```
# Import necessary PySpark functions
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, to_date, lit, lag, datediff, min as pyspark_min, max as pyspark_max, count, when, expr, row_number
from pyspark.sql.window import Window
import os

# --- 0. 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 or google.colab.drive module not found. Assuming local file system.")
    google_drive_base_path = "" # Or set to your local base path

# Initialize SparkSession
spark = SparkSession.builder.appName("TimeToInactivityAnalysis") \
    .config("spark.sql.legacy.timeParserPolicy", "LEGACY") \
    .getOrCreate() # LEGACY policy for robust date parsing

# Define paths to data files
input_base_dir_drive = os.path.join(google_drive_base_path, 'Tables/')
login_data_dir_drive = os.path.join(google_drive_base_path, 'LOG_NEW/') # For login data

client_details_filename = "client_details.txt"
trade_data_filename = "trade_data.txt" # The one with CLIENTCODE,TRADE_DATE,TOTAL_GROSS_BROKERAGE_DAY

client_details_path = os.path.join(input_base_dir_drive, client_details_filename)
trade_data_path = os.path.join(input_base_dir_drive, trade_data_filename)
login_data_path_pattern = os.path.join(login_data_dir_drive, "LOGIN_*.txt")

print(f"Client details path: {client_details_path}")
print(f"Trade data path: {trade_data_path}")
print(f"Login data pattern: {login_data_path_pattern}")
```

```
Mounted at /content/drive
Google Drive mounted successfully.
Client details path: /content/drive/MyDrive/Tables/client_details.txt
Trade data path: /content/drive/MyDrive/Tables/trade_data.txt
Login data pattern: /content/drive/MyDrive/LOG_NEW/LOGIN_*.txt
```

```
# --- Load Client Master Data ---
try:
    client_master_df_raw = spark.read.format("csv") \
        .option("header", "true") \
        .option("delimiter", ",") \
        .load(client_details_path)

    client_master_df = client_master_df_raw.select(
        col("CLIENTCODE").alias("ClientCode"),
        to_date(col("ACTIVATIONDATE"), "dd/MM/yyyy").alias("ActivationDate") # Corrected format
    ).filter(col("ActivationDate").isNotNull()).distinct()

    print("Client master data loaded and processed:")
    client_master_df.show(5, truncate=False)
    print(f"Total distinct clients with activation date: {client_master_df.count()}")

except Exception as e:
    print(f"Error loading client_details.txt: {e}")
    # spark.stop() # Stop Spark if critical error
    # exit()
```

```
Client master data loaded and processed:
+----------+--------------+
|ClientCode|ActivationDate|
+----------+--------------+
|AA1291    |2007-01-15    |
|AA1365    |2007-02-27    |
|AA1505    |2007-05-10    |
|AA2120    |2007-11-22    |
|AA2248    |2007-12-15    |
+----------+--------------+
only showing top 5 rows

Total distinct clients with activation date: 1316511
```

```
# --- Load Trade Data ---
# Header: CLIENTCODE,TRADE_DATE,TOTAL_GROSS_BROKERAGE_DAY
# Delimiter: comma (,)
# Date Format: dd/MM/yyyy
```

```
try:
    trades_df = spark.read.format("csv") \
        .option("header", "true") \
        .option("delimiter", ",") \
        .load(trade_data_path)

    trades_df = trades_df.select(
        col("CLIENTCODE").alias("ClientCode"),
        to_date(col("TRADE_DATE"), "dd/MM/yyyy").alias("ActivityDate")
    ).withColumn("ActivityType", lit("Trade")) \
     .filter(col("ActivityDate").isNotNull())

    print("Trade data loaded and processed:")
    trades_df.show(5, truncate=False)
    print(f"Total trade activities: {trades_df.count()}")

except Exception as e:
    print(f"Error loading trade_data.txt: {e}")
```

```
Trade data loaded and processed:
+----------+------------+------------+
|ClientCode|ActivityDate|ActivityType|
+----------+------------+------------+
|SD9627    |2020-08-04  |Trade       |
|AV3818    |2020-08-04  |Trade       |
|NR2513    |2020-08-04  |Trade       |
|BN1496    |2020-08-04  |Trade       |
|UM1246    |2020-08-04  |Trade       |
+----------+------------+------------+
only showing top 5 rows

Total trade activities: 17254800
```

```
# --- Load Login Data ---
# Format: ClientCode,DD/MM/YYYY (no header)
try:
    # Define schema for login files as they have no header
    from pyspark.sql.types import StructType, StructField, StringType, DateType
    login_schema = StructType([
        StructField("ClientCode_raw", StringType(), True),
        StructField("LoginDate_str", StringType(), True)
    ])

    logins_df_raw = spark.read.format("csv") \
        .schema(login_schema) \
        .option("delimiter", ",") \
        .load(login_data_path_pattern) # Wildcard path

    logins_df = logins_df_raw.select(
        col("ClientCode_raw").alias("ClientCode"), # Assuming it's just client code, no extra chars
        to_date(col("LoginDate_str"), "dd/MM/yyyy").alias("ActivityDate")
    ).withColumn("ActivityType", lit("Login")) \
     .filter(col("ActivityDate").isNotNull())

    print("Login data loaded and processed:")
    logins_df.show(5, truncate=False)
    print(f"Total login activities: {logins_df.count()}")

except Exception as e:
    print(f"Error loading login data: {e}")
```

```
Login data loaded and processed:
+----------+------------+------------+
|ClientCode|ActivityDate|ActivityType|
+----------+------------+------------+
|PK70      |2023-07-03  |Login       |
|PK70      |2023-07-03  |Login       |
|PK70      |2023-07-03  |Login       |
|PK70      |2023-07-03  |Login       |
|PK70      |2023-07-03  |Login       |
+----------+------------+------------+
only showing top 5 rows

Total login activities: 176232060
```

```
# --- Combine All Activities ---
# Ensure columns match for union: ClientCode, ActivityDate, ActivityType
# Renaming during select in previous steps should handle this.
if 'trades_df' in locals() and 'logins_df' in locals():
    all_activities_df = trades_df.unionByName(logins_df)
    print("Combined activities:")
    all_activities_df.show(5, truncate=False)
    print(f"Total combined activities before distinct: {all_activities_df.count()}")
```

```
else:
    print("Skipping activity combination as one of the DFs is missing.")
```

```
Combined activities:
+----------+------------+------------+
|ClientCode|ActivityDate|ActivityType|
+----------+------------+------------+
|SD9627    |2020-08-04  |Trade       |
|AV3818    |2020-08-04  |Trade       |
|NR2513    |2020-08-04  |Trade       |
|BN1496    |2020-08-04  |Trade       |
|UM1246    |2020-08-04  |Trade       |
+----------+------------+------------+
only showing top 5 rows

Total combined activities before distinct: 193486860
```

```
# --- Join Activities with Client Master & Get Distinct Post-Activation Activities ---
if 'all_activities_df' in locals() and 'client_master_df' in locals():
    # Broadcast client_master_df if it's small enough, for optimization
    client_activity_joined_df = all_activities_df.join(
        broadcast(client_master_df), # Broadcast hint
        "ClientCode",
        "inner"
    )

    # Filter activities before activation date
    client_activity_filtered_df = client_activity_joined_df.filter(
        col("ActivityDate") >= col("ActivationDate")
    )

    # Select distinct activity dates per client post-activation
    client_distinct_activities_df = client_activity_filtered_df.select(
        "ClientCode", "ActivationDate", "ActivityDate"
    ).distinct()

    client_distinct_activities_df.persist() # Persist as this will be used multiple times

    print("Client distinct activities (post-activation):")
    client_distinct_activities_df.orderBy("ClientCode", "ActivityDate").show(10, truncate=False)
    print(f"Total client distinct activity records: {client_distinct_activities_df.count()}")
else:
    print("Skipping join as one of the DFs is missing.")
```

```
Client distinct activities (post-activation):
+----------+--------------+------------+
|ClientCode|ActivationDate|ActivityDate|
+----------+--------------+------------+
|AA001     |2014-10-28    |2020-08-03  |
|AA001     |2014-10-28    |2020-08-04  |
|AA001     |2014-10-28    |2020-08-05  |
|AA001     |2014-10-28    |2020-08-06  |
|AA001     |2014-10-28    |2020-08-07  |
|AA001     |2014-10-28    |2020-08-10  |
|AA001     |2014-10-28    |2020-08-11  |
|AA001     |2014-10-28    |2020-08-12  |
|AA001     |2014-10-28    |2020-08-13  |
|AA001     |2014-10-28    |2020-08-14  |
+----------+--------------+------------+
only showing top 10 rows

Total client distinct activity records: 41420852
```

```
# --- Phase 2: Calculate Inter-Activity Gaps ---

# Step 8 (Revised): Create a DataFrame of (ClientCode, ActivationDate as ActivityDate)
# and union it with actual distinct activities.
# This ensures ActivationDate is the first point of reference for calculating gaps.

if 'client_distinct_activities_df' in locals() and 'client_master_df' in locals():
    # DataFrame with just activation dates, aliased to 'ActivityDate'
    activation_event_df = client_master_df.select(
        col("ClientCode"),
        col("ActivationDate").alias("ActivityDate") # ActivationDate itself is an "event point"
    )

    # Union this with the actual distinct activity dates
    # client_distinct_activities_df already has ClientCode and ActivityDate
    # Ensure client_distinct_activities_df only has ClientCode and ActivityDate for this union

    # Get distinct actual activities (ClientCode, ActivityDate)
    actual_activities_for_union_df = client_distinct_activities_df.select("ClientCode", "ActivityDate")

    # Union activation points with actual activity points
```

```
    all_event_points_df = activation_event_df.unionByName(actual_activities_for_union_df).distinct()

    # Join back ActivationDate for reference (needed for datediff from activation)
    all_event_points_with_activation_df = all_event_points_df.join(
        client_master_df.select("ClientCode", "ActivationDate").alias("master"), # Use alias to avoid ambiguous ClientCode
        all_event_points_df.ClientCode == col("master.ClientCode")
    ).select(all_event_points_df.ClientCode, "ActivityDate", "ActivationDate")


    all_event_points_with_activation_df.persist() # Persist this as it's key for gap calculation

    print("All event points (Activation + Actual Activities) per client:")
    all_event_points_with_activation_df.orderBy("ClientCode", "ActivityDate").show(15, truncate=False)
    print(f"Total event points records: {all_event_points_with_activation_df.count()}")
else:
    print("Skipping step 8 as client_distinct_activities_df or client_master_df is missing.")
```

```
All event points (Activation + Actual Activities) per client:
+----------+------------+--------------+
|ClientCode|ActivityDate|ActivationDate|
+----------+------------+--------------+
|A*        |2005-04-13  |2005-04-13    |
|A-        |2014-09-04  |2014-09-04    |
|A.        |2004-03-10  |2004-03-10    |
|A..       |2004-12-23  |2004-12-23    |
|A...      |2005-05-23  |2005-05-23    |
|A....     |2006-02-01  |2006-02-01    |
|A.....    |2005-08-24  |2005-08-24    |
|A.......  |2006-01-10  |2006-01-10    |
|A.........|2005-11-16  |2005-11-16    |
|A1        |2014-09-05  |2014-09-05    |
|A180868878|2018-08-31  |2018-08-31    |
|A210322136|2021-03-24  |2021-03-24    |
|AA        |2005-03-02  |2005-03-02    |
|AA001     |2014-10-28  |2014-10-28    |
|AA001     |2020-08-03  |2014-10-28    |
+----------+------------+--------------+
only showing top 15 rows

Total event points records: 42735506
```

```
if 'all_event_points_with_activation_df' in locals():
    # Step 9: Calculate Previous Activity Date
    window_spec_activity = Window.partitionBy("ClientCode").orderBy("ActivityDate")

    activity_with_lag_df = all_event_points_with_activation_df.withColumn(
        "Previous_ActivityDate",
        lag("ActivityDate", 1).over(window_spec_activity)
    )

    # Step 10: Calculate Gap Durations
    # The first 'Previous_ActivityDate' for each client will be null.
    # The gap is between the current ActivityDate and the Previous_ActivityDate.
    # If Previous_ActivityDate is null (i.e., for the ActivationDate itself when considered as the first event),
    # the gap isn't meaningful in the same way as inter-activity gaps.
    # For the very first event point (which should be ActivationDate), Previous_ActivityDate is null.
    # The first *meaningful* gap is between the first *actual* activity and the ActivationDate.

    inter_activity_gaps_df = activity_with_lag_df.withColumn(
        "Gap_In_Days",
        datediff(col("ActivityDate"), col("Previous_ActivityDate"))
    )

    # Filter out rows where Previous_ActivityDate is null, as the gap isn't between two activities.
    # These rows correspond to the ActivationDate itself when it's the first point.
    # The gaps we are interested in are those *after* the activation date reference point.
    meaningful_gaps_df = inter_activity_gaps_df.filter(col("Previous_ActivityDate").isNotNull())

    meaningful_gaps_df.persist()

    print("Inter-activity gaps calculated:")
    meaningful_gaps_df.orderBy("ClientCode", "ActivityDate").show(15, truncate=False)
    print(f"Total meaningful gap records: {meaningful_gaps_df.count()}")

    # Unpersist the previous DFs if no longer directly needed
    if client_distinct_activities_df.is_cached:
        client_distinct_activities_df.unpersist()
    if all_event_points_with_activation_df.is_cached:
        all_event_points_with_activation_df.unpersist()
else:
    print("Skipping steps 9 & 10 as all_event_points_with_activation_df is missing.")
```

```
Inter-activity gaps calculated:
+----------+------------+--------------+---------------------+-----------+
```

```
|ClientCode|ActivityDate|ActivationDate|Previous_ActivityDate|Gap_In_Days|
+----------+------------+--------------+---------------------+-----------+
|AA001     |2020-08-03  |2014-10-28    |2014-10-28           |2106       |
|AA001     |2020-08-04  |2014-10-28    |2020-08-03           |1          |
|AA001     |2020-08-05  |2014-10-28    |2020-08-04           |1          |
|AA001     |2020-08-06  |2014-10-28    |2020-08-05           |1          |
|AA001     |2020-08-07  |2014-10-28    |2020-08-06           |1          |
|AA001     |2020-08-10  |2014-10-28    |2020-08-07           |3          |
|AA001     |2020-08-11  |2014-10-28    |2020-08-10           |1          |
|AA001     |2020-08-12  |2014-10-28    |2020-08-11           |1          |
|AA001     |2020-08-13  |2014-10-28    |2020-08-12           |1          |
|AA001     |2020-08-14  |2014-10-28    |2020-08-13           |1          |
|AA001     |2020-08-17  |2014-10-28    |2020-08-14           |3          |
|AA001     |2020-08-18  |2014-10-28    |2020-08-17           |1          |
|AA001     |2020-08-19  |2014-10-28    |2020-08-18           |1          |
|AA001     |2020-08-20  |2014-10-28    |2020-08-19           |1          |
|AA001     |2020-08-21  |2014-10-28    |2020-08-20           |1          |
+----------+------------+--------------+---------------------+-----------+
only showing top 15 rows

Total meaningful gap records: 41418995
```

```python
# --- Phase 3: Find Time to First N-Day Inactivity Spell ---

if 'meaningful_gaps_df' in locals() and 'client_master_df' in locals():
    n_day_windows = [60, 90, 270, 365]

    # Base DataFrame to join results onto
    client_survival_times_df = client_master_df.select("ClientCode", "ActivationDate")

    for n_days in n_day_windows:
        print(f"\nProcessing for N = {n_days} days...")

        # Find rows where Gap_In_Days >= N
        n_day_inactivity_spells_df = meaningful_gaps_df.filter(col("Gap_In_Days") >= n_days)

        # For each client, find the first such spell
        # The 'ActivityDate' is when the spell ended (or when activity resumed)
        # The 'Previous_ActivityDate' is when the N-day (or longer) spell BEGAN.
        window_first_spell = Window.partitionBy("ClientCode").orderBy("ActivityDate") # Order by when spell ended

        first_n_day_spell_df = n_day_inactivity_spells_df.withColumn(
            "spell_rank",
            row_number().over(window_first_spell)
        ).filter(col("spell_rank") == 1) \
         .select(
            col("ClientCode"),
            col("Previous_ActivityDate").alias(f"Start_First_{n_days}D_Inactivity"), # This is when the N-day spell started
            col("ActivationDate").alias(f"ActivationDate_spell_{n_days}") # Carry over for calculation
         )

        # Calculate Time_To_Start_of_First_N_Day_Inactivity
        # This is days from ActivationDate until the N-day inactivity spell BEGAN.
        first_n_day_spell_df = first_n_day_spell_df.withColumn(
            f"Time_To_First_{n_days}D_Inactivity_Start",
            datediff(col(f"Start_First_{n_days}D_Inactivity"), col(f"ActivationDate_spell_{n_days}"))
        ).select("ClientCode", f"Time_To_First_{n_days}D_Inactivity_Start") # Keep only necessary columns

        # Left join this result back to our main client survival DataFrame
        client_survival_times_df = client_survival_times_df.join(
            first_n_day_spell_df,
            "ClientCode",
            "left"
        )

        print(f"Finished processing N = {n_days}. Columns in client_survival_times_df: {client_survival_times_df.columns}")
        # client_survival_times_df.show(5, truncate=False) # Optional: show intermediate results

    client_survival_times_df.persist()
    print("\nFinal DataFrame with time to first N-day inactivity start (for clients who experienced it):")
    client_survival_times_df.show(10, truncate=False)
    print(f"Total clients in survival times df: {client_survival_times_df.count()}")

else:
    print("Skipping Phase 3, Step 11 as meaningful_gaps_df or client_master_df is missing.")
```

```
Processing for N = 60 days...
Finished processing N = 60. Columns in client_survival_times_df: ['ClientCode', 'ActivationDate', 'Time_To_First_60D_Inactivity_Star

Processing for N = 90 days...
Finished processing N = 90. Columns in client_survival_times_df: ['ClientCode', 'ActivationDate', 'Time_To_First_60D_Inactivity_Star

Processing for N = 270 days...
```

```
Finished processing N = 270. Columns in client_survival_times_df: ['ClientCode', 'ActivationDate', 'Time_To_First_60D_Inactivity_Sta

Processing for N = 365 days...
Finished processing N = 365. Columns in client_survival_times_df: ['ClientCode', 'ActivationDate', 'Time_To_First_60D_Inactivity_Sta

Final DataFrame with time to first N-day inactivity start (for clients who experienced it):
+----------+--------------+---------------------------------+---------------------------------+---------------------------------
|ClientCode|ActivationDate|Time_To_First_60D_Inactivity_Start|Time_To_First_90D_Inactivity_Start|Time_To_First_270D_Inactivity_Start
+----------+--------------+---------------------------------+---------------------------------+---------------------------------
|AA1139    |2006-10-19    |NULL                             |NULL                             |NULL
|AA1255    |2006-12-28    |NULL                             |NULL                             |NULL
|AA1408    |2007-03-20    |NULL                             |NULL                             |NULL
|AA1440    |2007-04-12    |NULL                             |NULL                             |NULL
|AA1474    |2007-05-04    |NULL                             |NULL                             |NULL
|AA1587    |2007-06-06    |NULL                             |NULL                             |NULL
|AA171     |2005-03-18    |NULL                             |NULL                             |NULL
|AA1839    |2007-08-29    |0                                |0                                |0
|AA1924    |2007-10-04    |NULL                             |NULL                             |NULL
|AA1944    |2007-10-09    |0                                |0                                |0
+----------+--------------+---------------------------------+---------------------------------+---------------------------------
only showing top 10 rows

Total clients in survival times df: 1316511
```

```python
if 'client_survival_times_df' in locals() and 'all_event_points_with_activation_df' in locals(): # Using all_event_points not just mean

    # First, get the last known activity date for all clients from the complete event timeline
    # (this includes their activation date if they had no other activity)
    last_activity_df = all_event_points_with_activation_df.groupBy("ClientCode") \
        .agg(
            pyspark_max("ActivityDate").alias("Last_Known_ActivityDate"),
            pyspark_min("ActivationDate").alias("ActivationDate_for_censoring") # Get ActivationDate consistently
        )

    # Join this to our survival times DataFrame
    client_survival_times_df = client_survival_times_df.join(
        last_activity_df,
        "ClientCode",
        "left" # Should be inner effectively, but left is safer if client_master had clients not in activities
    )

    print("\nJoined last known activity date:")
    client_survival_times_df.show(5, truncate=False)

    for n_days in n_day_windows:
        time_col_name = f"Time_To_First_{n_days}D_Inactivity_Start"
        censored_col_name = f"Is_Censored_{n_days}D"

        # Calculate censoring time: days from ActivationDate to Last_Known_ActivityDate
        # This is the duration they were observed without hitting the N-day inactivity.
        censoring_time_col_name = f"Censoring_Time_For_{n_days}D"
        client_survival_times_df = client_survival_times_df.withColumn(
            censoring_time_col_name,
            datediff(col("Last_Known_ActivityDate"), col("ActivationDate_for_censoring"))
        )

        # Identify censored clients and fill their 'Time_To_First_N_Day_Inactivity_Start'
        client_survival_times_df = client_survival_times_df.withColumn(
            censored_col_name,
            when(col(time_col_name).isNull(), True).otherwise(False)
        )

        client_survival_times_df = client_survival_times_df.withColumn(
            time_col_name, # This is the final duration column (either time to event or censoring time)
            when(col(censored_col_name), col(censoring_time_col_name)) \
            .otherwise(col(time_col_name))
        )

        # Ensure the time is not negative (can happen if Last_Known_ActivityDate is somehow before ActivationDate_for_censoring due to d
        client_survival_times_df = client_survival_times_df.withColumn(
            time_col_name,
            when(col(time_col_name) < 0, 0).otherwise(col(time_col_name))
        )

    # Select final columns for clarity
    final_columns = ["ClientCode", "ActivationDate"] + \
                    [f"Time_To_First_{n}D_Inactivity_Start" for n in n_day_windows] + \
                    [f"Is_Censored_{n}D" for n in n_day_windows]

    final_client_survival_df = client_survival_times_df.select(final_columns)
    final_client_survival_df.persist()

    print("\nFinal DataFrame with censoring information:")
```

```
    final_client_survival_df.show(10, truncate=False)
    print(f"Total clients in final survival df: {final_client_survival_df.count()}")

    # Unpersist previous DFs
    if meaningful_gaps_df.is_cached:
        meaningful_gaps_df.unpersist()
    if 'client_survival_times_df' in locals() and client_survival_times_df.is_cached: # Check if it was created
         client_survival_times_df.unpersist() # Unpersist the intermediate one
else:
    print("Skipping Phase 3, Steps 12 & 13 due to missing DataFrames.")
```

```
Joined last known activity date:
+----------+--------------+---------------------------------+---------------------------------+---------------------------------
|ClientCode|ActivationDate|Time_To_First_60D_Inactivity_Start|Time_To_First_90D_Inactivity_Start|Time_To_First_270D_Inactivity_Start
+----------+--------------+---------------------------------+---------------------------------+---------------------------------
|AA1139    |2006-10-19    |NULL                             |NULL                             |NULL
|AA1255    |2006-12-28    |NULL                             |NULL                             |NULL
|AA1408    |2007-03-20    |NULL                             |NULL                             |NULL
|AA1440    |2007-04-12    |NULL                             |NULL                             |NULL
|AA1474    |2007-05-04    |NULL                             |NULL                             |NULL
+----------+--------------+---------------------------------+---------------------------------+---------------------------------
only showing top 5 rows


Final DataFrame with censoring information:
+----------+--------------+---------------------------------+---------------------------------+---------------------------------
|ClientCode|ActivationDate|Time_To_First_60D_Inactivity_Start|Time_To_First_90D_Inactivity_Start|Time_To_First_270D_Inactivity_Start
+----------+--------------+---------------------------------+---------------------------------+---------------------------------
|AA1139    |2006-10-19    |0                                |0                                |0
|AA1255    |2006-12-28    |0                                |0                                |0
|AA1408    |2007-03-20    |0                                |0                                |0
|AA1440    |2007-04-12    |0                                |0                                |0
|AA1474    |2007-05-04    |0                                |0                                |0
|AA1587    |2007-06-06    |0                                |0                                |0
|AA171     |2005-03-18    |0                                |0                                |0
|AA1839    |2007-08-29    |0                                |0                                |0
|AA1924    |2007-10-04    |0                                |0                                |0
|AA1944    |2007-10-09    |0                                |0                                |0
+----------+--------------+---------------------------------+---------------------------------+---------------------------------
only showing top 10 rows

Total clients in final survival df: 1316511
```

```
# --- Phase 4: Analysis and Visualization ---
from pyspark.sql.functions import avg, min as pyspark_min, max as pyspark_max, col, count # Explicitly import here

if 'final_client_survival_df' in locals():
    print("\n--- Descriptive Statistics for Time to First N-Day Inactivity Start ---")

    n_day_windows = [60, 90, 270, 365] # Redefine if not in scope from previous cell run

    summary_stats = []

    for n_days in n_day_windows:
        time_col = f"Time_To_First_{n_days}D_Inactivity_Start"
        censored_col = f"Is_Censored_{n_days}D"

        # Total clients
        total_clients = final_client_survival_df.count()

        # Number of clients who experienced the event (not censored)
        event_clients_df = final_client_survival_df.filter(col(censored_col) == False)
        event_count = event_clients_df.count()

        percentage_event = (event_count / total_clients) * 100 if total_clients > 0 else 0
        percentage_censored = 100 - percentage_event

        print(f"\n--- Stats for {n_days}-Day Inactivity Window ---")
        print(f"Total Clients: {total_clients}")
        print(f"Number of Clients Experiencing {n_days}D Inactivity: {event_count} ({percentage_event:.2f}%)")
        print(f"Number of Clients Censored for {n_days}D Inactivity: {total_clients - event_count} ({percentage_censored:.2f}%)")

        current_summary = {
            "N_Day_Window": n_days,
            "Total_Clients": total_clients,
            "Event_Count": event_count,
            "Percentage_Event": percentage_event,
            "Percentage_Censored": percentage_censored
        }

        if event_count > 0:
            # Calculate stats only for those who experienced the event
            stats_for_event_clients = event_clients_df.select(time_col).agg(
```

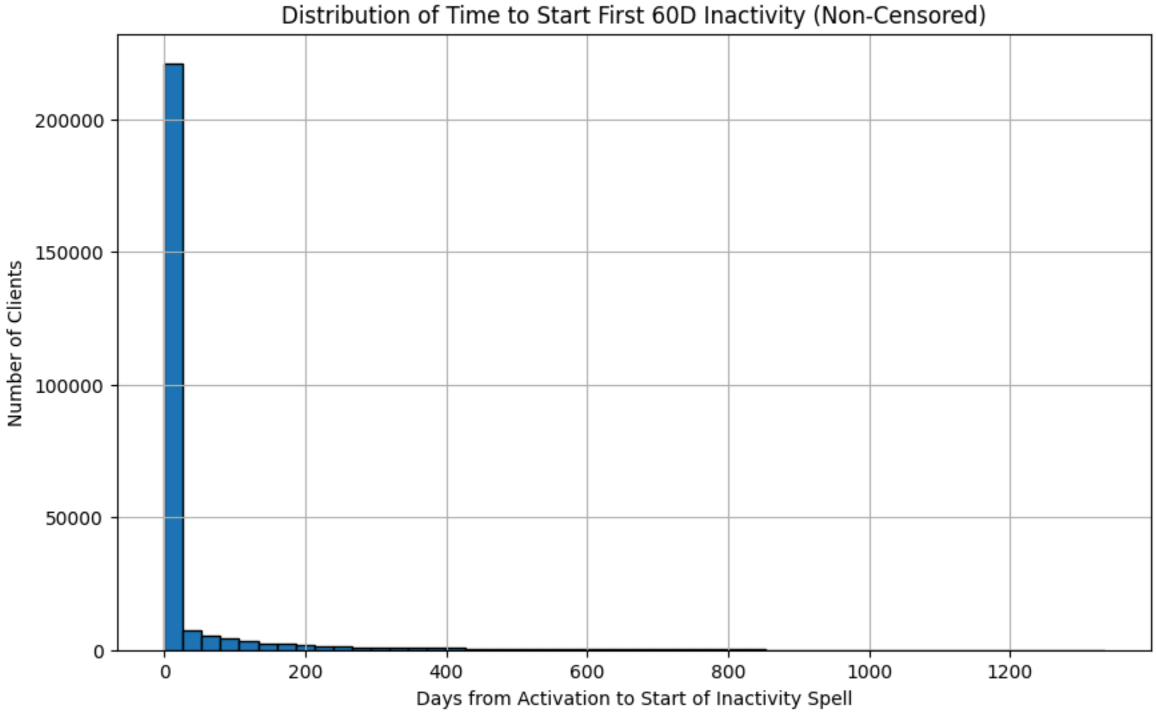
```
                avg(time_col).alias("Mean_TimeToEvent"),
                pyspark_min(time_col).alias("Min_TimeToEvent"),
                pyspark_max(time_col).alias("Max_TimeToEvent")
            ).first() # Get the Row object

            # For median and percentiles, it's better to use approxQuantile
            # Ensure time_col is numeric for approxQuantile; it should be from datediff
            quantiles = event_clients_df.approxQuantile(time_col, [0.25, 0.50, 0.75, 0.90], 0.01) # error tolerance 0.01

            print(f"  Mean Time to Start {n_days}D Inactivity (for those who experienced it): {stats_for_event_clients['Mean_TimeToEvent
            print(f"  Min Time to Start {n_days}D Inactivity: {stats_for_event_clients['Min_TimeToEvent']:.2f} days")
            print(f"  Max Time to Start {n_days}D Inactivity: {stats_for_event_clients['Max_TimeToEvent']:.2f} days")
            if quantiles:
                print(f"  25th Percentile Time to Start {n_days}D Inactivity: {quantiles[0]:.2f} days")
                print(f"  Median Time to Start {n_days}D Inactivity: {quantiles[1]:.2f} days")
                print(f"  75th Percentile Time to Start {n_days}D Inactivity: {quantiles[2]:.2f} days")
                print(f"  90th Percentile Time to Start {n_days}D Inactivity: {quantiles[3]:.2f} days")

            current_summary.update({
                "Mean_TimeToEvent": stats_for_event_clients['Mean_TimeToEvent'],
                "Min_TimeToEvent": stats_for_event_clients['Min_TimeToEvent'],
                "Max_TimeToEvent": stats_for_event_clients['Max_TimeToEvent'],
                "P25_TimeToEvent": quantiles[0] if quantiles else None,
                "Median_TimeToEvent": quantiles[1] if quantiles else None,
                "P75_TimeToEvent": quantiles[2] if quantiles else None,
                "P90_TimeToEvent": quantiles[3] if quantiles else None,
            })
        else:
            print(f"  No clients experienced {n_days}D inactivity.")

        summary_stats.append(current_summary)

    # Convert summary_stats list of dicts to a Spark DataFrame for nice display (optional)
    if summary_stats:
        summary_stats_df = spark.createDataFrame(summary_stats)
        print("\n--- Overall Summary Table ---")
        summary_stats_df.orderBy("N_Day_Window").show(truncate=False)

    # It's good practice to unpersist only if the DataFrame exists and is cached.
    if 'final_client_survival_df' in locals() and final_client_survival_df.is_cached:
        final_client_survival_df.unpersist()
else:
    print("Skipping Phase 4, Step 14 as final_client_survival_df is missing.")
```

```
--- Descriptive Statistics for Time to First N-Day Inactivity Start ---

--- Stats for 60-Day Inactivity Window ---
Total Clients: 1316511
Number of Clients Experiencing 60D Inactivity: 261851 (19.89%)
Number of Clients Censored for 60D Inactivity: 1054660 (80.11%)
  Mean Time to Start 60D Inactivity (for those who experienced it): 32.25 days
  Min Time to Start 60D Inactivity: 0.00 days
  Max Time to Start 60D Inactivity: 1333.00 days
  25th Percentile Time to Start 60D Inactivity: 0.00 days
  Median Time to Start 60D Inactivity: 0.00 days
  75th Percentile Time to Start 60D Inactivity: 0.00 days
  90th Percentile Time to Start 60D Inactivity: 74.00 days

--- Stats for 90-Day Inactivity Window ---
Total Clients: 1316511
Number of Clients Experiencing 90D Inactivity: 247789 (18.82%)
Number of Clients Censored for 90D Inactivity: 1068722 (81.18%)
  Mean Time to Start 90D Inactivity (for those who experienced it): 35.07 days
  Min Time to Start 90D Inactivity: 0.00 days
  Max Time to Start 90D Inactivity: 1312.00 days
  25th Percentile Time to Start 90D Inactivity: 0.00 days
  Median Time to Start 90D Inactivity: 0.00 days
  75th Percentile Time to Start 90D Inactivity: 0.00 days
  90th Percentile Time to Start 90D Inactivity: 81.00 days

--- Stats for 270-Day Inactivity Window ---
Total Clients: 1316511
Number of Clients Experiencing 270D Inactivity: 200831 (15.25%)
Number of Clients Censored for 270D Inactivity: 1115680 (84.75%)
  Mean Time to Start 270D Inactivity (for those who experienced it): 23.11 days
  Min Time to Start 270D Inactivity: 0.00 days
  Max Time to Start 270D Inactivity: 1320.00 days
  25th Percentile Time to Start 270D Inactivity: 0.00 days
  Median Time to Start 270D Inactivity: 0.00 days
  75th Percentile Time to Start 270D Inactivity: 0.00 days
  90th Percentile Time to Start 270D Inactivity: 0.00 days

--- Stats for 365-Day Inactivity Window ---
Total Clients: 1316511
Number of Clients Experiencing 365D Inactivity: 188525 (14.32%)
```

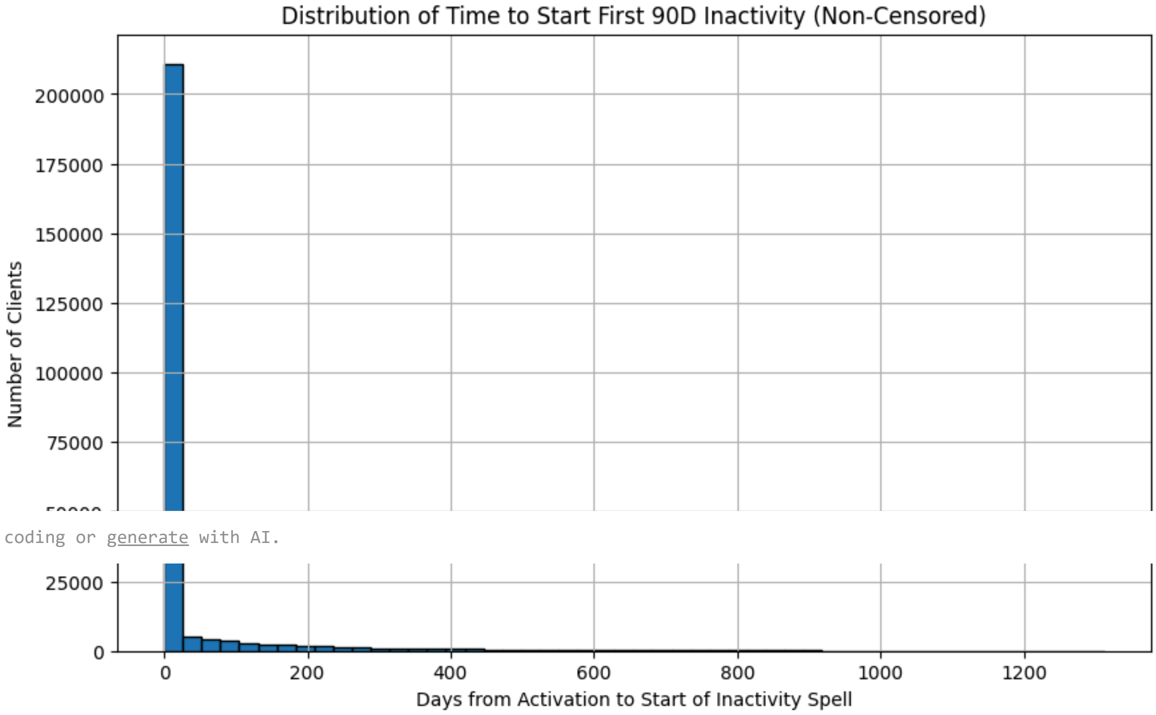
```
Number of Clients Censored for 365D Inactivity: 1127986 (85.68%)
  Mean Time to Start 365D Inactivity (for those who experienced it): 14.56 days
  Min Time to Start 365D Inactivity: 0.00 days
  Max Time to Start 365D Inactivity: 1296.00 days
  25th Percentile Time to Start 365D Inactivity: 0.00 days
  Median Time to Start 365D Inactivity: 0.00 days
  75th Percentile Time to Start 365D Inactivity: 0.00 days
  90th Percentile Time to Start 365D Inactivity: 0.00 days

--- Overall Summary Table ---
+-----------+---------------+------------------+------------------+---------------+------------+---------------+---------------+--
|Event_Count|Max_TimeToEvent|Mean_TimeToEvent  |Median_TimeToEvent|Min_TimeToEvent|N_Day_Window|P25_TimeToEvent|P75_TimeToEvent|P9
+-----------+---------------+------------------+------------------+---------------+------------+---------------+---------------+--
|261851     |1333           |32.24843517878488 |0.0               |0              |60          |0.0            |0.0            |74
|247789     |1312           |35.069405825117336|0.0               |0              |90          |0.0            |0.0            |81
```

```
# Conceptual Step for Visualization (more advanced plotting often done in Pandas/matplotlib after collecting data)
# For a quick look, we can plot histograms of Time_To_First_N_Day_Inactivity_Start for non-censored clients

import matplotlib.pyplot as plt
import pandas as pd

if 'final_client_survival_df' in locals():
    print("\n--- Visualizing Time to Event (for non-censored clients) ---")

    # Ensure it's persisted if not already, or re-evaluate if needed
    # final_client_survival_df.persist()

    n_day_windows_viz = [60, 90] # Let's visualize for shorter windows first to manage data size

    for n_days in n_day_windows_viz:
        time_col = f"Time_To_First_{n_days}D_Inactivity_Start"
        censored_col = f"Is_Censored_{n_days}D"

        # Collect data for non-censored clients for plotting
        # Be cautious with collect() on very large datasets. Sample if necessary.
        # For this exploration, if event_count is huge, consider sampling.

        event_data_df = final_client_survival_df.filter(col(censored_col) == False).select(time_col)

        # Limiting the amount of data brought to Pandas for histogram plotting
        # If event_data_df.count() is very large, this can be slow / cause OOM.
        # Consider sampling: event_data_pd = event_data_df.sample(fraction=0.1, seed=42).toPandas()

        print(f"Collecting data for histogram for {n_days}D window (non-censored)...")
        # Check count before collecting
        num_event_clients = event_data_df.count()
        if num_event_clients == 0:
            print(f"No non-censored clients for {n_days}D window to plot.")
            continue
        elif num_event_clients > 500000: # Arbitrary threshold for large data
            print(f"Warning: Collecting {num_event_clients} data points for {n_days}D. Sampling to 100k for plotting.")
            event_data_pd = event_data_df.sample(fraction=min(1.0, 100000.0/num_event_clients), seed=42).toPandas()
        else:
            event_data_pd = event_data_df.toPandas()

        if not event_data_pd.empty:
            plt.figure(figsize=(10, 6))
            plt.hist(event_data_pd[time_col], bins=50, edgecolor='black') # You might want to adjust bins or range
            plt.title(f'Distribution of Time to Start First {n_days}D Inactivity (Non-Censored)')
            plt.xlabel('Days from Activation to Start of Inactivity Spell')
            plt.ylabel('Number of Clients')
            plt.grid(True)
            plt.show()
        else:
            print(f"No data to plot for {n_days}D window (non-censored).")

    # final_client_survival_df.unpersist() # Already unpersisted at end of cell 11
else:
    print("Skipping Phase 4, Step 15 as final_client_survival_df is missing.")

# Stop Spark Session
spark.stop()
```

```
--- Visualizing Time to Event (for non-censored clients) ---
Collecting data for histogram for 60D window (non-censored)...
```




```
Collecting data for histogram for 90D window (non-censored)...
```



Start coding or generate with AI.