```
import pandas as pd
import numpy as np
from datetime import datetime
import os
from openpyxl import load_workbook
from openpyxl.styles import Font, Alignment, Border, Side, PatternFill
from copy import copy as copy_style
import warnings

# Suppress FutureWarning for groupby operations
warnings.filterwarnings('ignore', category=FutureWarning, module='pandas')

def create_zbm_hierarchical_reports():
    """
    Create separate ZBM reports showing ABM hierarchy with perfect tallies
    Each ZBM gets a report showing all ABMs under them
    """

    print("🔄 Starting ZBM Hierarchical Reports Creation...")

    # Read master tracker data from Excel file
    print("📖 Reading ZBM Automation Email 2410252.xlsx...")
    try:
        df = pd.read_excel('ZBM Automation Email 2410252.xlsx')
        print(f"✅ Successfully loaded {len(df)} records")
    except Exception as e:
        print(f"❌ Error reading file: {e}")
        return

    print(f"📋 Columns in file: {list(df.columns)}")

    # Clean and prepare data
    print("🧹 Cleaning and preparing data...")

    # Ensure required columns exist
    required_columns = ['ZBM Terr Code', 'ZBM Name', 'ZBM EMAIL_ID',
                        'ABM Terr Code', 'ABM Name', 'ABM EMAIL_ID',
                        'TBM HQ', 'TBM EMAIL_ID',
                        'Doctor: Customer Code', 'Assigned Request Ids', 'Request Status', 'Rto
Reason']
    missing = [c for c in required_columns if c not in df.columns]
    if missing:
        print(f"❌ Missing required columns: {missing}")
        return

    # **CRITICAL FIX 1**: Check for NULL/blank Request IDs
    print(f"📊 Total rows before cleaning: {len(df)}")
    print(f"   Rows with NULL Request IDs: {df['Assigned Request Ids'].isna().sum()}")
```

```
    print(f"   Rows with blank Request IDs: {(df['Assigned Request Ids'].astype(str).str.strip()
== '').sum()}")

    # Remove rows with NULL or blank Request IDs
    df = df[df['Assigned Request Ids'].notna()]
    df = df[df['Assigned Request Ids'].astype(str).str.strip() != '']
    print(f"📊 Rows after removing NULL/blank Request IDs: {len(df)}")

    # Remove rows where key fields are null or empty
    original_len = len(df)
    df = df.dropna(subset=['ZBM Terr Code', 'ZBM Name', 'ABM Terr Code', 'ABM Name', 'TBM HQ'])
    df = df[df['ZBM Terr Code'].astype(str).str.strip() != '']
    df = df[df['ABM Terr Code'].astype(str).str.strip() != '']
    df = df[df['TBM HQ'].astype(str).str.strip() != '']
    print(f"📊 Rows after cleaning ZBM/ABM/TBM fields: {len(df)} (removed {original_len -
len(df)})")

    # **CRITICAL FIX 2**: Standardize ZBM and ABM codes/names (remove extra spaces, case issues)
    df['ZBM Terr Code'] = df['ZBM Terr Code'].astype(str).str.strip().str.upper()
    df['ZBM Name'] = df['ZBM Name'].astype(str).str.strip()
    df['ABM Terr Code'] = df['ABM Terr Code'].astype(str).str.strip().str.upper()
    df['ABM Name'] = df['ABM Name'].astype(str).str.strip()
    df['ZBM EMAIL_ID'] = df['ZBM EMAIL_ID'].astype(str).str.strip().str.lower()
    df['ABM EMAIL_ID'] = df['ABM EMAIL_ID'].astype(str).str.strip().str.lower()

    print(f"📊 Unique ZBM codes in raw data: {df['ZBM Terr Code'].nunique()}")
    print(f"📊 Unique Request IDs in raw data: {df['Assigned Request Ids'].nunique()}")

    # Compute Final Answer per unique request id using rules from logic.xlsx
    print("🧠 Computing final status per unique Request Id using rules...")
    try:
        xls_rules = pd.ExcelFile('logic.xlsx')
        sheet2 = pd.read_excel(xls_rules, 'Sheet2')

        def normalize(text):
            return str(text).strip().casefold()

        rules = {}
        for _, row in sheet2.iterrows():
            statuses = [normalize(s) for s in row.drop('Final Answer').dropna().tolist()]
            statuses = tuple(sorted(set(statuses)))
            rules[statuses] = row['Final Answer']

        # Group statuses by request id from master data
        grouped = df.groupby('Assigned Request Ids')['Request Status'].apply(list).reset_index()

        def get_final_answer(status_list):
            key = tuple(sorted(set(normalize(s) for s in status_list)))
            return rules.get(key, '❌ No matching rule')
```

```
        grouped['Request Status'] = grouped['Request Status'].apply(lambda lst: sorted(set(lst),
key=str))
        grouped['Final Answer'] = grouped['Request Status'].apply(get_final_answer)

        # Merge Final Answer back to main dataframe
        df = df.merge(grouped[['Assigned Request Ids', 'Final Answer']], on='Assigned Request
Ids', how='left')

        # Check for unmapped requests
        unmapped_count = (df['Final Answer'] == '❌ No matching rule').sum()
        if unmapped_count > 0:
            print(f"⚠️ WARNING: {unmapped_count} rows have no matching rule in logic.xlsx")
            print(f"   Unique Request IDs with no rule: {df[df['Final Answer'] == '❌ No
matching rule']['Assigned Request Ids'].nunique()}")

    except Exception as e:
        print(f"❌ Error computing final status from logic.xlsx: {e}")
        return

    # **CRITICAL FIX 3**: Create deduplicated dataset at REQUEST ID + ZBM + ABM level
    # This handles cases where same Request ID appears under multiple ZBMs or ABMs (data quality
issue)
    print("🔧 Deduplicating data at Request ID + ZBM + ABM level...")

    # First, check if Request IDs span multiple ZBMs or ABMs (data quality check)
    request_zbm_count = df.groupby('Assigned Request Ids')['ZBM Terr Code'].nunique()
    multi_zbm_requests = request_zbm_count[request_zbm_count > 1]
    if len(multi_zbm_requests) > 0:
        print(f"⚠️ WARNING: {len(multi_zbm_requests)} Request IDs appear under multiple ZBMs!")
        print(f"   Examples: {list(multi_zbm_requests.head().index)}")

    request_abm_count = df.groupby('Assigned Request Ids')['ABM Terr Code'].nunique()
    multi_abm_requests = request_abm_count[request_abm_count > 1]
    if len(multi_abm_requests) > 0:
        print(f"⚠️ WARNING: {len(multi_abm_requests)} Request IDs appear under multiple ABMs!")
        print(f"   Examples: {list(multi_abm_requests.head().index)}")

    # **KEY FIX**: Deduplicate at Request ID + ZBM + ABM combination level
    # This ensures each request is counted once per ABM (which is correct for the hierarchy)
    df_dedup = df.groupby(['Assigned Request Ids', 'ZBM Terr Code', 'ABM Terr Code']).agg({
        'ZBM Name': 'first',
        'ZBM EMAIL_ID': 'first',
        'ABM Name': 'first',
        'ABM EMAIL_ID': 'first',
        'TBM HQ': 'first',
        'TBM EMAIL_ID': 'first',
        'Doctor: Customer Code': 'first',
        'Final Answer': 'first',
        'Rto Reason': 'first',
        'ABM HQ': 'first' if 'ABM HQ' in df.columns else lambda x: None
```

```
    }).reset_index()

    print(f"📊 Deduplicated from {len(df)} rows to {len(df_dedup)} unique (Request ID + ZBM +
ABM) combinations")

    # **CRITICAL FIX 4**: Get unique ZBMs correctly with proper email handling
    # Handle cases where same ZBM code has multiple emails or names (data quality)
    zbm_check = df_dedup.groupby('ZBM Terr Code').agg({
        'ZBM Name': 'nunique',
        'ZBM EMAIL_ID': 'nunique'
    })
    zbm_multi_name = zbm_check[zbm_check['ZBM Name'] > 1]
    zbm_multi_email = zbm_check[zbm_check['ZBM EMAIL_ID'] > 1]

    if len(zbm_multi_name) > 0:
        print(f"⚠️ WARNING: {len(zbm_multi_name)} ZBM codes have multiple names!")
        for zbm_code in zbm_multi_name.index[:5]:
            names = df_dedup[df_dedup['ZBM Terr Code'] == zbm_code]['ZBM Name'].unique()
            print(f"   {zbm_code}: {names}")

    if len(zbm_multi_email) > 0:
        print(f"⚠️ WARNING: {len(zbm_multi_email)} ZBM codes have multiple emails!")
        for zbm_code in zbm_multi_email.index[:5]:
            emails = df_dedup[df_dedup['ZBM Terr Code'] == zbm_code]['ZBM EMAIL_ID'].unique()
            print(f"   {zbm_code}: {emails}")

    # Get unique ZBMs using mode (most frequent) for name/email
    zbms = df_dedup.groupby('ZBM Terr Code').agg({
        'ZBM Name': lambda x: x.mode()[0] if len(x.mode()) > 0 else x.iloc[0],
        'ZBM EMAIL_ID': lambda x: x.mode()[0] if len(x.mode()) > 0 else x.iloc[0]
    }).reset_index().sort_values('ZBM Terr Code')

    print(f"📋 Found {len(zbms)} unique ZBMs (expected 140)")

    if len(zbms) != 140:
        print(f"⚠️ WARNING: Expected 140 ZBMs but found {len(zbms)}!")
        print(f"   Difference: {len(zbms) - 140}")

    # Debug: Show all ZBMs and their ABMs
    print("\n🔍 ZBM-ABM Mapping (first 10):")
    for idx, (_, zbm_row) in enumerate(zbms.head(10).iterrows()):
        zbm_code = zbm_row['ZBM Terr Code']
        zbm_name = zbm_row['ZBM Name']
        zbm_data_temp = df_dedup[df_dedup['ZBM Terr Code'] == zbm_code]
        abms_temp = zbm_data_temp[['ABM Terr Code', 'ABM Name']].drop_duplicates()
        requests_temp = zbm_data_temp['Assigned Request Ids'].nunique()
        print(f"   {idx+1}. {zbm_code} ({zbm_name}): {len(abms_temp)} ABMs, {requests_temp}
requests")

    # Create output directory
```

```
    timestamp = datetime.now().strftime('%Y%m%d')
    output_dir = f"ZBM_Reports_{timestamp}"
    os.makedirs(output_dir, exist_ok=True)
    print(f"📁 Created output directory: {output_dir}")

    # Process each ZBM
    file_count = 0
    for _, zbm_row in zbms.iterrows():
        zbm_code = zbm_row['ZBM Terr Code']
        zbm_name = zbm_row['ZBM Name']
        zbm_email = zbm_row['ZBM EMAIL_ID']

        print(f"\n🔄 Processing ZBM: {zbm_code} - {zbm_name}")

        # Filter data for this ZBM (using deduplicated data)
        zbm_data = df_dedup[df_dedup['ZBM Terr Code'] == zbm_code].copy()

        if len(zbm_data) == 0:
            print(f"⚠️ No data found for ZBM: {zbm_code}")
            continue

        # Get unique ABMs under this ZBM
        abms = zbm_data.groupby(['ABM Terr Code', 'ABM Name']).agg({
            'ABM EMAIL_ID': lambda x: x.mode()[0] if len(x.mode()) > 0 else x.iloc[0],
            'TBM HQ': 'first',
            'ABM HQ': 'first' if 'ABM HQ' in zbm_data.columns else lambda x: None
        }).reset_index()

        abms = abms.sort_values('ABM Terr Code')
        print(f"   📊 Found {len(abms)} ABMs under this ZBM")

        # Create summary data for this ZBM
        summary_data = []

        for _, abm_row in abms.iterrows():
            abm_code = abm_row['ABM Terr Code']
            abm_name = abm_row['ABM Name']
            abm_email = abm_row['ABM EMAIL_ID']
            tbm_hq = abm_row['TBM HQ']

            # Filter data for this specific ABM (using deduplicated data)
            abm_data = zbm_data[(zbm_data['ABM Terr Code'] == abm_code) & (zbm_data['ABM Name']
== abm_name)].copy()

            # Calculate metrics for this ABM using ORIGINAL df for TBM and HCP counts
            abm_original = df[(df['ZBM Terr Code'] == zbm_code) &
                              (df['ABM Terr Code'] == abm_code) &
                              (df['ABM Name'] == abm_name)]
            unique_tbms = abm_original['TBM EMAIL_ID'].nunique() if 'TBM EMAIL_ID' in
abm_original.columns else 0
```

```
            unique_hcps = abm_original['Doctor: Customer Code'].nunique()

            # All request counts use deduplicated data
            unique_requests = len(abm_data)

            # HO Section (A + B) - Count requests by Final Answer
            request_cancelled_out_of_stock = len(abm_data[abm_data['Final Answer'].isin(['Out of
stock', 'On hold', 'Not permitted'])])
            action_pending_at_ho = len(abm_data[abm_data['Final Answer'].isin(['Request Raised',
'Action pending / In Process At HO'])])

            # HUB Section (D + E)
            pending_for_invoicing = len(abm_data[abm_data['Final Answer'].isin(['Action pending
/ In Process At Hub'])])
            pending_for_dispatch = len(abm_data[abm_data['Final Answer'].isin(['Dispatch
Pending'])])

            # Delivery Status (G + H)
            delivered = len(abm_data[abm_data['Final Answer'].isin(['Delivered'])])
            dispatched_in_transit = len(abm_data[abm_data['Final Answer'].isin(['Dispatched & In
Transit'])])

            # RTO Reasons - Count requests with RTO reasons
            incomplete_address = len(abm_data[abm_data['Rto
Reason'].astype(str).str.contains('Incomplete Address', na=False, case=False)])
            doctor_non_contactable = len(abm_data[abm_data['Rto
Reason'].astype(str).str.contains('Dr. Non contactable', na=False, case=False)])
            doctor_refused_to_accept = len(abm_data[abm_data['Rto
Reason'].astype(str).str.contains('Doctor Refused to Accept', na=False, case=False)])

            # Calculate RTO as sum of RTO reasons
            rto_total = incomplete_address + doctor_non_contactable + doctor_refused_to_accept

            # Calculated fields using formulas
            requests_dispatched = delivered + dispatched_in_transit + rto_total  # F = G + H + I
            sent_to_hub = pending_for_invoicing + pending_for_dispatch + requests_dispatched  #
C = D + E + F
            requests_raised = request_cancelled_out_of_stock + action_pending_at_ho +
sent_to_hub  # Total = A + B + C
            hold_delivery = 0

            # Verify tally
            if requests_raised != unique_requests:
                print(f"      ⚠️ TALLY MISMATCH for {abm_code}: Calculated={requests_raised},
Actual={unique_requests}")
                print(f"         A={request_cancelled_out_of_stock}, B={action_pending_at_ho},
C={sent_to_hub}")
                print(f"         D={pending_for_invoicing}, E={pending_for_dispatch}, F=
{requests_dispatched}")
                print(f"         G={delivered}, H={dispatched_in_transit}, I={rto_total}")
```

```
                # Show which requests don't have matching Final Answer
                all_counted = (abm_data['Final Answer'].isin(['Out of stock', 'On hold', 'Not
permitted',
                                                              'Request Raised', 'Action pending
/ In Process At HO',
                                                              'Action pending / In Process At
Hub', 'Dispatch  Pending',
                                                              'Delivered', 'Dispatched & In
Transit']))
                uncounted = abm_data[~all_counted]
                if len(uncounted) > 0:
                    print(f"         Uncounted Final Answers: {uncounted['Final
Answer'].unique()}")

            # Create Area Name
            if 'ABM HQ' in abm_row and pd.notna(abm_row['ABM HQ']):
                abm_hq = abm_row['ABM HQ']
            else:
                abm_hq = tbm_hq
            area_name = f"{abm_code} - {abm_hq}"

            summary_data.append({
                'Area Name': area_name,
                'ABM Name': abm_name,
                'Unique TBMs': unique_tbms,
                'Unique HCPs': unique_hcps,
                'Requests Raised': requests_raised,
                'Request Cancelled Out of Stock': request_cancelled_out_of_stock,
                'Action Pending at HO': action_pending_at_ho,
                'Sent to HUB': sent_to_hub,
                'Pending for Invoicing': pending_for_invoicing,
                'Pending for Dispatch': pending_for_dispatch,
                'Requests Dispatched': requests_dispatched,
                'Delivered': delivered,
                'Dispatched In Transit': dispatched_in_transit,
                'RTO': rto_total,
                'Incomplete Address': incomplete_address,
                'Doctor Non Contactable': doctor_non_contactable,
                'Doctor Refused to Accept': doctor_refused_to_accept,
                'Hold Delivery': hold_delivery
            })

        # Create DataFrame for this ZBM
        zbm_summary_df = pd.DataFrame(summary_data)

        # Create Excel file for this ZBM
        create_zbm_excel_report(zbm_code, zbm_name, zbm_email, zbm_summary_df, output_dir)
        file_count += 1
```

```
    print(f"\n🎉 Successfully created {file_count} ZBM reports in directory: {output_dir}")
    print(f"📊 Expected 140 ZBMs, created {file_count} files")
    if file_count != 140:
        print(f"⚠️ WARNING: File count mismatch! Difference: {file_count - 140}")

def create_zbm_excel_report(zbm_code, zbm_name, zbm_email, summary_df, output_dir):
    """Create Excel report for a specific ZBM with perfect formatting"""

    try:
        # Load template
        wb = load_workbook('zbm_summary.xlsx')
        ws = wb['ZBM']

        def get_cell_value_handling_merged(row, col):
            """Get cell value even if it's part of a merged cell"""
            cell = ws.cell(row=row, column=col)

            # Check if this cell is part of a merged range
            for merged_range in ws.merged_cells.ranges:
                if cell.coordinate in merged_range:
                    # Get the top-left cell of the merged range
                    top_left_cell = ws.cell(row=merged_range.min_row,
column=merged_range.min_col)
                    return top_left_cell.value

            return cell.value

        # Search for header row
        header_row = None
        for row_idx in range(1, 15):
            for col_idx in range(1, min(30, ws.max_column + 1)):
                cell_value = get_cell_value_handling_merged(row_idx, col_idx)
                if cell_value and 'Area Name' in str(cell_value):
                    header_row = row_idx
                    break
            if header_row:
                break

        if header_row is None:
            header_row = 7

        data_start_row = header_row + 1

        # Read actual column positions from template header row
        column_mapping = {}
        for col_idx in range(1, min(30, ws.max_column + 1)):
            header_val = get_cell_value_handling_merged(header_row, col_idx)
            if header_val:
                header_str = str(header_val).strip()
```

```
                if 'Area Name' in header_str:
                    column_mapping['Area Name'] = col_idx
                elif 'ABM Name' in header_str:
                    column_mapping['ABM Name'] = col_idx
                elif 'Unique TBMs' in header_str or '# Unique TBMs' in header_str:
                    column_mapping['Unique TBMs'] = col_idx
                elif 'Unique HCPs' in header_str or '# Unique HCPs' in header_str:
                    column_mapping['Unique HCPs'] = col_idx
                elif 'Requests Raised' in header_str or '# Requests Raised' in header_str:
                    column_mapping['Requests Raised'] = col_idx
                elif 'Request Cancelled' in header_str or 'Out of Stock' in header_str:
                    column_mapping['Request Cancelled Out of Stock'] = col_idx
                elif 'Action pending' in header_str and 'HO' in header_str:
                    column_mapping['Action Pending at HO'] = col_idx
                elif 'Sent to HUB' in header_str:
                    column_mapping['Sent to HUB'] = col_idx
                elif 'Pending for Invoicing' in header_str:
                    column_mapping['Pending for Invoicing'] = col_idx
                elif 'Pending for Dispatch' in header_str:
                    column_mapping['Pending for Dispatch'] = col_idx
                elif 'Requests Dispatched' in header_str or '# Requests Dispatched' in
header_str:
                    column_mapping['Requests Dispatched'] = col_idx
                elif header_str == 'Delivered' or 'Delivered (G)' in header_str:
                    column_mapping['Delivered'] = col_idx
                elif 'Dispatched & In Transit' in header_str or 'Dispatched In Transit' in
header_str:
                    column_mapping['Dispatched In Transit'] = col_idx
                elif header_str == 'RTO' or 'RTO (I)' in header_str:
                    column_mapping['RTO'] = col_idx
                elif 'Incomplete Address' in header_str:
                    column_mapping['Incomplete Address'] = col_idx
                elif 'Doctor Non Contactable' in header_str or 'Dr. Non contactable' in
header_str:
                    column_mapping['Doctor Non Contactable'] = col_idx
                elif 'Doctor Refused' in header_str or 'Refused to Accept' in header_str:
                    column_mapping['Doctor Refused to Accept'] = col_idx
                elif 'Hold Delivery' in header_str:
                    column_mapping['Hold Delivery'] = col_idx

        # Clear existing data rows
        max_clear_rows = max(len(summary_df) + 10, 50)
        for r in range(data_start_row, data_start_row + max_clear_rows):
            for c in range(1, ws.max_column + 1):
                try:
                    cell = ws.cell(row=r, column=c)
                    cell.value = None
                except:
                    pass
```

```
    def copy_row_style(src_row_idx, dst_row_idx):
        """Copy formatting from source row to destination row"""
        for c in range(1, ws.max_column + 1):
            try:
                src = ws.cell(row=src_row_idx, column=c)
                dst = ws.cell(row=dst_row_idx, column=c)

                if src.font:
                    dst.font = copy_style(src.font)
                if src.alignment:
                    dst.alignment = copy_style(src.alignment)
                if src.border:
                    dst.border = copy_style(src.border)
                if src.fill:
                    dst.fill = copy_style(src.fill)
                dst.number_format = src.number_format
            except:
                pass

    # Write data rows
    template_data_row = data_start_row
    for i in range(len(summary_df)):
        target_row = data_start_row + i
        copy_row_style(template_data_row, target_row)

        for col_name, col_idx in column_mapping.items():
            if col_name in summary_df.columns:
                value = summary_df.iloc[i][col_name]

                try:
                    cell = ws.cell(row=target_row, column=col_idx)
                    cell.value = value

                    if isinstance(value, (int, float)) and not pd.isna(value):
                        cell.number_format = '0'
                except:
                    pass

    # Add total row
    total_row = data_start_row + len(summary_df)
    copy_row_style(template_data_row, total_row)

    if 'ABM Name' in column_mapping:
        try:
            cell = ws.cell(row=total_row, column=column_mapping['ABM Name'])
            cell.value = "Total"
            cell.font = Font(bold=True, name='Arial', size=10)
            cell.alignment = Alignment(horizontal='center', vertical='center')
        except:
            pass
```

```
        # Calculate and write totals
        for col_name, col_idx in column_mapping.items():
            if col_name in summary_df.columns and col_name not in ['Area Name', 'ABM Name']:
                total_value = int(summary_df[col_name].sum())

                try:
                    cell = ws.cell(row=total_row, column=col_idx)
                    cell.value = total_value
                    cell.font = Font(bold=True, name='Arial', size=10)
                    cell.alignment = Alignment(horizontal='center', vertical='center')
                    cell.number_format = '0'
                except:
                    pass

        # Save file
        safe_zbm_name = str(zbm_name).replace(' ', '_').replace('/', '_').replace('\\', '_')
        filename =
f"ZBM_Summary_{zbm_code}_{safe_zbm_name}_{datetime.now().strftime('%Y%m%d')}.xlsx"
        filepath = os.path.join(output_dir, filename)

        wb.save(filepath)
        print(f"   ✅ Created: {filename}")

    except Exception as e:
        print(f"   ❌ Error creating Excel report for {zbm_code}: {e}")
        import traceback
        traceback.print_exc()

if __name__ == "__main__":
    create_zbm_hierarchical_reports()
```