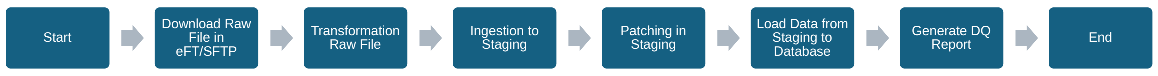
Full Process in eTRPlus

Main Process in eTRPlus



Download raw file in eft/sftp server

Automate file retrieval from an SFTP server based on a contributor list and organize the files into specific local directories, then clean up based on copy status.

Step-by-Step Summary:

1. Load Contributor Info
   - Read Excel file list_ref_comid.xlsx (sheet list_refcomid) into a DataFrame.

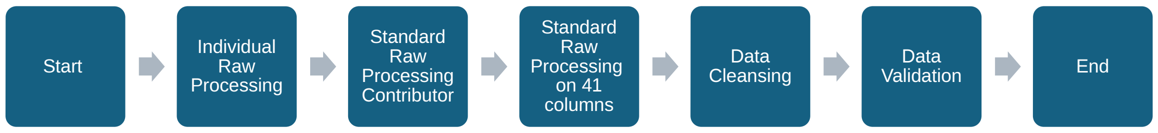
list_ref_comid (4).xlsx

   - Select and rename columns: ref_comid, email, prefix.
   - Generate each contributor's SFTP path: /eTR Plus/{email}.

2. Get Current Date Info
   - Generate current Kuala Lumpur time (today, year, year_month).
   - Used to build dynamic folder structures for local copies.

3. SFTP File Listing
    - Connect to SFTP server: eft.ctos.com.my:225.
    - Recursively list all files under each contributor's path.
    - Explode file list into individual rows, extracting file name and directory.
4. Path Construction
    - Create local destination paths:
        - talend_path: /talend_prod/.../{ref_comid}/{year}/{year_month}/{today}/{filename}
        - staging_eft_folder: /etrplus/.../{ref_comid}/raw/{year}/{year_month}/{filename}
5. Copy Files from SFTP
    - Function copy_talend_path_files_from_sftp:
        - Copy each file to the constructed talend_path.
    - Function copy_staging_eft_path_files_from_sftp:
        - Copy each file to staging_eft_folder.
    - Both update the DataFrame with copy status: 'ok' or 'fail'.
6. Cleanup Based on Status
    - Function delete_files_based_on_copy_status:
        - If either copy failed → delete local files.
        - If both copies succeeded → delete the source SFTP file.
7. Finish
    - Print "done" when complete.

<u>**Tranformation on Raw File Processing**</u>


Start
Individual Raw Processing
Standard Raw Processing Contributor
Standard Raw Processing on 41 columns
Data Cleansing
Data Validation
End

- For individual raw processing, each contributor has their own process. The details in excel below.

individual%20transformation.xlsx

Function: standard_raw_processing_contributor

🎯 Function Purpose:

To clean, standardize, and validate raw contributor data before integrating it into the fixed payment pipeline.

🧱 Key Processing Steps:

1. 💲 Convert Currency Columns
    - Converts dollar_columns to numeric types using pd.to_numeric.
2. 🧹 Clean Nulls and Apply Defaults
    - Replaces all standard nulls via replace_null_values.
    - Fills missing values with predefined contributor-specific defaults (patch_default_value_if_null).
3. 🗓️ Format Dates
    - Fixes agreement_status, lodgement_date, and others using:
        - replace_date_formating_agreement_status_lod
        - convert_date_to_correct_format
4. 🆔 Fix Registration Numbers
    - Applies customer and sponsor registration ID fixes:
        - ssm_customer_fixing_registeration_id
        - ssm_sponsor_fixing_registeration_id
5. 🛠️ Populate Key Fields
    - Fills missing op_code with 'A'.
    - Sets acc_status and del_reason_code conditionally based on op_code.
    - Ensures correct logic:
        - A and M op codes with invalid or missing acc_status → set to '1'.
    - Converts acc_status to string for consistency.
6. 📑 Reindex and Filter
    - Reorders columns to match fixed_payment_columns_list.
    - Filters out rows with:
        - Null seq

        - Invalid dates like '19700101'
        - Missing identifiers (old_ic, reg_no, passport_no)

7. 📱 Clean Mobile Numbers
    - Separates missing and non-missing mobile_no.
    - Cleans format of present mobile_no values via clean_mobile_numbers.
    - Handles multiple numbers (via .explode()).
8. 🔗 Combine Cleaned Data
    - Concatenates cleaned and missing mobile number data into a final DataFrame.
9. 📤 Returns the fully processed DataFrame.

) 📄 Function: Standard raw processing 41 columns

🔧 Function Purpose:

The function standard_raw_processing_41_columns processes and cleans raw contributor data with 41 columns for fixed payment (FP) ingestion. It handles data standardization, classification, fuzzy matching for SSM validation, deduplication, formatting, and exports cleaned and error datasets along with summary statistics.

🧼 Main Processing Steps:

1. Data Standardization:

- Converts strings to uppercase.
- Removes special characters (excluding email).
- Replaces null values.

2. Party Type Classification:

- Splits data into:
  - df_I: Individual
  - df_C: Company (ROC)
  - df_B: Business (ROB)

3. SSM Fuzzy Matching (Parallel Execution):

- Uses fuzzy_match_ssm_roc for companies (ROC)
- Uses fuzzy_match_ssm_rob for businesses (ROB)
- Uses ThreadPoolExecutor for parallel processing
- Returns:
  - Cleaned DataFrames (df_C_new, df_B_new)
  - Rejected rows (reject_C, reject_B)

4. SSM Cross-Check for Individuals:

- Validates df_I against SSM data.

5. Merging & Deduplication:

- Combines df_I, df_B_new, df_C_new
- Removes duplicates using checksum priority

6. Further Cleaning:

- Trims whitespace from ID columns
- Fixes null arrears/balance values
- Formats dollar columns
- Converts data types to str where necessary
- Replaces null again for consistency

7. SSM Fixing (Customer & Sponsor):

    1. Applies functions to correct registration ID issues.

8. Validation & Error Segregation:

- Applies mandatory field validation
- Returns:
    - final_clean_df: Valid rows
    - full_list_errors: All error rows
    - unique_list_errors: Unique errors only

9. Statistics Collection:

Creates summary including:

- Total raw rows
- Rows after deduplication
- Clean/error counts
- Rejected SSM records

10. File Saving (Parallel):

Saves:

- Clean data
- Error data
- Statistics
- SSM rejects

🧾 Outputs:

Returns:

- final_clean_df: Fully cleaned DataFrame
- full_list_errors: All rows with validation errors

) 📄 Function: Data Cleansing

📌 Function Purpose:

To perform comprehensive data cleansing on a DataFrame (df) specific to fixed payment data for the eTR Plus system.

🧼 Key Cleansing Steps Performed:

1. Reordering Columns:
   - Reorders DataFrame to match the required schema (fixed_payment_columns_list).
2. Data Type and Value Conversion:
   - Converts columns to appropriate types (string, date, dollar, number).
   - Replaces empty strings with appropriate defaults based on column type.
3. Enhancements and Formatting:
   - Adds a sequential column for row tracking (seq).
   - Removes excess whitespace and special characters (excluding emails).
   - Applies consistent casing (upper/lower) for specific columns.
   - Converts float values stored as strings to clean string representations.
   - Converts float-like arrears values to integer-style strings.
4. Null and Missing Value Handling:
   - Fills nulls with defaults.
   - Replaces nulls in specific sponsor-related columns with domain-specific values.
   - Removes leading, trailing, and all whitespaces from key columns.
5. Status and Duplicate Cleanup:
   - Updates account status and operation codes based on specific rules.
   - Drops duplicate rows based on the sequential column (seq), keeping the last one.

6. Date Formatting:
    - Converts datetime fields into YYYYMMDD string format.
7. Registration Number Formatting:
    - Formats reg_no for individuals.
    - Formats new_ic_no for sponsors.

) 📄 Function: Data Validation

📌 Function Purpose:

To validate raw, cleansed data in a structured and modular way—checking for rule violations, data integrity issues, and compliance with formatting and business requirements.

🔍 Validation Workflow Overview:

1. Split Data by Operation Type:
   - df_sponsor, df_add, df_delete — separates the dataset based on the type of transaction for more targeted validation.
2. Mandatory Field Checks:
   - Ensures required fields are not missing:
     - Sponsor-related (validate_sponsor_mandatory_value)
     - Opcode field
     - Add and Delete operations
3. Dictionary Value Validation:
   - Verifies if column values are allowed/expected based on predefined dictionaries (value_validation_dictionary).
4. Date Validations:
   - Checks date differences and logical ordering:
     - Between statement date and relationship start (validate_diff_stmnt_date_relationship)
     - Between statement date and agreement date (validate_diff_stmnt_date_agreement)
     - Ensures future dates are not used (compare_dates_today_dates_columns)
5. Amount and Format Checks:
   - Validates:
     - Instalment amount (validate_instalment_amt)
     - Maximum character lengths
     - Alphanumeric/numeric constraints

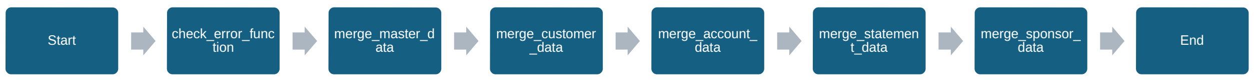
        - IC formats for sponsor and individual
        - Date format consistency

6. Compulsory Custom Rules:
    - Applies critical business rules via apply_compulsory_checks.
7. Error Consolidation:
    - Merges all validation error results into one DataFrame (df_combined_errors).
8. Unique Error Filtering:
    - Drops duplicates based on unique row identifier seq.
    - Retains only one error per row for clean-up filtering (df_unique_errors).
9. Clean Valid Rows:
    - Filters out rows with any validation error using filter_clean_unique.

✅ Returns:

- df: The filtered DataFrame with only valid (clean) records.
- df_combined_errors: A comprehensive list of all validation errors found.
- df_unique_errors: A deduplicated version used for clean filtering.

Ingestion to Staging



Check Error Function

✅ **Purpose:**

To validate each row of a DataFrame for required fields and data rules for eTR+ submissions. If any validation fails, it assigns an appropriate error message in the Error column.

🔍 **Validation Categories:**

1. **Operation Code**:
   - Must be one of 'A', 'M', or 'D'.
   - Mandatory for eTR+.
2. **Party Type & Name**:
   - Must be one of 'I', 'B', 'C'.
   - Name must be provided and ≤ 100 characters.
3. **Identification Fields** (At least one required):
   - Old IC No./ Business Reg No (≤ 20 chars, alphanumeric).
   - New IC No./ Company Reg No (≤ 20 chars, alphanumeric).
   - Passport No (≤ 20 chars, alphanumeric).
4. **Account Information**:
   - Account No is mandatory, ≤ 30 characters.
   - Account Status must be '1', '2', '3', or '4'.
5. **Date Fields**:
   - Relationship Start Date, Agreement Date, and Statement Date must be in 'yyyyMMdd' format.
   - Required when Op Code is 'A' or 'M'.
6. **Relationship & Role**:
   - Relationship Type: Must be from '1' to '7'.
   - Capacity: Must be '1' or '2'.
   - Facility: Must be from '1' to '5'.
7. **Financial Fields** (Required for 'A'/'M'):
   - Limit, Instalment Amount, Amount In Arrears, Month In Arrears, Tenure Month, Total Balance Outstanding, Principal Repayment Term, Collateral Type
   - Must be numeric and positive where applicable.
8. **Optional Numeric Fields**:
   - Late Payment Interest, Tenure Day must be numeric and positive (if provided).

Merge Master Data

- Function Purpose:
  - To merge new account-related records (input_data) into an existing master dataset (data), ensuring:
    - Keys are correctly managed.
    - Duplicates are avoided.
    - New unique entries receive fresh tref_id_plus values.
- Key Steps:
  - Data Preparation:
    - Convert relevant columns in both data and input_data to appropriate types (str or int).
    - Ensure column order in input_data matches the desired structure.

- Master Key Generation:
  1. Call generate_master_keys() to populate keys in new_data.

- Existing Key Matching:
  1. Match 1: Rows where both account_ref_comid_key and customer_key already exist in data.
  2. Match 2: Rows where only customer_key exists (possibly with different account_ref_comid_key).
- New Key Identification:
  1. Identify truly new customer_key values not already matched.
  2. Assign new tref_id_plus values sequentially starting from the current max.
- Master Key Consolidation:
  1. Combine matched and newly assigned records into one unified master_key.
- Merge and Finalize:
  1. Merge new_data with master_key to get corresponding tref_id_plus.
  2. Concatenate new_data to data.
  3. Drop duplicate account_ref_comid_key entries to maintain uniqueness.
  4. Reorder columns to match the desired structure.
- Performance Logging:
  1. Print the final number of rows and processing time.

Returns:

The updated data DataFrame with merged and deduplicated records.

Mapping Logic	Description
data['Account No'] ⇄ input_data['Account No']	Ensures account numbers are strings.
customer_key + account_ref_comid_key ⇄ tref_id_plus	This pair uniquely identifies a record in the master. If it exists, reuse the existing tref_id_plus.
customer_key only match	If only customer_key exists but not the full account_ref_comid_key, create a mapping using both account_ref_comid_key_x and _y versions and assign the existing tref_id_plus.
New customer_key (not in master)	Assigns a new tref_id_plus using the next max value from the existing master.
master_key ⇄ new_data	The generated or existing mappings are merged back into new_data to assign tref_id_plus.

Merge Customer Data

- Function Purpose:

To prepare and deduplicate customer-level data from the full master_data DataFrame for downstream use (e.g., customer master records).

- Key Steps:
- Start Timer:
    - Measure how long the function takes to run for performance monitoring.
- Select Relevant Columns:
    - Extract only customer-related fields from master_data.
- Reorder Columns:
    - Reorder the selected columns to match the desired format for consistency.
- Sort & Deduplicate:
    - Sort data by tref_id_plus (ascending).
    - Drop duplicate records based on tref_id_plus, keeping only the first occurrence.
- Clean Up Values:
    - Apply replace_values_with_nan() to clean or normalize invalid/missing data fields.
- Logging:
    - Print the number of rows and processing time.
- Returns:

  A cleaned and deduplicated customer-level DataFrame with standardized structure.

Merge Account Data

- Function Purpose:

To process, merge, deduplicate, and clean account-level data from multiple sources, specifically handling account additions and deletions for a given ref_comid.

- Key Steps and Logic:

1. Start Timer:
    - Captures start time to measure total processing duration.
2. Prepare New Input Data:
    - Select relevant columns from input_data.
    - Create a composite key account_ref_comid_key by combining Account No and ref_comid.
3. Merge With Master:
    - Join new input data with master_data to bring in customer_key and tref_id_plus.
    - Create account_key for more granular uniqueness (based on account + transaction).
4. Sort and Deduplicate:
    - Sort new data by Statement Date.
    - Drop duplicates based on account_key, keeping the latest record.
5. Split Into Add and Delete Records:
    - Filter deletion_status == '0' for additions.
    - Filter deletion_status == '1' for deletions.
6. Ensure Column Alignment:
    - Select and align columns in both new and existing add and delete datasets.
7. Type Conversion:
    - Convert all datasets to str type for consistency.
8. Concatenate and Final Deduplication:
    - Combine old and new add/delete data.
    - Deduplicate using account_ref_comid_key and customer_key, keeping latest.

9. Final Merge and Cleanup:
    - Merge both add and delete datasets.
    - Final deduplication based on account_ref_comid_key and customer_key.
    - Clean data using replace_values_with_nan().
10. Print Logs:
    - Output the number of rows processed and total duration.

Merge Statement Data

🔧 Function Purpose: merge_statement_data

Processes and merges fixed payment statement data (additions and deletions), handles duplicates, applies transformations in parallel, and returns a consolidated DataFrame.

🧠 Key Processing Steps:

1. Initial Processing:
    - Calls new_data_statement_processing() to separate raw input into:
        - new_data_add
        - new_data_delete
2. Parallel Transformation:
    - Uses ThreadPoolExecutor to speed up processing of 4 datasets (data_delete, new_data_add, data_add, new_data_delete) in parallel using:
        - parallel_process_dataframe() on specified desired_columns.
3. Filter Deletions:
    - Filters data_delete where "Payment Trend Operation Code" == 'D'.
4. Process Additions:
    - Concatenates data_add + new_data_add.
    - Filters to only 'A' (Add) and 'M' (Modify) operation codes.
    - Removes duplicates by statement_key, keeping the latest.
    - Writes to temp file STATEMENT_ADD_FILE_TEMP.
5. Process Deletions:
    - Concatenates data_delete + new_data_delete.
    - Filters to 'D' (Delete) operation code only.
    - Removes duplicates by statement_key, keeping the latest.
    - Writes to temp file STATEMENT_DELETE_FILE_TEMP.
6. Final Concatenation:
    - Merges processed data_add and data_delete into a full data DataFrame.

   - Merges new_data_add and new_data_delete into a full new_data DataFrame.
   - Deduplicates both by statement_key.
7. Append & Reorder:
   - Combines data and new_data using append_and_reorder().
   - Deduplicates the final output based on statement_key.

📊 Output:

- Returns the final merged and deduplicated DataFrame data.

Merge Sponsor Data

Function Purpose: merge_sponsor_data

Merges new sponsor information with existing data, assigns unique sp_id values for new sponsors, and returns an updated sponsor dataset.

🧠 Key Processing Steps:

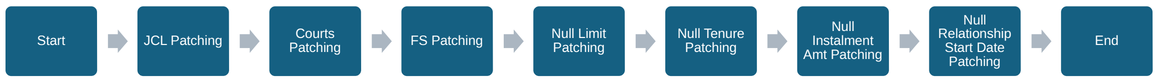
1. Extract Relevant Columns:
    - Pulls sponsor-related fields from input_data and key identifiers from master_data.
2. Generate Keys:
    - Constructs account_ref_comid_key by combining Account No and ref_comid.
    - Merges to obtain tref_id_plus.
    - Creates a unique sponsor_key for each record using:
        - tref_id_plus, Account No, Sponsor Constitution, Old IC/Reg No, New IC/Reg No, Passport No.
3. Deduplicate:
    - Drops duplicate sponsor records based on sponsor_key, keeping the latest.
4. Split Records:
    - Identifies:
        - Existing sponsors: Found in current data.
        - New sponsors: Not found in current data.
5. Assign sp_id:
    - Existing sponsors inherit sp_id from current data.
    - New sponsors are assigned new sequential sp_id values starting from the max of existing ones.
6. Concatenate & Clean:
    - Merges existing and new sponsor data.
    - Keeps only the latest entries for each sponsor_key.
    - Ensures consistent data types (sp_id as int, tref_id_plus as str).
7. Reorder & Clean Columns:

    - Ensures final DataFrame has standardized column order.
    - Applies replace_values_with_nan() for data sanitization.

📊 Outputs:

- Returns the cleaned and updated sponsor DataFrame.

Patching In Staging



Patching for Courts (account level)

📝 Script Purpose

This script patches missing or incorrect 'Relationship Start Date' values in an account-level staging Parquet file by referencing corrected values from an Excel patching file, then updates the Parquet file.

🧠 Main Steps:

1. Import Libraries & Set Up:
    - Imports pandas, numpy, and a logging module.
    - Adds a custom path to sys.path to allow importing from a global project directory.
2. Load Excel Patch File:
    - Loads the Excel file into df_account (with string data types).
    - Adds a fixed ref_comid value 'E243500'.
    - Converts 'Relationship Start Date' to a new formatted string column 'new_date'.
    - Creates a key column as a combination of ref_comid + '_' + Account No.
3. Load Account Parquet File:
    - Reads the existing Parquet file into account.
    - Adds the same key column for merging.
4. Merge and Split:
    - Merges account with the patching file using the key column.
    - Splits merged data into:
        - df_clean: rows with no patch (new_date is NaN)
        - df_dirty: rows with available patch (new_date is not NaN)
    - Cleans up df_dirty:
        - Drops original 'Relationship Start Date'
        - Renames 'new_date' → 'Relationship Start Date'
        - Sets 'Agreement Date' to the new 'Relationship Start Date'

5. Recombine and Finalize:
    - Concatenates clean and updated dirty records.
    - Drops helper columns like 'key' and 'new_date'.
    - Removes duplicates by account_key, keeping the last.
6. Save and Finish:
    - Overwrites the original ACCOUNT_FILE Parquet with the patched new_account data.
    - Prints 'done'.

📤 Output:

- Updated ACCOUNT_FILE Parquet file with corrected 'Relationship Start Date'.

✅ Key Outcome:

Fixes outdated or missing start dates in account records by patching from a known, reliable source.

Patching for Courts (statement level)

📝 Script Purpose

This script patches multiple columns (Limit, Tenure Month, and Instalment Amount) in a Parquet-based statement file by using updated values from an Excel patching file. The updated data is merged and used to overwrite outdated values, then the cleaned result is saved back to the same Parquet file.

🧠 Main Steps:

1. Setup and Load Patching Data:
    - Imports necessary libraries.
    - Loads the patching Excel file (df).
    - Adds a constant ref_comid = 'E243500'.
    - Selects and renames key columns to:
        - 'Limit' → 'new_limit'
        - 'Instalment Amount' → 'new_instalment_amount'
        - 'Tenure' → 'new_tenure_month'
    - Creates a key column using Account No + '_' + ref_comid.
2. Patch Limit:
    - Loads the Parquet statement file.
    - Adds a similar key column for matching.
    - Merges the statement with new_limit values.
    - Splits into:
        - df_clean: rows without patches.
        - df_dirty: rows with updated Limit.
    - Drops old Limit and replaces it with the new one.
    - Recombines, removes duplicates, and prepares for next patching step.
3. Patch Tenure Month:
    - Repeats similar logic using new_tenure_month.
    - Replaces old Tenure Month with new value if available.
4. Patch Instalment Amount:
    - Final patching step using new_instalment_amount.

    - Updates and cleans the final dataset.

5. Save Output:
    - Drops helper columns like key and temporary patch columns.
    - Saves the updated DataFrame back to STATEMENT_FILE.

📤 Output:

- An updated Parquet file (STATEMENT_FILE) with corrected values for:
    - Limit
    - Tenure Month
    - Instalment Amount

✅ Key Outcome:

This script ensures that any known corrections from the Excel patch file are applied to the statement data, keeping only the most recent and accurate version of each record based on statement_key.

Patching for JCL(Amend acc_status = 4 due to change multiple limit)

⚙️ Helper Functions

1. check_limit_changes(df)

- Identifies if the account Limit changes over time.
- Adds a new column limit_remarks:
  - 'change' if a limit change is detected,
  - 'no_change' otherwise.

2. update_account_status(df)

- For rows where limit_remarks == 'change' and Account Status == '1', changes Account Status to '4' from the change date onwards.

3. change_account_status(df)

- Converts all Account Status values from '1' to '4' (used for accounts data).

4. patch_change_limit(df_list_account, statements, accounts)

- Filters both statements and accounts based on the list of affected accounts.
- Applies check_limit_changes and update_account_status on statements.
- Applies change_account_status on accounts.
- Saves updated results to temporary Parquet files:
  - statement_status.pq
  - account_status.pq

📁 Main Execution Workflow

🔄 For each of the 3 dataset types:

1. STATEMENT_FILE & ACCOUNT_FILE
2. STATEMENT_ADD_FILE & ACCOUNT_ADD_FILE
3. STATEMENT_DELETE_FILE & ACCOUNT_DELETE_FILE

The following steps are performed:

- Load relevant statements/accounts Parquet files.
- Call patch_change_limit() to generate updated account statuses.
- Merge updated status into the original datasets.

- Drop duplicate records by statement_key or account_key, keeping the latest.
- Convert Account Status to string.
- Save the cleaned and updated DataFrames back to their respective Parquet files.

📎 Input

- df_list_account: List of accounts with inconsistent limits loaded from CSV.

✅ Output

- Updated Parquet files for:
    - Main statements and accounts
    - "Add" and "Delete" statements and accounts

🧠 Summary

This script processes financial account data to detect and patch changes in credit limits. If a limit changes:

- The corresponding Account Status in statements is updated conditionally.
- The Account Status in the master account list is also updated accordingly.
- Final datasets are merged, deduplicated, and written back to disk.

Patching for FS (instalment amount)

📝 Script Purpose

This script patches column (Instalment Amount) in a Parquet-based statement file by using updated values from an Excel patching file. The updated data is merged and used to overwrite outdated values, then the cleaned result is saved back to the same Parquet file.

🧠 Main Steps:

1. Setup and Load Patching Data:
    1. Setup and Load Patching Data:
    2. Imports necessary libraries.
    3. Loads the patching Excel file (df).
    4. Adds a constant ref_comid = 'E243500'.
    5. Selects and renames key columns to:
        a. 'Instalment Amount' → 'new_instalment_amount'
    6. Creates a key column using Account No + '_' + ref_comid.
2. Patch Instalment Amount:
    1. Final patching step using new_instalment_amount.
    2. Updates and cleans the final dataset.
3. Save Output:
    1. Drops helper columns like key and temporary patch columns.
    2. Saves the updated DataFrame back to STATEMENT_FILE.

📤 Output:

- An updated Parquet file (STATEMENT_FILE) with corrected values for:
    - Instalment Amount

✅ Key Outcome:

This script ensures that any known corrections from the Excel patch file are applied to the statement data, keeping only the most recent and accurate version of each record based on statement_key.

Null Patching Script on Relationship Start Date

🔧 Function Purpose: null_relationship_start_date_account_vs_fixed_template

Cleans and fills missing Relationship Start Date values in an account DataFrame using reference data from a fixed template file. The cleaned DataFrame is saved back to the original file.

🧠 Key Steps:

1. Read Account Data:
    - Loads the Parquet file into account_df.
    - Extracts base ref_comid from full_ref_comid (e.g., CTOS from CTOS_ABC).
2. Identify Dirty Records:
    - Filters account_df to find rows with:
        - ref_comid matching base ID.
        - Relationship Start Date is null → stored in account_df_refcomid_dirty.
3. Skip If Clean:
    - If no dirty rows exist (i.e., no missing start dates), returns the original DataFrame.
4. Prepare Clean Records:
    - Remaining records (non-matching ref_comid or non-null start dates) go into account_df_refcomid_clean.
5. Read Fixed Template Data:
    - Loads the corresponding fixed template file from a report directory.
    - Parses and formats relation_start_date to string-formatted Relationship Start Date.
    - Retains only the latest entry per acc_no.
6. Merge to Fill Missing Dates:
    - Left merges account_df_refcomid_dirty with the template based on Account No.
    - Drops the original null column and renames the filled one.
7. Fill Agreement Date if Missing:
    - If Agreement Date is missing, it's filled with the Relationship Start Date.

8. Recombine and Save:
    - Concatenates the clean and newly corrected dirty DataFrames.
    - Saves the updated DataFrame back to the original file path.

Null Patching Script on

🔧 Function Purpose: null_statement_limit_vs_fixed_template

Cleans and fills invalid or missing Limit values in a statement-level DataFrame using data from a reference fixed template. Saves the corrected DataFrame back to the original file.

🧠 Key Steps:

1. Load Data:
    - Reads the statement data from a Parquet file.
    - Extracts base ref_comid (e.g., CTOS from CTOS_ABC).
2. Identify Dirty Records:
    - Filters records with matching ref_comid and Limit values that are:
        - '0', '0.0', or NaN.
3. Skip If Clean:
    - If there are no such dirty records, returns the original DataFrame immediately.
4. Filter Out Clean Records:
    - Removes dirty records from the main DataFrame and stores them separately as statement_df_refcomid_clean.
5. Load Fixed Template Data:
    - Loads a reference fixed template statement file.
    - Extracts Account No, Limit, and Statement Date.
    - Converts Limit to string (after forcing it through float → int → str).
    - Filters out rows where Limit == '0'.
    - Keeps only the most recent record per Account No.
6. Merge to Fill Missing Limits:
    - Left joins the dirty records with the fixed template on Account No.
    - Drops old Limit and replaces it with the newly joined value.
7. Recombine and Save:
    - Merges clean and corrected dirty records.
    - Saves the updated DataFrame back to the original Parquet file.

📤 Output:

- Updated and cleaned statement DataFrame written to the original file location.

🔧 Function Purpose: null_statement_instalment_amt_vs_fixed_template

Cleans and fills missing or zero 'Instalment Amount' values in a statement-level DataFrame using a trusted fixed template, then updates the original file.

🧠 Key Steps:

1. Load Data:
    - Reads the statement Parquet file into statement_df.
    - Extracts base ref_comid (e.g., CTOS from CTOS_ABC).
2. Identify Dirty Records:
    - Filters rows where:
        - ref_comid matches the extracted one AND
        - 'Instalment Amount' is '0', '0.0', or NaN.
3. Early Exit If Clean:
    - If no such dirty records exist, returns the original DataFrame immediately.
4. Filter Clean Records:
    - Removes the dirty rows from the original dataset to isolate clean rows.
5. Load Fixed Template Data:
    - Loads trusted statement data from a template.
    - Extracts and converts:
        - acc_no → 'Account No'
        - instalment_amt → 'Instalment Amount' as string
        - stmnt_date → 'Statement Date' in datetime format.
    - Removes rows with 'Instalment Amount' equal to '0'.
    - Keeps the most recent record for each 'Account No'.
6. Merge Fixed Data into Dirty Records:
    - Merges the dirty records with cleaned template data by 'Account No'.
    - Drops the original 'Instalment Amount' and replaces it with the new one from the template.

    - Fills missing values with '0'.

7. Recombine and Save:
    - Concatenates the cleaned original and updated dirty records.
    - Saves the updated DataFrame back to the original Parquet file.

📤 Output:

- Overwrites the original Parquet file with cleaned statement data.

🔧 Function Purpose: null_statement_tenure_vs_fixed_template

Cleans and fills missing or zero 'Tenure Month' values in a statement-level DataFrame using trusted data from a fixed template, then updates the original file.

🧠 Key Steps:

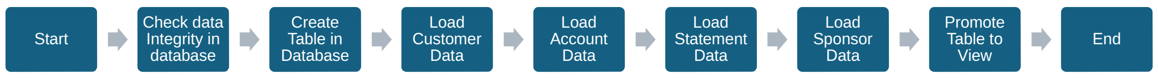
1. Load Data:
    - Reads the statement-level data from the provided Parquet file into statement_df.
    - Extracts the base ref_comid from full_ref_comid (e.g., 'CTOS' from 'CTOS_ABC').
2. Filter Dirty Records:
    - Selects rows where:
        - ref_comid matches the extracted one AND
        - 'Tenure Month' is '0', '0.0', or NaN.
3. Exit Early if Clean:
    - If no dirty records are found, returns the original statement_df.
4. Filter Clean Records:
    - Removes the dirty records from statement_df, creating a clean subset.
5. Load and Prepare Fixed Template Data:
    - Loads fixed template statement data.
    - Prepares relevant columns:
        - acc_no → 'Account No'
        - tenure_month → 'Tenure Month' (converted to string after filling blanks/NaNs and converting to integer)
        - stmnt_date → 'Statement Date' as datetime.
    - Filters out rows where 'Tenure Month' is '0'.
    - Keeps the latest 'Tenure Month' per 'Account No' based on the most recent 'Statement Date'.
6. Merge Template into Dirty Records:

   - Joins the dirty records with cleaned template data by 'Account No'.
   - Drops old 'Tenure Month', replaces with updated value from the template.
   - Ensures the columns match the original expected layout.
7. Recombine and Save:
   - Merges clean and updated dirty records.
   - Overwrites the original Parquet file with the updated statement_df.

📤 Output:

- Overwrites the input Parquet file with a cleaned version of the statement-level data.

Load Data to Staging



Check Data Integrity

Purpose

The script performs data integrity checks by ensuring that certain values in the application data (customer_data, statement_data, account_data, sponsor_data) exist in their corresponding reference tables in a database.

Key Components

1. Helper Functions

- format_values(list):
    - Formats a list of values into SQL-compatible string tuples for querying.
- check_integrity(table, fields, values):
    - Filters out NaN and None values.
    - Constructs a SQL SELECT DISTINCT query to find existing records.
    - Compares expected values against actual database results.
    - Prints an error and raises an Exception if any values are missing.

Integrity Checks Performed

1. ref_id (from customer_data)
   → Validated against referee.ref_id
2. Collateral Type (from statement_data)
   → Validated against collateral_type.collateral_type_code
3. Facility (from statement_data)
   → Validated against facility.facility_code
4. Principal Repayment Term (from statement_data)
   → Validated against principal_repayment_term.principal_repayment_term_code
5. ref_comid (from customer_data)
   → Validated against ref_com.ref_comid
6. Pair of ref_comid and Relationship Type (from account_data)
   → Validated against rel_type(ref_comid, rel_type_code)
7. Debt Type (from account_data)
   → Validated against debt_type.debt_type_code
8. Deletion Reason Code (from account_data)
   → Validated against deletion_reason.deletion_reason_code

9. Sponsor Status (from sponsor_data)
   → Validated against sponsor_status.sp_type_code

EER Diagram for eTRPlus MYSQL.

etrplus_EER_diagram.png

Create table in etrplus database

Purpose

To create timestamped tables in the etrplus database using predefined SQL template files.

Key Components

1. create_table Function

def create_table(table, cursor, LOAD_TIME_STR):

- Constructs a new table name by appending the LOAD_TIME_STR (usually a timestamp).
- Reads a corresponding SQL template file from the sql_jcl_41_columns directory, e.g.:

sql_jcl_41_columns/create_table_keytbl_jcl_uat.tpl

- Replaces placeholders in the SQL file:
    - %TABLE_NAME% → actual table name (with timestamp)
    - %LOAD_TIME_STR% → timestamp string
- Executes the final SQL command using the given cursor.

2. Tables Created

Using create_table, the script creates the following tables (with timestamp suffix):

- keytbl_jcl_uat
- relationship_jcl_uat
- acc_stts_jcl_uat
- sponsor_jcl_uat

Summary

This script dynamically creates multiple versioned or time-tagged tables in the etrplus database using templated SQL scripts. It helps in managing data loads by organizing tables with a consistent naming pattern based on the load time.

Load Customer Data to MYSQL Table

✅ Purpose

To clean, transform, and batch-insert customer_data into a MySQL table using SQLAlchemy.

🔧 Key Steps

1. Database Connection

db = mysql.connect(...)

- Connects to a MySQL database using mysql.connector.
- Enables autocommit and initializes a buffered cursor.
- Logs a confirmation message when the connection is successful.

2. Data Cleaning & Transformation

- Replaces invalid/missing values (e.g., 'nan', 'None', np.nan) with empty strings.
- Standardizes the 'Capacity' column:
  - '1' → 'OWN'
  - '2' → 'JOINT OWNERS'
  - Otherwise → None
- Drops the customer_key column.
- Renames selected columns to match the target database schema.

3. MySQL Table Name

table_name = 'keytbl_jcl_uat_{}'.format(LOAD_TIME_STR)

- Sets a timestamped table name using LOAD_TIME_STR.

5. SQLAlchemy Setup

engine = create_engine(...)

- Uses SQLAlchemy for efficient bulk inserts.
- Constructs the MySQL connection string (with encoded password).

6. Batch Insert Using Chunking

chunk_size = 100000

- Breaks customer_data into chunks of 100,000 rows to avoid memory or timeout issues.
- Inserts each chunk into the database using to_sql(if_exists='append').
- Logs progress after each chunk.

7. Exception Handling

try ... except ... finally

- Catches and logs any exception that occurs during insertion.
- Ensures the end log is printed even if an error occurs.

Load Account Data to MYSQL Table

✅ **Purpose**

To **clean, transform, and insert** account data from a Parquet file (ACCOUNT_FILE) into a dynamically named MySQL table using **chunked inserts** with SQLAlchemy.

🔧 **Main Steps**

1. MySQL Connection

```
db = mysql.connect(...)

cursor = db.cursor(buffered=True)
```

- Establishes a MySQL database connection using mysql.connector.
- Enables autocommit and buffered cursor.
- Prints a confirmation message upon successful connection.

2. Reading and Preprocessing Data

```
account_data = pd.read_parquet(ACCOUNT_FILE)
```

- Reads account data from a Parquet file into a DataFrame.

3. Date Feature Extraction

```
account_data['rel_syear'] = ...

account_data['deletion_year'] = ...
```

- Extracts year/month/day from:
  - Relationship Start Date
  - deletion_date

4. Cleaning and Standardizing

```
account_data.replace(..., inplace=True)

account_data.fillna('')

account_data.astype(str)
```

- Replaces null-like values and fills missing entries.
- Converts all data to string type to ensure compatibility with SQL.

Key transformations:

- Fixes values in deletion_status, Deletion Reason Code, Debt Type, etc.
- Handles NaT for datetime fields (Date of Notice/Letter of Demand, Agreement Date).

6. Column Cleanup & Renaming

```
account_data = account_data.drop([...])
```

```
account_data.rename(columns={...}, inplace=True)
```

- Drops unnecessary columns (e.g., account_key, Statement Date).
- Renames columns to match target DB schema (e.g., Account No → acc_no).

7. Column Selection and Ordering

```
account_data = account_data[account_table]
```

- Selects and reorders relevant columns for DB insertion.

8. SQLAlchemy Engine Setup

```
engine = create_engine(f'mysql+mysqlconnector://...')
```

- Prepares an SQLAlchemy engine for batch inserts.

9. Batch Insert with Chunking

```
chunk_size = 100000

for i in range(0, len(account_data), chunk_size):

    ...
```

- Inserts data in chunks to optimize performance and avoid memory/time issues.
- Logs success messages for each chunk.

10. Exception Handling and Final Logging

```
try ... except ... finally
```

- Logs errors if any issue occurs during insertion.
- Ensures proper completion log regardless of errors.

Load Statement Data to MYSQL Table

✅ Purpose

To clean, transform, and insert account data from a Parquet file (STATEMENT_FILE) into a dynamically named MySQL table using chunked inserts with SQLAlchemy.

✅ Main Step

1. MySQL Database Connection
   - Uses mysql.connector and sqlalchemy to establish connection to a MySQL database using credentials from environment variables (or external config).
   - Sets autocommit = True and creates a buffered cursor for DB operations.
2. Load and Filter Statement Data
   - Reads a Parquet file into a Pandas DataFrame.
   - Filters records from the last 5 years based on 'Statement Date'.
   - Sorts the data by 'Statement Date'.
3. Split Data by Year-Month and Save Parquet
   - Adds a new column YearMonth extracted from 'Statement Date'.
   - Creates (or cleans and recreates) a folder for storing split Parquet files.
   - Splits the DataFrame by YearMonth, saves each subset as its own .pq file.
4. Iterate Over Split Parquet Files for Preprocessing
   - Loads each split Parquet file.
   - Adds 'month' and 'year' columns.
   - Cleans and transforms multiple columns:
     - Casts values, handles NaN, empty strings, missing values.
     - Applies conditional logic to create the missed_payments column.
     - Drops unnecessary columns used for processing.
     - Renames columns to match the MySQL table schema.
5. Insert into MySQL Table
   - Uses to_sql() with SQLAlchemy to insert records in chunks (100,000 rows per chunk) into MySQL.
   - Table name is dynamically created as acc_stts_jcl_uat_<LOAD_TIME_STR>.
6. Error Handling

- Wrapped in try-except-finally to handle and log any errors that occur during processing and loading.

Load Sponsor Data to MYSQL Table

📝 Purpose

To load Sponsor data from a Parquet file into a MySQL database table, using chunked inserts via SQLAlchemy.

📦 Steps Overview

1. Connect to MySQL database using mysql.connector:
   - Host, user, password, and database are set using variables like MYSQL_HOST.
2. Load Parquet file into a Pandas DataFrame:
   - File path: SPONSOR_FILE.
3. Clean and transform data:
   - Replace various null-like values (nan, NaN, None, NaT, etc.) with empty strings.
   - Drop sponsor_key column.
   - Rename columns to match database schema.
   - Reorder columns based on the sponsor_table list.
4. Set up SQLAlchemy engine for MySQL connection using encoded credentials.
5. Insert data in chunks (100,000 rows each):
   - Loop through DataFrame and load into table sponsor_jcl_uat_<LOAD_TIME_STR>.
6. Handle errors and log success or failure of each chunk.

Promote Tables to Views

📝 Purpose

To refresh and recreate database views for the ETRPLUS system, specifically for:

- Data Analytics (DA)
- Reporting Environment (RE)

This process is triggered after loading new batch data (staging tables with timestamp suffixes).

🔧 Steps Overview

1. Database Connection:
    - Uses mysql.connector to connect to MySQL.
    - Enables autocommit and creates a buffered cursor.

📁 Part 1: Update DA Views

Drops and recreates 4 views to point to the new staging tables with timestamp:

- keytbl_jcl_uat
- relationship_jcl_uat
- acc_stts_jcl_uat
- sponsor_jcl_uat

Each view is recreated using:

CREATE VIEW <view_name> AS SELECT * FROM
<view_name>_<LOAD_TIME_STR>

📁 Part 2: Create RE Views

Drops and recreates 4 new views tailored for reporting:

- keytbl_plus_non_bank
- relationship_plus_non_bank
- acc_stts_plus_non_bank
- sponsor_plus_non_bank

Each view:

- Joins or transforms columns (e.g., complex logic for selecting IC/Passport based on availability).
- Filters data (e.g., acc_stts_plus_non_bank limits to the last 24 months using stmdate).

🛠️ Logic Highlights

- Dynamic View Naming: Appends LOAD_TIME_STR to select the latest data.
- Null Handling: Uses SQL CASE logic to prioritize available IDs.
- Performance-Oriented: acc_stts_plus_non_bank filters on date range for efficiency.
- Redundancy: DA view creation block is repeated twice — possibly an oversight unless needed.

✅ Output

- Logs confirmation (done) and timestamps for:
    - DA view updates
    - RE view updates

Generate DQ Report

📌 Purpose

This script generates a Data Quality (DQ) Staging Report for a specified company. It performs data validation, reconciliation, and reporting on various datasets related to fixed payment processing.

🧩 Main Components

1. 📥 Input Parameters
    - --company_code: Short code for the company.
    - --full_company_code: Full identifier used for folder paths.
2. 📂 Folder Setup
    - Sets up paths for input/output folders like:
        - Fixed template data
        - Reporting output
        - SSM rejection
        - Error and ingestion folders
3. 🧪 Data Preparation & Validation
    - Loads and processes fixed template data (customer, account, statement).
    - Loads SSM, error, and ingestion rejection files.
    - Saves intermediate results as .parquet and .csv.
4. ⚙️ Parallel Processing
    - Uses multiprocessing to speed up:
        - Fixed template processing
        - Staging data processing (master, customer, account, statement)
5. 📊 Reconciliation Checks

- Compares raw vs staging data for:
    - Limits
    - Tenure
    - Instalment amounts
    - Amounts in arrears
    - Relationship start dates
- Also checks for missing values and inconsistencies.

6. 📉 Data Quality Checks
    - Identifies:
        - Duplicates
        - Nulls
        - Wrong formats
        - Invalid or extreme values
        - Missing relationships
7. 📈 Summary Report
    - Compiles all findings into a summary DataFrame.
    - Saves the summary as a CSV file.
8. 📧 Email Notification
    - Sends the summary report via email with the CSV attached.