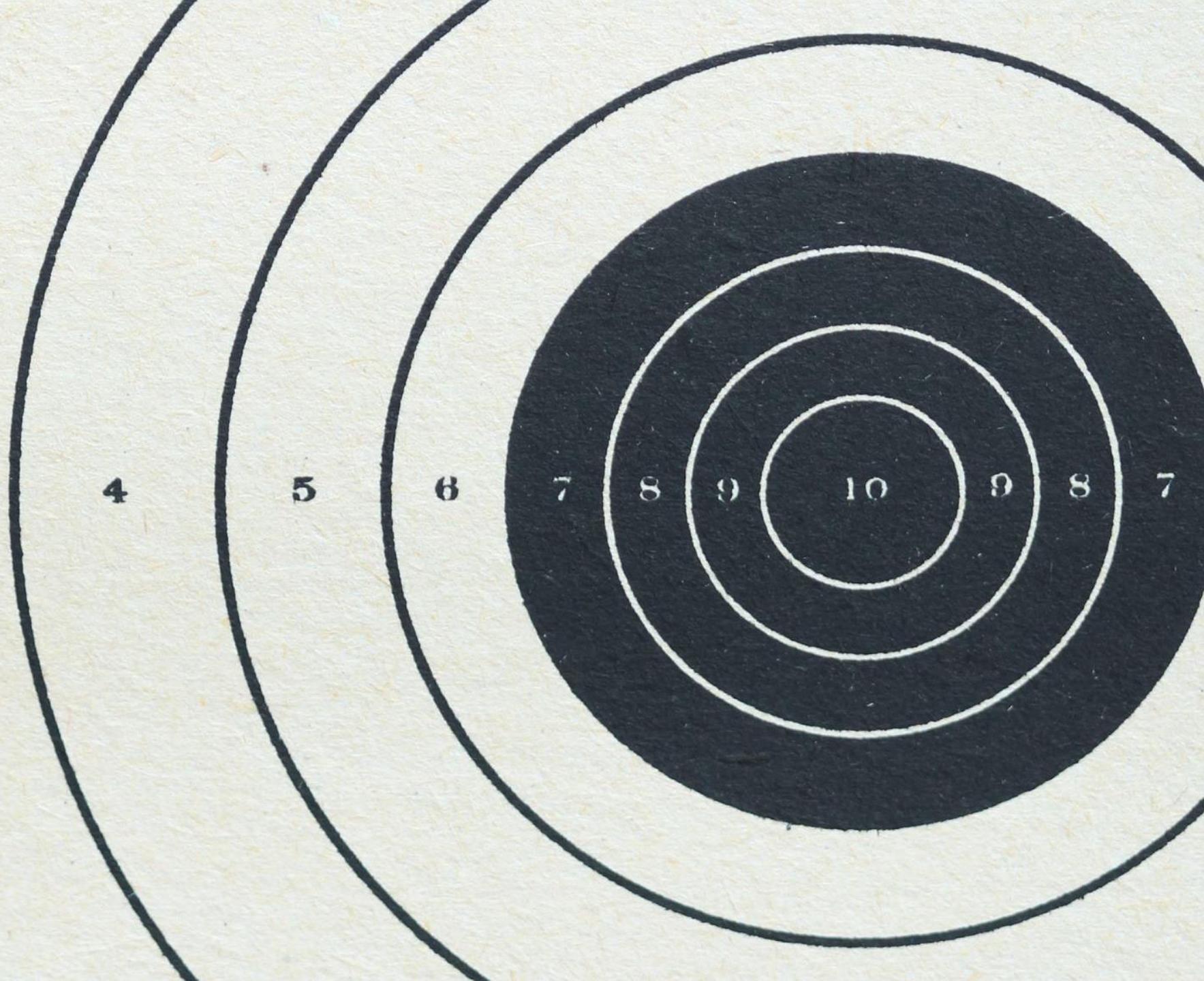
CROSS JOIN

- Devuelve **todas las combinaciones posibles** entre filas de ambas tablas (producto cartesiano).
- **Ejemplo:**

```
SELECT e.nombre, d.nombre_departamento  
FROM empleados e  
CROSS JOIN departamentos d;
```

- Si hay 5 empleados y 3 departamentos, se generarán 15 filas.

DEMO





PACKAGE

- Un **Package** es un **contenedor lógico** en Oracle que agrupa:
 - **Procedimientos (PROCEDURES)**
 - **Funciones (FUNCTIONS)**
 - **Variables y constantes**
 - **Tipos de datos y cursores**
- Todo esto en un solo objeto, lo que permite **modularizar** y **reusar código**.
- Un package tiene **dos partes**:

ESPECIFICACIONES

- Aquí defines **qué es público**: funciones, procedimientos y variables que otros pueden usar.
- No se implementa la lógica aquí, solo la interfaz.
- **Ejemplo:**

```
CREATE OR REPLACE PACKAGE mi_paquete AS  
    PROCEDURE saludar(nombre IN VARCHAR2);  
    FUNCTION suma(a IN NUMBER, b IN NUMBER) RETURN NUMBER;  
END mi_paquete;  
/
```

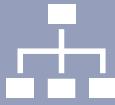
CUERPO

- Aquí defines **la implementación** de lo que declaraste en la especificación.

- **Ejemplo:**

```
CREATE OR REPLACE PACKAGE BODY mi_paquete AS
    PROCEDURE saludar(nombre IN VARCHAR2) IS
        BEGIN
            DBMS_OUTPUT.PUT_LINE('Hola, ' || nombre || '!');
        END saludar;
    FUNCTION suma(a IN NUMBER, b IN NUMBER)
    RETURN NUMBER IS
        BEGIN
            RETURN a + b;
        END suma;
    END mi_paquete;
```

VENTAJAS



Organización: Agrupa procedimientos, funciones y variables relacionadas.



Encapsulamiento: Puedes ocultar la implementación y exponer solo lo necesario.



Rendimiento: Oracle carga el package completo en memoria, lo que acelera su ejecución.



Reutilización: Varias aplicaciones pueden usar el mismo package.

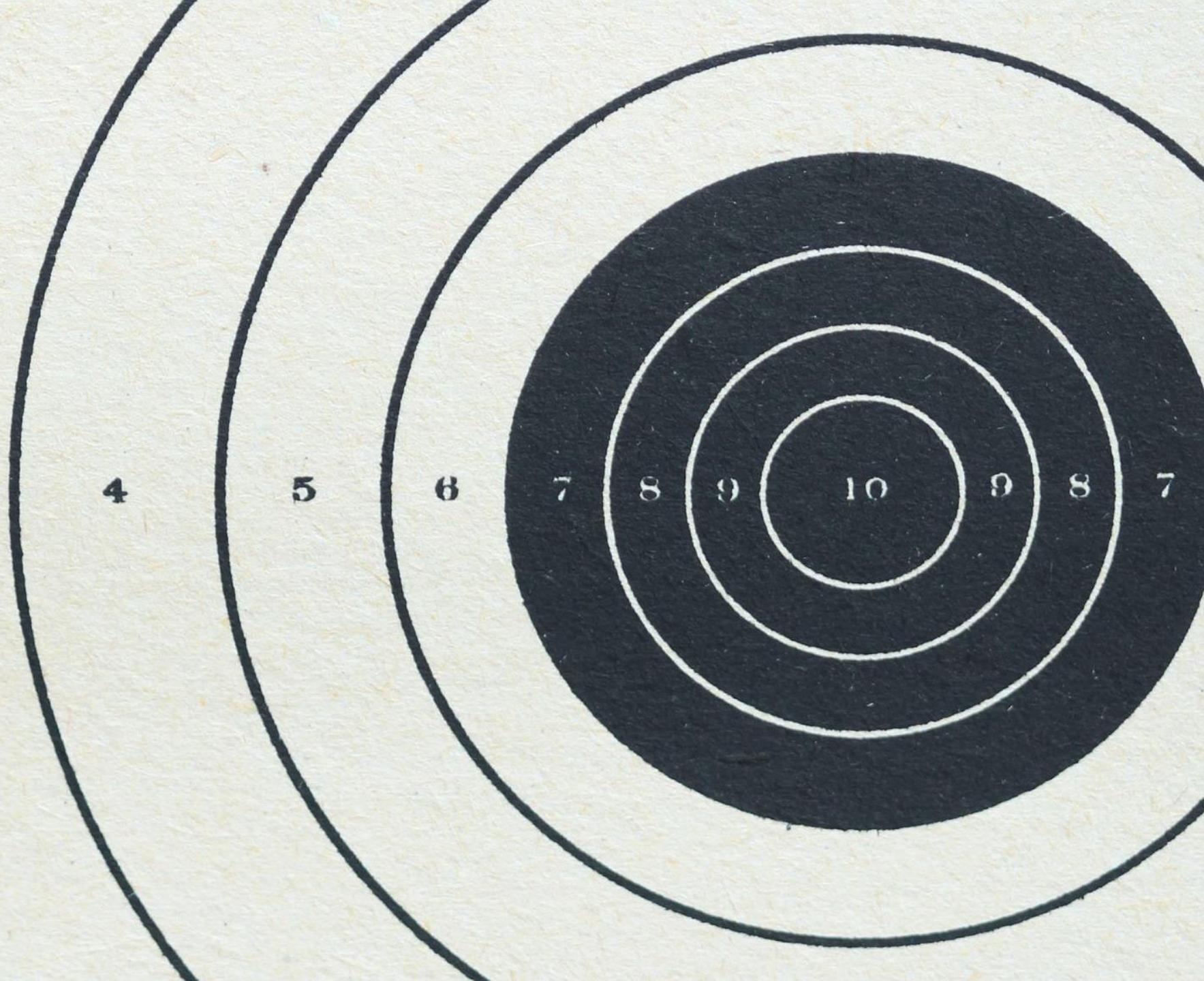
COMO USARLO

```
BEGIN
    -- Llamar a un procedimiento
    mi_paquete.saludar('Juan');

    -- Llamar a una función
DECLARE
    resultado NUMBER;
BEGIN
    resultado := mi_paquete.suma(5, 10);
    DBMS_OUTPUT.PUT_LINE('Resultado: ' || resultado);
END;
/

```

DEMO



LABORATORIO 6

