

SQL AVANZADO

“Para saber algo, no basta con haberlo aprendido”.

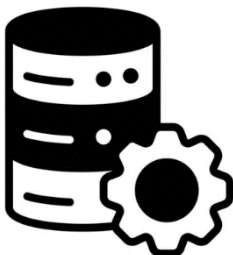
(Séneca)

PLANES DE EJECUCIÓN DE CONSULTAS

El gran poder de SQL reside en su naturaleza declarativa. El desarrollador le indica a la base de datos lo que desea obtener y el motor encuentra la mejor manera de mostrarlo. Ahora la cuestión es, si el motor decide como ejecutar la consulta ¿Cómo podemos hacer que las mismas sean más rápidas? Para saber eso necesitamos el plan de ejecución de la consulta.



Al enviar una consulta para que el motor la ejecute, el optimizador de SQL determina cómo ejecutarla, y este proceso es el que crea el plan de ejecución de la consulta. Un plan de ejecución es un mapa detallado de cómo Oracle recupera datos para completar una consulta SQL. Muestra el orden de las operaciones, los métodos de acceso a las tablas y los índices utilizados.



Nota

Un optimizador de consultas es un componente de un sistema de base de datos cuya función es determinar un plan para ejecutar una consulta lo más rápido posible basándose en la estructura de la consulta, la información estadística disponible sobre los objetos subyacentes y todas las funciones para su correcta ejecución.

Por ejemplo, supongamos la siguiente consulta:

```
SELECT e.employee_id, e.first_name, e.last_name, e.email,
e.salary,d.department_name
FROM employees e INNER JOIN departments d ON e.department_id = d.department_id
ORDER BY e.last_name ASC;
```

Ahora ejecutemos un plan de ejecución básico:

```
EXPLAIN PLAN FOR
SELECT e.employee_id, e.first_name, e.last_name, e.email,
e.salary,d.department_name
FROM employees e INNER JOIN departments d ON e.department_id = d.department_id
ORDER BY e.last_name ASC;
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY);
```

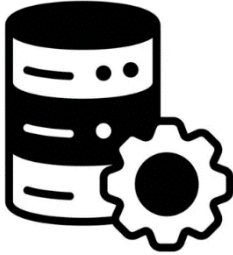
Una salida puede ser:

Plan hash value: 3011238288

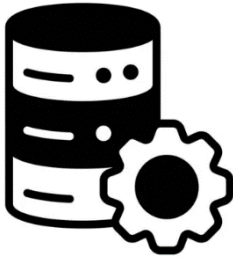
Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		108	6372	7 (29)	00:00:01
1	SORT ORDER BY		108	6372	7 (29)	00:00:01
2	MERGE JOIN		108	6372	6 (17)	00:00:01
3	TABLE ACCESS BY INDEX ROWID	DEPARTMENTS	28	448	2 (0)	00:00:01
4	INDEX FULL SCAN	DEPT_ID_PK	28		1 (0)	00:00:01
* 5	SORT JOIN		109	4687	4 (25)	00:00:01
6	TABLE ACCESS FULL	EMPLOYEES	109	4687	3 (0)	00:00:01

La instrucción EXPLAIN PLAN FOR le dice a Oracle que genere y almacene el plan de ejecución para la consulta sin ejecutarla realmente. El plan se guarda en una tabla del sistema llamada PLAN_TABLE. La instrucción DBMS_XPLAN.DISPLAY muestra la información de una manera legible. Cada fila de esta salida tiene un orden (Se lee como un árbol, de las hojas hacia la raíz) y cada fila es una operación independiente. Estas operaciones para poder ejecutarse están vinculadas. Cada fila se ejecuta de adentro hacia afuera (mayor a menor indentación). La declaración SELECT es la raíz del árbol de ejecución. Primero se accede por el índice de la tabla DEPARTMENTS escaneándolo (28 Filas), luego accede a la tabla EMPLOYEES escaneándola completamente. (109 Filas). Hace un JOIN entre ambas tablas (MERGE JOIN) usando department_id. (108 Filas), Luego hay un ordenamiento por la columna LAST_NAME. Finalmente devuelve las 108 Filas (SELECT STATEMENT). Fíjese que el plan de muestra, indica un escaneo completo a la tabla. En el contexto de Oracle SQL, la cardinalidad y el coste son conceptos importantes que utiliza el optimizador de SQL para evaluar y elegir el plan de ejecución más eficiente para una consulta. La cardinalidad es una estimación aproximada del número de filas devueltas por operación. Con una cardinalidad más alta se obtendrán más filas, se realizará más trabajo y la consulta tardará más. El costo es la cantidad estimada de trabajo que realizará el plan. Es un valor numérico que representa el consumo estimado de recursos (CPU, memoria, E/S) necesarios para ejecutar una consulta. El optimizador utiliza el valor del costo para comparar diferentes planes de ejecución posibles para una consulta y elige el que tiene el costo más bajo. El costo no siempre es igual al tiempo de ejecución.

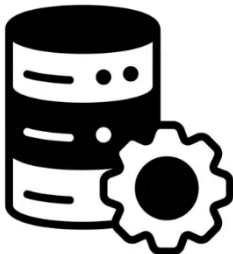
Representa el gasto relativo estimado, no el tiempo real, pero, por lo general, un menor costo implica una ejecución más rápida de la consulta. En el ejemplo anterior para saber el costo debe leer el ID 0 (SELECT STATEMENT) que devuelve en este caso el valor 7. Este plan tiene un costo moderadamente bueno, pero puede mejorarse eliminando el table access full en EMPLOYEES. Una solución para mejorar la performance seria poner un índice compuesto. Al igual que un índice único, un índice compuesto también es una estructura de datos ordenados según un valor. Sin embargo, a diferencia de un índice único, ese valor no es un campo, sino una concatenación de varios campos.

**Nota**

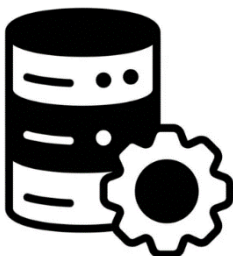
Los índices permiten a Oracle encontrar filas más rápidamente. Asegúrese de que existan índices en las columnas que se usan con frecuencia en las cláusulas WHERE, JOIN o ORDER BY.

**Nota**

Verifique el plan de ejecución para asegurarse de que la consulta utilice índices INDEX FULL SCAN en lugar de TABLE ACCESS FULL.

**Nota**

Evite seleccionar mas campos de los necesarios. Usar SELECT * cuando no se necesitan todas las columnas puede aumentar la E/S y ralentizar la consulta.

**Nota**

Las consultas pueden ser lentas si los índices no se utilizan eficientemente. Por ejemplo, aplicar una función a una columna en una cláusula WHERE impide el uso del índice. Si es necesaria una función, cree un índice basado en funciones:

```
CREATE INDEX idx_lastname_upper  
ON empleados ( UPPER (last_name));
```

Por ejemplo, podemos crear un índice compuesto para esta consulta que puede usarse en forma periódica:

```
CREATE INDEX idx_emp_covering ON employees(department_id, last_name,  
employee_id, first_name, email, salary);
```

¿Por qué este orden? El campo `department_id` es necesario para el JOIN, el campo `last_name` porque es usado por ORDER BY y el resto para crear un covering index. Fíjese como mejora el plan de ejecución:

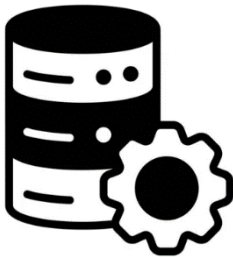
Plan hash value: 1004167179

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		108	6372	5 (40)	00:00:01
1	SORT ORDER BY		108	6372	5 (40)	00:00:01
2	MERGE JOIN		108	6372	4 (25)	00:00:01
3	TABLE ACCESS BY INDEX ROWID	DEPARTMENTS	28	448	2 (0)	00:00:01
4	INDEX FULL SCAN	DEPT_ID_PK	28		1 (0)	00:00:01
* 5	SORT JOIN		109	4687	2 (50)	00:00:01
6	INDEX FULL SCAN	IDX_EMP_COVERING	109	4687	1 (0)	00:00:01

Los índices compuestos, al igual que los individuales, implican velocidades de escritura más lentas y requieren mayor espacio de almacenamiento. Para determinar qué campos deben formar parte de un índice compuesto y su orden óptimo, es necesario considerar los siguientes aspectos:

- Si ciertos campos suelen aparecer juntos en las consultas, conviene crear un índice compuesto para ellos.
- Si vamos a crear un índice en el campo, `field1`, pero también un índice compuesto en (`field1`, `field2`), basta con crear solo el índice compuesto en (`field1`, `field2`) ya que puede usarse para realizar consultas `field1` únicamente.

El Costo total de este plan de ejecución es 5 mucho mejor que el plan anterior.



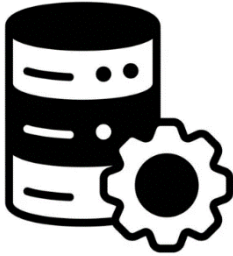
Nota

Si quisiera ver los índices de la tabla `employees` debería escribir la siguiente instrucción: `SELECT index_name, status, visibility, last_analyzed FROM user_indexes WHERE table_name = 'EMPLOYEES';`

TABLAS TEMPORALES GLOBALES

Además de las tablas permanentes, Oracle permite la creación de tablas temporales globales (GTT). Una tabla temporal en Oracle es un tipo especial de tabla que almacena datos temporales durante una sesión o transacción. Se utilizan ampliamente para almacenar resultados intermedios, procesar grandes conjuntos de datos y aislar datos temporales de las tablas permanentes. La sentencia `CREATE GLOBAL TEMPORARY TABLE` crea una tabla temporal cuya temporalidad puede ser definida a nivel de transacción (los datos existen mientras se realiza la transacción) o a nivel de sesión (los datos existen mientras dura la sesión).

Para especificar si los datos de una tabla temporal son por sesión o por transacción, a la hora de crear la definición de la tabla, utilizaremos la cláusula `ON COMMIT DELETE ROWS` para indicar que la temporalidad es a nivel de transacción, o la cláusula `ON COMMIT PRESERVE ROWS` si queremos que la temporalidad sea a nivel de sesión.



Nota

Las tablas temporales se eliminan como tablas normales. Aunque sean temporales sus estructuras son permanentes. Solo los datos son temporales no la tabla en sí. Para eliminar una tabla temporal debe usar el comando `DROP TABLE`.

BENEFICIOS DE LAS TABLAS TEMPORALES

Hay una multitud de beneficios cuando se trata de tablas temporales, algunos de los más importantes son:

- **SEGURIDAD DE DATOS:** El uso de tablas temporales permite el acceso de todos los usuarios a una tabla, limitando su visibilidad y protegiendo la información confidencial. (Esto se puede lograr omitiendo ciertas columnas de la tabla principal en la tabla temporal).
- **ESCRITURA REDUCIDA:** Otra ventaja es usar una tabla temporal cuando es necesario escribir el mismo código complejo una y otra vez para obtener un resultado. Un buen ejemplo son las sentencias `JOINS`, donde dos o más tablas deben unirse con frecuencia. En lugar de escribir la unión una y otra vez, simplemente introdúzcala en una tabla temporal.
- **INDEXACIÓN / RENDIMIENTO MEJORADO:** Las tablas temporales se pueden indexar, lo que permite optimizar aún más el rendimiento de las consultas que implican búsquedas u ordenaciones basadas en columnas específicas. Esto puede aumentar significativamente la velocidad de recuperación de datos.

Ahora haremos un ejemplo sobre tabla temporales:

```
CREATE GLOBAL TEMPORARY TABLE gtt_employees_salary (  
    employee_id    NUMBER,  
    full_name      VARCHAR2(100),  
    current_salary NUMBER,  
    proposed_salary NUMBER  
) ON COMMIT DELETE ROWS;
```

En este ejemplo, creamos una tabla llamada `gtt_employees_salary` con cuatro columnas: `employee_id`, `full_name`, `current_salary` y `proposed_salary`. La cláusula `ON COMMIT DELETE ROWS` significa que los datos se eliminarán después de cada transacción (es decir, después de cada `COMMIT`). Después de crear la tabla vera el siguiente mensaje:

```
Global temporary TABLE creado.
```

Podemos hacer la siguiente consulta para comprobar que es una tabla temporaria:

```
SELECT object_type, temporary
FROM user_objects
WHERE object_name = 'GTT_EMPLOYEES_SALARY';
```

Ahora insertaremos dos registros:

```
INSERT INTO gtt_employees_salary (employee_id, full_name,
current_salary,proposed_salary)
VALUES (101, 'Martin Bach', 8000,8500);

INSERT INTO gtt_employees_salary (employee_id, full_name,
current_salary,proposed_salary)
VALUES (102, 'Anders Donner', 11000,12500);
```

Ahora haremos un select:

```
SELECT * FROM gtt_employees_salary;
```

Y veremos los registros insertados. Ahora haremos un commit:

```
commit;
```

El nuevo mensaje es el siguiente:

```
Confirmación terminada.
```

Después de ejecutar la declaración COMMIT, se eliminarán los datos en gtt_employees_salary, ya que especificamos ON COMMIT DELETE ROWS. Cuando hagamos un select:

```
SELECT count(*) FROM gtt_employees_salary;
```

Veremos que indica 0 registros. Si desea que los datos persistan durante toda la sesión, puede definir la tabla de la siguiente manera:

```
CREATE GLOBAL TEMPORARY TABLE gtt_employees_salary_data (  
    employee_id    NUMBER,  
    full_name      VARCHAR2(100),  
    current_salary NUMBER,  
    proposed_salary NUMBER  
) ON COMMIT PRESERVE ROWS;
```

En este caso, los datos permanecerán en la tabla hasta que se cierre la sesión, incluso si confirma la transacción. Ahora insertemos los mismos registros que antes:

```
INSERT INTO gtt_employees_salary_data (employee_id, full_name,  
current_salary,proposed_salary)  
VALUES (101, 'Martin Bach', 8000,8500);  
  
INSERT INTO gtt_employees_salary_data (employee_id, full_name,  
current_salary,proposed_salary)  
VALUES (102, 'Anders Donner', 11000,12500);
```

Hagamos un select:

```
SELECT * FROM gtt_employees_salary_data;
```

Y veremos los registros insertados. Ahora haremos un commit:

```
commit;
```

El nuevo mensaje es el siguiente:

```
Confirmación terminada.
```

Cuando hagamos un select:

```
SELECT count(*) FROM gtt_employees_salary_data;
```

Y veremos que existen los dos registros ingresados. Para poder eliminar los datos necesita cerrar la sesión desde el SQL Developer, para ello hacemos click derecho sobre la conexión abierta y elegimos la opción Desconectar.

Cuando se conecte nuevamente y realice el siguiente comando:

```
SELECT * FROM gtt_employees_salary_data;
```

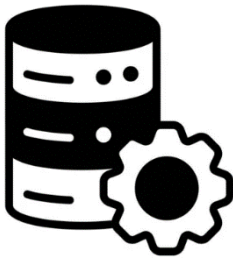

Vera que la tabla no tienen registros. También puede crear una tabla basado en registros anteriores de otra tabla:

```
CREATE GLOBAL TEMPORARY TABLE gtt_high_salary_employees
ON COMMIT PRESERVE ROWS
AS SELECT employee_id, first_name, last_name, department_id, salary
FROM employees
WHERE salary > 5000;
```

Esta nueva tabla denominada gtt_high_salary_employees contiene los registros de la tabla EMPLOYEES cuyo salario de los empleados es mayor a 5000. Si hacemos un select sobre la tabla gtt_high_salary_employees:

```
SELECT * FROM gtt_high_salary_employees;
```

Vera los registros que cumplen dicha condición de la tabla EMPLOYEES sobre la nueva tabla gtt_high_salary_employees



Nota

Oracle hace un COMMIT automático después de comandos DDL como CREATE TABLE. Si usa ON COMMIT DELETE ROWS en el ejemplo anterior no podrá ver los registros que cumplan la condición ya que se eliminan automáticamente.

COMMON TABLE EXPRESSIONS (CTE)

Las expresiones de tabla comunes (CTE) son una característica poderosa de SQL que le permite definir conjuntos de resultados temporales que se pueden usar dentro de una consulta más amplia. Proporcionan una forma de dividir consultas complejas en partes más manejables y comprensibles, lo que mejora la legibilidad y la facilidad en el mantenimiento. No se almacenan en el disco y se recalculan cada vez que se invocan dentro de la consulta. Esto puede ser beneficioso para subconsultas complejas que se utilizan varias veces en una consulta más grande. Una CTE se implementa mediante la cláusula WITH, que forma parte del estándar ANSI SQL-99 y existe en Oracle desde la versión 9.2. La diferencia principal con las subconsultas es que puede darle un nombre a ese resultado y hacerlo referenciar a él varias veces dentro de la consulta principal. Puede utilizar una CTE dentro de los siguientes tipos de consultas:

- SELECT
- INSERT
- UPDATE
- DELETE

La sintaxis de este tipo de consultas es la siguiente:

```
WITH cte_name (column1, column2, ...) AS (  
    -- CTE query here  
    SELECT column1, column2, ...  
    FROM table_name  
    WHERE conditions  
)  
-- Main query using the CTE  
SELECT *  
FROM cte_name;
```

Para poder ver un poco mejor este tipo de consultas intentemos por otros medios (como una subconsulta) mostrar los empleados cuyos salarios superan al promedio de su departamento.

```
SELECT e.first_name,  
e.last_name,e.department_id, e.salary,  
round(d.avg_salary,2) as "Salario Promedio"  
FROM employees e  
INNER JOIN (  
    SELECT department_id, AVG(salary) AS avg_salary  
    FROM employees  
    GROUP BY department_id  
) d  
ON e.department_id = d.department_id  
WHERE e.salary>d.avg_salary;
```

En la subconsulta resolvemos el promedio de masa salarial de cada departamento. Este resultado me servirá para la consulta principal donde mostrará solo los salarios mayores al promedio obtenido en la subconsulta anterior. Si usáramos una CTE le permitirá mover la subconsulta y definirla por separado. La consulta tendrá el siguiente aspecto:

```
WITH depto_avg_salary AS(  
    SELECT department_id, AVG(salary) AS avg_salary  
    FROM employees  
    GROUP BY department_id  
)  
SELECT e.first_name,  
e.last_name,e.department_id, e.salary, round(d.avg_salary,2) as "Salario  
Promedio"  
FROM employees e  
INNER JOIN depto_avg_salary d  
ON e.department_id = d.department_id  
WHERE e.salary>d.avg_salary;
```

Vamos a desglosarlo. Este es el CTE:

```
WITH depto_avg_salary AS (  
  SELECT department_id, AVG(salary) AS avg_salary  
    FROM employees  
   GROUP BY department_id  
)
```

Podemos saber que es una CTE porque comienza con una palabra clave **WITH**. Hemos definido esta consulta **SELECT** utilizando la palabra clave **WITH** y le hemos dado el nombre **depto_avg_salary**. El resto de la consulta viene después del cierre del paréntesis y es la consulta principal:

```
SELECT e.first_name,  
       e.last_name,e.department_id, e.salary, round(d.avg_salary,2) as "Salario  
       Promedio"  
FROM employees e  
INNER JOIN depto_avg_salary d  
ON e.department_id = d.department_id  
WHERE e.salary>d.avg_salary;
```

Esta es la consulta que finalmente se ejecuta y toma el conjunto de resultados de la anterior CTE. Como la CTE tiene un nombre y columnas se puede tratar como si fuera una tabla temporal.

MÚLTIPLES CTEs EN UNA SIMPLE CONSULTA

Puede definir múltiples CTE en una sola consulta, lo cual resulta especialmente útil para abordar problemas complejos donde cada CTE representa una etapa diferente en la resolución del problema. Este tipo de consulta permite descomponer consultas complejas en pasos más manejables y comprensibles, facilitando tanto el desarrollo como el mantenimiento del código SQL. Cada CTE actúa como un bloque de código que puede ser referenciado por los CTEs posteriores, creando un flujo lógico y secuencial en la resolución del problema. Por ejemplo, supongamos que queremos encontrar los empleados que fueron contratados en el año 2013 y que además han tenido más de un puesto de trabajo a lo largo de su historial dentro de la empresa.

```
WITH
cte_departaments_2013 AS (
  -- Primer CTE: Devuelve el employee_id que fue contratado en el año 2013.
  SELECT
    employee_id
  FROM
    EMPLOYEES
  WHERE EXTRACT(YEAR FROM hire_date) = 2013
),
cte_multiple_jobs AS (
  -- Segundo CTE: Empleados que han tenido más de un puesto dentro de la
  -- empresa.
  SELECT
    employee_id,
    COUNT(job_id) AS count_jobs
  FROM
    job_history
  GROUP BY
    employee_id
  HAVING
    COUNT(job_id) > 1
)
-- Consulta final: Une los resultados de los CTEs con la tabla de empleados.
-- Devuelve los empleados que fueron contratados en el 2013 y tuvieron más de
-- un puesto en la organización.
SELECT
  e.first_name,
  e.last_name,
  cte_jobs.count_jobs
FROM
  employees e
  INNER JOIN cte_departaments_2013 cte_depto
    ON e.employee_id = cte_depto.employee_id
  INNER JOIN cte_multiple_jobs cte_jobs
    ON e.employee_id = cte_jobs.employee_id
ORDER BY
  e.last_name;
```

El primer CTE devuelve los legajos de los empleados que fueron contratados en el año 2013. El segundo CTE analiza la tabla `job_history` buscando los empleados que han tenido más de un trabajo. La cláusula `HAVING` es crucial aquí, ya que filtra el resultado para incluir únicamente a aquellos empleados que han tenido más de un puesto de trabajo. La consulta principal une la tabla `employees` con los dos CTEs que hemos creado anteriormente. El resultado final es el nombre, el apellido, el departamento que trabaja y la cantidad de trabajos de los empleados que cumplen ambas condiciones.

CONSULTAS RECURSIVAS USANDO CTE

Una consulta SQL recursiva es un tipo especial de consulta que se llama a sí misma repetidamente hasta que no se encuentran nuevos resultados. Se usan para procesar datos jerárquicos. Algunos ejemplos de datos jerárquicos incluyen la estructura de una organización (jefes/empleados) o, por ejemplo, un conjunto de tareas que se ejecutan en un proyecto (tareas/subtareas). La recursión se logra mediante una CTE (Common Table Expressions). Esta permite nombrar el resultado y referenciarlo posteriormente en otras consultas. Para que la recursión funcione, necesitamos empezar con algo y decidir cuándo debe terminar. La estructura básica de una consulta recursiva en ORACLE es la siguiente:

```
WITH cte_name (column_names) AS (  
    -- Anchor member  
    SELECT ...  
    UNION ALL  
    -- Recursive member  
    SELECT ...  
)  
SELECT * FROM cte_name;
```

El primer SELECT del CTE es denominado Anchor member y selecciona las filas iniciales para la recursión.

El operador UNION ALL combina los resultados del Anchor member y el Recursive member. El segundo SELECT del CTE es la parte recursiva, que define la relación entre las filas principales y secundarias.

Por ejemplo, supongamos que queremos visualizar en distintos niveles la estructura de los empleados de nuestra organización del esquema HR:

```
WITH employee_hierarchy (EmployeeID, FirstName, LastName, ManagerID, Nivel) AS (  
    -- Anchor member  
    SELECT employee_id, first_name, last_name, manager_id, 1 as nivel  
    FROM employees  
    WHERE manager_id IS NULL  
    UNION ALL  
    -- Recursive member  
    SELECT e.employee_id, e.first_name, e.last_name, e.manager_id, eh.nivel+ 1  
    FROM employees e  
    INNER JOIN employee_hierarchy eh ON e.manager_id = eh.employeeID  
)  
SELECT * FROM employee_hierarchy
```

Esta consulta devolverá todas las columnas de la CTE denominada employee_hierarchy o sea EmployeeID, FirstName, LastName, ManagerID y Nivel. Para crear una consulta SQL recursiva, deberá definir una CTE con dos partes: el Anchor member y el Recursive member. El Anchor member establece el punto de inicio de la recursión.

Esta primera consulta se fija en los empleados cuyo `manager_id` es NULL y le asigna el nivel 1 (O sea muestra al presidente o los directores de la empresa, ya que no tienen jefe). Luego continua la consulta ya que el operador UNION ALL permite combinar los resultados de las dos consultas dentro de la CTE. La segunda parte de la consulta CTE, interviene un llamado a la misma CTE tomando el resultado anterior. Cuando el resultado anterior no tiene mas datos para trabajar la recursión se detiene.



Nota

Las CTE recursivas pueden consumir muchos recursos si la profundidad de recursión es alta o se procesan grandes conjuntos de datos.

PIVOTS EN SQL

Las tablas dinámicas son una herramienta eficaz para el análisis de datos, ya que permiten organizar y resumir grandes conjuntos de datos de forma coherente. En SQL, una tabla dinámica es una técnica utilizada para transformar datos de filas a columnas. ORACLE puede crear tablas dinámicas mediante los operadores PIVOT y UNPIVOT. La pivotación de datos, es una tarea común en el análisis de datos, implica transformar filas en columnas para obtener nuevas perspectivas. Por ejemplo, supongamos que tenemos la siguiente información sobre distintas temperaturas promedios obtenidas en distintos meses y años:

YEAR_NUM	MONTH_NUM	AVG_TEMP
2017	1	27
2017	2	26
2017	3	26
2017	4	24
2017	5	21
2017	6	17
2018	1	28
2018	2	24
2018	3	25
2018	4	24
2018	5	22
2018	6	18
2018	7	16
2019	1	27
2019	2	28
2019	3	28
2019	4	26
2019	5	24
2019	6	22

Un investigador podría querer visualizar estos datos de una mejor forma. Una tabla dinámica sería útil en este caso, con dos dimensiones de datos: los años a analizar y los meses. De esta forma identificaría patrones y tendencias. Con esta información, el investigador puede tomar decisiones basadas en datos, como saber como viene la temperatura en invierno:

YEAR_NUM	ENERO	FEBRERO	MARZO	ABRIL	MAYO	JUNIO	JULIO
2017	27	26	26	24	21	17	
2018	28	24	25	24	22	18	16
2019	27	28	28	26	24	22	20

Es decir, los años en la primera columna (mostrándose una sola vez) y los distintos meses serían las otras columnas.

YEAR_NUM	MONTH_NUM	AVG_TEMP
2017	1	27
2017	2	26
2017	3	26
2017	4	24
2017	5	21
2017	6	17
2018	1	28
2018	2	24
2018	3	25
2018	4	24
2018	5	22
2018	6	18
2018	7	16
2019	1	27
2019	2	28
2019	3	28
2019	4	26
2019	5	24
2019	6	22

Transformación de filas a columnas



YEAR_NUM	ENERO	FEBRERO	MARZO	ABRIL	MAYO	JUNIO	JULIO
2017	27	26	26	24	21	17	
2018	28	24	25	24	22	18	16
2019	27	28	28	26	24	22	20

La sintaxis del operador PIVOT en Oracle es la siguiente:

```
SELECT *
FROM (
    SELECT column1, column2, column3
    FROM table_name
)
PIVOT (
    aggregate_function(column3)
    FOR column2 IN (value1, value2, ..., valueN)
) Alias;
```

Por ejemplo, supongamos que tenemos esta consulta hecha con la instrucción CASE:

```
SELECT  d.department_name,  
        CASE  
            WHEN e.salary < 5000 THEN 'Bajo'  
            WHEN e.salary BETWEEN 5000 AND 10000 THEN 'Medio'  
            ELSE 'Alto'  
        END as rango_salarial,  
        e.employee_id  
FROM employees e  
INNER JOIN departments d ON e.department_id = d.department_id  
WHERE d.department_name IN ('IT', 'Sales', 'Finance')  
ORDER BY d.department_name;
```

En el mismo se muestra un listado detallado de cada empleado de los departamentos IT, Sales y Finance con tres columnas, en la primera se muestra el nombre del departamento donde trabaja el empleado, en la segunda a través de una estructura CASE se calcula si el empleado obtiene un salario bajo, medio o alto y en la tercera se muestra el legajo del empleado. Ahora vamos a usar el operador PIVOT de Oracle. mostraremos un resumen estadístico de cuantos empleados de los departamentos IT, Sales y Finance hay por cada rango salarial. Para ello las columnas serán cada uno de los rangos salariales obtenidos (bajo, medio y alto) y las filas serán solo tres (una por cada departamento: Finance, IT, Sales):

```
SELECT * FROM (  
    SELECT  
        d.department_name,  
        CASE  
            WHEN e.salary < 5000 THEN 'Bajo'  
            WHEN e.salary BETWEEN 5000 AND 10000 THEN 'Medio'  
            ELSE 'Alto'  
        END as rango_salarial,  
        e.employee_id  
    FROM employees e  
    INNER JOIN departments d ON e.department_id = d.department_id  
    WHERE d.department_name IN ('IT', 'Sales', 'Finance')  
)  
PIVOT (  
    COUNT(employee_id)  
    FOR rango_salarial IN (  
        'Bajo' AS Salario_Bajo,  
        'Medio' AS Salario_Medio,  
        'Alto' AS Salario_Alto  
    )  
)  
ORDER BY department_name;
```


La consulta interna relaciona las tablas employees y departments a través de un INNER JOIN, el WHERE se encarga de filtrar solo los departamentos IT, Sales y Finance y el CASE clasifica a cada empleado según su salario. ¿Qué hace el Operador PIVOT? Cuenta cuantos empleados hay en cada clasificación, el FOR..IN toma los valores Salario_Bajo, Salario_Medio y Salario_Alto y los transforma en columnas separadas. Colocando debajo de cada columna la cantidad obtenida. La sintaxis del operador PIVOT es sencilla. Primero, se aplica una función de agregación a la columna cuyos valores se desean mostrar en las columnas pivotadas. En nuestro caso, queremos mostrar las columnas Salario_Bajo, Salario_Medio y Salario_Alto. Finalmente, se utiliza una sentencia FOR para especificar la columna pivotada y sus valores.

DEPARTMENT_NAME	SALARIO_BAJO	SALARIO_MEDIO	SALARIO_ALTO
IT	3	2	1
Sales	0	26	8
Finance	0	5	1

¿QUE ES UN TRIGGER?

Un disparador o un trigger es una pieza de código que se almacena en la base de datos y se activa cuando ocurre un evento específico. El evento asociado que hace que se ejecute un trigger puede vincularse a una tabla de base de datos específica, una vista de la base de datos, un esquema de base de datos o la propia base de datos. También se lo definen como procedimientos especiales porque no se puede llamar directamente como otros procedimientos almacenados en SQL.

¿PARA QUE SE USAN LOS TRIGGERS?

Los triggers son herramientas esenciales tanto para los administradores como para los desarrolladores de bases de datos porque pueden garantizar la integridad de los datos, automatizar procesos y hacer cumplir las reglas de negocios.

Mantener la integridad de la base de datos

Los disparadores se pueden usar para garantizar que los datos en una base de datos permanezcan consistentes y precisos. Por ejemplo, puede definir un disparador de SQL para asegurarse de que el valor de una clave foránea exista cuando se inserta un nuevo registro.

Automatización de tareas

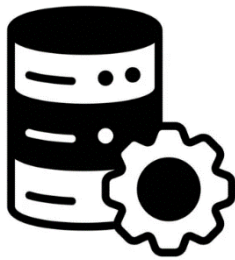
Los disparadores son excelentes para automatizar tareas en una base de datos y evitar realizar tareas de programación. Por ejemplo, puede usar disparadores de SQL para registrar datos automáticamente en una tabla.

Hacer cumplir las reglas de negocio

Los disparadores de SQL se pueden usar para hacer cumplir automáticamente las reglas de negocios de la base de datos. Por ejemplo, se puede usar un disparador para garantizar que el precio de un producto nunca se establezca por debajo de su costo.

Los disparadores se pueden ejecutar en:

- Sentencias DML como DELETE, INSERT o UPDATE.
- Sentencias DDL como CREATE, ALTER o DROP.
- Eventos y operaciones de base de datos como: SERVERERROR, LOGON, LOGOFF, STARTUP o SHUTDOWN.

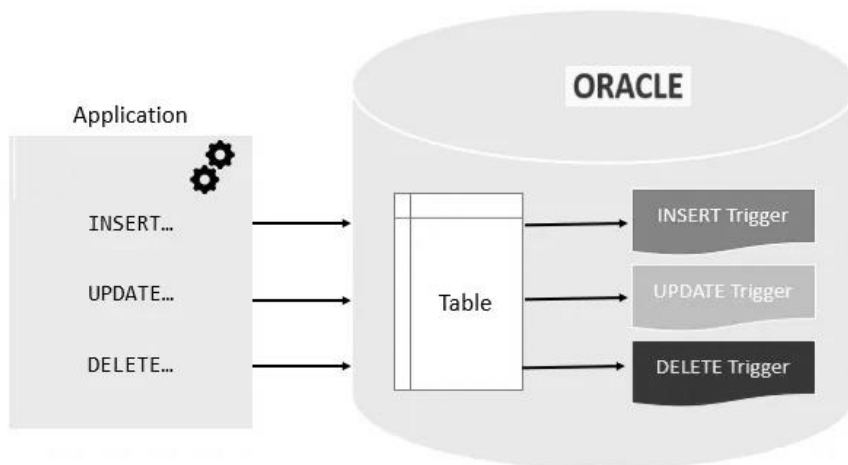


Nota

No todas son ventajas. Mal usados los triggers pueden afectar negativamente el rendimiento especialmente en bases de datos con un alto volumen de transacciones.

TRIGGERS DML

El principal tipo de TRIGGER son los TRIGGERS DML. Se llaman así porque se ejecutan cuando se ejecuta una instrucción DML (INSERTAR, ACTUALIZAR o ELIMINAR). Puede programar su disparador para que se ejecute ANTES o DESPUÉS de ejecutar su instrucción DML. Por ejemplo, puede crear un trigger que se activará antes de la actualización. De manera similar, puede crear un trigger que se activará después de la ejecución de su instrucción INSERT DML.



Para crear un trigger, se utiliza la sentencia CREATE TRIGGER. Los detalles de la instrucción cambiarán dependiendo de si está creando un trigger DML, un trigger de esquema o un trigger de base de datos. Por ejemplo, para crear un trigger DML podemos usar la siguiente sintaxis:

```
CREATE [OR REPLACE] TRIGGER trigger_name
{BEFORE | AFTER} dml_event ON table_name
[FOR EACH ROW]
[DECLARE variables]
BEGIN
    pl_sql_code
    [EXCEPTION exception_code]
END;
```

Después del comando CREATE TRIGGER debemos indicar cuando debe ejecutarse el trigger (BEFORE/AFTER) indicando si el trigger se ejecutara ANTES o DESPUES del comando DML lanzado por el USUARIO, luego debemos indicaremos sobre que tabla actuará. La línea FOR EACH ROW indica que el trigger se activará para cada fila afectada. Finalmente, la sección BEGIN ... END contiene el código que se ejecutará cuando se active el trigger. Cuando se escribe el código de un trigger muchas veces debe hacerse referencia a los viejos y/o a los nuevos datos. Por ejemplo, cuando se realiza un UPDATE capaz que necesita hacer referencia a los viejos datos, así como también a los nuevos valores a actualizar. Estos datos se pueden referenciar usando dos variables llamadas: NEW y :OLD. La variable :OLD hace referencia al valor de una columna antes de que la incidencia se produzca. La variable :NEW hace referencia a una columna afectada por la incidencia, una vez que haya pasado. Por ejemplo, si desea hacer referencia a la columna first_name antes de que los datos se actualicen con una instrucción UPDATE, usaría lo siguiente

```
:old.first_name
```

Si desea hacer referencia al nuevo salario del empleado puede usar:

```
:new.salary
```

Para ver el uso de estas dos variables realizaremos el siguiente ejemplo, Este trigger denominado print_salary_changes se disparará antes de que se realice un UPDATE. La idea es que permita mostrar los cambios de salarios producidos por las distintas alteraciones.

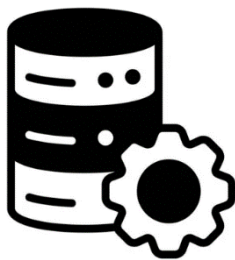
```
CREATE OR REPLACE TRIGGER print_salary_changes
BEFORE UPDATE ON employees
FOR EACH ROW
WHEN (new.employee_id > 0)
DECLARE
    sal_diff number;
BEGIN
    sal_diff := :new.salary - :old.salary;
    dbms_output.put_line('Old salary: ' || :old.salary);
    dbms_output.put_line('New salary: ' || :new.salary);
    dbms_output.put_line('Difference ' || sal_diff);
END;
```

Este trigger posee una sección de declaración donde se declara una variable `sal_diff` que contendrá la diferencia de sueldos. Como se puede apreciar este trigger mostrara el valor antes de la alteración (variable `:OLD.salary`), después de la alteración (variable `:NEW.salary`) y el valor de la diferencia entre ambas variables. Para poder ver el trigger en funcionamiento debe realizar un `UPDATE` sobre la tabla `employees`. Por ejemplo, podríamos realizar lo siguiente:

```
UPDATE employees SET salary=9000 WHERE employee_id=104
```

Y una posible salida en la ventana Salida DBMS seria:

```
Old salary: 8000 New salary: 9000 Difference 1000
```



Nota

Para listar los procedimientos creados debe usar la siguiente instrucción SQL:
`SELECT * FROM user_objects WHERE object_type='TRIGGER';`

Para poder hacer el siguiente ejercicio primero crearemos una tabla denominada `employees_audit`:

```
CREATE TABLE employees_audit(
    audit_id NUMBER GENERATED BY DEFAULT ON NULL AS IDENTITY PRIMARY KEY,
    employee_id NUMBER,
    action VARCHAR2(50),
    audit_date DATE default sysdate,
    created_by VARCHAR2(255)
);
```

Esta tabla contendrá información sobre los usuarios y los momentos en que inserten nuevos empleados en la tabla `employees`. Ahora falta construir un trigger:

```
CREATE OR REPLACE TRIGGER trg_employee_audit
AFTER INSERT ON employees
FOR EACH ROW
BEGIN
    INSERT INTO employees_audit(employee_id,action,created_by) VALUES
    (:new.employee_id,'insert',user);
END;
```

Este trigger se disparará después de haber insertado un nuevo empleado en la tabla `employees` (la variable del sistema `USER` almacenan el nombre del usuario). La idea del trigger es insertar en la tabla `employees_audit` información sobre el usuario que realizado el ingreso del nuevo empleado. Para probar esto realizaremos un comando `INSERT`:

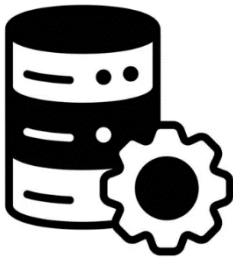
```
INSERT INTO employees VALUES (791,'Dave', 'Greenfield',  
'davegreen@example.com', '1134190112',TO_DATE('04/01/2020','DD/MM/YYYY')  
,12,10000,101,6);
```

Para comprobar si el procedimiento hace disparar al trigger debemos primero visualizar el listado de empleados:

```
SELECT * FROM employees;
```

Y luego comprobar si se insertó un nuevo registro en employees_audit:

```
SELECT * FROM employees_audit;
```



Nota

Es conveniente desactivar los triggers cuando se realiza una carga masiva de datos para ello debe usarse el siguiente comando: ALTER TRIGGER <nombre_del_trigger> DISABLE. En el caso que deba activarlo nuevamente debe escribir lo siguiente: ALTER TRIGGER <nombre_del_trigger> ENABLE.

También podemos hacer uso de predicados condicionales como INSERTING esto permite detectar una sentencia la sentencia DML INSERT para poder ejecutar una acción. Los predicados condicionales pueden ser: INSERTING, UPDATING o DELETING. Por ejemplo, ahora modificaremos el TRIGGER anterior reemplazándolo por esta nueva versión:

```
CREATE OR REPLACE TRIGGER trg_employee_audit  
AFTER INSERT OR DELETE ON employees  
FOR EACH ROW  
BEGIN  
    IF INSERTING THEN  
        INSERT INTO employees_audit(employee_id,action,created_by) VALUES  
        (:new.employee_id,'insert',user);  
    END IF;  
    IF DELETING THEN  
        INSERT INTO employees_audit(employee_id,action,created_by) VALUES  
        (:old.employee_id,'delete',user);  
    END IF;  
END;
```

Si probamos con instrucciones INSERT o DELETE sobre distintos registros podemos comprobar las nuevas auditorías en employees_audit:

```
SELECT * FROM employees_audit;
```

Podemos hacer un nuevo ejemplo probando nacimientos y fallecimientos sobre la tabla DEPENDENTS:

```
CREATE OR REPLACE TRIGGER trg_upd_dependents
AFTER INSERT OR DELETE ON dependents
FOR EACH ROW
BEGIN
    IF INSERTING THEN
        UPDATE employees SET salary = salary + 500 WHERE employee_id =
            :new.employee_id;
    END IF;
    IF DELETING THEN
        UPDATE employees SET salary = salary + 1500 WHERE employee_id =
            :old.employee_id;
    END IF;
END;
```

Por nacimiento se incrementa un salario de 500 dólares más, por fallecimiento un incremento de 1500 dólares. En el próximo ejercicio usaremos excepciones. La idea es verificar que el salario a modificar este entre los valores mínimos y máximos establecidos en la tabla JOBS.

```
CREATE OR REPLACE TRIGGER trg_check_sal_range
BEFORE UPDATE ON employees
FOR EACH ROW
DECLARE
    v_min_salary jobs.min_salary%TYPE;
    v_max_salary jobs.max_salary%TYPE;
BEGIN
    SELECT min_salary, max_salary
    INTO    v_min_salary, v_max_salary
    FROM    jobs WHERE job_id=:NEW.job_id;
    dbms_output.put_line(v_min_salary);
    dbms_output.put_line(v_max_salary);
    IF ((:NEW.salary<v_min_salary) OR (:NEW.salary>v_max_salary)) THEN
        raise_application_error(-20000,'Salario fuera de rango ');
    END IF;
END;
```

El código RAISE_APPLICATION_ERROR genera una excepción basada en un código de error y un mensaje proporcionados por el desarrollador. El código de error especificado acepta valores entre -20000 a -20999. Para probar este código podemos escribir lo siguiente:

```
UPDATE employees SET salary=3500 WHERE employee_id=110
```

Como el salario del job_id de este empleado tiene que estar entre 4200 y 9000. La excepción mandara un error y el cambio en el registro cuyo código de empleado es 110 no se realizará.

TRIGGERS DE BASE DE DATOS

Estos tipos de disparadores se ejecutan cada vez que ocurre un evento específico de la base de datos. Para poder desarrollar un ejemplo debemos crear la siguiente tabla sobre auditorías:

```
CREATE TABLE logon_audit (  
    id NUMBER GENERATED BY DEFAULT ON NULL AS IDENTITY,  
    current_date DATE NOT NULL,  
    current_username VARCHAR2(255),  
    CONSTRAINT pk_log_audit PRIMARY KEY (id)  
)
```

Una vez creada la tabla crearemos el siguiente trigger:

```
CREATE OR REPLACE TRIGGER trg_logon_audit  
AFTER LOGON ON DATABASE  
BEGIN  
    INSERT INTO logon_audit (current_username, current_date)  
    VALUES (USER, SYSDATE);  
END;
```

Este trigger se disparará después de un login cargando en la tabla de auditorías el nombre del usuario que inicio la sesión y el momento en que ocurrió dicho inicio.

ELIMINAR TRIGGERS

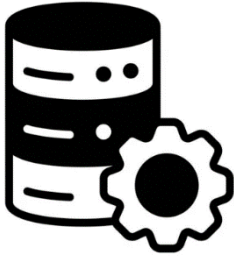
Como muchos otros objetos de la base de datos, los activadores se pueden eliminar. Esto se hace descartando el disparador mediante la sentencia DROP TRIGGER.

```
DROP TRIGGER <trigger_name>;
```

DICCIONARIO DE DATOS

Un diccionario de datos es una colección de tablas y vistas de solo lectura que brindan información útil sobre la base de datos incluyendo tablas, usuarios, vistas, índices, secuencias, procedimientos, disparadores, etc. El diccionario de datos consta de muchas tablas y vistas creadas por la instancia de la base de datos y es el motor de Oracle quien mantiene dicho diccionario de datos. Los esquemas de usuario generalmente no tienen privilegios a las tablas del diccionario, ORACLE otorga solo acceso SELECT a las vistas que comienzan con el prefijo USER_, ALL_ o DBA_:

- La vista **USER**: Devuelven información sobre objetos que son de su propiedad o que ha creado.
- La vista **ALL**: Devuelven información sobre los objetos a los que tiene acceso el usuario actual independientemente de su propietario.
- La vista **DBA**: Devuelven información sobre todos los objetos de la base de datos, independientemente de quién sea su propietario.



Nota

Para las vistas que tienen los prefijos ALL o DBA, suele existir una columna adicional en la vista denominada OWNER para identificar de quién es propiedad el objeto.

Veamos un ejemplo. Supongamos que quiero obtener una lista de los objetos (tablas, vistas, paquetes, etc.) definidos en la base de datos. En la siguiente consulta devuelve todos los objetos definidos en nuestro esquema:

```
SELECT * FROM user_objects
```

Esta otra consulta devuelve todos los objetos que están definidos en mi esquema o para los cuales se me ha otorgado el privilegio de usar esos objetos de alguna manera:

```
SELECT * FROM all_objects
```

Finalmente, la siguiente consulta devuelve una lista de todos los objetos definidos en la instancia de la base de datos (suponiendo que tengo los privilegios para hacer un select de la vista):

```
SELECT * FROM dba_objects
```

Por ejemplo, podemos listar las tablas que tenemos en nuestro esquema tipeando el siguiente comando:

```
SELECT table_name FROM user_tables ORDER BY table_name;
```

Este comando SQL devolverá una lista de todas las tablas en las que el usuario actual es propietario. Si está interesado en mostrar las tablas donde el usuario actual puede tener acceso debe tipear:

```
SELECT table_name FROM all_tables ORDER BY table_name;
```

Podemos listar también las columnas de una tabla específica:

```
SELECT column_name, data_type  
FROM user_tab_columns  
WHERE table_name = 'EMPLOYEES'  
ORDER BY column_id;
```

La vista USER_TAB_COLUMNS es una vista que muestra información detallada perteneciente al usuario que está actualmente conectado.

Es una vista fundamental para entender la estructura de tus propias tablas. En este caso listaremos las columnas y el tipo de datos de la tabla EMPLOYEES. También podemos buscar un campo determinado en las tablas que poseemos:

```
SELECT table_name, column_name, data_type
FROM user_tab_columns
WHERE column_name LIKE '%EMAIL%';
```

En este caso buscamos el campo EMAIL en todas nuestras tablas. También podemos buscar un tipo de dato en cualquier tabla que tengamos:

```
SELECT
    table_name,
    column_name,
    data_type
FROM
    user_tab_columns
WHERE
    data_type = 'DATE';
```

Como los procedimientos almacenados son objetos puede consultarse al diccionario de datos de esta manera:

```
select * from user_objects where object_type='PROCEDURE'
```

Los triggers pueden consultarse al diccionario de datos de la siguiente forma:

```
select * from user_objects where object_type='TRIGGER'
```

Si necesito acceder a todos los triggers definidos en la tabla EMPLOYEES debemos escribir lo siguiente:

```
SELECT *
FROM user_triggers
WHERE table_name = 'EMPLOYEES'
```

Si queremos mostrar todos los triggers que se disparan cuando se realiza una operación de actualización (UPDATE) debemos escribir lo siguiente:

```
SELECT *
FROM user_triggers
WHERE triggering_event LIKE '%UPDATE%'
```

Otra consulta interesante, es mostrar los objetos modificados en la fecha actual:

```
SELECT object_type, object_name,  
       last_ddl_time  
FROM user_objects  
WHERE last_ddl_time >= TRUNC (SYSDATE)  
ORDER BY object_type, object_name
```

Podemos listar también todos los objetos que dependen o hacen referencia a la tabla EMPLOYEES:

```
SELECT type, name  
FROM user_dependencies  
WHERE referenced_name = 'EMPLOYEES'  
ORDER BY type, name
```

GESTION DE USUARIOS EN ORACLE

Los usuarios son las personas que utilizarán la base de datos de Oracle. En Oracle Database incluye tanto usuarios comunes como locales. Un usuario común se crea en la base de datos container (CDB) y tiene el mismo nombre de usuario y contraseña en todas las PDB (pluggable databases) que forman parte de esa CDB. Los usuarios comunes pueden tener privilegios que se otorgan a nivel de contenedor y otros privilegios que se conceden en cada una de las bases "pluggable". Un privilegio es el derecho a ejecutar una tarea o consulta específica en una base de datos. Los privilegios pueden ser diferentes en cada una de las PDB, pero el usuario no necesita ser creado de manera individual en cada una de ellas. Un usuario local, por el contrario, se crea en la base de datos PDB y no tiene acceso al contenedor. Esto es bueno para el DBA que gestiona un PDB pero no administra el sistema en general. Para poder trabajar con estos comandos crearemos un usuario local. Para poder crear un usuario local debe conectarse Oracle como usuario System o SYS (esos usuarios son una de las pocas cuentas comunes administrativas predefinidas que se generan automáticamente cuando se instala Oracle). Si desea acceder a través de la herramienta SQLPLUS debe escribir el siguiente comando:

```
C:\>sqlplus system/maria123456
```

Con este comando nos conectamos al usuario administrador (system) con la contraseña que colocho en el momento de la instalación. Puede usar el siguiente comando para desplegar el usuario actual:

```
SQL> show user;
```

Al momento aparecerá el siguiente mensaje:

```
USER es "SYSTEM"
```

El usuario system es un usuario creado por Oracle para tareas de administración de base de datos. También podemos conectarnos como:

```
c:\> sqlplus / as sysdba
```

En este caso cuando escribamos el comando:

```
SQL> show user;
```

Nos mostrara:

```
USER es "SYS"
```

En este caso no fue necesario escribir la contraseña ya que se ejecuta desde el mismo servidor donde corre Oracle. Al usar el privilegio sysdba tiene los privilegios administrativos máximos. El usuario SYS es el usuario dueño de Oracle y debe usarse en emergencias y para operaciones críticas. La conexión también puede establecerse a través de SQL Developer, Si queremos conectarnos desde SQL Developer esta tendría que ser la configuración para el usuario sys:

Nueva / Seleccionar Conexión a Base de Datos

Nombre de Cone... Detalles de Cone...
sys sys@//DESKTOP-...

Name Color

Tipo de Base de Datos

Información de usuario Usuario de Proxy

Tipo de autenticación

Usuario Rol

Contraseña ☐ Guardar Contraseña

Tipo de Conexión

Detalles Avanzado

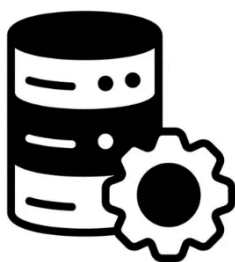
Nombre del Host

Puerto

☒ SID

☐ Nombre del Servicio

Estado:



Nota

El usuario SYS se utiliza para arrancar y parar la base de datos, para ejecutar recuperaciones y crear y eliminar PDBs.

Desde el usuario SYS podemos conectarnos como SYSTEM escribiendo el siguiente comando:

```
SQL> connect system/maria123456@localhost;
```

El comando SHOW CON_NAME muestra el contenedor al cual estoy conectado:

```
SQL> show con_name;
```

Al momento aparecerá el siguiente mensaje:

```
CON_NAME
```

```
-----
```

```
CDB$ROOT
```

El contenedor ROOT es el primer contenedor que se crea en una base de datos CDB. Obtiene el nombre predeterminado de CDB\$ROOT. Un contenedor ROOT está destinado a almacenar el diccionario de datos del sistema, los objetos utilizados internamente por las bases de datos y los propietarios creados por la base de datos en el momento de su creación, es decir, SYS y SYSTEM. También contiene información sobre todas las bases de datos conectables asociadas con él. Los usuarios comunes son aquellos que están disponibles no sólo dentro de un contenedor ROOT sino también en todas las bases de datos conectables o PDB. Para crear usuarios locales en Oracle, debe estar conectado a la base de datos conectable (PDB), no a la base de datos de contenedor (CDB). Con el siguiente comando mostramos los Pluggable Database (PDB) disponibles:

```
SELECT con_id, name from v$pdbs;
```

El resultado es el siguiente:

CON_ID	NAME
2	PDB\$SEED
3	FREEPDB1

Nuestra base de datos FREEPDB1 se encuentra en el container 3. El container 2 contiene la PDB denominada PDB\$SEED que viene de fábrica y es la semilla para la creación de nuevas PDBs. Por ejemplo, si quisiera crear una base de datos para el área de recursos humanos, usaría este contenedor inicial y crearía una nueva base de datos conectable. Dado que el contenedor semilla solo está destinado a usarse como plantilla para crear bases de datos conectables, siempre está en modo de solo lectura, lo que hace imposible cualquier cambio en sus objetos subyacentes. Los contenedores conectables son las bases de datos que interesarán a la mayoría de los DBA. La razón es que los PDB son creados por el DBA y se usarán para contener los datos relacionados con el negocio (por ejemplo, hay un PDB de Recursos Humanos y por ejemplo un PDB de Ventas). Cada PDB contiene datos de usuario, por lo que el diccionario de datos que contiene se utiliza para contener la información de estos objetos. Como se mencionó anteriormente, una CDB contiene usuarios comunes, mientras que una PDB contiene usuarios locales, es decir, usuarios que están confinados únicamente dentro de esa PDB.

Para crear un usuario local en una base de datos conectable o PDB se debe indicar la base de datos FREEPDB1. Para ello se debe tipear el siguiente comando:

```
SQL> ALTER SESSION SET CONTAINER=FREEPDB1;
```

Aparecerá el siguiente mensaje:

```
sesión modificada
```

Puede tipear el comando:

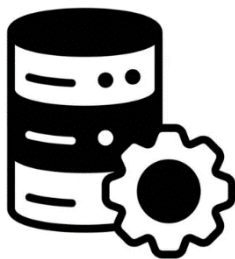
```
SQL> show con_name;
```

Y al momento aparecerá el siguiente mensaje:

```
CON_NAME
```

```
-----
```

```
FREEPDB1
```



Nota

Para cambiar a CDB, especifique el nombre CDB\$ROOT. Por ejemplo: ALTER SESSION SET container=CDB\$ROOT;

Puede listar todos los servicios en la base de datos, que son los nombres que se especifican cuando desea crear una nueva conexión. Esto es útil para tener una idea de los PDB en la base de datos y encontrar los detalles si desea crear una nueva conexión.

```
SQL> SELECT name, pdb FROM v$services;
```

El resultado es el siguiente:

```
NAME
```

```
PDB
```

```
-----
```

```
freepdb1
```

```
FREEPDB1
```

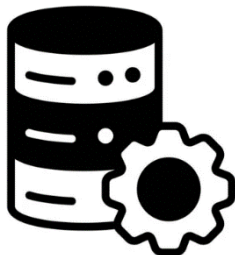
Una vez conectado como SYSTEM, simplemente use el comando CREATE USER para generar una nueva cuenta:

```
CREATE USER <username> IDENTIFIED BY <password>;
```

Por ejemplo, para crear un nuevo usuario llamado bob con la contraseña Superpassword:

```
CREATE USER bob IDENTIFIED BY Superpassword;
```

La contraseña puede contener letras, números y la mayoría de los caracteres especiales.



Nota

Si la contraseña contiene caracteres especiales, escríbala entre comillas dobles. Esto asegurará que el comando se ejecute correctamente.

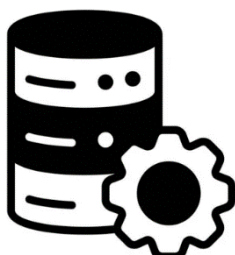
Después de haber lanzado el comando aparecerá el siguiente mensaje:

```
Usuario creado.
```

Ahora podemos crear una nueva conexión con este usuario. Si estamos usando la herramienta SQLPLUS debe escribir lo siguiente:

```
C:\>sqlplus bob/Superspassword@hostname:1521/freepdb1
```

Donde hostname es el nombre del equipo y 1521 es el puerto de conexión.



Nota

Si usa SQL Developer y quiere crear una nueva conexión cuando solicite el nombre del servicio debe colocar xe, en el caso de usar una base de datos CDB. En cambio, si la base de datos es PDB, en nombre del servicio debe colocar freepdb1.

Luego cuando intentemos probar la conexión aparecerá el siguiente mensaje de error:

```
Estado: Fallo - Fallo de la prueba: ORA-01045: el usuario BOB no tiene  
privilegio CREATE SESSION; conexión denegada
```

Este error ocurre porque el nuevo usuario no tiene permisos. De forma predeterminada, un usuario de la base de datos no tiene privilegios. Ni siquiera para conectarse.



Nota

Simplemente crear un nuevo usuario no hará que el nuevo usuario acceda a la base de datos. Hay roles y privilegios necesarios que se deben asignar al usuario.

Cuando crea un nuevo usuario, debe al menos asignar el privilegio CREATE SESSION para que el usuario pueda conectarse a la base de datos desde una cuenta SYSTEM.


```
GRANT CREATE SESSION TO bob;
```

y aparecerá el siguiente mensaje:

```
Grant correcto.
```

El error que puede suceder si intenta realizar una conexión nuevamente es el siguiente:

```
ORA-01031: privilegios insuficientes
```

Para solucionar este problema, otorgaremos el privilegio de administrar los disparadores de la base de datos en la cuenta SYSTEM. Este privilegio es necesario para modificar correctamente el disparador de la base de datos:

```
GRANT ADMINISTER DATABASE TRIGGER TO bob;
```

Una vez hecho esto, el usuario BOB debería poder conectarse a la base de datos, pero aún no puede hacer nada útil. Además del privilegio CREATE SESSION, existen muchos otros privilegios que se pueden otorgar a los usuarios según sus necesidades, para ello tenemos los privilegios de sistema que son aquellos que permiten al usuario ejecutar acciones de nivel de sistema en la base de datos. Algunos ejemplos de privilegios de sistema son:

- **CREATE TABLE:** Permite al usuario crear nuevas tablas en la base de datos.
- **CREATE VIEW:** Permite al usuario crear nuevas vistas en la base de datos.
- **CREATE SESSION:** Permite al usuario iniciar una sesión en la base de datos.
- **ALTER USER:** Permite al usuario modificar la configuración de otro usuario.
- **CREATE PROCEDURE:** Permite al usuario crear nuevos procedimientos almacenados en la base de datos.

Por otro lado, tenemos los privilegios de objeto en Oracle que determinan cómo un usuario puede acceder y modificar los datos en la base de datos. Algunos ejemplos de privilegios de objeto son:

- **SELECT:** Permite al usuario realizar consultas de lectura en una tabla.
- **INSERT:** Permite al usuario insertar nuevos registros en una tabla.
- **UPDATE:** Permite al usuario actualizar registros existentes en una tabla.
- **DELETE:** Permite al usuario eliminar registros de una tabla.
- **EXECUTE:** Permite al usuario ejecutar un procedimiento almacenado.

Para asignar privilegios de sistema, puedes utilizar la siguiente sintaxis:

```
GRANT privilege_name TO username;
```

Por ejemplo, podríamos otorgar el siguiente permiso desde una cuenta SYSTEM al usuario BOB:

```
GRANT CREATE TABLE TO bob;
```

Este permiso le permitirá crear tablas, así como índices y restricciones sobre las mismas tablas. Ahora intente crear una tabla en el usuario BOB. Por ejemplo, la siguiente:

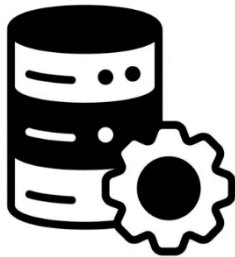
```
CREATE TABLE friend_email (  
    name      VARCHAR2(30),  
    email     VARCHAR2(100)  
);
```

Puede verificar la creación de la tabla consultando al diccionario de datos:

```
SELECT table_name FROM user_tables ORDER BY table_name;
```

El resultado obtenido es el siguiente:

```
TABLE_NAME  
-----  
FRIEND_EMAIL
```



Nota

La mejor práctica de seguridad es el Principio de Mínimo Privilegio (PoLP). Esto implica otorgar a los usuarios solo los privilegios específicos que necesitan para realizar su trabajo, y nada más. Esto mejora significativamente la seguridad de su base de datos.

Sin embargo, todavía no tendríamos permiso para algo elemental como almacenar datos e información en la tabla. Para evitar esto, debe darle a su usuario una cuota de espacio en su tablespace:

```
GRANT UNLIMITED TABLESPACE TO bob;
```

Los usuarios tienen acceso completo a sus propios objetos lo que significa que puede ejecutar cualquier consulta sobre sus tablas como INSERT, UPDATE o DELETE. Pruebe realizando un INSERT:

```
INSERT INTO friend_email VALUES('Martina','martina@testmail.com');
```

Puede otorgar muchos privilegios al usuario de una sola vez separándolos con una coma de esta manera:

```
GRANT CREATE VIEW, CREATE PROCEDURE TO bob;
```

Dejar al usuario acceso a datos que no debería manejar es un riesgo para la seguridad. Una vez que tenga un privilegio puede también revocarlo.

```
REVOKE CREATE TABLE FROM bob;
```

De manera similar, para asignar privilegios de objeto, puede utilizar la siguiente sintaxis:

```
GRANT privilege_name ON object_name TO username;
```

Por ejemplo, creemos el usuario patricia:

```
CREATE USER patricia IDENTIFIED BY patol23;
```

Luego asignemos el permiso de sistema para poder iniciar sesión:

```
GRANT CREATE SESSION TO patricia;
```

Ahora para asignar el privilegio de lectura en la tabla EMPLOYEES debe escribir el siguiente comando:

```
GRANT SELECT ON hr.regions TO patricia;
```

Luego si se conecta como patricia el usuario puede lanzar el siguiente comando:

```
SELECT count(*) FROM hr.regions;
```

Ahora procederemos a usar uno de los nuevos privilegios de Oracle23ai GRANT SELECT ANY TABLE ON SCHEMA, una característica que simplifica significativamente la forma en que los administradores de bases de datos administran el acceso de solo lectura a los esquemas de bases de datos. Primero crearemos un usuario:

```
CREATE USER sheena IDENTIFIED BY secret;
```

Ahora le concederemos al usuario el acceso de lectura de todas las tablas y vistas del esquema HR de manera automática:

```
GRANT SELECT ANY TABLE ON SCHEMA HR TO sheena;
```

Oracle 23ai introduce una nueva vista dentro de los diccionarios de datos denominado DBA_SCHEMA_PRIVS.

```
SELECT grantee, privilege, schema FROM DBA_SCHEMA_PRIVS WHERE  
grantee='SHEENA';
```

Esta consulta devuelve el siguiente resultado:

GRANTEE	PRIVILEGE	SCHEMA
SHEENA	SELECT ANY TABLE	HR_ADMIN

Luego asignemos el permiso de sistema para poder iniciar sesión:

```
GRANT CREATE SESSION TO sheena;
```

Si ahora entramos con el usuario sheena, podemos hacer la siguiente consulta:

```
select count(*) from hr.employees;
```

Un rol es un grupo de privilegios en un solo paquete. Esto permite al administrador de la base de datos otorgar o revocar fácilmente privilegios entre los usuarios. En lugar de asignar privilegios individualmente a cada usuario, se le puede asignar un rol. Oracle ya proporciona algunos roles predefinidos para ayudar en la administración de la base de datos;

- **CONNECT:** Permite conexión a la Base de Datos y consultar los esquemas.
- **RESOURCE:** Permite la gestión de Objetos de su propio esquema.
- **DBA:** Tiene la mayoría de los privilegios, no es recomendable asignarlo a usuarios que no son administradores.



Nota

Una forma sencilla de visualizar los roles de un usuario es usando SQL DEVELOPER. Para ello ingrese al usuario SYSTEM y en el explorador de objetos visualice OTROS USUARIOS. Haciendo click con el botón derecho del Mouse seleccione la opción EDITAR USUARIOS y seleccione la solapa ROLES OTORGADOS.

Ahora escriba el siguiente comando en el usuario SYSTEM:

```
GRANT RESOURCE TO bob;
```

Si necesita saber que roles posee un usuario determinado escriba el siguiente comando usando la cuenta SYSTEM:

```
SELECT granted_role FROM dba_role_privs WHERE grantee='BOB';
```



Nota

Una forma sencilla de visualizar los roles de un usuario es usando SQL DEVELOPER. Para ello ingrese al usuario SYSTEM y en el explorador de objetos visualice OTROS USUARIOS. Haciendo click con el botón derecho del Mouse seleccione la opción EDITAR USUARIOS y seleccione la solapa ROLES OTORGADOS.

Probablemente se pregunte cómo identificar qué usuarios de la base de datos Oracle tienen actualmente el rol DBA. Puede usar la siguiente consulta SQL para listar todos los usuarios con el rol de DBA (excluyendo las cuentas SYS y SYSTEM):

```
SELECT grantee FROM dba_role_privs WHERE granted_role='DBA' AND grantee NOT IN ('SYS','SYSTEM');
```

A partir de Oracle23 existe un nuevo rol denominado DB_DEVELOPER_ROLE, diseñado para reemplazar a los roles CONNECT y RESOURCE que permite a los administradores asignar rápidamente todos los privilegios necesarios que los desarrolladores necesitan para diseñar, crear e implementar aplicaciones para Oracle Database. Para ello crearemos un nuevo usuario:

```
CREATE USER dev_app IDENTIFIED BY qwerty123;
```

Luego le otorgamos el rol DB_DEVELOPER_ROLE:

```
GRANT DB_DEVELOPER_ROLE TO dev_app;
```

Los usuarios con el rol DB_DEVELOPER_ROLE aún necesitarán cuotas de espacio específicos por ello debe escribir el siguiente comando:

```
GRANT UNLIMITED TABLESPACE TO dev_app;
```

Si necesito modificar la contraseña del usuario dev_app debemos hacer lo siguiente:

```
ALTER USER dev_app IDENTIFIED BY Superpassword123456;
```

Si desea que el usuario cambie la contraseña inmediatamente después de iniciar sesión debe escribir lo siguiente:

```
ALTER USER dev_app PASSWORD EXPIRE;
```

Donde dev_app es el usuario al cual se solicitará el cambio de contraseña:

```
Cambiando la contraseña para dev_app  
Contraseña nueva:
```

Si necesita bloquear a un usuario temporalmente puede usar el siguiente comando:

```
ALTER USER bob ACCOUNT LOCK;
```

Si necesita desbloquear al usuario para que pueda seguir usando la cuenta el comando es el siguiente:

```
ALTER USER bob ACCOUNT UNLOCK;
```

A veces, las cuentas deben eliminarse del sistema por varias razones, pero suponga que el nombre de la cuenta esta mal escrita y como no existe el comando ALTER USER para modificar el nombre debemos eliminar la cuenta existente para ello usaremos el comando:

```
DROP USER bob
```

Sólo puede hacer esto si el usuario no está conectado a la base de datos. Así que asegúrese de desconectarse de cualquier sesión antes de hacerlo. Si el usuario posee objetos debe usar esta versión:

```
DROP USER bob CASCADE
```

TRANSACCIONES EN SQL

Una transacción es una unidad atómica e indivisible que contiene una o más instrucciones SQL donde se ejecutan en su totalidad o no se realizan en absoluto. ¿Por qué necesitamos transacciones? Consideremos un escenario en el que se deben realizar múltiples operaciones para mantener la integridad de los datos y no se utilizan transacciones, si se produce un error durante cualquiera de estas operaciones, la base de datos podría acabar en un estado inconsistente. Para que una unidad se considere una transacción, debe cumplir con cuatro propiedades, conocidas como ACID, que significa atomicidad, consistencia, aislamiento y durabilidad.

Atomicidad: Se realizan todas las tareas de una transacción o no se realiza ninguna de ellas. No hay transacciones parciales.

Consistencia: La transacción lleva la base de datos de un estado coherente a otro estado coherente. Por ejemplo, en una transacción bancaria que debita una cuenta de ahorros y acredita una cuenta corriente, una falla no debe provocar que la base de datos acredite solo una cuenta, lo que generaría datos inconsistentes.

Aislamiento: El efecto de una transacción no es visible para otras transacciones hasta que se confirma la transacción. Por ejemplo, un usuario que actualiza la tabla employees no podrá ver los cambios no confirmados de otro usuario sobre employees cuando se trabaja simultáneamente. Por ello a los usuarios les parece que las transacciones se están ejecutando en serie.

Durabilidad: Los cambios realizados por transacciones confirmadas son permanentes. La durabilidad significa que el resultado de esta transacción se almacena en la base de datos y no se perderá si la base de datos falla o se bloquea.

Una transacción comienza cuando se encuentra la primera instrucción SQL ejecutable. A su vez existen ciertos comandos para trabajar con transacciones. El comando COMMIT (comprometerse) permite indicar que la transacción ha finalizado exitosamente y que los datos deben guardarse permanentemente o confirmarse en la base de datos. El comando

ROLLBACK (deshacer) permite deshacer los cambios de una transacción y restaurar la base de datos a un estado anterior al inicio de la transacción. Es lo opuesto a una confirmación. El comando SAVEPOINT se utiliza para especificar un punto de restauración en la transacción al que puede revertirse más tarde. Para ver todo lo recién visto generemos un INSERT en el usuario HR:

```
INSERT INTO regions VALUES(100,'Latin American');
```

Si todo anduvo bien ingrese a través de la herramienta SQL PLUS con el usuario patricia y realice un SELECT de las regiones:

```
SELECT * FROM hr.regions;
```

En ese listado puede ver que la nueva región no aparece. Vuelva al usuario HR y escriba el comando:

```
COMMIT;
```

Ahora sobre el usuario patricia vuelva a ejecutar el mismo comando que tipeo anteriormente y vera que ahora si puede ver el cambio.

```
SELECT * FROM hr.regions;
```



Nota

Una forma sencilla de realizar un commit en SQL DEVELOPER es presionando la tecla <F11>.

Ahora realice un nuevo INSERT en el usuario HR:

```
INSERT INTO regions VALUES(101,'Australasia');
```

Ahora realice un SELECT en el mismo usuario:

```
SELECT * FROM regions;
```

Y vera la nueva región ingresada. Ahora tipee el siguiente comando:

```
ROLLBACK;
```


Y realice el siguiente listado:

```
SELECT * FROM regions;
```

Podrá ver que la región recién insertada no aparece. Podemos hacer el siguiente ejemplo para incorporar el concepto de punto de restauración:

```
BEGIN
  INSERT INTO regions VALUES(101, 'Australasia');
  SAVEPOINT after_insert;
  UPDATE regions SET region_name='Australia y Nueva Zelanda' WHERE
    region_id=101;
  ROLLBACK TO SAVEPOINT after_insert;
END;
```

En este ejemplo podemos ver que tenemos un punto de restauración con el nombre `after_insert` que se sitúa debajo de la instrucción `INSERT`. Luego de esta se realiza un `UPDATE` sobre la región insertada, pero esta no se aplica porque se realiza un `ROLLBACK` al punto de restauración.

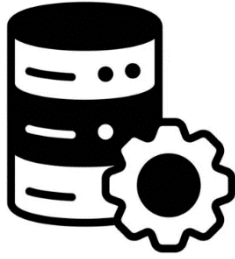
COPIAS DE RESGUARDO y RECUPERACION

En el entorno empresarial actual, la pérdida de datos representa un riesgo inaceptable que puede generar consecuencias devastadoras: desde impactos financieros significativos hasta daños irreparables en la reputación corporativa. Cuando se trata de administración de bases de datos, es prácticamente necesario crear e implementar una estrategia sólida para las tareas de copia de seguridad y recuperación. Normalmente, los administradores siguen la regla general del 3-2-1 al gestionar cualquier copia de seguridad crítica del sistema. La regla es la siguiente:

3 copias de sus datos : Conserve al menos tres copias de sus datos. Esto incluye los datos originales y dos copias de seguridad. Esta redundancia ayuda a proteger contra la pérdida de datos.

2 medios de almacenamiento diferentes : Almacene las copias en al menos dos tipos de medios de almacenamiento diferentes. Por ejemplo, podría usar un disco duro externo y por lado usar la nube. Esto reduce el riesgo de que ambas copias de seguridad fallen simultáneamente.

1 Copia de seguridad externa : Conserve al menos una copia externa. Esto significa almacenar la copia de seguridad en una ubicación física diferente, como un servicio en la nube o un servidor remoto. Esto protege sus datos de desastres locales como incendios o inundaciones.



Nota

En un ataque de ransomware, los intrusos, tras acceder al sistema, pueden borrar los servidores, cifrar los datos críticos y los archivos de copia de seguridad, y solicitar un rescate en bitcoins.

CLASIFICACION DE LAS COPIAS DE SEGURIDAD FISICA

Las copias de seguridad de bases de datos se pueden clasificar en copias de seguridad físicas y copias de seguridad lógicas.

LAS COPIAS DE SEGURIDAD FÍSICAS

Los backups físicos son copias de seguridad a nivel de sistema operativo que incluyen todos los archivos críticos de la base de datos Oracle, Los componentes principales son:

Archivos de datos (datafiles): Archivos con extensión .dbf o .dat donde Oracle almacena físicamente toda la información de tablas, índices y datos de usuario.

Archivos de control (control files): Contienen los metadatos esenciales sobre la estructura física y lógica de la base de datos, incluyendo ubicaciones de archivos e información de configuración. Son indispensables para el arranque de la base de datos.

Archivos de Redo Log: Registran secuencialmente todos los cambios transaccionales realizados en la base de datos, permitiendo la recuperación y mantenimiento de la consistencia de datos.

Siempre es recomendable realizar una copia de seguridad física completa semanal o mensualmente. Es mejor realizarla durante un periodo de inactividad total para evitar la aparición de datos inconsistentes durante la recuperación. La desventaja de las copias de seguridad físicas es que, por lo general, solo se utilizan para recrear el sistema durante la restauración y no permiten realizar una restauración completa si faltan archivos. En este caso, debe asegurarse de recrear la copia de seguridad con los archivos faltantes y luego proceder con el proceso de restauración.

LAS COPIAS DE SEGURIDAD LOGICA

Una copia de seguridad lógica contiene copias de la información de una base de datos (como tablas, esquemas y procedimientos) y suele exportarse como archivos binarios mediante herramientas de exportación e importación. Si necesita restaurar o mover una copia de la base de datos a otro entorno las copias de seguridad lógicas son una excelente opción. Las copias de seguridad lógicas completas deben realizarse semanalmente y se utilizan comúnmente como sustituto si la copia de seguridad física completa no está disponible. La desventaja de este tipo de copia de seguridad es que no tiene la información del sistema de archivos (lo que dificulta el proceso de restauración).

RMAN

Oracle Recovery Manager (RMAN) es una potente herramienta que simplifica enormemente la copia de seguridad, la restauración y la recuperación de archivos de bases de datos. La herramienta RMAN toma automáticamente una copia instantánea de la base de datos, sus datos y todos los objetos. El proceso de copia de seguridad requiere una base de datos de origen de la que se debe realizar una copia de seguridad y una ubicación de destino para almacenar los archivos de copia de seguridad. Antes de comenzar iniciaremos una sesión en Oracle:

```
sqlplus / as sysdba
```

y tipearemos el siguiente comando:

```
select name, log_mode from v$database;
```

Este comando nos devuelve la siguiente información:

NAME	LOG_MODE
FREE	ARCHIVELOG

El modo ARCHIVELOG es un mecanismo de protección ante fallos de disco implementado por Oracle que nos permitirá realizar backups sin detener la base de datos (en caliente). Si la salida llega a ser la siguiente

NAME	LOG_MODE
FREE	NOARCHIVELOG

Debemos poner la base de datos en modo Archive log. Este procedimiento deberemos hacerlo en un momento del día en que el impacto sea menor ya que debemos detener la base de datos. Para cambiar el modo de la base de datos, esta debe estar en modo MOUNT, por lo que es necesario primero detenerla:

```
shutdown immediate;
```

Y luego debe iniciar en modo MOUNT:

```
startup mount;
```

Una vez en modo MOUNT, alteramos la base de datos para cambiar a modo Archive log;

```
alter database archivelog;
```

y luego debe abrirse para que los usuarios puedan volver a conectarse.

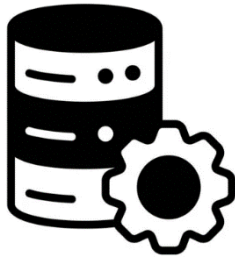
```
alter database open;
```

Se puede verificar que la base de datos está en modo Archivelog, escribiendo nuevamente el siguiente comando:

```
select name, log_mode from v$database;
```

Este comando ahora nos devolverá la siguiente información:

NAME	LOG_MODE
FREE	ARCHIVELOG



Nota

El modo archivelog/noarchivelog es una propiedad de la base de datos que se modifica mediante la sentencia SQL alter database. Este modo de funcionamiento se guarda en el archivo de control de la base de datos, no es necesario volver a indicarlo en cada backup.

Para usar rman debemos tipear lo siguiente:

```
rman target /
```

O sino usar el usuario system:

```
rman target sys/ maria123456@hostname;
```



Nota

Recordar que cuando ingreso como rman target / o rman target sys/maria123456@hostname; resguardo TODO es decir el CDB Root, todos los PDBS (incluyendo freepdb1) y el template pdb\$seed. Pero si me conecto como rman target sys/maria123456@hostname:1521/FREEPDB1 solo hago copia del freepdb1.

Lo primero que haremos es cambiar la ubicación de nuestro backup:

```
configure channel device type disk format 'c:\rman\backup/%d_%T_%u';
```

La ubicación puede ser una unidad local, una unidad de red o un dispositivo externo. Pero debe existir. Los modificadores que se pueden usar en la creación de los archivos son:

- %d nombre de la base de datos
- %T año, mes, día en formato: AAAAMMDD
- %u identificador único del backup.

Podemos ver si este parámetro está configurado correctamente escribiendo el siguiente comando:

```
show all;
```






Finalmente, para realizar la copia de seguridad completa de la base de datos debemos escribir el siguiente comando.

```
backup database tag='FULL_BACKUP';
```

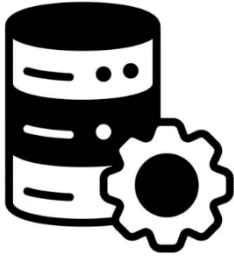
Para que este comando funcione la base de datos debe estar montada o abierta, porque RMAN necesita acceder al archivo de control de la base de datos de destino fundamentalmente para registrar en él la existencia de la copia de seguridad. Las copias de seguridad de base de datos solo son posibles si la base de datos funciona en modo ARCHIVELOG. El atributo tag en rman permite etiquetar y organizar los backups para identificarlos fácilmente. Si queremos la copia completa y que también incluye el historial de cambios para poder recuperar transacciones posteriores al backup debemos escribir lo siguiente:

```
backup database plus archivelog tag='FULL_BACKUP';
```

Luego de aguardar un tiempo podremos dar por terminado nuestro backup y comprobaremos yendo al directorio que los archivos se encuentren:

Nombre	Fecha de modificación	Tipo	Tamaño
 FREE_20250805_0V40C2KV	5/8/2025 14:18	Archivo	438.996 KB
 FREE_20250805_1040C2LF	5/8/2025 14:19	Archivo	1.890.856 KB
 FREE_20250805_1140C2NG	5/8/2025 14:20	Archivo	963.552 KB
 FREE_20250805_1240C2OK	5/8/2025 14:20	Archivo	562.008 KB
 FREE_20250805_1340C2PE	5/8/2025 14:20	Archivo	38 KB

RMAN emplea dos tipos principales de copias de seguridad: completas e incrementales. Una copia de seguridad completa, como su nombre indica, implica realizar una copia de seguridad completa de la base de datos. Esta copia de seguridad contiene todos los archivos de datos, archivos de control y registros de rehacer archivados necesarios para restaurar la base de datos a su estado original. Las copias completas proporcionan una protección integral de todos los datos, aunque consumen considerablemente más espacio de almacenamiento. Por el contrario, las copias de seguridad incrementales capturan únicamente los cambios realizados en la base de datos desde la última copia de seguridad. Este tipo de copia de seguridad suele ser más rápida y requiere menos espacio de almacenamiento que una copia de seguridad completa, pero a cambio aumenta la complejidad durante la recuperación de datos.



Nota

La idea principal de las copias de seguridad incrementales es que no todas las tablas se modifican a diario. Entonces, ¿por qué realizar una copia de seguridad completa a diario? Basta con hacer una copia de seguridad de los cambios en lugar de la base de datos completa.

Una copia de seguridad de nivel 0 es el punto de partida para una copia de seguridad incremental. Las copias de seguridad incrementales solo hacen copias de los datos que han cambiado desde la copia de seguridad anterior. Ahorran tiempo y espacio en lugar de tener que realizar una copia completa de cada copia de seguridad. Para realizar una copia incremental debe escribir el siguiente comando:

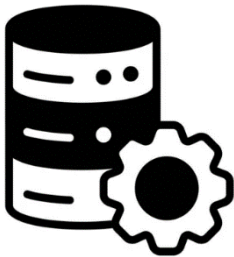
```
backup incremental level 0 database tag='BACKUP_BASE';
```

Las copias de seguridad incrementales de nivel 0 en cierto modo son equivalentes a una copia de seguridad completa.

```
backup incremental level 1 database tag='INCREMENTAL_01';
```

Este último comando especifica que se realizará una copia de seguridad incremental de nivel 1. Esto significa que realiza una copia de seguridad únicamente de los bloques que han cambiado desde la última copia de seguridad de nivel 0 (completa) o nivel 1 (incremental)

Nota



Supongamos que tenemos un sistema e-commerce podemos realizar una copia de seguridad incremental de Nivel 0 todos los sábados por la noche y copias de seguridad de Nivel 1 cada noche a partir de dicho sábado. Esto reduce el tamaño y el tiempo de copia de seguridad necesarios, garantizando que las copias de seguridad nocturnas capturen únicamente los cambios realizados durante el día. Si perdemos la base de datos el primer viernes, primero debemos recuperarla con la copia de seguridad completa realizada el sábado y, posteriormente, aplicar las copias de seguridad incrementales realizadas el domingo, lunes, martes, miércoles y jueves de dicha semana.

Puede listar todos los backups que se hicieron con el comando:

```
list backup;
```

Si necesita buscar información de una copia específica necesitara usar el atributo tag, como en este ejemplo:

```
list backup tag='FULL_BACKUP';
```

Para probar la restauración haremos una modificación intencional en nuestro sistema para que luego podamos recuperar la base de datos. Buscaremos los archivos de datos de nuestra base en el directorio c:\app\username\product\23ai\oradata\free y cambiaremos el nombre de uno de los archivos, por ejemplo, modificaremos la extensión al archivo SYSTEM01.DBF por la extensión .DEL. Puede ser que no aparezca ningún error todavía. Pero intentemos iniciar una sesión en Oracle:

```
sqlplus / as sysdba
```

y tipearemos el siguiente comando para detener la base de datos:

```
shutdown immediate;
```

Al momento aparecerá el error:

```
ORA-01116: error al abrir el archivo de base de datos 1
ORA-01110: archivo de datos 1:
'C:\APP\USERNAME\PRODUCT\23AI\ORADATA\FREE\SYSTEM01.DBF'
ORA-27041: no se ha podido abrir el archivo
OSD-04002: no se ha podido abrir el archivo
O/S-Error: (OS 2) El sistema no puede encontrar el archivo especificado.
```

El problema se encuentra en el archivo recién renombrado. Para solucionar este problema haremos una restauración: Antes de ello cerraremos la base de datos en forma abrupta con el comando:

```
shutdown abort;
```

El comando shutdown abort en Oracle fuerza a detener la base de datos cuando esta no puede detenerse con el comando shutdown immediate. Luego iniciaremos la base de datos con el comando:

```
startup mount;
```

El comando startup mount en Oracle arranca la instancia y monta la base de datos, pero no la abre para que los usuarios se puedan conectar. Ahora si podemos hacer la restauración completa para ello usaremos la utilidad rman de la siguiente manera:

```
rman target /
```

Para restaurar una base de datos a partir de una copia de seguridad completa con todas las copias incrementales, utilice los siguientes comandos:

```
restore database;
```

```
recover database;
```

De esta forma ya tendremos la Base de Datos consistente y actualizada. Luego de unos minutos ya tendremos la restauración de la base de datos y veremos al archivo renombrado con el nombre anterior. Ahora probaremos la restauración. Para salir del cliente RMAN, ingrese el comando exit.

```
exit;
```

Ingresamos a sqlplus nuevamente:

```
sqlplus / as sysdba
```

Y tipeamos el siguiente comando:

```
alter database open;
```

El comando alter database open abre la base de datos para uso normal después de que haya sido montada.

El mensaje que aparecerá a continuación será:

```
Base de datos modificada.
```

Ahora suponga que no desea restaurar ni recuperar toda la base de datos ni todos los archivos de datos asociados. Solo desea restaurar y recuperar el archivo de datos que sufrió el fallo. Para ello primero listaremos los DBFs que disponemos con el comando;

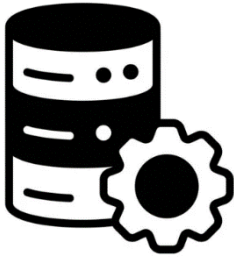
```
SELECT name FROM v$datafile;
```

Este comando nos devolverá todos los data files. Para recuperar un data file debe usar el comando restore datafile. Por ejemplo:

```
restore datafile  
'c:\app\username\product\23ai\oradata\free\freepdb1\system01.dbf'
```

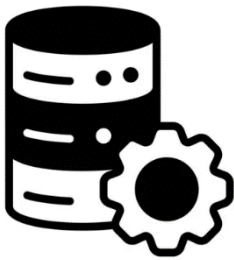
EXPORTACION DE DATOS

RMAN está diseñado principalmente para copias de seguridad y recuperación. Realiza una copia exacta de los archivos físicos que componen la base de datos en el disco. La utilidad Data Pump fue introducida en la versión 10g de Oracle. Es principalmente una herramienta de respaldo utilizada principalmente para migrar datos entre diferentes bases de datos de Oracle, pero también se puede utilizar para exportar datos y metadatos en un formato estructurado que Oracle puede interpretar.

**Nota**

Las exportaciones de Oracle son copias de seguridad lógicas de bases de datos (no físicas), ya que extraen datos y definiciones lógicas de la base de datos a un archivo.

Hay varios métodos para usar esta utilidad. Primero, usaremos SQL Developer y luego la línea de comando. Para este ejemplo exportaremos el esquema HR.

**Nota**

Un esquema es la cuenta de usuario y la colección de todos los objetos que contiene.

Luego para continuar debemos crear el directorio a nivel lógico en Oracle que especifica la ruta que Oracle usará para realizar los exports e imports. Esto permitirá organizar en directorios distintos los backups. Esto lo podemos hacer desde la línea de comando o desde SQL Developer utilizando el usuario SYS o uno con rol de DBA como hr_admin:

```
create directory backup_2025 as 'c:\orabackup';
```

El nombre back_2025 es el nombre lógico que asignamos para nuestra utilidad de exportación. El directorio físico será c:\orabackup. Obviamente ese directorio ya debe estar creado. Al momento aparecerá el siguiente mensaje:

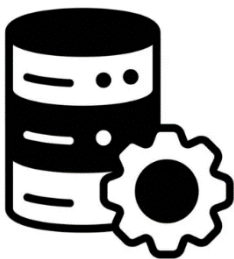
```
Directory BACKUP_2025 creado
```

Ahora daremos permisos de lectura y escritura al directorio creado:

```
grant read, write on directory backup_2025 to hr_admin;
```

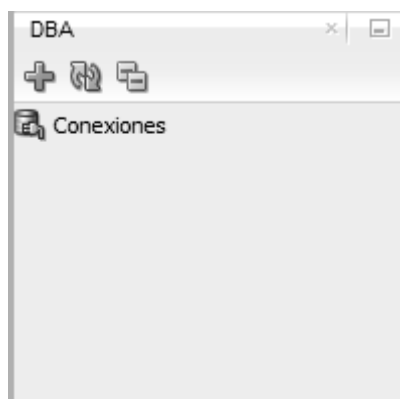
A continuación, verificaremos la creación de dicho directorio a través del comando:

```
select * from dba_directories;
```

**Nota**

Si la exportación es parte de una estrategia de copia de seguridad es recomendable evitar crear el directorio o carpeta en la misma partición donde está el directorio de Oracle o el arranque del Sistema Operativo.

Para abrir el asistente de exportación deberá ir al menú Ver de SQL Developer y seleccionar la opción DBA. Luego desde este panel debemos seleccionar la conexión que contiene los datos que serán exportados (por ejemplo: hr_admin).



Luego desde la conexión seleccionada debe seleccionar la opción Pump de Datos y luego con el botón derecho seleccionar la opción Asistente de exportación de Pump de Datos. En el Asistente de exportación debemos seleccionar la opción que deseamos exportar. Los tipos que se pueden exportar usando Data Pump son los siguientes:

1. Base de datos
2. Tablespace
3. Esquema
4. Tablas

Por ejemplo, podemos seleccionar la opción Esquemas y luego presionar el botón siguiente.



Nota

Cada destacar que las ventanas de los asistentes variarán según la información a exportar.

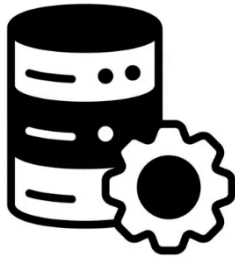
En la próxima ventana seleccionamos el esquema que deseamos exportar y presionamos el botón > para incluirlo en el proceso de exportación (Podemos exportar uno o varios esquemas a la vez seleccionando los que queramos). En la sección Filtro otra vez realice siguiente, lo mismo en la opción Datos de la Tabla. En la siguiente pantalla, seleccione el DIRECTORIO LOGICO que contendrá los archivos de volcado de salida (en este caso backup_2025) y seleccione siguiente.



Nota

Tenga en cuenta en dicho menú la versión de Oracle. Si va a importar más tarde a una base de datos de Oracle de una versión anterior, desea establecer el número de versión en el de la base de datos de destino.

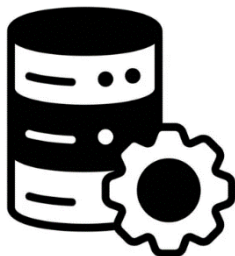
Observe que el nombre de archivo en la siguiente termina con %U. Esto le dice al asistente que use un incremento numérico en el nombre del archivo. Por ultimo haga click en Finalizar.



Nota

Un archivo .dmp de Oracle es un archivo binario creado por la utilidad Oracle Data Pump (expdp), contiene una copia de uno o más objetos de la base de datos Oracle, como tablas, índices, procedimientos almacenados etc. e incluye instrucciones SQL para crear los objetos de la bases de datos.

Una vez finalizado el trabajo, dentro del panel DBA sobre Pump de Datos seleccione el Nodo Trabajos de Exportación donde puede monitorear la exportación realizada. Ahora vaya a la carpeta y verifique el archivo .DMP y el archivo .LOG se encuentran ya creados dentro de la carpeta.



Nota

Tome nota del nombre del archivo .DMP ya que servirá para la posterior importación.

Si desea realizar el mismo proceso a través de la línea de comando deberá usar el comando expdp. Para ello abra una ventana del símbolo del sistema usando la sintaxis que se muestra a continuación:

```
expdp <username>/<password> schemas=<schema_name>  
directory=<directory_object_name> dumpfile=<dump_file_name>  
logfile=<log_file_name>
```

Para este ejemplo usaremos la cuenta de usuario HR_ADMIN para exportar su esquema en el directorio lógico backup_2025:

```
expdp hr_admin/pwd123456hr@hostname:1521/freepdb1 schemas=hr_admin directory=  
backup_2025 dumpfile=data_back.dump logfile=data_back.log
```

El archivo del volcado generado será llamado data_back.dump y el archivo log generado se llamara data_back.log. Por otro lado, si desea exportar toda la base de datos deberá usar el parámetro full=Y. Ejemplo:

```
expdp hr_admin/pwd123456hr@hostname:1521/freepdb1 full=Y directory=backup_2025  
dumpfile=data_full_back.dump logfile=data_full_back.log
```

Otra forma, es pasar un archivo de parámetros al comando expdp. Para indicarle a expdp el archivo con los parámetros emplearemos el parámetro parfile. En el directorio c:\orabackup crearemos el archivo export.par con el siguiente contenido:

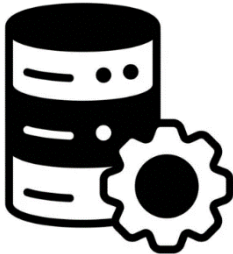
```
DIRECTORY=backup_2025  
DUMPFILE=export.dmp  
LOGFILE=export.log  
SCHEMAS=hr_admin
```

Donde:

- DIRECTORY será el objeto DIRECTORY que hemos creado previamente en oracle y que apunta al directorio de exportación.
- DUMPFILE definirá el nombre del archivo de exportación.
- LOGFILE definirá el archivo con el detalle de la exportación.
- SCHEMAS es el nombre del esquema a exportar.

Una vez almacenado el archivo desde el mismo directorio tipearemos el siguiente comando:

```
C:\orabackup> expdp hr_admin/pwd123456hr@hostname:1521/freepdb1 -parfile  
export.par
```



Nota

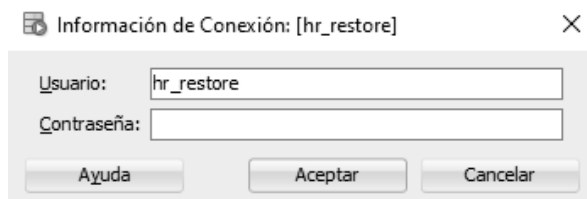
Si necesita exportar un numero de tablas especificas deberá utilizar el parámetro `tables`. Por ejemplo podría escribirlo de la siguiente manera: `TABLES=employees,jobs`. No es necesario especificar el nombre del esquema para las tablas porque el usuario `hr` está exportando tablas en su propio esquema.

IMPORTAR USANDO SQL DEVELOPER

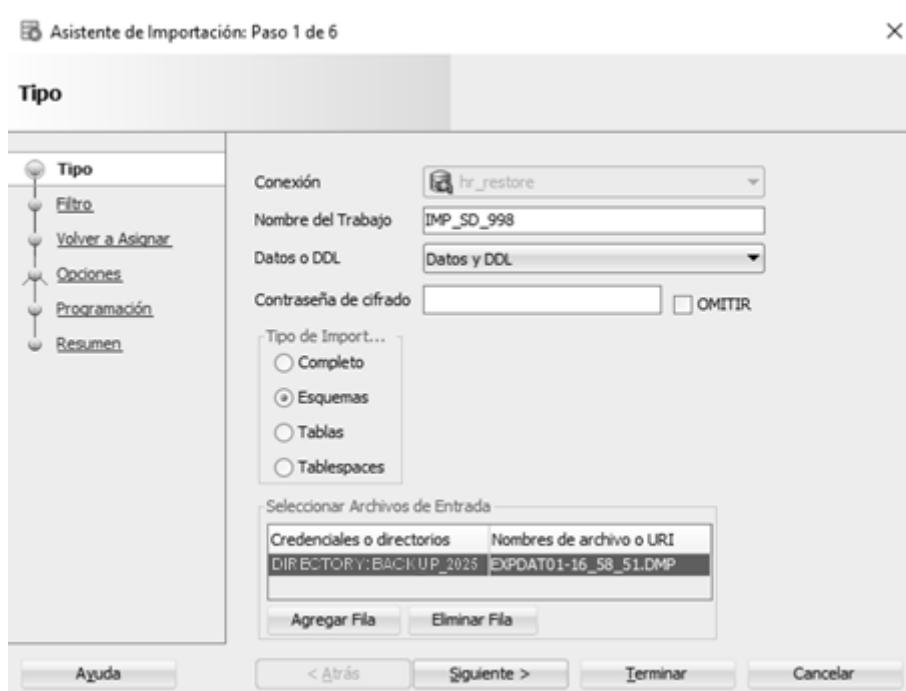
En esta parte importaremos el esquema anterior. Para poder simular la importación crearemos un nuevo usuario denominado `hr_restore` con su correspondiente tablespace que recibirá la importación del archivo DMP generado anteriormente:

```
create user hr_restore identified by pwdOracle123456;  
grant create session to hr_restore with admin option;  
grant connect, resource,dba to hr_restore;  
grant unlimited tablespace to hr_restore;
```

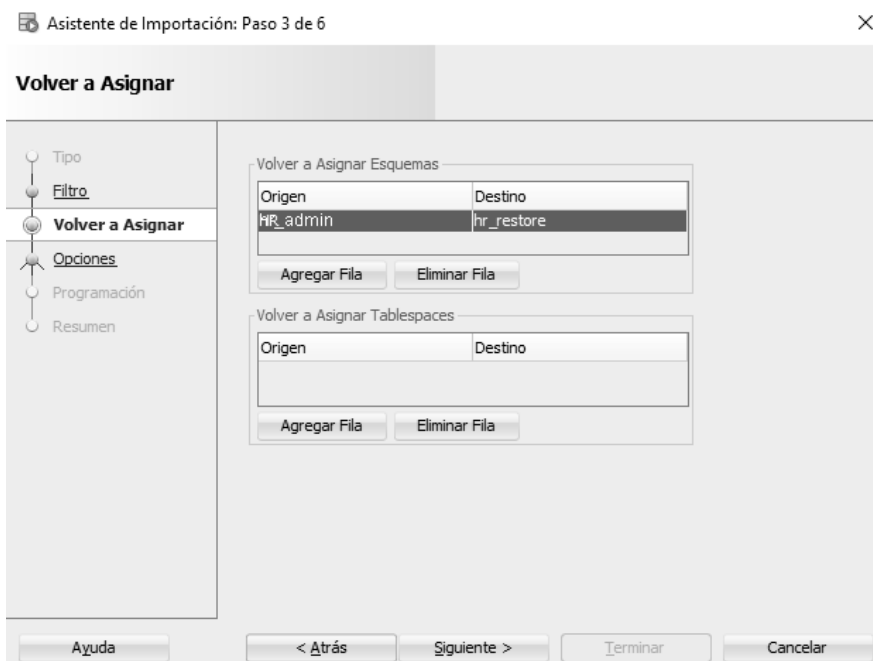
Ahora crearemos una nueva conexión denominada `hr_restore` y luego la abriremos para importar lo realizado anteriormente.



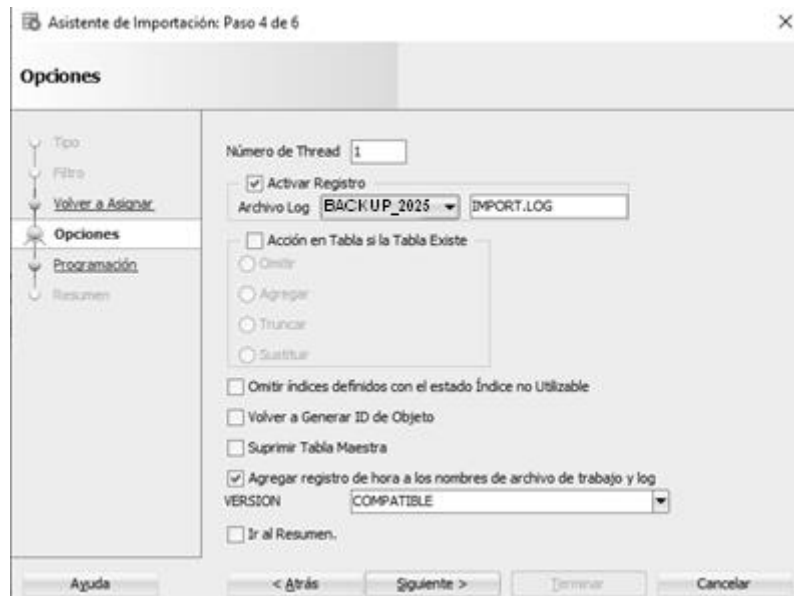
Para realizar la importación dentro del Panel DBA deberá seleccionar la opción Pump de Datos y escoger la opción Asistente de Importación de Pump de Datos. La primera ventana es clave en la importación acá deberá elegir el directorio lógico donde se encuentra el archivo .DMP (en este caso backup_2025) y el nombre del archivo (El nombre del archivo de exportación que se tomó nota en el punto anterior). También es importante seleccionar lo que se quiere importar (En este caso un Esquema).



Luego en un momento aparecerá un listado de lo que se podrá importar. Dentro de este listado seleccione el esquema HR_ADMIN. En la ventana de reasignación, haga clic en "agregar fila" y seleccione en destino el esquema recién creado (HR_RESTORE). Luego haga click en siguiente.



En la próxima ventana seleccione el directorio lógico del datadump y un nombre para el archivo de registro (log) y realice un click en siguiente.



En la ventana de programación de trabajo seleccione la opción Inmediatamente y haga clic en el botón siguiente y en la próxima ventana haga click en Finalizar.



Finalmente, en el Panel Izquierdo de SQL Developer pude clickear en la sección Tablas del usuario hr_copy y ver el listado de tablas importadas. También puede realizar la siguiente query en una hoja de trabajo de SQL de dicha conexión:

```
SELECT table_name FROM user_tables
```

COMANDO IMPDP

El funcionamiento general del comando de importación impdp es análogo al del expdp. La sintaxis es la siguiente:

```
impdp <username>/<password> schemas=<schema_name>  
directory=<directory_object_name> dumpfile=<dump_file_name>  
logfile=<log_file_name> table_exists_actions=replace
```

Ahora crearemos otro usuario distinto:

```
sqlplus / as sysdba
```

Como es un usuario común debemos escribir lo siguiente:

```
ALTER SESSION SET CONTAINER=FREEPDB1;
```

Y crearemos el usuario user_operation:

```
create user user_operation identified by pwdOracle123456;  
grant create session to user_operation with admin option;  
grant connect, resource, dba to user_operation;  
grant unlimited tablespace to user_operation;
```

Salimos del usuario sys:

```
exit;
```

Ahora volcaremos el esquema del usuario HR que obtuvimos con el comando expdp:

```
impdp user_operation/pwdOracle123456@hostname:1521/freepdb1  
directory=backup_2025 dumpfile=EXPDAT01-16_58_51.DMP logfile=new_import.log  
remap_schema=hr:user_operation table_exists_actions=replace
```

El parámetro REMAP_SCHEMA toma dos valores. El primero es el nombre del esquema que estamos importando y el segundo es al esquema donde va a parar el volcado. La opción table_exists_actions permite reemplazar las tablas que ya existan.