```python
# Continuando con el DataFrame anterior

# Registrar el DataFrame como una vista temporal
df.createOrReplaceTempView("ventas")

# Ejecutar una consulta SQL compleja
resultadoSQL = spark.sql("""
    WITH ventas_ranking AS (
        SELECT
            fecha,
            producto,
            total_ventas,
            RANK() OVER (PARTITION BY producto ORDER BY total_ventas DESC) as rank
        FROM (
            SELECT
                fecha,
                producto,
                SUM(cantidad * precio) as total_ventas
            FROM ventas
            GROUP BY fecha, producto
        )
    )
    SELECT
        v.fecha,
        v.producto,
        v.total_ventas,
        vr.rank
    FROM ventas_ranking vr
    JOIN (
        SELECT
            fecha,
            producto,
            SUM(cantidad * precio) as total_ventas
        FROM ventas
        GROUP BY fecha, producto
    ) v ON v.fecha = vr.fecha AND v.producto = vr.producto
    WHERE vr.rank <= 3
    ORDER BY v.producto, vr.rank
""")

resultadoSQL.show()
```

Este código continúa trabajando con el DataFrame `df` que creaste en el ejemplo anterior y demuestra cómo interactuar con Spark SQL utilizando vistas temporales y consultas SQL complejas directamente dentro de tu código PySpark. Vamos a analizarlo:

1. **`df.createOrReplaceTempView("ventas")`**:
    - Este método registra el DataFrame `df` como una vista temporal llamada "ventas". Una vista temporal es una tabla virtual que existe solo durante la vida de la `SparkSession`. Puedes consultar esta vista utilizando sintaxis SQL a través del método `spark.sql()`.
    - `createOrReplaceTempView` asegura que si ya existe una vista temporal con el nombre "ventas", se reemplace con la nueva definición.
2. **`resultadoSQL = spark.sql(...)`**:
    - Este es el punto donde se ejecuta la consulta SQL. El método `spark.sql()` toma una cadena que contiene una consulta SQL y devuelve un nuevo DataFrame (`resultadoSQL`) que contiene los resultados de esa consulta.
    - La consulta SQL utiliza una expresión de tabla común (Common Table Expression - CTE) llamada `ventas_ranking` para lograr el resultado deseado.
3. **Análisis de la Consulta SQL:**
    - **`WITH ventas_ranking AS (...)`**: Define la CTE llamada `ventas_ranking`.
        - **`SELECT fecha, producto, total_ventas, RANK() OVER (PARTITION BY producto ORDER BY total_ventas DESC) as rank`**: Selecciona la fecha, el producto, las ventas totales y calcula el rango de las ventas totales dentro de cada producto.
            - **`RANK() OVER (PARTITION BY producto ORDER BY total_ventas DESC)`**: Esta es una función de ventana SQL.
                - **`PARTITION BY producto`**: Divide los datos en particiones basadas en los valores únicos de la columna `producto`. El ranking se calculará independientemente para cada producto.
                - **`ORDER BY total_ventas DESC`**: Ordena las filas dentro de cada partición de producto por la columna `total_ventas` en orden descendente (de mayor a menor venta).
                - **`as rank`**: Asigna el alias "rank" al resultado de la función `RANK()`. Los productos con las mayores ventas para una fecha dada dentro de su grupo tendrán un rango más bajo (1, 2, 3, etc.).
        - **`FROM (SELECT fecha, producto, SUM(cantidad * precio) as total_ventas FROM ventas GROUP BY fecha, producto)`**: Esta subconsulta interna calcula las ventas totales para cada combinación única de `fecha` y `producto` desde la vista temporal "ventas".
    - **`SELECT v.fecha, v.producto, v.total_ventas, vr.rank FROM ventas_ranking vr JOIN (...) v ON v.fecha = vr.fecha AND v.producto = vr.producto WHERE vr.rank <= 3 ORDER BY v.producto, vr.rank`**: Esta es la consulta principal que selecciona los resultados.
        - **`FROM ventas_ranking vr JOIN (...) v ON v.fecha = vr.fecha AND v.producto = vr.producto`**: Realiza una unión (JOIN) entre la CTE `ventas_ranking` (alias `vr`) y una subconsulta idéntica a la interna de la CTE (alias `v`). La unión se basa en la igualdad de las columnas `fecha` y `producto`. Esta unión es redundante en este caso, ya que toda la información necesaria ya está en `ventas_ranking`. Podría simplificarse.
        - **`WHERE vr.rank <= 3`**: Filtra los resultados para incluir solo las filas donde el rango (`rank`) es menor o igual a 3. Esto significa que se seleccionan las tres fechas con las mayores ventas para cada producto.
        - **`ORDER BY v.producto, vr.rank`**: Ordena los resultados primero por el `producto` y luego por el `rank` dentro de cada producto.

4. **`resultadoSQL.show()`**: Muestra el DataFrame resultante (`resultadoSQL`) que contiene las fechas, productos, ventas totales y su ranking dentro de cada producto, limitado a los tres primeros rangos.

En resumen, este código realiza las siguientes acciones:

- Registra el DataFrame de ventas como una vista temporal llamada "ventas".
- Ejecuta una consulta SQL que:
    - Calcula las ventas totales por fecha y producto.
    - Calcula el ranking de las ventas totales para cada producto en diferentes fechas (el día con las mayores ventas para un producto tendrá el rango 1).
    - Selecciona las fechas, productos y ventas totales para los tres primeros rankings de ventas dentro de cada producto.
- Muestra el DataFrame resultante, que contendrá las tres fechas con las mayores ventas para cada producto.

Este ejemplo ilustra la poderosa capacidad de PySpark para combinar la manipulación de DataFrames con la expresividad de SQL, permitiéndote realizar análisis de datos complejos de manera eficiente.