```
# --- 1. Setup ---
import os
from pyspark.sql import SparkSession
from pyspark.sql.functions import (
    col, to_date, lit, datediff, add_months, expr, lag,
    sum as pyspark_sum, avg as pyspark_avg, count as pyspark_count,
    min as pyspark_min, max as pyspark_max,
    when, broadcast, coalesce, last_day, trunc,
    countDistinct, year, month, dayofmonth, date_add, date_sub,
    trim # Added trim for potential use
)
from pyspark.sql.window import Window
from pyspark.sql.types import StructType, StructField, StringType, DateType, DoubleType, IntegerType, LongType
import pandas as pd

# --- Mount Google Drive (if using Google Colab) ---
try:
    from google.colab import drive
    drive.mount('/content/drive')
    print("Google Drive mounted successfully.")
    google_drive_base_path = '/content/drive/MyDrive/'
except ImportError:
    print("Not running in Google Colab. Assuming local file system.")
    google_drive_base_path = ""

# Initialize SparkSession
spark = SparkSession.builder.appName("ReligareABTGeneration") \
    .config("spark.sql.legacy.timeParserPolicy", "LEGACY") \
    .config("spark.sql.shuffle.partitions", "200") \
    .config("spark.sql.adaptive.enabled", "true") \
    .getOrCreate()

# Define base paths
input_tables_dir = os.path.join(google_drive_base_path, 'Tables/')
input_log_new_dir = os.path.join(google_drive_base_path, 'LOG_NEW/')
output_abt_dir = os.path.join(google_drive_base_path, 'Tables/output_abt_final_pred/')

# --- Input File Paths ---
client_details_path = os.path.join(input_tables_dir, "client_details.txt")
trade_data_path = os.path.join(input_tables_dir, "trade_data.txt")
deposit_data_path = os.path.join(input_tables_dir, "deposit_data.txt")
payout_data_path = os.path.join(input_tables_dir, "payout_data.txt")
login_data_path_pattern = os.path.join(input_log_new_dir, "LOGIN_*.txt")
aum_data_path = os.path.join(input_tables_dir, "AUM.txt")
cashbal_data_path = os.path.join(input_tables_dir, "CASHBAL.txt")

# --- Output Path ---
output_file_name_base = "predictive_abt_religare_churn_2021_2023"
output_path_parquet = os.path.join(output_abt_dir, f"{output_file_name_base}.parquet")

# In Cell 1 (Setup)
output_abt_dir = os.path.join(google_drive_base_path, 'Tables/output_abt_final_pred/')
temp_abt_path = os.path.join(output_abt_dir, "temp_abt_in_progress.parquet") # temp_abt_path is INSIDE output_abt_dir

# Ensure the output directory exists (this covers the parent of temp_abt_path)
if not os.path.exists(output_abt_dir): # Checks if /content/drive/MyDrive/Tables/output_abt_final_pred/ exists
    try:
        os.makedirs(output_abt_dir) # Creates it if it doesn't
        print(f"Created directory: {output_abt_dir}")
    except Exception as e:
        print(f"Could not create directory {output_abt_dir}: {e}")

# --- Constants ---
LOOKBACK_PERIODS_DAYS = [30, 60, 90, 180, 270, 365]
CHURN_WINDOWS_DAYS = [60, 90, 270, 365]
MAX_ACTIVITY_LOOKFORWARD_NEEDED = max(CHURN_WINDOWS_DAYS)
SNAPSHOT_START_DATE_STR = "2021-01-01"

print("Setup Complete.")
print(f"Client Details Path: {client_details_path}")
print(f"Trade Data Path: {trade_data_path}")
print(f"Deposit Data Path: {deposit_data_path}")
print(f"Payout Data Path: {payout_data_path}")
print(f"Login Data Pattern: {login_data_path_pattern}")
print(f"AUM Data Path: {aum_data_path}")
print(f"Cash Balance Data Path: {cashbal_data_path}")
print(f"Output ABT Path: {output_path_parquet}")
```

```
Mounted at /content/drive
Google Drive mounted successfully.
```

```
Setup Complete.
Client Details Path: /content/drive/MyDrive/Tables/client_details.txt
Trade Data Path: /content/drive/MyDrive/Tables/trade_data.txt
Deposit Data Path: /content/drive/MyDrive/Tables/deposit_data.txt
Payout Data Path: /content/drive/MyDrive/Tables/payout_data.txt
Login Data Pattern: /content/drive/MyDrive/LOG_NEW/LOGIN_*.txt
AUM Data Path: /content/drive/MyDrive/Tables/AUM.txt
Cash Balance Data Path: /content/drive/MyDrive/Tables/CASHBAL.txt
Output ABT Path: /content/drive/MyDrive/Tables/output_abt_final_pred/predictive_abt_religare_churn_2021_2023.parquet
```

```
# --- 2. Data Loading Functions ---

def load_client_details(spark, path):
    print(f"Loading Client Master from: {path}")
    df = spark.read.format("csv") \
        .option("header", "true") \
        .option("delimiter", ",") \
        .load(path)
    df = df.select(
        trim(col("CLIENTCODE")).alias("ClientCode"),
        to_date(col("ACTIVATIONDATE"), "dd/MM/yyyy").alias("ActivationDate") # Confirmed
    ).filter(col("ActivationDate").isNotNull()).distinct()
    print(f"Loaded {df.count()} distinct clients with activation dates.")
    return df

def load_trade_data(spark, path):
    print(f"Loading Trade Data from: {path}")
    df = spark.read.format("csv") \
        .option("header", "true") \
        .option("delimiter", ",") \
        .load(path)
    df = df.select(
        trim(col("CLIENTCODE")).alias("ClientCode"),
        to_date(col("TRADE_DATE"), "dd/MM/yyyy").alias("ActivityDate"), # Confirmed
        col("TOTAL_GROSS_BROKERAGE_DAY").cast(DoubleType()).alias("GrossBrokerage")
    ).filter(col("ActivityDate").isNotNull())
    print(f"Loaded {df.count()} trade records.")
    return df

def load_login_data(spark, path_pattern):
    print(f"Loading Login Data from: {path_pattern}")
    login_schema = StructType([
        StructField("ClientCode_raw", StringType(), True),
        StructField("LoginDate_str", StringType(), True)
    ])
    df_raw = spark.read.format("csv") \
        .schema(login_schema) \
        .option("delimiter", ",") \
        .load(path_pattern)
    df = df_raw.select(
        trim(col("ClientCode_raw")).alias("ClientCode"),
        to_date(col("LoginDate_str"), "dd/MM/yyyy").alias("ActivityDate") # Confirmed format style
    ).filter(col("ActivityDate").isNotNull())
    print(f"Loaded {df.count()} login records.")
    return df

def load_funding_data(spark, path, date_col_name, amount_col_name, activity_type_name):
    print(f"Loading {activity_type_name} Data from: {path}")
    # Identifier: CLIENTCODE
    # Date Format: dd/MM/yyyy
    # Delimiter: comma (,)
    df = spark.read.format("csv") \
        .option("header", "true") \
        .option("delimiter", ",") \
        .load(path)
    df = df.select(
        trim(col("CLIENTCODE")).alias("ClientCode"), # Standardized identifier
        to_date(col(date_col_name), "dd/MM/yyyy").alias("ActivityDate"), # UPDATED date format
        col(amount_col_name).cast(DoubleType()).alias("Amount")
    ).filter(col("ActivityDate").isNotNull() & col("Amount").isNotNull())
    print(f"Loaded {df.count()} {activity_type_name} records.")
    return df

def load_aum_data(spark, path):
    print(f"Loading AUM Data from: {path}")
    # MONTH,CLIENTCODE,MONTHLYAUM,RUNNINGTOTALAUM
    # MONTH format: DD/MM/YYYY (start of month)
    # Delimiter: comma (,)
    df = spark.read.format("csv") \
        .option("header", "true") \
        .option("delimiter", ",") \
        .load(path)
```

```
        df = df.select(
            to_date(col("MONTH"), "dd/MM/yyyy").alias("AUMMonthStartDate"),
            trim(col("CLIENTCODE")).alias("ClientCode"),
            col("MONTHLYAUM").cast(DoubleType()).alias("MonthlyAUM"),
            col("RUNNINGTOTALAUM").cast(DoubleType()).alias("RunningTotalAUM")
        ).filter(col("AUMMonthStartDate").isNotNull())
        print(f"Loaded {df.count()} AUM records.")
        return df

    def load_cash_balance_data(spark, path):
        print(f"Loading Cash Balance Data from: {path}")
        # CLIENTCODE,DDATE,CASHBAL
        # DDATE format: DD/MM/YYYY (end of month)
        # Delimiter: comma (,)
        df = spark.read.format("csv") \
            .option("header", "true") \
            .option("delimiter", ",") \
            .load(path)
        df = df.select(
            trim(col("CLIENTCODE")).alias("ClientCode"),
            to_date(col("DDATE"), "dd/MM/yyyy").alias("BalanceDateEOM"),
            col("CASHBAL").cast(DoubleType()).alias("CashBalance")
        ).filter(col("BalanceDateEOM").isNotNull())
        print(f"Loaded {df.count()} EOM cash balance records.")
        return df

    print("Data loading functions defined with updated formats.")
```

```
Data loading functions defined with updated formats.
```

```
    # --- 3. Load All Raw Data ---
    client_master_df = load_client_details(spark, client_details_path)
    trades_master_df = load_trade_data(spark, trade_data_path)
    logins_master_df = load_login_data(spark, login_data_path_pattern)

    # For deposits:
    deposits_master_df = load_funding_data(spark, deposit_data_path,
                                           date_col_name="DEPOSIT_DATE",     # Confirmed
                                           amount_col_name="DEPOSIT_AMOUNT", # Confirmed
                                           activity_type_name="Deposit")

    # For payouts: Corrected column names based on your sample
    payouts_master_df = load_funding_data(spark, payout_data_path,
                                          date_col_name="PAYOUT_DATE",       # <--- CORRECTED
                                          amount_col_name="PAYOUT_AMOUNT",   # <--- CORRECTED
                                          activity_type_name="Payout")

    # Load AUM and Cash Balance Data
    aum_master_df = load_aum_data(spark, aum_data_path)
    cash_balance_master_df = load_cash_balance_data(spark, cashbal_data_path)

    # Persist key master dataframes
    persisted_dfs = [client_master_df, trades_master_df, logins_master_df,
                     deposits_master_df, payouts_master_df, aum_master_df, cash_balance_master_df]
    persisted_df_names = ["ClientMaster", "Trades", "Logins", "Deposits", "Payouts", "AUM", "CashBalance"] # For better print messages

    for i, df_to_persist in enumerate(persisted_dfs):
        df_name = persisted_df_names[i]
        if df_to_persist:
            try:
                # Check if DataFrame is not empty before persisting
                if df_to_persist.head(1):
                     df_to_persist.persist()
                     print(f"Persisted DataFrame: {df_name}")
                else:
                    print(f"DataFrame {df_name} is empty. Not persisting.")
            except Exception as e_persist:
                print(f"Error during persist/check for DataFrame {df_name}: {e_persist}")

    print("\nSample of loaded data & Schemas:")
    for df_name, df_sample in zip(persisted_df_names, persisted_dfs):
        if df_sample:
            print(f"\n--- Sample and Schema for {df_name} ---")
            df_sample.show(3, truncate=False)
            df_sample.printSchema()
```

```
Loading Client Master from: /content/drive/MyDrive/Tables/client_details.txt
Loaded 1316511 distinct clients with activation dates.
Loading Trade Data from: /content/drive/MyDrive/Tables/trade_data.txt
Loaded 17254800 trade records.
```

```
Loading Login Data from: /content/drive/MyDrive/LOG_NEW/LOGIN_*.txt
Loaded 176232060 login records.
Loading Deposit Data from: /content/drive/MyDrive/Tables/deposit_data.txt
Loaded 3107112 Deposit records.
Loading Payout Data from: /content/drive/MyDrive/Tables/payout_data.txt
Loaded 1868336 Payout records.
Loading AUM Data from: /content/drive/MyDrive/Tables/AUM.txt
Loaded 10432022 AUM records.
Loading Cash Balance Data from: /content/drive/MyDrive/Tables/CASHBAL.txt
Loaded 13341533 EOM cash balance records.
Persisted DataFrame: ClientMaster
Persisted DataFrame: Trades
Persisted DataFrame: Logins
Persisted DataFrame: Deposits
Persisted DataFrame: Payouts
Persisted DataFrame: AUM
Persisted DataFrame: CashBalance

Sample of loaded data & Schemas:

--- Sample and Schema for ClientMaster ---
+----------+--------------+
|ClientCode|ActivationDate|
+----------+--------------+
|AA1291    |2007-01-15    |
|AA1365    |2007-02-27    |
|AA1505    |2007-05-10    |
+----------+--------------+
only showing top 3 rows

root
 |-- ClientCode: string (nullable = true)
 |-- ActivationDate: date (nullable = true)


--- Sample and Schema for Trades ---
+----------+------------+----------------+
|ClientCode|ActivityDate|GrossBrokerage  |
+----------+------------+----------------+
|SD9627    |2020-08-04  |596.815017700195|
|AV3818    |2020-08-04  |40.0050010681152|
|NR2513    |2020-08-04  |56.8474998474121|
+----------+------------+----------------+
only showing top 3 rows

root
 |-- ClientCode: string (nullable = true)
 |-- ActivityDate: date (nullable = true)
 |-- GrossBrokerage: double (nullable = true)


--- Sample and Schema for Logins ---
+----------+------------+
|ClientCode|ActivityDate|
```

```
# --- 4. Determine Overall Data Date Range and Generate Snapshot Dates ---

# To determine a reasonable snapshot range, find min/max dates from activity data
# We need a robust way to get min/max across multiple activity DataFrames
# Ensure all activity DataFrames have an 'ActivityDate' column after loading

activity_dfs_for_range = []
if 'trades_master_df' in locals() and trades_master_df: activity_dfs_for_range.append(trades_master_df.select("ActivityDate"))
if 'logins_master_df' in locals() and logins_master_df: activity_dfs_for_range.append(logins_master_df.select("ActivityDate"))
if 'deposits_master_df' in locals() and deposits_master_df: activity_dfs_for_range.append(deposits_master_df.select("ActivityDate"))
if 'payouts_master_df' in locals() and payouts_master_df: activity_dfs_for_range.append(payouts_master_df.select("ActivityDate"))

overall_min_date = None
overall_max_date = None

if activity_dfs_for_range:
    # Union all activity dates to find global min/max
    all_activity_dates_unioned_df = activity_dfs_for_range[0]
    for i in range(1, len(activity_dfs_for_range)):
        all_activity_dates_unioned_df = all_activity_dates_unioned_df.unionByName(activity_dfs_for_range[i])

    min_max_dates_row = all_activity_dates_unioned_df.agg(
        pyspark_min("ActivityDate").alias("GlobalMinDate"),
        pyspark_max("ActivityDate").alias("GlobalMaxDate")
    ).first()

    if min_max_dates_row:
        overall_min_date = min_max_dates_row["GlobalMinDate"]
        overall_max_date = min_max_dates_row["GlobalMaxDate"]

print(f"Overall Min Activity Date from loaded data: {overall_min_date}")
print(f"Overall Max Activity Date from loaded data: {overall_max_date}")
```

```
# Define snapshot period
if overall_max_date:
    snapshot_start_date_pd = pd.to_datetime(SNAPSHOT_START_DATE_STR)
    # Max snapshot date allows for the longest churn window look-forward
    snapshot_end_date_pd = pd.to_datetime(overall_max_date) - pd.Timedelta(days=MAX_ACTIVITY_LOOKFORWARD_NEEDED)

    if snapshot_end_date_pd < snapshot_start_date_pd:
        print(f"Warning: Snapshot end date ({snapshot_end_date_pd}) is before start date ({snapshot_start_date_pd}). "
              f"Not enough data for full look-forward for {MAX_ACTIVITY_LOOKFORWARD_NEEDED} days. "
              f"Adjusting analysis period or MAX_ACTIVITY_LOOKFORWARD_NEEDED might be necessary if this is unexpected.")
        snapshots_df = None # Or handle by creating an empty DataFrame with schema
    else:
        print(f"Snapshot Start Date for generation: {snapshot_start_date_pd.strftime('%Y-%m-%d')}")
        print(f"Snapshot End Date for generation (calculated): {snapshot_end_date_pd.strftime('%Y-%m-%d')}")

        # Generate monthly snapshot dates (end of month)
        snapshot_dates_pd_series = pd.date_range(start=snapshot_start_date_pd, end=snapshot_end_date_pd, freq='ME') # 'ME' for Month-End
        snapshot_dates_list_of_tuples = [(d.strftime('%Y-%m-%d'),) for d in snapshot_dates_pd_series]

        if snapshot_dates_list_of_tuples:
            snapshots_df = spark.createDataFrame(snapshot_dates_list_of_tuples, ["SnapshotDate_str"])
            snapshots_df = snapshots_df.withColumn("SnapshotDate", to_date(col("SnapshotDate_str"), "yyyy-MM-dd")) \
                                       .select("SnapshotDate")
            if snapshots_df.count() > 0:
                snapshots_df.persist()
                print(f"\nGenerated {snapshots_df.count()} snapshot dates:")
                snapshots_df.orderBy("SnapshotDate").show(5)
                snapshots_df.orderBy(col("SnapshotDate").desc()).show(5)
            else:
                print("No snapshot dates generated (empty list). Check date ranges and logic.")
                snapshots_df = None # Ensure it's defined as None if not created
        else:
            print("No snapshot dates generated (empty list after pandas date_range). Check date ranges.")
            snapshots_df = None
else:
    print("Could not determine overall_max_date from activity data. Cannot generate snapshots.")
    snapshots_df = None
```

```
Overall Min Activity Date from loaded data: 2020-08-03
Overall Max Activity Date from loaded data: 2024-04-30
Snapshot Start Date for generation: 2021-01-01
Snapshot End Date for generation (calculated): 2023-05-01

Generated 28 snapshot dates:
+------------+
|SnapshotDate|
+------------+
|  2021-01-31|
|  2021-02-28|
|  2021-03-31|
|  2021-04-30|
|  2021-05-31|
+------------+
only showing top 5 rows

+------------+
|SnapshotDate|
+------------+
|  2023-04-30|
|  2023-03-31|
|  2023-02-28|
|  2023-01-31|
|  2022-12-31|
+------------+
only showing top 5 rows
```

```
# --- 5. Create Client Universe and Base ABT Structure ---

if 'client_master_df' in locals() and client_master_df and \
   'snapshots_df' in locals() and snapshots_df is not None and snapshots_df.count() > 0:

    print("\n--- Creating Client-Snapshot Base ABT ---")
    client_universe_df = client_master_df.select("ClientCode", "ActivationDate").distinct()
    print(f"Total unique clients from master data: {client_universe_df.count()}")

    client_snapshot_base_df = client_universe_df.crossJoin(broadcast(snapshots_df))
    client_snapshot_base_df = client_snapshot_base_df.filter(col("SnapshotDate") >= col("ActivationDate"))

    client_snapshot_base_df = client_snapshot_base_df.withColumn(
        "Tenure_Days",
        datediff(col("SnapshotDate"), col("ActivationDate"))
```

```
    )

    # Persist before writing and count for verification
    client_snapshot_base_df.persist()
    base_abt_count = client_snapshot_base_df.count() # Action to materialize
    print(f"Total client-snapshot records in initial base ABT: {base_abt_count}")
    print("Sample of initial base ABT:")
    client_snapshot_base_df.orderBy("ClientCode", "SnapshotDate").show(5, truncate=False)

    # --- Initial Save of Base ABT ---
    if base_abt_count > 0:
        print(f"Writing initial base ABT to: {temp_abt_path}")
        client_snapshot_base_df.write.mode("overwrite").parquet(temp_abt_path)
        print("Initial base ABT written successfully.")
    else:
        print("Base ABT is empty, not writing to disk.")

    if client_snapshot_base_df.is_cached:
        client_snapshot_base_df.unpersist()
else:
    print("Skipping client-snapshot base generation/save due to missing client_master_df or snapshots_df.")
```

```
--- Creating Client-Snapshot Base ABT ---
Total unique clients from master data: 1316511
Total client-snapshot records in initial base ABT: 33681785
Sample of initial base ABT:
+----------+--------------+------------+-----------+
|ClientCode|ActivationDate|SnapshotDate|Tenure_Days|
+----------+--------------+------------+-----------+
|A*        |2005-04-13    |2021-01-31  |5772       |
|A*        |2005-04-13    |2021-02-28  |5800       |
|A*        |2005-04-13    |2021-03-31  |5831       |
|A*        |2005-04-13    |2021-04-30  |5861       |
|A*        |2005-04-13    |2021-05-31  |5892       |
+----------+--------------+------------+-----------+
only showing top 5 rows

Writing initial base ABT to: /content/drive/MyDrive/Tables/output_abt_final_pred/temp_abt_in_progress.parquet
Initial base ABT written successfully.
```

```
# --- 6. Recency Features ---
print("\n--- Calculating Recency Features ---")
try:
    # Read the current ABT from disk
    abt_df = spark.read.parquet(temp_abt_path)
    print(f"Read ABT from {temp_abt_path} with {abt_df.count()} rows for Recency features.")

    # Helper function calculate_single_recency_feature
    def calculate_single_recency_feature(main_abt_df, activity_df, activity_df_alias_str,
                                         activity_pk_col, activity_date_col_in_activity_df,
                                         feature_prefix):
        print(f"    Calculating recency for {feature_prefix}...")
        act_df_aliased = activity_df.alias(activity_df_alias_str)
        last_activity_dates = main_abt_df.alias("abt").join(
            act_df_aliased,
            (col(f"abt.ClientCode") == col(f"{activity_df_alias_str}.{activity_pk_col}")) & \
            (col(f"{activity_df_alias_str}.{activity_date_col_in_activity_df}") <= col("abt.SnapshotDate")),
            "left"
        ).groupBy(col("abt.ClientCode"), col("abt.SnapshotDate")) \
         .agg(pyspark_max(col(f"{activity_df_alias_str}.{activity_date_col_in_activity_df}")).alias(f"Last_{feature_prefix}_Date"))

        updated_abt_df = main_abt_df.join(
            last_activity_dates,
            ["ClientCode", "SnapshotDate"],
            "left"
        )

        updated_abt_df = updated_abt_df.withColumn(
            f"Days_Since_Last_{feature_prefix}",
            when(col(f"Last_{feature_prefix}_Date").isNotNull(),
                 datediff(col("SnapshotDate"), col(f"Last_{feature_prefix}_Date")))
            .otherwise(None)
        )
        return updated_abt_df

    # Apply for each activity type
    abt_df = calculate_single_recency_feature(abt_df, trades_master_df, "t", "ClientCode", "ActivityDate", "Trade")
    abt_df = calculate_single_recency_feature(abt_df, logins_master_df, "l", "ClientCode", "ActivityDate", "Login")
    abt_df = calculate_single_recency_feature(abt_df, deposits_master_df, "d", "ClientCode", "ActivityDate", "Deposit")
    abt_df = calculate_single_recency_feature(abt_df, payouts_master_df, "p", "ClientCode", "ActivityDate", "Payout")
```

```
        # Persist before writing and count
        abt_df.persist()
        recency_abt_count = abt_df.count()
        print("Recency features calculated. Sample:")
        recency_cols_to_show = ["ClientCode", "SnapshotDate", "Tenure_Days"] + \
                               [f"Last_{pfx}_Date" for pfx in ["Trade", "Login", "Deposit", "Payout"] if f"Last_{pfx}_Date" in abt_df.columns] + \
                               [f"Days_Since_Last_{pfx}" for pfx in ["Trade", "Login", "Deposit", "Payout"] if f"Days_Since_Last_{pfx}" in abt_df.columns]
        existing_recency_cols = [c for c in recency_cols_to_show if c in abt_df.columns]
        abt_df.select(existing_recency_cols).orderBy("ClientCode", "SnapshotDate").show(5, truncate=False)
        print(f"ABT DF Count after Recency: {recency_abt_count}")

        # --- Write Updated ABT to Disk ---
        if recency_abt_count > 0 :
            print(f"Writing ABT with Recency features to: {temp_abt_path}")
            abt_df.write.mode("overwrite").parquet(temp_abt_path)
            print("ABT with Recency features written successfully.")
        else:
            print("ABT with Recency features is empty. Not writing.")

        if abt_df.is_cached:
            abt_df.unpersist()

    except FileNotFoundError:
        print(f"ERROR: Could not read temporary ABT from {temp_abt_path}. Ensure Cell 5 (or previous step) ran successfully and wrote the fi
    except Exception as e:
        print(f"Error in Recency Feature Engineering or writing: {e}")
        raise e
```

```
--- Calculating Recency Features ---
Read ABT from /content/drive/MyDrive/Tables/output_abt_final_pred/temp_abt_in_progress.parquet with 33681785 rows for Recency featur
    Calculating recency for Trade...
    Calculating recency for Login...
    Calculating recency for Deposit...
    Calculating recency for Payout...
Recency features calculated. Sample:
+----------+------------+-----------+---------------+---------------+-----------------+----------------+---------------------+-----
|ClientCode|SnapshotDate|Tenure_Days|Last_Trade_Date|Last_Login_Date|Last_Deposit_Date|Last_Payout_Date|Days_Since_Last_Trade|Days_S
+----------+------------+-----------+---------------+---------------+-----------------+----------------+---------------------+-----
|A*        |2021-01-31  |5772       |NULL           |NULL           |NULL             |NULL            |NULL                 |NULL
|A*        |2021-02-28  |5800       |NULL           |NULL           |NULL             |NULL            |NULL                 |NULL
|A*        |2021-03-31  |5831       |NULL           |NULL           |NULL             |NULL            |NULL                 |NULL
|A*        |2021-04-30  |5861       |NULL           |NULL           |NULL             |NULL            |NULL                 |NULL
|A*        |2021-05-31  |5892       |NULL           |NULL           |NULL             |NULL            |NULL                 |NULL
+----------+------------+-----------+---------------+---------------+-----------------+----------------+---------------------+-----
only showing top 5 rows

ABT DF Count after Recency: 33681785
Writing ABT with Recency features to: /content/drive/MyDrive/Tables/output_abt_final_pred/temp_abt_in_progress.parquet
ABT with Recency features written successfully.
```

```
# --- 7. Frequency and Monetary Features (Lookback Periods - Independent Blocks) ---
print("\n--- Calculating Frequency and Monetary Features ---")
try:
    # Read the current ABT (with Recency features) from disk
    abt_df = spark.read.parquet(temp_abt_path)
    print(f"Read ABT from {temp_abt_path} with {abt_df.count()} rows for Freq/Monetary features.")

    base_keys_df = abt_df.select("ClientCode", "SnapshotDate").distinct()
    if base_keys_df.is_cached: base_keys_df.unpersist()
    base_keys_df.persist()
    base_keys_count = base_keys_df.count() # Action
    print(f"Persisted base_keys_df with {base_keys_count} distinct client-snapshot pairs.")

    # Helper function calculate_feature_block (as defined in the "Independent Blocks" strategy from previous response)
    # ... (definition of calculate_feature_block remains the same - ensure it's the one that returns a block of features for one activit
    def calculate_feature_block(keys_df_input, activity_df_input,
                                activity_pk_col, activity_date_col_in_activity_df,
                                value_col_name_for_sum,
                                activity_alias_str, feature_name_prefix, lookback_days_list):
        print(f"  Calculating feature block for {feature_name_prefix}...")
        keys_df = keys_df_input.alias("s_keys")
        activity_df_aliased = activity_df_input.alias(activity_alias_str)
        current_block_features_df = keys_df.select(
            col("s_keys.ClientCode").alias("ClientCode_block_key"),
            col("s_keys.SnapshotDate").alias("SnapshotDate_block_key")
        )
        for days in lookback_days_list:
            print(f"    Calculating for {days}-day lookback for {feature_name_prefix}...")
            join_condition = (
                (col("s_keys.ClientCode") == col(f"{activity_alias_str}.{activity_pk_col}")) &
                (col(f"{activity_alias_str}.{activity_date_col_in_activity_df}") <= col("s_keys.SnapshotDate")) &
```

```
            (col(f"{activity_alias_str}.{activity_date_col_in_activity_df}") > date_sub(col("s_keys.SnapshotDate"), days))
        )
        aggregated_lookback_df = keys_df.join(
            activity_df_aliased, join_condition, "left"
        ).groupBy(col("s_keys.ClientCode"), col("s_keys.SnapshotDate")) \
         .agg(
            countDistinct(col(f"{activity_alias_str}.{activity_date_col_in_activity_df}")).alias(f"{feature_name_prefix}_Days_Count_
            pyspark_count(col(f"{activity_alias_str}.{activity_date_col_in_activity_df}")).alias(f"{feature_name_prefix}_Txns_Count_
            *(
                [pyspark_sum(col(f"{activity_alias_str}.{value_col_name_for_sum}")).alias(f"{feature_name_prefix}_Sum_{days}D")]
                if value_col_name_for_sum else []
            )
        ).select(
            col("s_keys.ClientCode").alias("ClientCode_agg_key"),
            col("s_keys.SnapshotDate").alias("SnapshotDate_agg_key"),
            col(f"{feature_name_prefix}_Days_Count_{days}D"),
            col(f"{feature_name_prefix}_Txns_Count_{days}D"),
            *(
                [col(f"{feature_name_prefix}_Sum_{days}D")]
                if value_col_name_for_sum else []
            )
        )
        current_block_features_df = current_block_features_df.join(
            aggregated_lookback_df,
            (current_block_features_df.ClientCode_block_key == aggregated_lookback_df.ClientCode_agg_key) &
            (current_block_features_df.SnapshotDate_block_key == aggregated_lookback_df.SnapshotDate_agg_key),
            "left"
        ).drop("ClientCode_agg_key", "SnapshotDate_agg_key")
        fill_cols_this_iter = [f"{feature_name_prefix}_Days_Count_{days}D", f"{feature_name_prefix}_Txns_Count_{days}D"]
        if value_col_name_for_sum:
            fill_cols_this_iter.append(f"{feature_name_prefix}_Sum_{days}D")
        current_block_features_df = current_block_features_df.fillna(0, subset=fill_cols_this_iter)
    current_block_features_df = current_block_features_df \
        .withColumnRenamed("ClientCode_block_key", "ClientCode") \
        .withColumnRenamed("SnapshotDate_block_key", "SnapshotDate")
    return current_block_features_df

# --- Calculate each feature block ---
# (master data DFs should still be persisted from Cell 3)
trade_features_block_df = calculate_feature_block(base_keys_df, trades_master_df, "ClientCode", "ActivityDate", "GrossBrokerage", "t
# Optional: Persist individual blocks if they are very large and reused, but for one-time join, maybe not needed.

login_features_block_df = calculate_feature_block(base_keys_df, logins_master_df, "ClientCode", "ActivityDate", None, "l", "Login",

deposit_features_block_df = calculate_feature_block(base_keys_df, deposits_master_df, "ClientCode", "ActivityDate", "Amount", "d", '

payout_features_block_df = calculate_feature_block(base_keys_df, payouts_master_df, "ClientCode", "ActivityDate", "Amount", "p", "Pa

feature_blocks_to_join_list = [trade_features_block_df, login_features_block_df,
                               deposit_features_block_df, payout_features_block_df]
block_names = ["Trades", "Logins", "Deposits", "Payouts"]

# --- Join all feature blocks to the ABT read from disk ---
print("\nJoining all feature blocks to the ABT...")
current_abt_for_fm_df = abt_df # This is the ABT with recency features

for i, block_df in enumerate(feature_blocks_to_join_list):
    block_name = block_names[i]
    print(f"Joining block: {block_name}...")
    current_abt_for_fm_df = current_abt_for_fm_df.join(block_df, ["ClientCode", "SnapshotDate"], "left")

    # FillNA for columns introduced by this block, in case of any mismatches (though fillna(0) inside block calc should handle it)
    feature_cols_in_block = [c for c in block_df.columns if c not in ["ClientCode", "SnapshotDate"]]
    current_abt_for_fm_df = current_abt_for_fm_df.fillna(0, subset=feature_cols_in_block)

abt_df = current_abt_for_fm_df

if base_keys_df.is_cached: base_keys_df.unpersist()

# Persist before writing and count
abt_df.persist()
fm_abt_count = abt_df.count()
print("Frequency and Monetary features calculated and joined. Sample:")
# ... (show sample code) ...
sample_cols_fm = ["ClientCode", "SnapshotDate", "Trade_Days_Count_30D", "Trade_Sum_30D", "Login_Days_Count_30D", "Deposit_Sum_90D",
existing_sample_cols = [c for c in sample_cols_fm if c in abt_df.columns]
if existing_sample_cols: abt_df.select(existing_sample_cols).orderBy("ClientCode", "SnapshotDate").show(5, truncate=False)
else: abt_df.show(5, truncate=False)
print(f"ABT DF Count after Frequency/Monetary: {fm_abt_count}")

# --- Write Updated ABT to Disk ---
if fm_abt_count > 0:
```

```
        print(f"Writing ABT with Freq/Monetary features to: {temp_abt_path}")
        abt_df.write.mode("overwrite").parquet(temp_abt_path)
        print("ABT with Freq/Monetary features written successfully.")
    else:
        print("ABT with Freq/Monetary features is empty. Not writing.")

    if abt_df.is_cached:
        abt_df.unpersist()

except FileNotFoundError:
    print(f"ERROR: Could not read temporary ABT from {temp_abt_path} for Freq/Monetary. Ensure previous step ran and wrote the file.")
except Exception as e:
    print(f"Error in Frequency/Monetary Feature Engineering or writing: {e}")
    raise e
```

```
--- Calculating Frequency and Monetary Features ---
Read ABT from /content/drive/MyDrive/Tables/output_abt_final_pred/temp_abt_in_progress.parquet with 33681785 rows for Freq/Monetary
Persisted base_keys_df with 33681785 distinct client-snapshot pairs.
  Calculating feature block for Trade...
    Calculating for 30-day lookback for Trade...
    Calculating for 90-day lookback for Trade...
    Calculating for 180-day lookback for Trade...
    Calculating for 270-day lookback for Trade...
    Calculating for 365-day lookback for Trade...
  Calculating feature block for Login...
    Calculating for 30-day lookback for Login...
    Calculating for 90-day lookback for Login...
    Calculating for 180-day lookback for Login...
    Calculating for 270-day lookback for Login...
    Calculating for 365-day lookback for Login...
  Calculating feature block for Deposit...
    Calculating for 30-day lookback for Deposit...
    Calculating for 90-day lookback for Deposit...
    Calculating for 180-day lookback for Deposit...
    Calculating for 270-day lookback for Deposit...
    Calculating for 365-day lookback for Deposit...
  Calculating feature block for Payout...
    Calculating for 30-day lookback for Payout...
    Calculating for 90-day lookback for Payout...
    Calculating for 180-day lookback for Payout...
    Calculating for 270-day lookback for Payout...
    Calculating for 365-day lookback for Payout...

Joining all feature blocks to the ABT...
Joining block: Trades...
Joining block: Logins...
Joining block: Deposits...
Joining block: Payouts...
Frequency and Monetary features calculated and joined. Sample:
+----------+------------+--------------------+-------------+--------------------+---------------+--------------+
|ClientCode|SnapshotDate|Trade_Days_Count_30D|Trade_Sum_30D|Login_Days_Count_30D|Deposit_Sum_90D|Payout_Sum_90D|
+----------+------------+--------------------+-------------+--------------------+---------------+--------------+
|A*        |2021-01-31  |0                   |0.0          |0                   |0.0            |0.0           |
|A*        |2021-02-28  |0                   |0.0          |0                   |0.0            |0.0           |
|A*        |2021-03-31  |0                   |0.0          |0                   |0.0            |0.0           |
|A*        |2021-04-30  |0                   |0.0          |0                   |0.0            |0.0           |
|A*        |2021-05-31  |0                   |0.0          |0                   |0.0            |0.0           |
+----------+------------+--------------------+-------------+--------------------+---------------+--------------+
only showing top 5 rows

ABT DF Count after Frequency/Monetary: 33681785
Writing ABT with Freq/Monetary features to: /content/drive/MyDrive/Tables/output_abt_final_pred/temp_abt_in_progress.parquet
ABT with Freq/Monetary features written successfully.
```

```
# --- 8. Funding Flow Features & AUM Features ---
print("\n--- Calculating Funding Flow and AUM Features ---")
try:
    # Read the current ABT from disk
    abt_df = spark.read.parquet(temp_abt_path)
    print(f"Read ABT from {temp_abt_path} with {abt_df.count()} rows for Funding/AUM features.")

    # --- Funding Flow Features ---
    print("  Calculating Net Funding Flow and Ratios...")
    for days in LOOKBACK_PERIODS_DAYS:
        deposit_sum_col = f"Deposit_Sum_{days}D"
        payout_sum_col = f"Payout_Sum_{days}D"
        net_flow_col = f"Net_Funding_Flow_{days}D"
        payout_to_deposit_ratio_col = f"Payout_To_Deposit_Ratio_{days}D"

        if deposit_sum_col in abt_df.columns and payout_sum_col in abt_df.columns:
            abt_df = abt_df.withColumn(net_flow_col, col(deposit_sum_col) - col(payout_sum_col))
            abt_df = abt_df.withColumn(
                payout_to_deposit_ratio_col,
```

```
                when(col(deposit_sum_col) > 0, col(payout_sum_col) / col(deposit_sum_col))
                .otherwise(when(col(payout_sum_col) > 0, 99999.0).otherwise(0.0)))
        else:
            print(f"    Skipping funding flow for {days}D as source sum columns are missing.")

    # --- AUM Features ---
    print("\n  Calculating AUM Features (AUM of Snapshot Month)...")
    abt_df_with_ym = abt_df.withColumn("SnapshotYearMonth", expr("date_format(SnapshotDate, 'yyyy-MM')"))
    aum_master_df_with_ym = aum_master_df.withColumn("AUMYearMonth", expr("date_format(AUMMonthStartDate, 'yyyy-MM')"))

    aum_to_join_df = aum_master_df_with_ym.select(
        col("ClientCode").alias("AUM_ClientCode"), "AUMYearMonth",
        col("MonthlyAUM").alias("AUM_SnapshotMonth_Monthly"),
        col("RunningTotalAUM").alias("AUM_SnapshotMonth_RunningTotal")
    )

    abt_df = abt_df_with_ym.join(
        aum_to_join_df,
        (abt_df_with_ym.ClientCode == aum_to_join_df.AUM_ClientCode) & \
        (abt_df_with_ym.SnapshotYearMonth == aum_to_join_df.AUMYearMonth),
        "left"
    ).drop("SnapshotYearMonth", "AUM_ClientCode", "AUMYearMonth")

    aum_feature_cols = ["AUM_SnapshotMonth_Monthly", "AUM_SnapshotMonth_RunningTotal"]
    for c_aum in aum_feature_cols:
        if c_aum not in abt_df.columns: abt_df = abt_df.withColumn(c_aum, lit(0.0))
    abt_df = abt_df.fillna(0.0, subset=aum_feature_cols)

    # Persist before writing and count
    abt_df.persist()
    ff_aum_abt_count = abt_df.count()
    print("Funding Flow and AUM features calculated. Sample:")
    # ... (show sample code) ...
    sample_cols_aum_flow = ["ClientCode", "SnapshotDate", "Net_Funding_Flow_30D", "Payout_To_Deposit_Ratio_30D", "AUM_SnapshotMonth_Mont
    existing_sample_cols_af = [c for c in sample_cols_aum_flow if c in abt_df.columns]
    if existing_sample_cols_af: abt_df.select(existing_sample_cols_af).orderBy("ClientCode", "SnapshotDate").show(5, truncate=False)
    else: abt_df.select("ClientCode", "SnapshotDate").show(5,truncate=False)
    print(f"ABT DF Count after Funding Flow/AUM: {ff_aum_abt_count}")

    # --- Write Updated ABT to Disk ---
    if ff_aum_abt_count > 0:
        print(f"Writing ABT with Funding/AUM features to: {temp_abt_path}")
        abt_df.write.mode("overwrite").parquet(temp_abt_path)
        print("ABT with Funding/AUM features written successfully.")
    else:
        print("ABT with Funding/AUM features is empty. Not writing.")

    if abt_df.is_cached:
        abt_df.unpersist()

except FileNotFoundError:
    print(f"ERROR: Could not read temporary ABT from {temp_abt_path} for Funding/AUM. Ensure previous step ran and wrote the file.")
except Exception as e:
    print(f"Error in Funding Flow/AUM Feature Engineering or writing: {e}")
    raise e
```

```
--- Calculating Funding Flow and AUM Features ---
Read ABT from /content/drive/MyDrive/Tables/output_abt_final_pred/temp_abt_in_progress.parquet with 33681785 rows for Funding/AUM fe
  Calculating Net Funding Flow and Ratios...

  Calculating AUM Features (AUM of Snapshot Month)...
Funding Flow and AUM features calculated. Sample:
+----------+------------+--------------------+--------------------------+-------------------------+
|ClientCode|SnapshotDate|Net_Funding_Flow_30D|Payout_To_Deposit_Ratio_30D|AUM_SnapshotMonth_Monthly|
+----------+------------+--------------------+--------------------------+-------------------------+
|A*        |2021-01-31  |0.0                 |0.0                       |0.0                      |
|A*        |2021-02-28  |0.0                 |0.0                       |0.0                      |
|A*        |2021-03-31  |0.0                 |0.0                       |0.0                      |
|A*        |2021-04-30  |0.0                 |0.0                       |0.0                      |
|A*        |2021-05-31  |0.0                 |0.0                       |0.0                      |
+----------+------------+--------------------+--------------------------+-------------------------+
only showing top 5 rows

ABT DF Count after Funding Flow/AUM: 33681785
Writing ABT with Funding/AUM features to: /content/drive/MyDrive/Tables/output_abt_final_pred/temp_abt_in_progress.parquet
ABT with Funding/AUM features written successfully.
```

```
# --- 9. Payout Risk Features ---
print("\n--- Calculating Payout Risk Features ---")
try:
    # Read the current ABT from disk
```

```
        abt_df = spark.read.parquet(temp_abt_path)
        print(f"Read ABT from {temp_abt_path} with {abt_df.count()} rows for Payout Risk features.")

        # 1. Calculate Total Payouts in Snapshot Month
        abt_df_with_dates = abt_df.withColumn("StartOfMonth", trunc(col("SnapshotDate"), "MM"))

        payouts_in_month_df = abt_df_with_dates.alias("s").join( # Use abt_df_with_dates here
            payouts_master_df.alias("p"), # Ensure payouts_master_df is persisted from Cell 3
            (col("s.ClientCode") == col("p.ClientCode")) & \
            (col("p.ActivityDate") >= col("s.StartOfMonth")) & \
            (col("p.ActivityDate") <= col("s.SnapshotDate")),
            "left"
        ).groupBy(col("s.ClientCode"), col("s.SnapshotDate")) \
         .agg(coalesce(pyspark_sum(col("p.Amount")), lit(0.0)).alias("Total_Payout_In_Snapshot_Month"))

        abt_df = abt_df.join(payouts_in_month_df, ["ClientCode", "SnapshotDate"], "left")
        if "Total_Payout_In_Snapshot_Month" not in abt_df.columns:
            abt_df = abt_df.withColumn("Total_Payout_In_Snapshot_Month", lit(0.0))
        else:
            abt_df = abt_df.fillna(0.0, subset=["Total_Payout_In_Snapshot_Month"])

        # 2. Get EOM Cash Balance from Previous Month
        abt_df = abt_df.withColumn("PreviousMonthEOM", last_day(add_months(col("SnapshotDate"), -1)))
        cb_df = cash_balance_master_df.alias("cb") # Ensure cash_balance_master_df is persisted from Cell 3

        abt_df = abt_df.join(
            cb_df.select(
                col("cb.ClientCode").alias("CB_ClientCode"),
                col("cb.BalanceDateEOM").alias("CB_BalanceDateEOM"),
                col("cb.CashBalance").alias("CashBalance_EOM_PreviousMonth")
            ),
            (abt_df.ClientCode == col("CB_ClientCode")) & (abt_df.PreviousMonthEOM == col("CB_BalanceDateEOM")),
            "left"
        ).drop("CB_ClientCode", "CB_BalanceDateEOM") # Keep PreviousMonthEOM for inspection if needed

        # 3. Calculate Payout_As_Pct_Of_CashBalance
        abt_df = abt_df.withColumn(
            "Payout_As_Pct_Of_CashBalance",
            when((col("CashBalance_EOM_PreviousMonth").isNotNull()) & (col("CashBalance_EOM_PreviousMonth") != 0),
                 (col("Total_Payout_In_Snapshot_Month") / col("CashBalance_EOM_PreviousMonth")) * 100)
            .when((col("CashBalance_EOM_PreviousMonth").isNotNull()) & (col("CashBalance_EOM_PreviousMonth") == 0) & (col("Total_Payout_In_S
            .otherwise(None)
        )
        if "Payout_As_Pct_Of_CashBalance" not in abt_df.columns:
             abt_df = abt_df.withColumn("Payout_As_Pct_Of_CashBalance", lit(None).cast(DoubleType()))

        # 4. Create Payout_Risk_Flag
        abt_df = abt_df.withColumn(
            "Payout_Risk_Flag",
            when(col("Payout_As_Pct_Of_CashBalance") > 70, "CHURNRISK").otherwise(None)
        )
        if "Payout_Risk_Flag" not in abt_df.columns:
             abt_df = abt_df.withColumn("Payout_Risk_Flag", lit(None).cast(StringType()))

        # Persist before writing and count
        abt_df.persist()
        pr_abt_count = abt_df.count()
        print("Payout Risk features calculated. Sample:")
        # ... (show sample code) ...
        payout_risk_cols_show = ["ClientCode", "SnapshotDate", "Total_Payout_In_Snapshot_Month", "CashBalance_EOM_PreviousMonth", "Payout_As
        existing_pr_cols = [c for c in payout_risk_cols_show if c in abt_df.columns]
        if existing_pr_cols: abt_df.select(existing_pr_cols).orderBy("ClientCode", "SnapshotDate").show(5, truncate=False)
        else: abt_df.select("ClientCode", "SnapshotDate").show(5,truncate=False)
        print(f"ABT DF Count after Payout Risk: {pr_abt_count}")

        # --- Write Updated ABT to Disk ---
        if pr_abt_count > 0:
            print(f"Writing ABT with Payout Risk features to: {temp_abt_path}")
            abt_df.write.mode("overwrite").parquet(temp_abt_path)
            print("ABT with Payout Risk features written successfully.")
        else:
            print("ABT with Payout Risk features is empty. Not writing.")

        if abt_df.is_cached:
            abt_df.unpersist()

    except FileNotFoundError:
        print(f"ERROR: Could not read temporary ABT from {temp_abt_path} for Payout Risk. Ensure previous step ran and wrote the file.")
    except Exception as e:
        print(f"Error in Payout Risk Feature Engineering or writing: {e}")
        raise e
```

```
--- Calculating Payout Risk Features ---
Read ABT from /content/drive/MyDrive/Tables/output_abt_final_pred/temp_abt_in_progress.parquet with 33681785 rows for Payout Risk fe
Payout Risk features calculated. Sample:
+----------+------------+-----------------------------+-----------------------------+-----------------------------+-----------------+
|ClientCode|SnapshotDate|Total_Payout_In_Snapshot_Month|CashBalance_EOM_PreviousMonth|Payout_As_Pct_Of_CashBalance|Payout_Risk_Flag |
+----------+------------+-----------------------------+-----------------------------+-----------------------------+-----------------+
|A*        |2021-01-31  |0.0                          |NULL                         |NULL                         |NULL             |
|A*        |2021-02-28  |0.0                          |NULL                         |NULL                         |NULL             |
|A*        |2021-03-31  |0.0                          |NULL                         |NULL                         |NULL             |
|A*        |2021-04-30  |0.0                          |NULL                         |NULL                         |NULL             |
|A*        |2021-05-31  |0.0                          |NULL                         |NULL                         |NULL             |
+----------+------------+-----------------------------+-----------------------------+-----------------------------+-----------------+
only showing top 5 rows

ABT DF Count after Payout Risk: 33681785
Writing ABT with Payout Risk features to: /content/drive/MyDrive/Tables/output_abt_final_pred/temp_abt_in_progress.parquet
ABT with Payout Risk features written successfully.
```

```
# --- 10. Delta Features ---
print("\n--- Calculating Delta Features ---")
try:
    # Read the current ABT from disk
    abt_df = spark.read.parquet(temp_abt_path)
    print(f"Read ABT from {temp_abt_path} with {abt_df.count()} rows for Delta features.")

    # Define features for which to calculate deltas and their lookback period
    # Format: (current_feature_col_name, lookback_days_for_this_feature, delta_feature_name_suffix)
    delta_configs = [
        ("Trade_Days_Count_90D", 90, "Trade_Days_90D_Delta"),
        ("Login_Days_Count_90D", 90, "Login_Days_90D_Delta"),
        ("Trade_Sum_90D", 90, "Brokerage_Sum_90D_Delta") # Assuming Trade_Sum_90D is brokerage
    ]

    # Window spec to get the previous snapshot's value for a client
    # Snapshots are monthly, so lag(1) gets the previous month's snapshot.
    # We need to ensure snapshots are ordered correctly for the lag.
    # client_snapshot_base_df was ordered by ClientCode, SnapshotDate for show,
    # but abt_df read from Parquet has no inherent order for window functions unless specified.

    window_prev_snapshot = Window.partitionBy("ClientCode").orderBy("SnapshotDate")

    for base_col, lookback_days, delta_col_name in delta_configs:
        if base_col in abt_df.columns:
            print(f"  Calculating delta for {base_col}...")

            # Get the value of the base_col from the PREVIOUS snapshot
            # Lagging by 1 assumes consistent monthly snapshots.
            # If snapshots are not perfectly monthly or have gaps, this lag needs careful thought.
            # For this ABT, we generated monthly snapshots.
            abt_df = abt_df.withColumn(f"Prev_Snapshot_{base_col}", lag(col(base_col), 1).over(window_prev_snapshot))

            # Calculate Delta: Current Value - Previous Value
            # Handle nulls: if previous is null (e.g., first snapshot for client), delta might be null or current value.
            # If current is null (should be 0 due to fillna), delta is -Previous value.
            abt_df = abt_df.withColumn(
                delta_col_name,
                when(col(f"Prev_Snapshot_{base_col}").isNotNull(),
                     coalesce(col(base_col), lit(0.0)) - col(f"Prev_Snapshot_{base_col}"))
                .otherwise(None) # Or coalesce(col(base_col), lit(0.0)) if we want delta to be current value for first period
            )
            # Drop the temporary previous snapshot column
            abt_df = abt_df.drop(f"Prev_Snapshot_{base_col}")

            # Fill NA for delta if desired (e.g., with 0 if no previous period to compare against)
            abt_df = abt_df.fillna(0.0, subset=[delta_col_name])
        else:
            print(f"    Skipping delta for {base_col} as it's not in ABT columns.")

    # Persist before writing and count
    abt_df.persist()
    delta_abt_count = abt_df.count()
    print("Delta features calculated. Sample:")
    delta_cols_to_show = ["ClientCode", "SnapshotDate"] + [dc[2] for dc in delta_configs if dc[2] in abt_df.columns]
    if len(delta_cols_to_show) > 2: # if any delta columns were actually created
        abt_df.select(delta_cols_to_show).orderBy("ClientCode", "SnapshotDate").show(5, truncate=False)
    else:
        abt_df.select("ClientCode", "SnapshotDate").show(5,truncate=False)
    print(f"ABT DF Count after Delta Features: {delta_abt_count}")

    # --- Write Updated ABT to Disk ---
    if delta_abt_count > 0:
```

```
        print(f"Writing ABT with Delta features to: {temp_abt_path}")
        abt_df.write.mode("overwrite").parquet(temp_abt_path)
        print("ABT with Delta features written successfully.")
    else:
        print("ABT with Delta features is empty. Not writing.")

    if abt_df.is_cached:
        abt_df.unpersist()

except FileNotFoundError:
    print(f"ERROR: Could not read temporary ABT from {temp_abt_path} for Delta features. Ensure previous step ran and wrote the file.")
except Exception as e:
    print(f"Error in Delta Feature Engineering or writing: {e}")
    raise e
```

```
--- Calculating Delta Features ---
Read ABT from /content/drive/MyDrive/Tables/output_abt_final_pred/temp_abt_in_progress.parquet with 33681785 rows for Delta features
  Calculating delta for Trade_Days_Count_90D...
  Calculating delta for Login_Days_Count_90D...
  Calculating delta for Trade_Sum_90D...
Delta features calculated. Sample:
+----------+------------+--------------------+--------------------+-----------------------+
|ClientCode|SnapshotDate|Trade_Days_90D_Delta|Login_Days_90D_Delta|Brokerage_Sum_90D_Delta|
+----------+------------+--------------------+--------------------+-----------------------+
|A*        |2021-01-31  |0.0                 |0.0                 |0.0                    |
|A*        |2021-02-28  |0.0                 |0.0                 |0.0                    |
|A*        |2021-03-31  |0.0                 |0.0                 |0.0                    |
|A*        |2021-04-30  |0.0                 |0.0                 |0.0                    |
|A*        |2021-05-31  |0.0                 |0.0                 |0.0                    |
+----------+------------+--------------------+--------------------+-----------------------+
only showing top 5 rows

ABT DF Count after Delta Features: 33681785
Writing ABT with Delta features to: /content/drive/MyDrive/Tables/output_abt_final_pred/temp_abt_in_progress.parquet
ABT with Delta features written successfully.
```

```
print("--- Verifying columns in temp_abt_path before Churn Label Generation ---")
temp_abt_check_df = spark.read.parquet(temp_abt_path)
temp_abt_check_df.printSchema()
print(f"Columns found: {temp_abt_check_df.columns}")
# Check for specific 90D columns
for col_name_to_check in ["Trade_Days_Count_90D", "Login_Days_Count_90D"]:
    if col_name_to_check in temp_abt_check_df.columns:
        print(f"Column '{col_name_to_check}' IS PRESENT.")
    else:
        print(f"COLUMN '{col_name_to_check}' IS MISSING!")
del temp_abt_check_df # Clean up
```

```
--- Verifying columns in temp_abt_path before Churn Label Generation ---
root
 |-- ClientCode: string (nullable = true)
 |-- SnapshotDate: date (nullable = true)
 |-- ActivationDate: date (nullable = true)
 |-- Tenure_Days: integer (nullable = true)
 |-- Last_Trade_Date: date (nullable = true)
 |-- Days_Since_Last_Trade: integer (nullable = true)
 |-- Last_Login_Date: date (nullable = true)
 |-- Days_Since_Last_Login: integer (nullable = true)
 |-- Last_Deposit_Date: date (nullable = true)
 |-- Days_Since_Last_Deposit: integer (nullable = true)
 |-- Last_Payout_Date: date (nullable = true)
 |-- Days_Since_Last_Payout: integer (nullable = true)
 |-- Trade_Days_Count_30D: long (nullable = true)
 |-- Trade_Txns_Count_30D: long (nullable = true)
 |-- Trade_Sum_30D: double (nullable = true)
 |-- Trade_Days_Count_90D: long (nullable = true)
 |-- Trade_Txns_Count_90D: long (nullable = true)
 |-- Trade_Sum_90D: double (nullable = true)
 |-- Trade_Days_Count_180D: long (nullable = true)
 |-- Trade_Txns_Count_180D: long (nullable = true)
 |-- Trade_Sum_180D: double (nullable = true)
 |-- Trade_Days_Count_270D: long (nullable = true)
 |-- Trade_Txns_Count_270D: long (nullable = true)
 |-- Trade_Sum_270D: double (nullable = true)
 |-- Trade_Days_Count_365D: long (nullable = true)
 |-- Trade_Txns_Count_365D: long (nullable = true)
 |-- Trade_Sum_365D: double (nullable = true)
 |-- Login_Days_Count_30D: long (nullable = true)
 |-- Login_Txns_Count_30D: long (nullable = true)
 |-- Login_Days_Count_90D: long (nullable = true)
 |-- Login_Txns_Count_90D: long (nullable = true)
 |-- Login_Days_Count_180D: long (nullable = true)
 |-- Login_Txns_Count_180D: long (nullable = true)
 |-- Login_Days_Count_270D: long (nullable = true)
 |-- Login_Txns_Count_270D: long (nullable = true)
```

```
 |-- Login_Days_Count_365D: long (nullable = true)
 |-- Login_Txns_Count_365D: long (nullable = true)
 |-- Deposit_Days_Count_30D: long (nullable = true)
 |-- Deposit_Txns_Count_30D: long (nullable = true)
 |-- Deposit_Sum_30D: double (nullable = true)
 |-- Deposit_Days_Count_90D: long (nullable = true)
 |-- Deposit_Txns_Count_90D: long (nullable = true)
 |-- Deposit_Sum_90D: double (nullable = true)
 |-- Deposit_Days_Count_180D: long (nullable = true)
 |-- Deposit_Txns_Count_180D: long (nullable = true)
 |-- Deposit_Sum_180D: double (nullable = true)
 |-- Deposit_Days_Count_270D: long (nullable = true)
 |-- Deposit_Txns_Count_270D: long (nullable = true)
 |-- Deposit_Sum_270D: double (nullable = true)
 |-- Deposit_Days_Count_365D: long (nullable = true)
 |-- Deposit_Txns_Count_365D: long (nullable = true)
 |-- Deposit_Sum_365D: double (nullable = true)
 |-- Payout_Days_Count_30D: long (nullable = true)
 |-- Payout_Txns_Count_30D: long (nullable = true)
 |-- Payout_Sum_30D: double (nullable = true)
```

```
# --- 11. Churn Label Generation (Iterative Write) ---
print("\n--- Generating Churn Labels (Iterative Write Strategy) ---")
try:
    # Initial read of ABT (from Delta features step)
    current_abt_file_path = temp_abt_path

    # Master data should still be persisted from Cell 3
    trades_for_churn_label = trades_master_df.select(col("ClientCode").alias("t_ClientCode"), col("ActivityDate").alias("t_ActivityDate"
    logins_for_churn_label = logins_master_df.select(col("ClientCode").alias("l_ClientCode"), col("ActivityDate").alias("l_ActivityDate"

    # CHURN_WINDOWS_DAYS is [60, 90, 270, 365] from Cell 1

    for n_days_churn_window in CHURN_WINDOWS_DAYS:
        print(f"\n  Generating churn label for {n_days_churn_window}-day window...")

        # Read the latest version of ABT from disk for this iteration
        abt_df_iter = spark.read.parquet(current_abt_file_path)
        print(f"    Read ABT from {current_abt_file_path} with {abt_df_iter.count()} rows.")

        # --- Condition A: Recent Engagement (Lookback) ---
        lookback_period_for_cond_A = n_days_churn_window
        if n_days_churn_window == 60:
            lookback_period_for_cond_A = 30
            print(f"      For {n_days_churn_window}D churn, using {lookback_period_for_cond_A}D lookback features for Condition A.")

        lookback_trade_col_A = f"Trade_Days_Count_{lookback_period_for_cond_A}D"
        lookback_login_col_A = f"Login_Days_Count_{lookback_period_for_cond_A}D"
        churn_label_col_name = f"Is_Churned_Engage_{n_days_churn_window}Days"

        if lookback_trade_col_A not in abt_df_iter.columns or lookback_login_col_A not in abt_df_iter.columns:
            print(f"      ERROR: Lookback columns '{lookback_trade_col_A}' or '{lookback_login_col_A}' are missing. Skipping label for {
            # Add the churn label column as 0 if it can't be computed, then write and continue
            abt_df_iter = abt_df_iter.withColumn(churn_label_col_name, lit(0).cast(IntegerType()))
            print(f"      Writing ABT (with skipped label for {n_days_churn_window}D) back to: {current_abt_file_path}")
            abt_df_iter.write.mode("overwrite").parquet(current_abt_file_path)
            print(f"      ABT with skipped label for {n_days_churn_window}D written successfully.")
            continue

        # Create temporary DF for this iteration's calculations to avoid modifying abt_df_iter directly until the end
        iter_calc_df = abt_df_iter.withColumn(
            "Temp_Condition_A_Flag",
            (col(lookback_trade_col_A) > 0) | (col(lookback_login_col_A) > 0)
        )

        # --- Condition B: Subsequent Inactivity (Look Forward) ---
        print(f"      Calculating Condition B for {n_days_churn_window}D...")
        cond_B_trades = iter_calc_df.alias("abt_c").join(
            trades_for_churn_label.alias("t"),
            (col("abt_c.ClientCode") == col("t.t_ClientCode")) & \
            (col("t.t_ActivityDate") > col("abt_c.SnapshotDate")) & \
            (col("t.t_ActivityDate") <= date_add(col("abt_c.SnapshotDate"), n_days_churn_window)),
            "left"
        ).groupBy("abt_c.ClientCode", "abt_c.SnapshotDate") \
         .agg(countDistinct(col("t.t_ActivityDate")).alias("Temp_Future_Trade_Days"))

        cond_B_logins = iter_calc_df.alias("abt_c").join(
            logins_for_churn_label.alias("l"),
            (col("abt_c.ClientCode") == col("l.l_ClientCode")) & \
            (col("l.l_ActivityDate") > col("abt_c.SnapshotDate")) & \
            (col("l.l_ActivityDate") <= date_add(col("abt_c.SnapshotDate"), n_days_churn_window)),
            "left"
        ).groupBy("abt_c.ClientCode", "abt_c.SnapshotDate") \
```

```
            .agg(countDistinct(col("l.l_ActivityDate")).alias("Temp_Future_Login_Days"))

        iter_calc_df = iter_calc_df.join(cond_B_trades, ["ClientCode", "SnapshotDate"], "left") \
                                   .join(cond_B_logins, ["ClientCode", "SnapshotDate"], "left")

        iter_calc_df = iter_calc_df.fillna(0, subset=["Temp_Future_Trade_Days", "Temp_Future_Login_Days"])

        iter_calc_df = iter_calc_df.withColumn(
            "Temp_Condition_B_Flag",
            (col("Temp_Future_Trade_Days") == 0) & (col("Temp_Future_Login_Days") == 0)
        )

        # Generate Churn Label and add it to abt_df_iter (which was the read version)
        abt_df_iter = iter_calc_df.withColumn(
            churn_label_col_name,
            when(col("Temp_Condition_A_Flag") & col("Temp_Condition_B_Flag"), 1).otherwise(0)
        ).drop("Temp_Condition_A_Flag", "Temp_Condition_B_Flag", "Temp_Future_Trade_Days", "Temp_Future_Login_Days")

        # Persist before writing
        abt_df_iter.persist()
        current_iter_count = abt_df_iter.count() # Action
        print(f"      Label for {n_days_churn_window}D generated. ABT row count: {current_iter_count}")

        if current_iter_count > 0:
            print(f"      Writing ABT with {churn_label_col_name} to: {current_abt_file_path}")
            abt_df_iter.write.mode("overwrite").parquet(current_abt_file_path)
            print(f"      ABT with {churn_label_col_name} written successfully.")
        else:
            print(f"      ABT for {n_days_churn_window}D label is empty. Not writing.")

        if abt_df_iter.is_cached:
            abt_df_iter.unpersist()
        print(f"  Finished processing for {n_days_churn_window}-day window.")


    # Final verification after all labels
    print("\nAll Churn labels generated. Verifying final ABT from temp path...")
    final_abt_with_labels = spark.read.parquet(current_abt_file_path)
    final_abt_count = final_abt_with_labels.count()
    print(f"Final ABT from {current_abt_file_path} has {final_abt_count} rows.")
    label_cols_to_show = ["ClientCode", "SnapshotDate"] + [f"Is_Churned_Engage_{n}Days" for n in CHURN_WINDOWS_DAYS if f"Is_Churned_Enga
    if len(label_cols_to_show) > 2:
        final_abt_with_labels.select(label_cols_to_show).orderBy("ClientCode", "SnapshotDate").show(5, truncate=False)
    else:
        final_abt_with_labels.select("ClientCode", "SnapshotDate").show(5,truncate=False)

except FileNotFoundError:
    print(f"ERROR: Could not read temporary ABT from {current_abt_file_path} for Churn Labels. Ensure previous step ran and wrote the f
except Exception as e:
    print(f"Error in Churn Label Generation or writing: {e}")
    raise e # Re-raise to see the full error
```

```
 --- Generating Churn Labels (Iterative Write Strategy) ---

  Generating churn label for 60-day window...
    Read ABT from /content/drive/MyDrive/Tables/output_abt_final_pred/temp_abt_in_progress.parquet with 33681785 rows.
      For 60D churn, using 30D lookback features for Condition A.
      Calculating Condition B for 60D...
      Label for 60D generated. ABT row count: 33681785
      Writing ABT with Is_Churned_Engage_60Days to: /content/drive/MyDrive/Tables/output_abt_final_pred/temp_abt_in_progress.parquet
      ABT with Is_Churned_Engage_60Days written successfully.
  Finished processing for 60-day window.

  Generating churn label for 90-day window...
    Read ABT from /content/drive/MyDrive/Tables/output_abt_final_pred/temp_abt_in_progress.parquet with 33681785 rows.
      Calculating Condition B for 90D...
      Label for 90D generated. ABT row count: 33681785
      Writing ABT with Is_Churned_Engage_90Days to: /content/drive/MyDrive/Tables/output_abt_final_pred/temp_abt_in_progress.parquet
      ABT with Is_Churned_Engage_90Days written successfully.
  Finished processing for 90-day window.

  Generating churn label for 270-day window...
    Read ABT from /content/drive/MyDrive/Tables/output_abt_final_pred/temp_abt_in_progress.parquet with 33681785 rows.
      Calculating Condition B for 270D...
      Label for 270D generated. ABT row count: 33681785
      Writing ABT with Is_Churned_Engage_270Days to: /content/drive/MyDrive/Tables/output_abt_final_pred/temp_abt_in_progress.parque
      ABT with Is_Churned_Engage_270Days written successfully.
  Finished processing for 270-day window.

  Generating churn label for 365-day window...
    Read ABT from /content/drive/MyDrive/Tables/output_abt_final_pred/temp_abt_in_progress.parquet with 33681785 rows.
      Calculating Condition B for 365D...
      Label for 365D generated. ABT row count: 33681785
      Writing ABT with Is_Churned_Engage_365Days to: /content/drive/MyDrive/Tables/output_abt_final_pred/temp_abt_in_progress.parque
```

```
      ABT with Is_Churned_Engage_365Days written successfully.
   Finished processing for 365-day window.

All Churn labels generated. Verifying final ABT from temp path...
Final ABT from /content/drive/MyDrive/Tables/output_abt_final_pred/temp_abt_in_progress.parquet has 33681785 rows.
+----------+------------+------------------------+------------------------+-------------------------+-------------------------+
|ClientCode|SnapshotDate|Is_Churned_Engage_60Days|Is_Churned_Engage_90Days|Is_Churned_Engage_270Days|Is_Churned_Engage_365Days|
+----------+------------+------------------------+------------------------+-------------------------+-------------------------+
|A*        |2021-01-31  |0                       |0                       |0                        |0                        |
|A*        |2021-02-28  |0                       |0                       |0                        |0                        |
|A*        |2021-03-31  |0                       |0                       |0                        |0                        |
|A*        |2021-04-30  |0                       |0                       |0                        |0                        |
|A*        |2021-05-31  |0                       |0                       |0                        |0                        |
+----------+------------+------------------------+------------------------+-------------------------+-------------------------+
only showing top 5 rows
```

```
# --- 12. Final Filtering, fillna, and Final ABT Save ---
print("\n--- Finalizing ABT ---")
try:
    # Read the current ABT from disk (contains all features and labels)
    final_abt_df = spark.read.parquet(temp_abt_path)
    print(f"Read ABT from {temp_abt_path} with {final_abt_df.count()} rows for final processing.")

    # 1. Final Filtering (Example: Minimum Tenure)
    # You might want to filter out snapshots where tenure is too short for meaningful prediction
    min_tenure_for_abt = 90 # days, example
    initial_count = final_abt_df.count()
    final_abt_df = final_abt_df.filter(col("Tenure_Days") >= min_tenure_for_abt)
    filtered_count = final_abt_df.count()
    print(f"Filtered ABT from {initial_count} to {filtered_count} rows based on Tenure_Days >= {min_tenure_for_abt}")

    # 2. Final fillna for all feature columns
    # Identify feature columns (exclude ClientCode, SnapshotDate, ActivationDate, and target labels)
    key_cols = ["ClientCode", "SnapshotDate", "ActivationDate"]
    target_cols = [f"Is_Churned_Engage_{n}Days" for n in CHURN_WINDOWS_DAYS]

    # Columns that are dates and should not be filled with 0/ -1
    date_feature_cols = [c for c in final_abt_df.columns if "Date" in c and c not in key_cols] # e.g., Last_Trade_Date

    feature_columns_to_fill = [
        c for c in final_abt_df.columns if c not in key_cols and c not in target_cols and c not in date_feature_cols
    ]

    # For many features, 0 is a sensible fill (e.g., counts, sums, deltas if no prior data).
    # For recency (Days_Since_Last_...), a large number (or a special category like -1) might be better if None means "never happened".
    # Current recency gives None if never happened. Let's fill Days_Since_Last_ with a large number if they are Null.
    # (e.g. tenure + 1, or a fixed large number like 9999, if tenure itself could be null for some reason initially)

    print(f"Feature columns identified for potential fillna: {len(feature_columns_to_fill)}")
    # For simplicity, filling numeric features with 0. More nuanced filling might be needed.
    # Recency 'Days_Since_Last_...' are often filled with a large number like 9999 if null.
    for rec_col_prefix in ["Trade", "Login", "Deposit", "Payout"]:
        dsl_col = f"Days_Since_Last_{rec_col_prefix}"
        if dsl_col in final_abt_df.columns:
            # Fill with a value larger than any likely tenure or a fixed large value
            # Using Tenure_Days + 1 if available, else 9999.
            # Max tenure could be ~3 years * 365 = 1095. So 9999 is distinct.
            final_abt_df = final_abt_df.withColumn(dsl_col,
                when(col(dsl_col).isNull(),
                     when(col("Tenure_Days").isNotNull(), col("Tenure_Days") + 1).otherwise(9999)
                ).otherwise(col(dsl_col))
            )
            print(f"Filled nulls in {dsl_col} with Tenure_Days+1 or 9999.")

    # For other numeric features (counts, sums, deltas, AUM, Payout_As_Pct_Of_CashBalance), fill with 0.0
    numeric_features_to_fill_zero = [
        c for c in feature_columns_to_fill
        if "Days_Since_Last_" not in c and # Already handled
           "Payout_Risk_Flag" not in c # This is string
    ]
    if numeric_features_to_fill_zero:
        final_abt_df = final_abt_df.fillna(0.0, subset=numeric_features_to_fill_zero)
        print(f"Filled nulls in {len(numeric_features_to_fill_zero)} other numeric feature columns with 0.0.")

    # Payout_Risk_Flag is string, fill with "NORISK" or "UNKNOWN" if null
    if "Payout_Risk_Flag" in final_abt_df.columns:
        final_abt_df = final_abt_df.fillna("UNKNOWN_RISK", subset=["Payout_Risk_Flag"])
        print("Filled nulls in Payout_Risk_Flag with UNKNOWN_RISK.")

    # Ensure all target columns exist and are integer
    for tc in target_cols:
```

```
            if tc in final_abt_df.columns:
                final_abt_df = final_abt_df.withColumn(tc, col(tc).cast(IntegerType()))
            else: # Should not happen if label generation was successful
                final_abt_df = final_abt_df.withColumn(tc, lit(0).cast(IntegerType()))


        # 3. Final Column Selection (Optional, but good for a clean ABT)
        # Reorder or select specific columns if needed. For now, keep all generated.
        print("Final ABT Schema:")
        final_abt_df.printSchema()
        final_abt_df.show(5, truncate=False)

        final_abt_count = final_abt_df.count()
        print(f"Final ABT has {final_abt_count} rows and {len(final_abt_df.columns)} columns.")

        # 4. Save Final ABT to the designated output path (not the temp path)
        if final_abt_count > 0:
            print(f"Writing final ABT to: {output_path_parquet}") # Using the actual output_path_parquet
            final_abt_df.write.mode("overwrite").parquet(output_path_parquet)
            print("Final ABT successfully saved.")
        else:
            print("Final ABT is empty. Not writing.")

        # Clean up master DataFrames that were persisted
        persisted_master_dfs_to_unpersist = [
            client_master_df, trades_master_df, logins_master_df,
            deposits_master_df, payouts_master_df, aum_master_df, cash_balance_master_df
        ]
        for m_df in persisted_master_dfs_to_unpersist:
            if m_df and m_df.is_cached:
                m_df.unpersist()
        print("Unpersisted master dataframes.")

    except FileNotFoundError:
        print(f"ERROR: Could not read temporary ABT from {temp_abt_path} for Final Processing. Ensure previous step ran and wrote the file.'
    except Exception as e:
        print(f"Error in Final ABT Processing or writing: {e}")
        raise e

    # Stop Spark Session at the very end
    spark.stop()
    print("Spark session stopped.")
```

```
--- Finalizing ABT ---
Read ABT from /content/drive/MyDrive/Tables/output_abt_final_pred/temp_abt_in_progress.parquet with 33681785 rows for final proces
Filtered ABT from 33681785 to 33197513 rows based on Tenure_Days >= 90
Feature columns identified for potential fillna: 80
Filled nulls in Days_Since_Last_Trade with Tenure_Days+1 or 9999.
Filled nulls in Days_Since_Last_Login with Tenure_Days+1 or 9999.
Filled nulls in Days_Since_Last_Deposit with Tenure_Days+1 or 9999.
Filled nulls in Days_Since_Last_Payout with Tenure_Days+1 or 9999.
Filled nulls in 75 other numeric feature columns with 0.0.
Filled nulls in Payout_Risk_Flag with UNKNOWN_RISK.
Final ABT Schema:
root
 |-- ClientCode: string (nullable = true)
 |-- SnapshotDate: date (nullable = true)
 |-- ActivationDate: date (nullable = true)
 |-- Tenure_Days: integer (nullable = true)
 |-- Last_Trade_Date: date (nullable = true)
 |-- Days_Since_Last_Trade: integer (nullable = true)
 |-- Last_Login_Date: date (nullable = true)
 |-- Days_Since_Last_Login: integer (nullable = true)
 |-- Last_Deposit_Date: date (nullable = true)
 |-- Days_Since_Last_Deposit: integer (nullable = true)
 |-- Last_Payout_Date: date (nullable = true)
 |-- Days_Since_Last_Payout: integer (nullable = true)
 |-- Trade_Days_Count_30D: long (nullable = true)
 |-- Trade_Txns_Count_30D: long (nullable = true)
 |-- Trade_Sum_30D: double (nullable = false)
 |-- Trade_Days_Count_90D: long (nullable = true)
 |-- Trade_Txns_Count_90D: long (nullable = true)
 |-- Trade_Sum_90D: double (nullable = false)
 |-- Trade_Days_Count_180D: long (nullable = true)
 |-- Trade_Txns_Count_180D: long (nullable = true)
 |-- Trade_Sum_180D: double (nullable = false)
 |-- Trade_Days_Count_270D: long (nullable = true)
 |-- Trade_Txns_Count_270D: long (nullable = true)
 |-- Trade_Sum_270D: double (nullable = false)
 |-- Trade_Days_Count_365D: long (nullable = true)
 |-- Trade_Txns_Count_365D: long (nullable = true)
 |-- Trade_Sum_365D: double (nullable = false)
 |-- Login_Days_Count_30D: long (nullable = true)
 |-- Login_Txns_Count_30D: long (nullable = true)
 |-- Login_Days_Count_90D: long (nullable = true)
```

```
 |-- Login_Txns_Count_90D: long (nullable = true)
 |-- Login_Days_Count_180D: long (nullable = true)
 |-- Login_Txns_Count_180D: long (nullable = true)
 |-- Login_Days_Count_270D: long (nullable = true)
 |-- Login_Txns_Count_270D: long (nullable = true)
 |-- Login_Days_Count_365D: long (nullable = true)
 |-- Login_Txns_Count_365D: long (nullable = true)
 |-- Deposit_Days_Count_30D: long (nullable = true)
 |-- Deposit_Txns_Count_30D: long (nullable = true)
 |-- Deposit_Sum_30D: double (nullable = false)
 |-- Deposit_Days_Count_90D: long (nullable = true)
 |-- Deposit_Txns_Count_90D: long (nullable = true)
 |-- Deposit_Sum_90D: double (nullable = false)
 |-- Deposit_Days_Count_180D: long (nullable = true)
```

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("ABTCheck").getOrCreate()
abt_final_path = "/content/drive/MyDrive/Tables/output_abt_final_pred/predictive_abt_religare_churn_2021_2023.parquet"
abt = spark.read.parquet(abt_final_path)
abt.printSchema()
abt.show(10, truncate=False)
print(f"ABT Row Count: {abt.count()}")
print(f"ABT Column Count: {len(abt.columns)}")
spark.stop()
```

```
root
 |-- ClientCode: string (nullable = true)
 |-- SnapshotDate: date (nullable = true)
 |-- ActivationDate: date (nullable = true)
 |-- Tenure_Days: integer (nullable = true)
 |-- Last_Trade_Date: date (nullable = true)
 |-- Days_Since_Last_Trade: integer (nullable = true)
 |-- Last_Login_Date: date (nullable = true)
 |-- Days_Since_Last_Login: integer (nullable = true)
 |-- Last_Deposit_Date: date (nullable = true)
 |-- Days_Since_Last_Deposit: integer (nullable = true)
 |-- Last_Payout_Date: date (nullable = true)
 |-- Days_Since_Last_Payout: integer (nullable = true)
 |-- Trade_Days_Count_30D: long (nullable = true)
 |-- Trade_Txns_Count_30D: long (nullable = true)
 |-- Trade_Sum_30D: double (nullable = true)
 |-- Trade_Days_Count_90D: long (nullable = true)
 |-- Trade_Txns_Count_90D: long (nullable = true)
 |-- Trade_Sum_90D: double (nullable = true)
 |-- Trade_Days_Count_180D: long (nullable = true)
 |-- Trade_Txns_Count_180D: long (nullable = true)
 |-- Trade_Sum_180D: double (nullable = true)
 |-- Trade_Days_Count_270D: long (nullable = true)
 |-- Trade_Txns_Count_270D: long (nullable = true)
 |-- Trade_Sum_270D: double (nullable = true)
 |-- Trade_Days_Count_365D: long (nullable = true)
 |-- Trade_Txns_Count_365D: long (nullable = true)
 |-- Trade_Sum_365D: double (nullable = true)
 |-- Login_Days_Count_30D: long (nullable = true)
 |-- Login_Txns_Count_30D: long (nullable = true)
 |-- Login_Days_Count_90D: long (nullable = true)
 |-- Login_Txns_Count_90D: long (nullable = true)
 |-- Login_Days_Count_180D: long (nullable = true)
 |-- Login_Txns_Count_180D: long (nullable = true)
 |-- Login_Days_Count_270D: long (nullable = true)
 |-- Login_Txns_Count_270D: long (nullable = true)
 |-- Login_Days_Count_365D: long (nullable = true)
 |-- Login_Txns_Count_365D: long (nullable = true)
 |-- Deposit_Days_Count_30D: long (nullable = true)
 |-- Deposit_Txns_Count_30D: long (nullable = true)
 |-- Deposit_Sum_30D: double (nullable = true)
 |-- Deposit_Days_Count_90D: long (nullable = true)
 |-- Deposit_Txns_Count_90D: long (nullable = true)
 |-- Deposit_Sum_90D: double (nullable = true)
 |-- Deposit_Days_Count_180D: long (nullable = true)
 |-- Deposit_Txns_Count_180D: long (nullable = true)
 |-- Deposit_Sum_180D: double (nullable = true)
 |-- Deposit_Days_Count_270D: long (nullable = true)
 |-- Deposit_Txns_Count_270D: long (nullable = true)
 |-- Deposit_Sum_270D: double (nullable = true)
 |-- Deposit_Days_Count_365D: long (nullable = true)
 |-- Deposit_Txns_Count_365D: long (nullable = true)
 |-- Deposit_Sum_365D: double (nullable = true)
 |-- Payout_Days_Count_30D: long (nullable = true)
 |-- Payout_Txns_Count_30D: long (nullable = true)
 |-- Payout_Sum_30D: double (nullable = true)
 |-- Payout_Days_Count_90D: long (nullable = true)
```

```
# --- 13. Add Historical Excel-Based Classification (with Dynamic Status Score using 365D proxies) ---
from pyspark.sql.functions import (
    col, to_date, lit, datediff, add_months, expr, greatest, least,
```

```
        when, coalesce, date_sub, sum as pyspark_sum, avg as pyspark_avg,
        count as pyspark_count, min as pyspark_min, max as pyspark_max,
        round as pyspark_round
    )
    from pyspark.sql.types import IntegerType, DoubleType, StringType

    print("\n--- Adding Historical Excel-Based Classification (with Dynamic Status Score & 365D Proxies) ---")
    try:
        abt_df_input_for_class = spark.read.parquet(output_path_parquet) # Use a different name for clarity
        print(f"Read final ABT from {output_path_parquet} with {abt_df_input_for_class.count()} rows.") # Action here, might fail if input i

        # It's good practice to persist the DataFrame you'll be transforming multiple times
        abt_df_input_for_class.persist()
        # Trigger action to cache
        initial_count_for_class_cell = abt_df_input_for_class.count()
        print(f"Successfully read and count verified for input ABT: {initial_count_for_class_cell} rows.")


        # Constants
        # ... (Constants remain the same) ...
        WEIGHT_TRADE_DAYS = 220.0; MAX_SCORE_TRADE_DAYS = 270.0; WEIGHT_AUM = 200.0; MAX_SCORE_AUM = 300.0
        WEIGHT_BROKERAGE = 300.0; MAX_SCORE_BROKERAGE = 350.0; BENCHMARK_RECENCY = 180.0; MAX_SCORE_RECENCY = 180.0
        TARGET_TRADE_DAYS_FIXED = 54.3; TARGET_AUM_FIXED = 258084.0; TARGET_BROKERAGE_FIXED = 6671.10
        CHURN_LABEL_FOR_STATUS_SCORE = "Is_Churned_Engage_365Days"

        # Start transformations on a new variable 'df_transformed'
        df_transformed = abt_df_input_for_class

        # --- Calculate Dynamic Status Score (S) ---
        print(f"  Calculating Dynamic Status Score based on {CHURN_LABEL_FOR_STATUS_SCORE}...")
        if CHURN_LABEL_FOR_STATUS_SCORE not in df_transformed.columns:
            raise ValueError(f"Churn label column '{CHURN_LABEL_FOR_STATUS_SCORE}' not found in ABT.")
        df_transformed = df_transformed.withColumn("Excel_Status_Score_S_Dynamic",
            when(col(CHURN_LABEL_FOR_STATUS_SCORE) == 0, 100.0)
            .when(col(CHURN_LABEL_FOR_STATUS_SCORE) == 1,
                  when(col("Last_Trade_Date").isNull(), 0.0)
                  .otherwise(75.0))
            .otherwise(100.0))

        # --- Calculate Intermediate Excel-like Columns ---
        print("  Calculating intermediate values for classification (using 365D as proxy for 36M)...")
        df_transformed = df_transformed.withColumn("term1_max_brok", coalesce(col("Trade_Sum_180D"), lit(float('-inf'))))
        df_transformed = df_transformed.withColumn("term2_max_brok", coalesce(col("Trade_Sum_365D") / 6.0, lit(float('-inf'))))
        df_transformed = df_transformed.withColumn("Excel_Max_6_12M_Brok_Calc_temp", greatest(col("term1_max_brok"), col("term2_max_brok")))
        df_transformed = df_transformed.withColumn("Excel_Max_6_12M_Brok_Calc",
            when(col("Excel_Max_6_12M_Brok_Calc_temp") == float('-inf'), lit(0.0))
            .otherwise(col("Excel_Max_6_12M_Brok_Calc_temp"))
        ).drop("term1_max_brok", "term2_max_brok", "Excel_Max_6_12M_Brok_Calc_temp")

        # --- Score (Trading Days) ---
        df_transformed = df_transformed.withColumn("Excel_TradeDays_Achievement",
                           when(lit(TARGET_TRADE_DAYS_FIXED) != 0, col("Trade_Days_Count_365D") / lit(TARGET_TRADE_DAYS_FIXED))
                           .otherwise(0.0))
        df_transformed = df_transformed.withColumn("Excel_Score_TradeDays",
                           least(
                               (pyspark_round(col("Excel_TradeDays_Achievement") * lit(WEIGHT_TRADE_DAYS)) + col("Excel_Status_Score_S_Dynar
                               lit(MAX_SCORE_TRADE_DAYS)
                           ))
        df_transformed = df_transformed.fillna(0.0, subset=["Excel_Score_TradeDays"])

        # ... (Rest of the score calculations: AUM, Brokerage, Recency, TOTAL_SCORE, Slab & Tag) ...
        # ... (Ensure all these transformations use 'df_transformed = df_transformed.withColumn(...)' ) ...

        # --- Score (AUM) ---
        df_transformed = df_transformed.withColumn("Excel_AUM_ForAchievement", greatest(coalesce(col("AUM_SnapshotMonth_Monthly"), lit(0.0)),
        df_transformed = df_transformed.withColumn("Excel_Score_AUM",
                           when(lit(TARGET_AUM_FIXED) != 0,
                                pyspark_round(least((col("Excel_AUM_ForAchievement") / lit(TARGET_AUM_FIXED)) * lit(WEIGHT_AUM), lit(MAX_SC(
                               )
                           .otherwise(0.0))
        df_transformed = df_transformed.fillna(0.0, subset=["Excel_Score_AUM"])

        # --- Score (Brokerage) ---
        df_transformed = df_transformed.withColumn("Excel_Brokerage_Achievement", col("Excel_Max_6_12M_Brok_Calc"))
        df_transformed = df_transformed.withColumn("Excel_Score_Brokerage",
                           when(lit(TARGET_BROKERAGE_FIXED) != 0,
                                pyspark_round(least((col("Excel_Brokerage_Achievement") / lit(TARGET_BROKERAGE_FIXED)) * lit(WEIGHT_BROKERA(
                               )
                           .otherwise(0.0))
        df_transformed = df_transformed.fillna(0.0, subset=["Excel_Score_Brokerage"])

        # --- Score (Recency) ---
```

```
df_transformed = df_transformed.withColumn("Max_Activity_Date_For_Recency",
                   greatest(
                       coalesce(col("Last_Trade_Date"), date_sub(col("SnapshotDate"), 99999)),
                       coalesce(col("ActivationDate"), date_sub(col("SnapshotDate"), 99999)),
                       coalesce(col("Last_Deposit_Date"), date_sub(col("SnapshotDate"), 99999)),
                       coalesce(col("Last_Login_Date"), date_sub(col("SnapshotDate"), 99999))
                   ))
df_transformed = df_transformed.withColumn("Days_Since_Max_Activity_For_Recency",
                    when(col("Max_Activity_Date_For_Recency").isNotNull(),
                         datediff(col("SnapshotDate"), col("Max_Activity_Date_For_Recency"))
                    ).otherwise(99999)
                  )
df_transformed = df_transformed.withColumn("Excel_Recency_RawScore",
                   greatest(lit(BENCHMARK_RECENCY) - col("Days_Since_Max_Activity_For_Recency"), lit(0.0)))
df_transformed = df_transformed.withColumn("Excel_Score_Recency", least(col("Excel_Recency_RawScore"), lit(MAX_SCORE_RECENCY)))
df_transformed = df_transformed.fillna(0.0, subset=["Excel_Score_Recency"])

# --- TOTAL SCORE ---
df_transformed = df_transformed.withColumn("Historical_Total_Score",
                   (coalesce(col("Excel_Score_Recency"), lit(0.0)) +
                    coalesce(col("Excel_Score_Brokerage"), lit(0.0)) +
                    coalesce(col("Excel_Score_AUM"), lit(0.0)) +
                    coalesce(col("Excel_Score_TradeDays"), lit(0.0))
                   ).cast(IntegerType()))

# --- Slab & Tag ---
df_transformed = df_transformed.withColumn("Temp_AG4_Numeric_Slab",
    when(col("Historical_Total_Score") >= 1100, 1100.0)
    .when(col("Historical_Total_Score") >= 900, 900.0)
    .when(col("Historical_Total_Score") >= 700, 700.0)
    .when(col("Historical_Total_Score") >= 450, 450.0)
    .when(col("Historical_Total_Score") >= 0, 0.0)
    .otherwise(-1.0))
df_transformed = df_transformed.withColumn("Temp_AH4_Score_Slab_Numeric_Final",
    when(col("Tenure_Days") > 90, col("Temp_AG4_Numeric_Slab"))
    .when((col("Tenure_Days") <= 90) & (col("Temp_AG4_Numeric_Slab") >= 450.0), col("Temp_AG4_Numeric_Slab"))
    .otherwise(-1.0))
df_transformed = df_transformed.withColumn("Historical_Tag",
    when(col("Temp_AH4_Score_Slab_Numeric_Final") >= 1100.0, "Platinum +")
    .when(col("Temp_AH4_Score_Slab_Numeric_Final") >= 900.0, "Platinum")
    .when(col("Temp_AH4_Score_Slab_Numeric_Final") >= 700.0, "Gold")
    .when(col("Temp_AH4_Score_Slab_Numeric_Final") >= 450.0, "Silver")
    .when(col("Temp_AH4_Score_Slab_Numeric_Final") >= 0.0, "Classic")
    .otherwise("New"))

# --- Final Selection ---
original_abt_cols = abt_df_input_for_class.columns # Columns from the ABT we read
new_classification_cols = ["Excel_Status_Score_S_Dynamic", "Historical_Total_Score", "Historical_Tag"]

# Ensure we select columns from df_transformed, which has all the new ones
# And ensure we don't try to select a column that might have been dropped or doesn't exist

# Build the list of all columns expected in df_transformed by this point
# This includes original_abt_cols + all intermediate Excel_ cols + new_classification_cols
# For the final output, we only want original_abt_cols + new_classification_cols

cols_to_select_ordered = []
for c in original_abt_cols: # Keep all original columns
    cols_to_select_ordered.append(c)
for c_new in new_classification_cols: # Add new ones if they don't conflict
    if c_new not in original_abt_cols and c_new in df_transformed.columns:
        cols_to_select_ordered.append(c_new)
    elif c_new in original_abt_cols: # If it somehow already existed (e.g. rerun)
        pass # Already included
    elif c_new not in df_transformed.columns:
        print(f"Warning: Expected new column {c_new} not found in df_transformed. Skipping.")


final_classified_abt_df = df_transformed.select(cols_to_select_ordered)

# final_classified_abt_df.persist() # <<--- DO NOT PERSIST HERE for this attempt

classified_abt_count = final_classified_abt_df.count() # Action that might cause OOM
print("\nHistorical classification added. Sample:")
# ... (show sample as before) ...
final_classified_abt_df.select("ClientCode", "SnapshotDate", CHURN_LABEL_FOR_STATUS_SCORE,
                               "Last_Trade_Date", "Excel_Status_Score_S_Dynamic",
                               "Historical_Total_Score", "Historical_Tag", "Tenure_Days") \
                       .orderBy("ClientCode", "SnapshotDate").show(10, truncate=False)
print(f"Final ABT with classification has {classified_abt_count} rows and {len(final_classified_abt_df.columns)} columns.")

if classified_abt_count > 0:
```

```
        print(f"Overwriting final ABT at: {output_path_parquet} with classification features.")
        final_classified_abt_df.write.mode("overwrite").parquet(output_path_parquet) # This is the important action
        print("Final ABT with classification features successfully saved.")
    else:
        print("Classified ABT is empty. Not writing.")

    if abt_df_input_for_class.is_cached: # Unpersist the input DF
        abt_df_input_for_class.unpersist()
    # No final_classified_abt_df.unpersist() as it wasn't persisted

except FileNotFoundError:
    print(f"ERROR: Could not read final ABT from {output_path_parquet}.")
except Exception as e:
    print(f"Error in Historical Classification or writing: {e}")
    raise e
```

```
--- Adding Historical Excel-Based Classification (with Dynamic Status Score & 365D Proxies) ---
Read final ABT from /content/drive/MyDrive/Tables/output_abt_final_pred/predictive_abt_religare_churn_2021_2023.parquet with 3319751
Successfully read and count verified for input ABT: 33197513 rows.
  Calculating Dynamic Status Score based on Is_Churned_Engage_365Days...
  Calculating intermediate values for classification (using 365D as proxy for 36M)...

Historical classification added. Sample:
+----------+------------+------------------------+---------------+----------------------------+----------------------+------------
|ClientCode|SnapshotDate|Is_Churned_Engage_365Days|Last_Trade_Date|Excel_Status_Score_S_Dynamic|Historical_Total_Score|Historical_Ta
+----------+------------+------------------------+---------------+----------------------------+----------------------+------------
|A*        |2021-01-31  |0                       |NULL           |100.0                       |100                   |Classic
|A*        |2021-02-28  |0                       |NULL           |100.0                       |100                   |Classic
|A*        |2021-03-31  |0                       |NULL           |100.0                       |100                   |Classic
|A*        |2021-04-30  |0                       |NULL           |100.0                       |100                   |Classic
|A*        |2021-05-31  |0                       |NULL           |100.0                       |100                   |Classic
|A*        |2021-06-30  |0                       |NULL           |100.0                       |100                   |Classic
|A*        |2021-07-31  |0                       |NULL           |100.0                       |100                   |Classic
|A*        |2021-08-31  |0                       |NULL           |100.0                       |100                   |Classic
|A*        |2021-09-30  |0                       |NULL           |100.0                       |100                   |Classic
|A*        |2021-10-31  |0                       |NULL           |100.0                       |100                   |Classic
+----------+------------+------------------------+---------------+----------------------------+----------------------+------------
only showing top 10 rows

Final ABT with classification has 33197513 rows and 94 columns.
Overwriting final ABT at: /content/drive/MyDrive/Tables/output_abt_final_pred/predictive_abt_religare_churn_2021_2023.parquet with c
Final ABT with classification features successfully saved.
```

Start coding or generate with AI.