Paso 0: Preparar los Datos (Jerarquía)

Copia y ejecuta esto en Oracle Live SQL para tener el escenario perfecto:

```
CREATE TABLE empleados (
   emp_id NUMBER PRIMARY KEY,
   nombre VARCHAR2(50),
   salario NUMBER,
   jefe_id NUMBER -- Referencia al ID de otro empleado (el jefe)
);

BEGIN
   -- El CEO (No tiene jefe)
   INSERT INTO empleados VALUES (1, 'Carlos El CEO', 10000, NULL);

   -- Gerentes (Reportan a Carlos)
   INSERT INTO empleados VALUES (2, 'Ana Gerente', 8000, 1);
   INSERT INTO empleados VALUES (3, 'Luis Gerente', 8000, 1);

   -- Analistas (Reportan a Ana)
   INSERT INTO empleados VALUES (4, 'Sofia Analista', 4000, 2);
   INSERT INTO empleados VALUES (5, 'Pedro Analista', 4200, 2);

   -- Becario (Reporta a Sofia)
   INSERT INTO empleados VALUES (6, 'Juan Becario', 1500, 4);

   COMMIT;
END;
/
```

1. Subconsultas (La forma clásica)

Concepto: Una consulta dentro de otra. Puede estar en el SELECT, en el FROM o, más comúnmente, en el WHERE.

Escenario: Queremos saber **quién gana más que el salario promedio** de toda la empresa.

Código:

```
SELECT nombre, salario
FROM empleados
WHERE salario > (
  -- Subconsulta: Calcula el promedio una vez
  SELECT AVG(salario) FROM empleados
);
```

- **Cómo funciona:** Oracle ejecuta primero lo que está entre paréntesis (calcula el promedio, digamos 5950) y luego ejecuta la consulta externa usando ese valor.
- **Ventaja:** Rápida de escribir para filtros sencillos.
- **Desventaja:** Si anidas muchas subconsultas, el código se vuelve ilegible (el famoso "Spaghetti Code").

2. Common Table Expressions (CTE) - La cláusula WITH

Concepto: Defines una tabla temporal con nombre *antes* de tu consulta principal. Es como declarar variables en programación. Hace el código mucho más legible.

Escenario: Vamos a hacer lo mismo (superar el promedio), pero calculando también la diferencia exacta.

Código:

```
WITH SalarioPromedio AS (
  -- Definimos la lógica compleja aquí arriba
  SELECT AVG(salario) as avg_sal FROM empleados
)
SELECT
  e.nombre,
  e.salario,
```

```
  ROUND(s.avg_sal, 2) as promedio_global,
  e.salario - s.avg_sal as diferencia
FROM empleados e, SalarioPromedio s
WHERE e.salario > s.avg_sal;
```

- **Cómo funciona:**
    1. Oracle crea SalarioPromedio en memoria.
    2. La consulta principal lee de esa "tabla virtual".
- **Ventaja clave:** Si necesitas usar el promedio 5 veces en tu consulta principal, con una Subconsulta tendrías que calcularlo 5 veces. Con un CTE, **lo defines una vez y lo reutilizas**.
- **Legibilidad:** Lees el código de arriba a abajo, lógicamente.

3. Consultas Recursivas (La Magia Jerárquica)

Concepto: Un CTE que **se llama a sí mismo**. Es la *única* forma estándar de recorrer árboles genealógicos, organigramas o rutas de red en SQL.

Escenario: Queremos ver el organigrama completo: Quién es el jefe, en qué nivel de jerarquía está cada uno (Nivel 1 CEO, Nivel 2 Gerentes, etc.).

Código (Estructura Estándar):

En Oracle se usa WITH (sin la palabra RECURSIVE, aunque la lógica lo es).

```
WITH organigrama (emp_id, nombre, jefe_id, nivel, ruta_jefes) AS (
  -- 1. MIEMBRO ANCLA: El punto de partida (El CEO)
  SELECT emp_id, nombre, jefe_id, 1, CAST(nombre AS VARCHAR2(200))
  FROM empleados
  WHERE jefe_id IS NULL

  UNION ALL

  -- 2. MIEMBRO RECURSIVO: Se une con el resultado anterior
  SELECT e.emp_id, e.nombre, e.jefe_id, o.nivel + 1, o.ruta_jefes || ' -> ' || e.nombre
  FROM empleados e
  JOIN organigrama o ON e.jefe_id = o.emp_id -- Unimos empleado con su jefe del nivel anterior
```

)

SELECT * FROM organigrama ORDER BY nivel;

Explicación paso a paso:

1. **Ancla:** Primero encuentra a Carlos (Jefe NULL). Nivel 1.
2. **Iteración 1:** Busca a quien tenga como jefe a Carlos (Ana, Luis). Nivel 2.
3. **Iteración 2:** El CTE ahora contiene a Ana y Luis. Busca quién los tiene de jefe a ellos (Sofia, Pedro). Nivel 3.
4. **Iteración 3:** Busca quien reporta a Sofia (Juan). Nivel 4.
5. **Fin:** Nadie reporta a Juan. Termina.

Guía Definitiva: Cuándo usar cuál

Aquí tienes la tabla comparativa para tomar decisiones profesionales:

Característica	**Subconsulta**	**CTE (WITH)**	**Recursiva**
Sintaxis	SELECT ... (SELECT ...)	WITH nombre AS (...) SELECT ...	WITH nombre AS (Ancla UNION ALL Recursiva)
Legibilidad	Baja (si es compleja).	**Alta**.	Media (requiere práctica).
Reutilización	No (se repite el código).	**Sí** (se define una vez, se usa N veces).	Sí.
Caso de Uso	Filtros rápidos "de una sola vez" (WHERE IN, EXISTS).	Lógica compleja, reportes por pasos, limpiar código.	**Jerarquías**, árboles, grafos, rutas.
Performance	A veces el motor la optimiza bien, a veces la re-ejecuta.	El motor suele materializarla (guardarla en temp) si se usa mucho.	Costosa, pero necesaria para su propósito.

Práctica para consolidar

Para probar si lo has entendido, intenta este reto en Live SQL:

El Reto:

Usa la tabla mis_pedidos (de los ejercicios anteriores) y mis_clientes.

1. Usa un **CTE** para calcular el "Total Gastado por Cliente".
2. Luego, en el SELECT final, muestra solo los clientes que han gastado más de 5000.

El operador **APPLY** (introducido en Oracle 12c) es una de las herramientas más potentes y modernas de SQL.

Piénsalo de esta manera:

- Un JOIN normal une dos conjuntos de datos estáticos.
- Un APPLY funciona como un **bucle "For Each"**: Toma una fila de la tabla izquierda, y *para esa fila específica*, ejecuta una consulta en la derecha.

Esto te permite hacer cosas que con un JOIN normal son imposibles o muy difíciles, como pasar parámetros de una tabla a la subconsulta de la otra.

Existen dos variantes:

1. **CROSS APPLY**: Equivalente a un INNER JOIN (Si no hay resultados a la derecha, descarta la fila).
2. **OUTER APPLY**: Equivalente a un LEFT JOIN (Si no hay resultados a la derecha, mantiene la fila con NULLs).

El Escenario: "Top N por Categoría"

Este es el caso de uso clásico. Imagina que te pido: **"Dame los 2 pedidos más recientes de CADA cliente".**

Con un JOIN normal no puedes usar FETCH FIRST 2 ROWS ONLY porque limitaría el total global, no el total *por cliente*. Aquí es donde APPLY brilla.

Usaremos las tablas mis_clientes y mis_pedidos que ya tenemos.

1. Ejemplo Práctico: CROSS APPLY

Vamos a obtener los clientes que tienen pedidos, y para cada uno, traeremos **solo sus 2 pedidos más caros**.

Copia y pega esto en Live SQL:

SQL

```
SELECT
  c.nombre AS Cliente,
  p.pedido_id,
  p.monto,
  p.fecha_pedido
FROM mis_clientes c
CROSS APPLY (
  -- Esta subconsulta se ejecuta UNA VEZ POR CADA CLIENTE
```

```
  SELECT * FROM mis_pedidos sub_p
  WHERE sub_p.cliente_id = c.cliente_id  -- ¡Aquí está la magia! Referenciamos 'c' desde dentro
  ORDER BY sub_p.monto DESC
  FETCH FIRST 2 ROWS ONLY
) p;
```

Análisis:

- Nota cómo dentro del paréntesis usamos c.cliente_id. Un JOIN normal daría error ahí porque no puede "ver" la tabla de afuera antes de unirse.
- Como es CROSS APPLY, si un cliente no tiene pedidos, **no aparecerá** en el resultado.

2. Ejemplo Práctico: OUTER APPLY

Ahora queremos ver el reporte completo de clientes. Si tienen pedidos, muestra los 2 más caros. Si no tienen nada, muestra al cliente con NULL.

```
-- Primero insertemos un cliente sin pedidos para probar
INSERT INTO mis_clientes VALUES (999, 'Cliente Fantasma', 'Region_X');
COMMIT;

-- Consulta con OUTER APPLY
SELECT
  c.nombre AS Cliente,
  p.pedido_id,
  p.monto
FROM mis_clientes c
OUTER APPLY (
  SELECT * FROM mis_pedidos sub_p
  WHERE sub_p.cliente_id = c.cliente_id
  ORDER BY sub_p.monto DESC
  FETCH FIRST 2 ROWS ONLY
) p;
```

Resultado:

Verás al "Cliente Fantasma" en la lista, pero con pedido_id y monto vacíos.

- Esto es vital para reportes tipo "Maestro-Detalle" donde no quieres perder al maestro si no hay detalles.

3. APPLY vs. JOIN: ¿Cuándo usar cuál?

Para consolidar tu aprendizaje, comparenos las estrategias:

Característica	JOIN Estándar	OPERADOR APPLY
Lógica	Une Conjunto A con Conjunto B.	Para cada fila de A, ejecuta B.
Correlación	No puedes usar columnas de A dentro de la definición de B.	**Sí puedes**. Permite lógica compleja por fila.
Top-N por grupo	Difícil (requiere ROW_NUMBER() y subconsultas complejas).	**Muy Fácil** (directo con FETCH FIRST).
Funciones	Difícil unir con funciones que requieren parámetros.	Ideal para unir con funciones de tabla.
Rendimiento	Generalmente mejor para uniones masivas simples.	Puede ser costoso si la tabla izquierda tiene millones de filas (hace millones de subconsultas).

---

4. Veamos el Explain Plan de un APPLY

Es interesante ver cómo Oracle interpreta esto. A menudo lo transforma en una operación llamada LATERAL VIEW.

Ejecuta esto:

SQL

```
EXPLAIN PLAN FOR
SELECT c.nombre, p.monto
FROM mis_clientes c
CROSS APPLY (
  SELECT * FROM mis_pedidos p
  WHERE p.cliente_id = c.cliente_id
  FETCH FIRST 1 ROWS ONLY
```

) p;

SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY);

Interpretación:

Busca en la columna Operation algo como VIEW PUSHED PREDICATE o LATERAL VIEW.

Esto indica que Oracle está empujando el ID del cliente dentro de la vista de pedidos repetidamente para filtrar los datos antes de devolverlos.

Resumen

- Usa **CROSS APPLY** para filtrar y calcular datos complejos por cada fila (como un INNER JOIN vitaminado).
- Usa **OUTER APPLY** para conservar todas las filas de la izquierda aunque el cálculo de la derecha no devuelva nada (como un LEFT JOIN vitaminado).