```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, rand, when, avg, count, sum, round, datediff,
current_date
from pyspark.sql.types import StructType, StructField, IntegerType, StringType,
DateType, DoubleType
import mysql.connector
from datetime import datetime, date

# Crear una sesión de Spark
spark = SparkSession.builder \
    .appName("Complex MySQL PySpark Query") \
    .getOrCreate()

# Parámetros de conexión MySQL
mysql_config = {
    'user': 'hive',
    'password': 'hive_password',
    'host': 'localhost',
    'database': 'pyspark_test_db',
    'raise_on_warnings': True
}

# Función para leer datos de MySQL usando el conector SQL de Python
def read_mysql_data():
    try:
        conn = mysql.connector.connect(**mysql_config)
        cursor = conn.cursor()

        query = "SELECT * FROM test_data"
        cursor.execute(query)

        # Obtener los resultados y los nombres de las columnas
        results = cursor.fetchall()
        column_names = [desc[0] for desc in cursor.description]

        print(f"Datos leídos de MySQL: {len(results)} filas")
        return results, column_names

    except mysql.connector.Error as err:
        print(f"Error: {err}")
        return None, None
    finally:
        if 'conn' in locals() and conn.is_connected():
            cursor.close()
            conn.close()
            print("Conexión MySQL cerrada.")
```

```
# Leer datos de MySQL
mysql_data, column_names = read_mysql_data()

if mysql_data and column_names:
    # Definir el esquema para el DataFrame de Spark
    schema = StructType([
        StructField("id", IntegerType(), True),
        StructField("random_value", DoubleType(), True),
        StructField("normal_distribution", DoubleType(), True),
        StructField("category", StringType(), True)
    ])

    # Crear un DataFrame de Spark a partir de los datos de MySQL
    mysql_df = spark.createDataFrame(mysql_data, schema=schema)

    # Mostrar una muestra de los datos de MySQL
    print("Muestra de datos de MySQL:")
    mysql_df.show(5)

    # Crear un DataFrame sintético en PySpark para simular datos adicionales
    synthetic_schema = StructType([
        StructField("user_id", IntegerType(), False),
        StructField("product", StringType(), False),
        StructField("purchase_date", DateType(), False),
        StructField("amount", IntegerType(), False)
    ])

    # Función para crear una fecha válida
    def create_date(i):
        year = 2023
        month = (i % 12) + 1
        day = (i % 28) + 1
        return date(year, month, day)

    # Generar datos sintéticos
    # Creación de una lista llamada 'synthetic_data' utilizando una comprensión de
listas.
    synthetic_data = [
        (
            i % 1000,                           # Toma el valor de 'i' y calcula
su módulo 1000, lo que limita el resultado entre 0 y 999.
            ["ProductA", "ProductB", "ProductC"][i % 3],  # Selecciona un producto
de la lista ["ProductA", "ProductB", "ProductC"] usando 'i % 3' como índice.
            create_date(i),                     # Llama a la función 'create_date'
pasando 'i' como argumento; esta función debe retornar un objeto de tipo 'date'.
            (i % 5 + 1) * 100                   # Calcula el módulo de 'i' con 5,
suma 1 (para obtener un valor entre 1 y 5), y luego lo multiplica por 100.
        )
```

```
        for i in range(10000)                    # Itera sobre el rango de 0 a
9999, generando 10,000 tuplas en total.
    ]

    synthetic_df = spark.createDataFrame(synthetic_data, schema=synthetic_schema)

    # Mostrar una muestra de los datos sintéticos
    print("Muestra de datos sintéticos:")
    synthetic_df.show(5)

    # Realizar una consulta compleja combinando ambos DataFrames

    # Se realiza un 'join' (unión) entre dos DataFrames: 'mysql_df' y
'synthetic_df'.
    complex_query_result = mysql_df.join(
        synthetic_df,
        mysql_df.id == synthetic_df.user_id,  # Condición de unión: la columna 'id'
de 'mysql_df' debe coincidir con la columna 'user_id' de 'synthetic_df'.
        "inner"  # Tipo de unión: inner join, solo mantiene las filas donde hay
coincidencia en ambas tablas.
    ).select(
        mysql_df.id.alias("user_id"),  # Selecciona y renombra la columna 'id' de
'mysql_df' a 'user_id'.
        mysql_df.category,             # Selecciona la columna 'category' del
DataFrame 'mysql_df'.
        synthetic_df.product,          # Selecciona la columna 'product' del
DataFrame 'synthetic_df'.
        synthetic_df.purchase_date,    # Selecciona la columna 'purchase_date' del
DataFrame 'synthetic_df'.
        synthetic_df.amount,           # Selecciona la columna 'amount' del
DataFrame 'synthetic_df'.
        mysql_df.random_value,         # Selecciona la columna 'random_value' del
DataFrame 'mysql_df'.
        mysql_df.normal_distribution   # Selecciona la columna 'normal_distribution'
del DataFrame 'mysql_df'.
    ).withColumn(
        "days_since_purchase", datediff(current_date(), col("purchase_date"))
        # Crea una nueva columna llamada 'days_since_purchase' que calcula el número
de días transcurridos desde 'purchase_date' hasta la fecha actual.
    ).withColumn(
        "value_segment", when(col("random_value") < 0.3, "Low")
                        .when(col("random_value") < 0.7, "Medium")
                        .otherwise("High")
        # Crea una nueva columna llamada 'value_segment' que categoriza
'random_value' en "Low", "Medium", o "High".
        # Si 'random_value' es menor a 0.3, se asigna "Low". Si está entre 0.3 y
0.7, se asigna "Medium". Para valores mayores o iguales a 0.7, se asigna "High".
    ).groupBy(
```

```
        "category", "product", "value_segment"
        # Agrupa los datos resultantes por las columnas 'category', 'product', y
'value_segment'.
    ).agg(
        count("user_id").alias("num_purchases"),
        # Cuenta el número de compras en cada grupo (cada combinación de 'category',
'product', y 'value_segment') y lo nombra como 'num_purchases'.
        round(avg("amount"), 2).alias("avg_purchase_amount"),
        # Calcula el promedio de la columna 'amount' en cada grupo, redondeado a 2
decimales, y lo nombra como 'avg_purchase_amount'.
        round(avg("days_since_purchase"), 2).alias("avg_days_since_purchase"),
        # Calcula el promedio de la columna 'days_since_purchase', redondeado a 2
decimales, y lo nombra como 'avg_days_since_purchase'.
        round(sum("amount"), 2).alias("total_revenue"),
        # Calcula la suma total de la columna 'amount' en cada grupo, redondeado a 2
decimales, y lo nombra como 'total_revenue'.
        round(avg("normal_distribution"), 2).alias("avg_normal_dist")
        # Calcula el promedio de la columna 'normal_distribution' en cada grupo,
redondeado a 2 decimales, y lo nombra como 'avg_normal_dist'.
    ).orderBy(
        "category", "product", "value_segment"
        # Ordena el resultado final por 'category', 'product', y 'value_segment'.
    )

    # Mostrar los resultados de la consulta compleja
    print("Resultados de la consulta compleja:")
    complex_query_result.show(20, truncate=False)

    # Realizar un análisis adicional
    print("Análisis adicional - Top 5 combinaciones de categoría y producto por
ingresos totales:")
    complex_query_result.orderBy(col("total_revenue").desc()).select(
        "category", "product", "total_revenue"
    ).show(5)

    # Calcular métricas globales
    print("Métricas globales:")
    global_metrics = complex_query_result.agg(
        round(sum("total_revenue"), 2).alias("global_revenue"),
        round(avg("avg_purchase_amount"), 2).alias("global_avg_purchase"),
        round(avg("avg_days_since_purchase"),
2).alias("global_avg_days_since_purchase")
    )
    global_metrics.show()
else:
    print("No se pudieron cargar los datos de MySQL.")

# Detener la sesión de Spark
```

```
spark.stop()
```

Este código realiza una serie de operaciones complejas combinando datos de MySQL con datos sintéticos creados en PySpark. Aquí está el desglose de lo que hace:

1. **Configuración Inicial:**
   - Importa las librerías necesarias de PySpark, `mysql.connector`, y `datetime`.
   - Crea una sesión de Spark, asegurándose de incluir el driver JDBC de MySQL forzado forzado con `.config("spark.jars.packages", "mysql:mysql-connector-java:8.0.28")`.
2. **Lectura de Datos de MySQL:**
   - Define una función `read_mysql_data()` para conectar a la base de datos MySQL y leer todos los datos de la tabla `test_data`.
   - Obtiene los resultados y los nombres de las columnas.
3. **Creación del DataFrame de PySpark a partir de MySQL:**
   - Define un esquema para el DataFrame de Spark que coincida con la estructura de la tabla `test_data` en MySQL. **Es importante que los tipos de datos en el esquema coincidan con los de la tabla MySQL.** En este caso, se asume que la columna `id` es de tipo `IntegerType`.
   - Crea un DataFrame de PySpark (`mysql_df`) utilizando los datos leídos de MySQL y el esquema definido.
4. **Creación de DataFrame Sintético:**
   - Define un esquema (`synthetic_schema`) para un DataFrame sintético que simula datos de compras de usuarios.
   - Genera una lista de tuplas (`synthetic_data`) con datos de ejemplo para usuarios, productos, fechas de compra y montos.
   - Crea un DataFrame de PySpark (`synthetic_df`) a partir de estos datos sintéticos.
5. **Consulta Compleja:**
   - **Join:** Realiza un `inner join` entre `mysql_df` y `synthetic_df` basándose en la columna `id` de `mysql_df` y la columna `user_id` de `synthetic_df`. Esto combina los datos de ambas fuentes donde hay una coincidencia en el ID de usuario.
   - **Selección de Columnas:** Selecciona un conjunto específico de columnas de ambos DataFrames, renombrando la columna `id` de `mysql_df` a `user_id` para claridad después del join.
   - **Cálculo de `days_since_purchase`:** Crea una nueva columna llamada `days_since_purchase` calculando la diferencia en días entre la fecha actual (`current_date()`) y la `purchase_date`.
   - **Creación de `value_segment`:** Crea una columna `value_segment` basada en la columna `random_value` de los datos de MySQL, categorizando los usuarios en "Low", "Medium" o "High".
   - **Agrupación:** Agrupa los datos combinados por `category` (de MySQL), `product` (del DataFrame sintético) y `value_segment`.
   - **Agregación:** Realiza varias funciones de agregación sobre los datos agrupados:
     - `count("user_id")`: Cuenta el número de compras (`num_purchases`).
     - `avg("amount")`: Calcula el promedio del monto de compra (`avg_purchase_amount`).

        - `avg("days_since_purchase")`: Calcula el promedio de días desde la última compra (`avg_days_since_purchase`).
        - `sum("amount")`: Calcula los ingresos totales (`total_revenue`).
        - `avg("normal_distribution")`: Calcula el promedio de la distribución normal (`avg_normal_dist`).
        - Todos los valores agregados se redondean a 2 decimales.
    - **Ordenamiento:** Ordena el resultado final por `category`, `product`, y `value_segment`.
6. **Mostrar Resultados:**
    - Imprime los resultados de la consulta compleja, mostrando hasta 20 filas sin truncar las columnas.
7. **Análisis Adicional:**
    - **Top 5 por Ingresos:** Ordena los resultados por `total_revenue` de forma descendente y muestra las 5 principales combinaciones de categoría y producto con sus ingresos totales.
    - **Métricas Globales:** Calcula métricas agregadas a nivel global, como los ingresos totales, el promedio del monto de compra y el promedio de días desde la última compra, sobre todos los grupos resultantes.
8. **Manejo de Errores y Cierre de Conexión:**
    - La función `read_mysql_data()` incluye un bloque `try...except...finally` para manejar posibles errores de conexión con MySQL y asegurar que la conexión se cierre correctamente.
9. **Detención de Spark:**
    - Finalmente, se detiene la sesión de Spark.

Este script es un ejemplo avanzado de cómo combinar datos de una base de datos relacional (MySQL) con datos generados en Spark y realizar análisis complejos utilizando las potentes funcionalidades de transformación y agregación de PySpark. Recuerda que para una integración más eficiente con bases de datos en entornos de producción, se suele preferir el uso directo de las fuentes de datos JDBC de Spark en lugar de cargar los datos al driver de Python.