Code Review:

1. Import Statements

- Importing several required modules: `os`, `requests`, PySpark's `DataFrame`, and various PySpark functions. This looks good and covers the essential components needed for data processing, Spark session management, and custom functions like `udf`.

2. Spark Session Initialization (`get_spark_session()`):

Creating a Spark session and set the log level to `"ERROR"`.

Error Handling The function catches exceptions and returns `False` in case of failure.

Suggestion to add logging in the exception block to better trace the issue instead of just returning `False`. Consider logging the exact exception for debugging.

```
except Exception as ex:
    print(f"Error initializing Spark session: {ex}")   Add this logging
```

3. **Dataframe Loading Function (`get_dataframes()`):**

Logic The function loads a CSV file into a DataFrame.

Error Handling Checks for `FileNotFoundError` and logs the error.

Optimization Add logging of successful file loading for tracking purposes and potential debugging.

Minor Issue The `raise` after catching `FileNotFoundError` should be removed if it's being tested only for a specific purpose (since raising again will crash the program).

```
print(f"Successfully loaded file: {file_path}")   Log successful load
```

4. Fetch NSE ID Function (`fetch_nse_id()`):

Logic The function makes an API call to fetch an NSE ID by hashing the `claim_id`.

Potential Optimization This is a network-dependent operation, and making API calls within a UDF can slow down processing for large datasets. It is generally not recommended to make API calls directly within UDFs as it can be inefficient in a distributed environment.

Suggestion If the data volume is large, consider batching API requests outside the Spark job, storing the results in a lookup table or cache, and then performing a join rather than making individual API calls.

5. Processing Claim Dates (`process_claim_dates()`):

Logic Converts the `DATE_OF_LOSS` and `CREATION_DATE` columns into the desired formats (`to_date`, `to_timestamp`) and adds a new column `SYSTEM_TIMESTAMP`.

Optimization This looks well-structured. The `cast("string")` part ensures the final format is correct, and `current_timestamp()` is efficient.

6. Claim Transformation (`transform_claim()`):

Logic Performs several transformations on the claim DataFrame:

- Removes prefixes from `CLAIM_ID`.
- Maps `CLAIM_TYPE` to `TRANSACTION_TYPE`.
- Maps `CLAIM_ID` to `TRANSACTION_DIRECTION`.
- Uses a UDF (`fetch_nse_udf`) to call the API and fetch the `NSE_ID`.

Optimization

UDF with API call As noted earlier, API calls in UDFs can cause performance bottlenecks. It's recommended to move API calls outside the Spark job if possible.

Use of `expr` The use of `expr` is good for conditional logic. It is efficient, but ensure the logic handles edge cases like missing values for `CLAIM_TYPE`.

Refactoring You could refactor this into smaller, modular transformations for easier debugging and readability, though performance-wise it's fine.

7. Processing Transactions (`process_transactions()`):

Logic Joins the `claim_df` and `contract_df` DataFrames based on certain conditions and performs additional transformations to match the desired schema.

Optimization

Join Efficiency Ensure that both DataFrames (`claim_df` and `contract_df`) are partitioned properly before the join. Large-scale joins can be optimized by repartitioning or bucketing on the join keys (`CONTRACT_ID`, `SOURCE_SYSTEM`).

Parquet Output Writing the DataFrame in Parquet format is efficient for structured data, and your schema definition using `StructType` ensures type consistency.

Suggestion If the datasets are large, you might want to tune the number of partitions or use `repartition` before writing to Parquet to avoid data skew or small file issues.

8. Main Logic

Logic This section handles loading the data, processing it, and then joining DataFrames before writing the result to disk.

Optimization

Spark Session Shutdown You're calling `spark.stop()` in the `finally` block, which is good practice to free resources.

File Existence Check Consider checking both the `claim_file` and `contract_file` in one go before processing to avoid unnecessary Spark session creation if files are missing.

Error Logging Error messages could be logged in more detail for each operation for better traceability.

Suggested Optimizations:

1. Avoid API Calls in UDFs

- API calls can severely slow down Spark jobs, especially in distributed environments. Consider moving this logic outside the UDF or using batch processing to get `NSE_ID` values beforehand.

2. Optimize Joins

- Ensure that your DataFrames are partitioned or broadcasted based on size and the join conditions. This will prevent expensive shuffles during join operations.

3. Refactor Transformations

- Refactor the code into smaller transformation steps and add appropriate logging in each step to make the code more maintainable.

4. Broadcast Smaller Tables

- If `contract_df` is small, consider using `broadcast()` in the join to avoid large shuffles.

```
from pyspark.sql.functions import broadcast

transactions_df = broadcast(ct_df_als).join(cl_df_als, ...)
```

5. Add Caching

- If you are reusing DataFrames, you could cache them to avoid recomputation.

```
claim_df.cache()
```

Final Thoughts:

Overall, your code is well-structured, with clear separation of concerns for data loading, transformations, and output writing. Addressing the UDF and join optimizations will improve performance, especially for large datasets. Adding detailed logging will also make debugging and monitoring more manageable in a production environment.

Suggestion-

Avoid API Calls in UDFs:

API calls can severely slow down Spark jobs, especially in distributed environments.

Consider moving this logic outside the UDF or using batch processing to get NSE_ID values beforehand.

To avoid making API calls within a UDF, you can move the logic of fetching NSE_ID values outside of the distributed Spark operations. The general approach is to first extract all unique CLAIM_ID values from your DataFrame, then make the API requests in a batch (outside of the UDF), store the results in a lookup table (like a dictionary), and finally join the original DataFrame with this lookup table.

Steps to Implement:

1. **Extract unique CLAIM_ID values** from the DataFrame.
2. **Batch the API requests** to fetch the NSE_ID values.
3. **Create a lookup dictionary** that maps CLAIM_ID to NSE_ID.
4. **Convert the dictionary into a DataFrame** and join it back with the original DataFrame.

To avoid making API calls within a UDF, you can move the logic of fetching `NSE_ID` values outside of the distributed Spark operations. The general approach is to first extract all unique `CLAIM_ID` values from your DataFrame, then make the API requests in a batch (outside of the UDF), store the results in a lookup table (like a dictionary), and finally join the original DataFrame with this lookup table.

Steps to Implement:

1. **Extract unique `CLAIM_ID` values** from the DataFrame.

2. **Batch the API requests** to fetch the `NSE_ID` values.

3. **Create a lookup dictionary** that maps `CLAIM_ID` to `NSE_ID`.

4. **Convert the dictionary into a DataFrame** and join it back with the original DataFrame.

Here's how you can write the code for this approach:

Code Example

```
import requests

from pyspark.sql import DataFrame

from pyspark.sql.functions import col

from pyspark.sql.types import StringType

from pyspark.sql import SparkSession


# Helper Method to batch fetch 'NSE_ID'
```

```
def batch_fetch_nse_ids(claim_ids):
    lookup_dict = {}
    url_template = "https://api.hashify.net/hash/md4/hex?value={}"

    for claim_id in claim_ids:
        try:
            response = requests.get(url_template.format(claim_id)).json()
            nse_id = response.get('Digest', None)
            lookup_dict[claim_id] = nse_id
        except Exception as ex:
            # Log or handle the exception for failed requests
            print(f"Failed to fetch NSE_ID for {claim_id}: {ex}")

    return lookup_dict


# Step 1: Extract unique CLAIM_IDs from the DataFrame
def get_unique_claim_ids(df: DataFrame):
    return df.select("CLAIM_ID").distinct().rdd.flatMap(lambda x: x).collect()


# Step 2: Convert lookup dictionary to a DataFrame
def create_nse_id_lookup_df(spark: SparkSession, lookup_dict: dict) -> DataFrame:
    lookup_list = [(k, v) for k, v in lookup_dict.items()]
    schema = ["CLAIM_ID", "NSE_ID"]

    return spark.createDataFrame(lookup_list, schema=schema)


# Step 3: Join the lookup DataFrame with the original claim DataFrame
def add_nse_id_to_claim_df(claim_df: DataFrame, nse_id_lookup_df: DataFrame) -> DataFrame:
    return claim_df.join(nse_id_lookup_df, on="CLAIM_ID", how="left")
```

```
# Example full pipeline using the new approach
def process_claim_with_nse_ids(spark: SparkSession, claim_df: DataFrame) -> DataFrame:
    # Extract unique CLAIM_IDs
    unique_claim_ids = get_unique_claim_ids(claim_df)

    # Batch fetch NSE_ID for unique CLAIM_IDs
    nse_id_lookup = batch_fetch_nse_ids(unique_claim_ids)

    # Create a lookup DataFrame from the NSE_ID dictionary
    nse_id_lookup_df = create_nse_id_lookup_df(spark, nse_id_lookup)

    # Join the lookup DataFrame with the original claim DataFrame
    claim_df_with_nse_id = add_nse_id_to_claim_df(claim_df, nse_id_lookup_df)

    return claim_df_with_nse_id


# Example usage:
if __name__ == "__main__":
    spark = SparkSession.builder.appName("TestApp").getOrCreate()

    # Load claim data
    claim_df = spark.read.option("header", "true").csv("claim.csv")

    # Process the claim DataFrame with batch API calls to get NSE_IDs
    claim_df_with_nse_ids = process_claim_with_nse_ids(spark, claim_df)

    # Show result
    claim_df_with_nse_ids.show()
```

Explanation:

1. **`batch_fetch_nse_ids(claim_ids)`:**

- This function takes a list of `CLAIM_ID` values, makes API calls in a loop to fetch `NSE_ID` for each `CLAIM_ID`, and stores the results in a Python dictionary (`lookup_dict`). The dictionary maps `CLAIM_ID` to `NSE_ID`.

2. **`get_unique_claim_ids(df)`:**

- This function extracts the unique `CLAIM_ID`s from the `claim_df` using `distinct()` and collects them into a list.

3. **`create_nse_id_lookup_df(spark, lookup_dict)`:**

- Converts the `lookup_dict` into a DataFrame, where each row contains a `CLAIM_ID` and its corresponding `NSE_ID`. This is necessary to perform a join operation with the original DataFrame.

4. **`add_nse_id_to_claim_df(claim_df, nse_id_lookup_df)`:**

- Joins the original `claim_df` with the newly created `nse_id_lookup_df` on `CLAIM_ID`, effectively adding the `NSE_ID` values to the original DataFrame.

5. **Pipeline Execution:**

- The full pipeline is executed in `process_claim_with_nse_ids()`, where the steps are integrated:
  - Extract unique `CLAIM_ID`s.
  - Batch fetch `NSE_ID` values.
  - Join the fetched `NSE_ID`s back to the `claim_df`.

Benefits:

- **Efficiency**: You avoid making API calls inside a UDF and instead fetch all `NSE_ID` values in one go (batching). This will be much faster than calling the API in a UDF during distributed operations.

- **Scalability**: By batching the API calls, you reduce the time complexity and overhead associated with making many individual API calls, improving overall job performance.

Optimizations:

- You could further optimize the API call process by introducing retries for failed API calls or parallelizing the API requests using multi-threading or asynchronous requests.