# Raw data to L1 Area Processing structured Data Extraction

# **SRC.UTIL:**

1. **Function Name:` readEncryptedConfig`:**
   1. **Purpose:**
      * The function reads encrypted configuration data from an Excel file and decrypts it using a specified encryption key. It returns the decrypted configuration as a dictionary.

* **Input:**
  + **excelFilePath**: A string representing the path to the Excel file containing the encrypted configuration details.
* **Read Excel File:**
* Use `pandas.read\_excel()` to read the Excel file located at `excelFilePath`.
* Store the contents of the Excel file into a pandas DataFrame (`pathsDf`).
* Convert the DataFrame into a dictionary (`pathsDict`) where:
* The Excel column `Key\_name` is used as dictionary keys.
* The Excel column `Path` is used as dictionary values.
* **Read Encryption Key:**
  + Extract the path to the encryption key file from `pathsDict` (under key `key\_path`).
  + Open the file at the specified path and read its contents (`scKey`).
  + Create a `Fernet` cipher suite using the encryption key.
* **Read Encrypted CSV File:**
  + Extract the path to the encrypted CSV file from `pathsDict` (under key `encrypted\_file`).
  + Use `pandas.read\_csv()` to read the CSV file and store its contents in a DataFrame (`df`).
* **Decrypt Data:**
  + Use a lambda function to map each value in the DataFrame (`df`) through the `decryptData()` function.
  + Apply the decryption to each element using the `cipherSuite`.
  + Store the decrypted data into a new DataFrame (`df\_decrypted`).
* **Extract Configuration:**
  + Convert the decrypted DataFrame into a dictionary called `config` with keys such as `host`, `database`, `user`, `password`, and `port` (with a default value of `3306`).
* **Return Configuration:**
  + Return the decrypted configuration as a dictionary.
* **Input Example:**

excelFilePath = "config/encrypted\_config.xlsx"

* **Output:**
  + `config`: A dictionary containing the decrypted configuration details:
  + `host`: The hostname of the database.
  + `database`: The name of the database.
  + `user`: The username for the database.
  + `password`: The decrypted password for the database.
  + `port`: The port number (default value: `3306`).

1. **Function Name:` readMetadata`:**
   1. **Purpose:**
      * The function connects to a MySQL database, retrieves metadata including AWS credentials, and returns essential information such as the database connection, session, table structure, and AWS keys.

* **Input:**
  + A dictionary `config` containing the following:
  + `user`: Database username.
  + `password`: Database password.
  + `host`: Database host.
  + `database`: The name of the database.
* **Database Connection:**
  + Construct the database connection string (`dbUrl`) using the input `config`.
  + Use `create\_engine()` to establish a connection to the MySQL database using the constructed URL.
* **Define Metadata:**
  + Create a `MetaData` object.
  + Reflect the current schema of the connected database to load its structure.
* **Create Session:**
  + Use `sessionmaker()` to create a session that will handle database transactions
  + Bind the session to the established database connection.
* **Define the Table:**
  + Load the `s3\_files` table from the database using the reflected metadata.
* **Execute SQL Query:**
  + Define a SQL query to retrieve AWS credentials from the `joblog\_metadata.prod\_l1\_context\_db` table.
  + Execute the query and store the results in a pandas DataFrame (`df\_mysql`).
* **Extract AWS Credentials:**
  + Convert the DataFrame into a dictionary format with keys and their corresponding values.
  + Extract `S3\_AccessKey` and `S3\_Secret\_Access\_Key` from the dictionary.
* **Return Values:**
  + `db\_connection`: The active connection to the MySQL database.
  + `session`: The session for interacting with the database.
  + `s3\_files\_table`: Metadata about the `s3\_files` table.
  + `aws\_access\_key\_id`: AWS access key ID.
  + `aws\_secret\_access\_key`: AWS secret access key.
* **Input Example:**

python

config = {

'user': 'db\_user',

'password': 'db\_pass',

'host': 'localhost',

'database': 'my\_database'

}

* **Output:**
  + db\_connection\*\*: A MySQL database connection object.
  + session\*\*: A session object to execute SQL commands.
  + s3\_files\_table\*\*: Table metadata of the `s3\_files` table.
  + aws\_access\_key\_id\*\*: AWS Access Key ID (e.g., `"AKIA..."`).
  + aws\_secret\_access\_key\*\*: AWS Secret Access Key (e.g., `"secret123..."`).
* **Example Input and Output Scenarios:**
* **Input**

python

config = {

'user': 'admin',

'password': 'password123',

'host': '127.0.0.1',

'database': 'aws\_metadata'

}

db\_connection, session, s3\_files\_table, aws\_access\_key\_id, aws\_secret\_access\_key = readMetadata(config)

1. **Class Name `databaseHandler`:**

* **Purpose:**
  + Handles database operations like creating tables, inserting log and audit entries, and updating audit records.

1. **Initialize `databaseHandler`:**
   * **Input:**
     + `dbUrl` A string representing the database URL.

* **Operations:**
  + Connect to the database using the provided URL.
  + Define metadata and create two tables:
    - `etl\_log\_final`: Stores ETL job logs.
    - `etl\_audit\_final`: Stores audit records.
  + Initialize a database session using SQLAlchemy's `sessionmaker`.

1. **Method: `insertAuditEntry(auditEntry)`:**

* **Input:**
  + auditEntry:
    - A dictionary containing audit information (e.g., `exec\_id`, `job\_name`, `file\_name`).
* **Operation:**
  + - Insert the audit entry into the `etl\_audit\_final` table.
    - Commit the changes to the database.
    - Roll back in case of errors.

1. **Method: `updateAuditEntry(exec\_id, updated\_values)`:**

* **Input:**
* `exec\_id`: A string representing the audit entry's unique execution ID.
* `updated\_values`: A dictionary containing the updated values for the audit record.
* **Operation:**
* Update the audit entry corresponding to the given `exec\_id` with `updated\_values`.
* Commit the changes to the database.
* Roll back in case of errors.

1. **Method: `close()`:**

* **Operation**: Closes the database session and connection.

1. **Class Name: `stepLogger`:**
   1. **Purpose:** Logs stepbystep execution details of an ETL job
2. **Initialize `stepLogger`:**
   * **Input:** `dbHandler` An instance of the `databaseHandler` class
3. **Method: `logStep()`:**

* **Input:**
  + `execId`: A string representing the execution ID.
  + `jobName`: Name of the ETL job.
  + `fileName`: Name of the processed file.
  + `startTime`: Start time of the job.
  + `endTime`: End time of the job.
  + `srcCnt`: Source record count.
  + `tgtCnt`: Target record count.
  + `lkpCnt`: Lookup record count.
  + `rejCnt`: Rejected record count.
  + `jobStatus`: Job execution status.
* **Operation:**
  + Constructs a log entry using the input parameters.
  + Calls the `insertLogEntry()` method of `databaseHandler` to insert the log entry into the database.

1. **Class Name: `auditLogger`:**
   1. **Purpose:** Logs and updates audit records for an ETL job.
2. **Initialize `auditLogger`:**
   * **Input:** `dbHandler` An instance of the `databaseHandler` class.
3. **Method: `logAudit()`:**
   * **Input:** `auditEntry` A dictionary containing the audit details for a specific ETL job.
   * **Operation:**Calls the `insertAuditEntry()` method of `databaseHandler` to insert the audit entry into the database.
4. **Method: `updateAudit()`:**
   * **Input:**

* **`exec\_id**`: A string representing the audit entry's unique execution ID.
* **`updated\_values`:** A dictionary containing the updated audit values.

1. **Operation:** Calls the `updateAuditEntry()` method of `databaseHandler` to update the audit record with the new values.

**Input Example:**

python

dbUrl = "mysql+pymysql://user:password@localhost/db\_name"

# Create a database handler instance

dbHandler = databaseHandler(dbUrl)

# Log a step

step\_logger = stepLogger(dbHandler)

step\_logger.logStep(

execId="123456789",

jobName="Data Load Job",

fileName="data\_file.csv",

startTime=datetime.now(),

endTime=datetime.now(),

srcCnt=1000,

tgtCnt=990,

lkpCnt=10,

rejCnt=0,

jobStatus="Completed"

)

# Log an audit entry

audit\_logger = auditLogger(dbHandler)

audit\_logger.logAudit({

'exec\_id': '123456789',

'job\_name': 'Data Load Job',

'system\_id': 'SYS1',

'file\_name': 'data\_file.csv',

'file\_date': datetime.now(),

'total\_rec\_cnt': 1000,

'processed\_cnt': 990,

'job\_status': 'Completed',

'rejection\_cnt': 0,

'rejection\_rsn': None,

's3\_last\_modified\_date': datetime.now()

})

# Close the database handler

dbHandler.close()

**Output Example:**

* + A new record will be inserted into the `etl\_log\_final` table with the details provided for the ETL job execution.
  + A new record will be inserted into the `etl\_audit\_final` table with the audit information.
  + The audit record corresponding to the execution ID `123456789` will be updated with new values, such as `processed\_cnt: 995` and `job\_status: 'Partially Completed'`.

The session and connection will be closed after the operations are complete.

1. **Function Name `decryptData`:**
2. **Imports:**
   * `pandas`: A library for handling data in tabular form.
   * `Fernet`: A cryptography class for symmetric encryption/decryption from the `cryptography.fernet` module.
3. **Load the Encryption Key:**
   * Input: `keyDirectory` The path to the encryption key stored as a `.key` file.
   * Operation: Read the encryption key from the file and initialize the `Fernet` object (`cipherSuite`) with this key.
4. **Function: `decryptData(data, cipherSuite)`:**

* **Input:**
* `data`: The data to be decrypted.
* `cipherSuite`: The `Fernet` object used for decryption.
* `keyDirectory`: Path to the encryption key file.
* **Operation:**
* Read the encryption key from `keyDirectory` and create a `Fernet` cipher suite.
* Check if `data` is a string.
* If `True`: Decrypt the data using the `cipherSuite` and return the decoded string.
* If `False`: Return the data as it is.
* Output: Returns the decrypted string if the input was encrypted, otherwise returns the original input.
* **Output:**
  + Decrypted data or the original data (if not encrypted).

**Input Example:**

```python

# Path to the encryption key file

keyDirectory = 'C:\\Users\\jvineet\\PycharmProjects\\PythonLearnings\\Data\\metadata\\secret.key'

# Load encryption key

scKey = open(keyDirectory, 'rb').read()

# Create Fernet cipher suite using the key

cipherSuite = Fernet(scKey)

# Example encrypted data (assuming data was encrypted with the same key)

encryptedData = cipherSuite.encrypt(b"Secret Message").decode()

# Decrypt the data using the decryptData function

decryptedData = decryptData(encryptedData, cipherSuite)

print("Decrypted Data:", decryptedData)

**Output Example:**

```text

Decrypted Data: Secret Message

```

**Explanation:**

1. The `keyDirectory` specifies where the encryption key is located.

2. The key is loaded and used to create a `Fernet` cipher suite (`cipherSuite`), which allows decryption.

3. The function `decryptData()` is used to decrypt a string of data. It checks if the input is a string and decrypts it if necessary, or returns the original data otherwise.

4. The decrypted data is printed, showing the original message that was encrypted.

This pseudo code explains how the encryption key is used to decrypt data using the `Fernet` encryption class.

1. **Function ` splitFile `:**
2. **Purpose:** To split a large CSV file into smaller files, each containing a specified number of records
3. **Inputs:**

* `input\_file\_path`: The path to the input file (string).
* `output\_dir`: The directory where the output files will be saved (string).
* `num\_records`: The maximum number of records each output file should contain (integer).
* `delimiter`: The delimiter used in the input and output files (string, default is `,`).

1. **Function Definition:**

* Define a function named `splitFile` that takes four parameters: `input\_file\_path`, `output\_dir`, `num\_records`, and `delimiter`.

1. **TryExcept Block:**

* Begin a tryexcept block to handle any potential errors.

1. **Extract Input File Name:**

* Use `os.path.basename(input\_file\_path)` to get the base name of the input file (including extension).
* Use `os.path.splitext(input\_file\_name)` to split the input file name into `input\_file\_name\_no\_ext` (name without extension) and `file\_extension` (the file extension).

1. **Read Header:**

* Use `pd.read\_csv` with `nrows=0` to read the header of the CSV file and store it in `header`.

1. **Iterate Over Chunks:**

* Use `pd.read\_csv` with `chunksize=num\_records` to read the input file in chunks.
* For each chunk:
* Construct the output filename using the format:
* output\_filename = output\_dir + '/' + input\_file\_name\_no\_ext + '\_file' + i + file\_extension.
* Save the chunk to a CSV file using `chunk.to\_csv()` with specified `header` and `delimiter`.

1. **Handle Exceptions:**

* In case of an error, print the error message.

1. **Output:**

* Multiple smaller CSV files are created in the specified output directory, each containing a maximum of `num\_records`.

**Example Usage:**

```python

# Define input file path, output directory, number of records per file, and delimiter

inputFilePath = 'C:\\Users\\jvineet\\PycharmProjects\\PythonLearnings\\Data\\processed\\LA\_processed\\Hurley\_Medical\_8.20217.2023.csv'

outputDir = 'C:\\Users\\jvineet\\PycharmProjects\\PythonLearnings\\Data\\processed\\SplitFile'

numRecords = 10000

delimiter = '|' # Specify the delimiter

**# Call the splitFile function**

splitFile(inputFilePath, outputDir, numRecords, delimiter)

**Output Example:**

If the input CSV file contains 30,000 records, the function will create three output files:

Hurley\_Medical\_8.20217.2023\_file0.csv` (contains records 09999)

Hurley\_Medical\_8.20217.2023\_file1.csv` (contains records 1000019999)

Hurley\_Medical\_8.20217.2023\_file2.csv` (contains records 2000029999)

Each file will contain headers from the original CSV file, and the records will be separated using the specified delimiter (`|`).

This pseudo code outlines the functionality of the `splitFile` function, providing a structured understanding of its operations and expected inputs and outputs.

1. **Function ` createS3Client `:**
   1. **Purpose:** To establish a connection to an AWS S3 client using the provided AWS credentials.
   2. **Inputs:**
      * `awsAccessKeyId`: AWS Access Key ID (string).
      * `awsSecretAccessKey`: AWS Secret Access Key (string).
      * `regionName`: The AWS region name where the S3 service is located (string).
   3. **Function Definition:**
      * Define a function named `createS3Client` that takes three parameters: `awsAccessKeyId`, `awsSecretAccessKey`, and `regionName`.
   4. **TryExcept Block:**
      * Begin a tryexcept block to handle potential credential errors.
   5. **Create S3 Client:**
      * Use `boto3.client()` to create an S3 client with the following parameters:
      * Service name: `'s3'`
      * `aws\_access\_key\_id`: Use the provided `awsAccessKeyId`.
      * `aws\_secret\_access\_key`: Use the provided `awsSecretAccessKey`.
      * region\_name`: Use the provided `regionName`.
   6. **Return S3 Client:**
      * If successful, return the created S3 client.
   7. **Handle Credential Errors:**
      * If a `NoCredentialsError` or `PartialCredentialsError` occurs, print an error message indicating the problem and return `None`.
   8. **Output**:
      * Returns an S3 client object if successful.
      * Returns `None` if there was an error in the credentials

**Example Usage:**

**python**

**# Define AWS credentials and region**

awsAccessKeyId = 'YOUR\_AWS\_ACCESS\_KEY\_ID'

awsSecretAccessKey = 'YOUR\_AWS\_SECRET\_ACCESS\_KEY'

regionName = 'uswest2'

**# Create S3 client**

s3\_client = createS3Client(awsAccessKeyId, awsSecretAccessKey, regionName)

**# Check if S3 client was created successfully**

if s3\_client:

print("S3 client created successfully.")

else:

print("Failed to create S3 client.")

**Output Example:**

**1. Successful Connection:**

* If the credentials are valid, the output will be:
* S3 client created successfully.
* The `s3\_client` variable will hold a valid S3 client object.

**2. Failed Connection Due to Credentials:**

If the credentials are missing or incomplete, the output will be:

Credentials error: {error message}

Failed to create S3 client.

The `s3\_client` variable will be `None`.

# **SRC.UTIL:**

1. **Class Name:` dataLoader `:**
   1. **Purpose:** To load data from files into a specified database table, adding metadata and hash keys during the process.
   2. **Attributes:**
      * file\_type\*\*: (string) Type of files to load (e.g., 'csv').
      * table\_name\*\*: (string) Name of the database table to load data into.
      * directory\_path\*\*: (string) Path to the directory containing the files.
      * delimiter\*\*: (string) Delimiter used in the files.
      * db\_uri\*\*: (string) Database URI for SQLAlchemy engine.
      * load\_user\*\*: (string) User responsible for the load process.
      * engine\*\*: (SQLAlchemy Engine) SQLAlchemy engine instance.
      * Session\*\*: (SQLAlchemy Session) SQLAlchemy sessionmaker bound to the engine.
      * job\_results\*\*: (list) List of dictionaries containing results for each file processed.
   3. **Initialization: `\_\_init\_\_`**
2. **Input Parameters:** 
   * `file\_type`: Type of files (e.g., 'csv').
   * `table\_name`: Database table name.
   * `directory\_path`: Directory path of input files.
   * `delimiter`: Delimiter in files.
   * `db\_uri`: Database URI for connection.
   * `load\_user`: User loading the data.
3. **Initialize Attributes:**
   * Set class attributes using input parameters.
   * Create a SQLAlchemy engine using `db\_uri`.
   * Create a sessionmaker object.
   * Initialize `job\_results` as an empty list.
   1. **Method: `load\_data`**
4. **List Files:**
   * Get all files from `directory\_path` that match `file\_type`.
5. **Process Each File:**
   * For each file:
   * Construct the file path.
   * Attempt to read the file using `\_read\_file(file\_path)`.
   * If successful:
   * Count source records.
   * Add a hash key to the data using `\_add\_hash\_key(data)`.
   * Add metadata using `\_add\_metadata(data)`.
   * Insert data into the database using `\_insert\_data(data)`.
   * Append results to `job\_results`.
   * If reading fails, log failure in `job\_results`.
   1. **Method: `\_read\_file`**
6. **1. Input Parameter:** 
   * `file\_path`: Path of the file to read.
7. **Return:**
   * Return a Pandas DataFrame of the read data.
   * Return `None` if an error occurs.
   1. **Method: `\_add\_hash\_key`**
8. **Input Parameter:** 
   * `data`: DataFrame to which the hash key will be added.
9. **Process:**

* For each row in the DataFrame:
* Concatenate column values.
* Generate a SHA256 HMAC hash.
* Add the hash as a new column called `hash\_key`.

1. **Return:**
   * Return the modified DataFrame with the `hash\_key` column.
   1. **Method: `\_add\_metadata`**
2. **Input Parameter:** 
   * data`: DataFrame to which metadata will be added.
3. **Process:**
   * Add `load\_user` and `load\_date` columns to the DataFrame.
4. **Return:**
   * Return the modified DataFrame with metadata columns.
   1. **Method: `\_insert\_data`**
5. **Input Parameter:** 
   * `data`: DataFrame to be inserted into the database.
6. **Process:**
   * Begin a database transaction.
   * Insert the DataFrame into the specified table.
   * Return the total number of records in the table after insertion.
7. **Return:**
   * Return the count of records after the insert operation.
   1. **Method: `close`**
8. **Process**:
   * Dispose of the SQLAlchemy engine.
9. **Return**:
   * Return the `job\_results` list containing the results for each processed file.

**Example Usage:**

**python**

**# Define parameters for data loading**

file\_type = 'csv'

table\_name = 'your\_table\_name'

directory\_path = 'path/to/your/files'

delimiter = ','

db\_uri = 'mysql+pymysql://user:password@host/database'

load\_user = 'admin\_user'

**# Create dataLoader instance**

loader = dataLoader(file\_type, table\_name, directory\_path, delimiter, db\_uri, load\_user)

**# Load data from files into the database**

loader.load\_data()

**# Retrieve job results**

results = loader.close()

**# Output results**

print(results)

**Output Example:**

1. Successful Processing:

If all files are processed successfully, the output will be a list of dictionaries indicating success:

python

[

{'file\_name': 'file1.csv', 'source\_records\_count': 100, 'records\_loaded': 100, 'status': 'Success'},

{'file\_name': 'file2.csv', 'source\_records\_count': 200, 'records\_loaded': 200, 'status': 'Success'}

]

2. Failed Processing:

If there was an error during processing, the output will indicate failure:

python

[

{'file\_name': 'file1.csv', 'source\_records\_count': 0, 'records\_loaded': 0, 'status': 'Failed: <error message>'},

{'file\_name': 'file2.csv', 'source\_records\_count': 0, 'records\_loaded': 0, 'status': 'Failed: <error message>'}

]

1. **Class Name:` fileMover `:**
   1. **Purpose:** To move files from a source directory to target directories based on their file types.
   2. **Attributes:**
      * source\_dir: (string) Path to the directory containing files to be moved.
      * target\_base\_dir: (string) Base directory where files will be moved to, organized by type.
   3. **Initialization: `\_\_init\_\_`**
      * Set `self.source\_dir` to `source\_dir`.
      * Set `self.target\_base\_dir` to `target\_base\_dir`.
   4. **Method: `identify\_file\_type`**
2. **Input Parameter:**
   * `filename`: The name of the file to identify its type.
3. **Process:**
   * Convert `filename` to lowercase for comparison.
   * Check for specific substrings in the filename:
   * If `'ap'` is in the filename, return `'AP'`.
   * If `'invoice'` or `'inv'` is in the filename, return `'INVOICE'`.
   * If `'po'` is in the filename, return `'PO'`.
   * If no match is found, return `None`.
4. **Return:**
   * The identified file type or `None` if no type matches.
   1. **Method: `move\_file**
5. **Input Parameters:**
   * `file\_path`: The path of the file to be moved.
   * `file\_type`: The identified type of the file (e.g., `'AP'`, `'INVOICE'`, or `'PO'`).
6. **Process:**
   * Construct the target directory path by joining `target\_base\_dir` and `file\_type`.
   * Ensure the target directory exists using `os.makedirs()` with `exist\_ok=True`.
   * Construct the destination file path by joining the target directory and the base name of the file.
7. **Attempt to Move File:**
   * Use `shutil.move()` to move the file to the destination path.
   * Print a success message if the move is successful.
   * If an error occurs, print an error messag.
   1. **Method: `move\_files`:**
      * Iterate through each filename in the `source\_dir` using `os.listdir()`.
      * Construct the full file path for each file.
      * Check if the path corresponds to a file using `os.path.isfile()`:
      * If a file type is returned:
      * Call `move\_file(file\_path, file\_type)` to move the file.
      * If no file type is identified, print a message indicating that the file will be skipped.

**Example Usage:**

python

**# Define parameters for the file mover**

source\_dir = 'path/to/source/directory'

target\_base\_dir = 'path/to/target/directory’

**# Create fileMover instance**

mover = fileMover(source\_dir, target\_base\_dir)

# Move files from the source directory to respective folders based on file type

mover.move\_files()

**Output Example:**

1. Successful Moves:

If files are moved successfully, the output will be:

Moved 'example\_ap\_file.txt' to 'path/to/target/directory/AP/example\_ap\_file.txt'

Moved 'invoice\_123.pdf' to 'path/to/target/directory/INVOICE/invoice\_123.pdf'

Moved 'purchase\_order\_456.docx' to 'path/to/target/directory/PO/purchase\_order\_456.docx'

2. Skipped Files:

If a file does not match any known type:

File 'unknown\_file.txt' does not match any known type. Skipping.

3. Error Handling:

If an error occurs while moving a file:

Error moving file 'path/to/source/directory/example\_ap\_file.txt': <error message>

1. **Function Name:` readDocumentsFromPath `:**
   1. **Purpose:** To retrieve all files from a specified directory and its subdirectories.
   2. **Input:**
      * `myPath`: (string) The root directory path from where to begin searching for files.
   3. **Process:**
      * Initialize an empty list called `allFiles`.
      * Use `os.walk(myPath)` to traverse the directory tree starting at `myPath`:
        + For each directory (`path`), subdirectory (`subdirs`), and file (`files`):

* For each `name` in `files`:
* Append the full path of the file (`os.path.join(path, name)`) to `allFiles`.
  1. **reate `allFileNames`:**
     + Extract just the file names from `allFiles`:
     + For each `file` in `allFiles`, split by `//` and take the last part.
     + Split the result by `\` and take the last part to get the file name with extension.
  2. **Return:**
     + `allFiles`: A list of full paths to all files found.
     + `allFileNames`: A list of file names (with extensions) extracted from the full paths.

**Example:**

**python**

allFiles, allFileNames = readDocumentsFromPath("/path/to/directory")

**Output:**

allFiles`: ['/path/to/directory/file1.csv', '/path/to/directory/subdir/file2.txt']

allFileNames`: ['file1.csv', 'file2.txt']

1. **Function Name:` find\_delimiter`:**
   1. Purpose:
      * To identify the delimiter used in a given file by analyzing the first line, which is assumed to be the header.
   2. Input:
      * `file`: (string) The full path to the file whose delimiter is to be identified.
      * `doc\_type`: (string) The file extension/type (e.g., "csv", "tsv").
   3. Process:
      * Open the file in read mode and read the first line (column headers).
      * If `doc\_type` is "tsv", return the tab character (`\t`).
      * dentify special characters in the column headers using a regex pattern:
        + Create a list of unique special characters found.
        + If an underscore (`\_`) is present in the list, remove it.

* Initialize two empty lists: `delimiter\_count\_list` and `delimiter\_value\_list`.
* For each special character in the list:
* Count how many times it occurs in the header (`times\_in\_string`).
  + - * Split the header using that character and check if the number of splits matches the count:
* If yes, append the count to `delimiter\_count\_list` and the character to `delimiter\_value\_list`.
* Create a DataFrame from the counts and delimiters.
* Sort the DataFrame by count in descending order.
* If the DataFrame is not empty, return the most frequent delimiter.
* If no delimiters were found, return `None`

#Example:

```python

delimiter = find\_delimiter("/path/to/file.csv", "csv")

#Output:

If the file has commas as delimiters:

delimiter = ','

If the file is a TSV:

delimiter = '\t'

If no delimiters are found:

delimiter = None

1. **Function Name:`** **getS3Directories `:**
   1. **Purpose**:
      * Generate a list of S3 directory names for the current month and, if applicable, the previous month based on a cutoff day.
   2. **Input**:
      * `daysS3`: (integer) The cutoff day of the current month.
   3. **Process:**
      * Get the current date (`today`).
      * Format the current month directory as `MM\_Data\_Upload\_MonthYear`.
      * If today's day is less than or equal to `daysS3`:
      * Calculate the previous month by subtracting the current day + 1 from today's date.
      * Format the previous month directory as `MM\_Data\_Upload\_MonthYear`.
      * Return a list containing paths for both the current and previous month.
      * Otherwise, return a list containing only the current month path.
      * If an error occurs, print an error message and return an empty list.

**Example**:

**python**

directories = getS3Directories(15)

Output:

If today is September 10, 2024:

directories = ['customer\_data/09\_Data\_Upload\_September2024', 'customer\_data/08\_Data\_Upload\_August2024']

If today is September 20, 2024:

directories = ['customer\_data/09\_Data\_Upload\_September2024']

1. **Function Name:`** **listFilesInS3Directories `:**
   1. **Purpose**:
      * List all files in the specified S3 directories, filtering out directory keys and extracting the filenames.
   2. **Input:**
      * `directories`: (List of strings) A list of S3 directory paths to search for files.
      * s3Client`: (boto3.client) A boto3 S3 client object used to interact with the S3 service.
      * `bucketName`: (string) The name of the S3 bucket.
   3. **Process:**
      * Initialize an empty list called `files`.
      * For each `directory` in `directories`:
        + Use `s3Client.list\_objects\_v2` to list objects in the directory.
        + For each object in the response:
        + Get the `Key`.
        + If the `Key` does not end with `/` (i.e., it's not a directory):
        + Extract the filename using `os.path.basename`.
        + Append a dictionary containing `Key` and `LastModified` timestamp to `files`.
      * If an error occurs, print an error message.
      * Return the list of `files`.

**Example:**

python

files = listFilesInS3Directories(['customer\_data/09\_Data\_Upload\_September2024'], s3Client, 'mybucket')

#Output:

files = [

{'Key': 'file1.csv', 'LastModified': '20240915T12:34:56Z'},

{'Key': 'file2.txt', 'LastModified': '20240916T12:34:56Z'}

]

1. **Function Name:`** **getIncrementalFiles `:**
   1. **Purpose**:
      * Identify new files in the S3 directories by comparing them with previously processed files in an audit table.
   2. **Input**:
      * `s3Files`: (List of dictionaries) A list of dictionaries representing files from S3 directories.
      * `auditFiles`: (pd.DataFrame) A DataFrame containing file records from the audit table.
   3. **Process**:
      * Convert `s3Files` to a DataFrame (`s3FilesDf`).
      * Rename the `file\_name` column in `auditFiles` to `Key`.
      * Extract only the `Key` column from `auditFiles`.
      * Create a DataFrame from `auditFiles`.
      * Identify files in `s3FilesDf` that are not in `auditFilesDf`.
      * Return the list of new incremental files as dictionaries.

**Example:**

python

incrementalFiles = getIncrementalFiles(s3Files, auditFiles)

**Output:**

incrementalFiles = [

{'Key': 'file3.csv', 'LastModified': '20240917T12:34:56Z'},

{'Key': 'file4.txt', 'LastModified': '20240918T12:34:56Z'}

]

1. **Function Name:`** **insertFilesToDb `:**
   1. **Purpose**:
      * Insert new incremental files into a MySQL table, marking them as ready for processing.
   2. **Input:**
      * `incrementalFiles`: (List of dictionaries) A list of incremental files to be inserted.
      * s3FilesTable`: (sqlalchemy.Table) The SQLAlchemy Table object.
      * `dbConnection`: (sqlalchemy.engine.Connection) The SQLAlchemy database connection object.
      * `insert`: (sqlalchemy.sql.expression.Insert) The SQLAlchemy Insert expression.
   3. **Process**:
      * For each `file` in `incrementalFiles`:
        + Create an insert statement with the file details.
        + Execute the insert statement.
      * Commit the database transaction.
      * If an error occurs, roll back the transaction and print an error message.
      * Close the database connection.

**Example**:

python

insertFilesToDb(incrementalFiles, s3FilesTable, dbConnection, insert)

**Output**:

Successful insertion does not return a value but modifies the database

1. **Function Name:`** **fetchUnprocessedFiles `:**
   1. **Purpose**:
      * Fetch unprocessed files from the MySQL table where files are marked with the flag 'Y'.
   2. **Input**:
      * `s3FilesTable`: (sqlalchemy.Table) The SQLAlchemy Table object.
      * `session`: (sqlalchemy.orm.Session) A SQLAlchemy session object.
      * `select`: (sqlalchemy.sql.expression.Select) The SQLAlchemy Select expression.
   3. **Process:**
      * Create a select statement to fetch files with `Flag` = 'Y'.
      * Execute the query.
      * Convert the results to a list of dictionaries.
      * Return the list of unprocessed files.
      * If an error occurs, print an error message and return an empty list.
      * Finally, close the session.

**Example**:

python

unprocessedFiles = fetchUnprocessedFiles(s3FilesTable, session, select)

**Output**:

unprocessedFiles = [

{'Key': 'file1.csv', 'Flag': 'Y', 's3\_load\_time': '20240919T12:34:56Z'},

{'Key': 'file2.txt', 'Flag': 'Y', 's3\_load\_time': '20240920T12:34:56Z'}

]

1. **Function Name:`** **downloadAndProcessFile `:**
   1. **Purpose**:
      * Download an unprocessed file from S3 to a local directory and process it.
   2. **Input**:
      * `s3Client`: (boto3.client) A boto3 S3 client object.
      * `bucketName`: (string) The name of the S3 bucket.
      * `unprocessedFile`: (Dict) A dictionary representing the unprocessed file.
      * `listDir`: (string) The S3 directory path.
      * `localDirectory`: (string) The local directory path.
   3. **Process:**
      * Construct the full S3 path by combining `listDir` and the file name.
      * Construct the local file path.
      * Print the S3 path and local path for debugging.
      * Attempt to download the file from S3 to the local directory:
        + If successful, print a success message.
        + If the file is not found, print an error message.
        + If an error occurs, print an error message.

**Example**:

python

downloadAndProcessFile(s3Client, 'mybucket', unprocessedFile, 'customer\_data/09\_Data\_Upload\_September2024', '/local/path')

Output:

If the download is successful:

"Downloaded customer\_data/09\_Data\_Upload\_September2024/file1.csv to /local/path/file1.csv"

If the file is not found:

"File not found in S3: customer\_data/09\_Data\_Upload\_September2024/file1.csv"

# **SRC.** **data\_quality:**

1. **Class Name:` delimiterValidator `:**
   1. **Purpose:**
      * This class validates rows in a delimiterseparated file and separates valid and invalid rows into different output files.
   2. **Attributes:**
      * `input\_file\_path`: (string) Path to the input file.
      * `expected\_columns`: (int) The expected number of columns in each row.
      * `delimiter`: (char) The delimiter used to separate fields in the file.
      * `valid\_output\_file\_path`: (string) Path to the directory where valid rows will be saved.
   3. **Method: `\_\_init\_\_`:**

* **Purpose:**Initialize the `delimiterValidator` object with the necessary paths, column count, and delimiter.
* **Input:**
  + `input\_file\_path`: Path to the input file directory.
  + `expected\_columns`: Expected number of columns in the file.
  + `delimiter`: Delimiter used in the file.
  + `valid\_output\_file\_path`: Directory path for valid rows.
  + `rejected\_output\_file\_path`: Directory path for invalid rows.
* **Process:**
  + Initialize instance variables with the provided arguments.
  + Create directories for valid and rejected output files if they don't already exist.

**Example:**

python

validator = delimiterValidator("input/", 3, ',', "output/valid", "output/rejected")

**Output**:

Initializes an instance of the `delimiterValidator` class.

* 1. **Method: `validate\_row`**
* **Purpose:**Validate a single row by counting delimiters outside of quoted text and checking if it has the expected number of columns.
* **Input**:
  + `row`: (string) A single row of the input file.
* **Process:**
  + Use the `csv.reader` to parse the row with the given delimiter.
  + Try to read the row into a list of columns.
  + If the number of columns matches the `expected\_columns`, return `True` (valid row).
  + If not, return `False` (invalid row).

**Example:**

**python**

is\_valid = validator.validate\_row('John,Doe,30')

**Output:**

If the row has 3 columns:

is\_valid = True

If the row has fewer or more columns:

is\_valid = False

* 1. **Method: `process\_file`**
* **Purpose:**Processes the input file and separates valid and invalid rows into separate files.
* **Input:**
  + `input\_file\_name`: (string) The name of the input file to be processed.
* process:
  + Initialize `valid\_rows` and `invalid\_rows` lists.
  + Construct full paths for the input file, valid output file, and rejected output file.
  + If the input file does not exist, raise a `FileNotFoundError`.
  + Initialize counters for `total\_rows`, `valid\_rows\_count`, and `invalid\_rows\_count`.
  + Open the input file and iterate through each row:
  + Increment `total\_rows`.
  + If the row is valid (using `validate\_row`), add it to `valid\_rows` and increment `valid\_rows\_count`.
  + Otherwise, add it to `invalid\_rows` and increment `invalid\_rows\_count`.
  + Write the `valid\_rows` to the valid output file.
  + Write the `invalid\_rows` to the rejected output file.
  + Return a tuple containing:
    - `input\_file\_name`
    - `total\_rows`
    - `valid\_rows\_count`
    - `invalid\_rows\_count`
    - `job\_status` ("Success" or "Failed" if an error occurs).

**Example:**

python

result = validator.process\_file('data.csv')

**#Output:**

If the input file contains the following rows:

John,Doe,30

Jane,Doe

Alice,Smith,25

and the `expected\_columns` is 3, the following files will be generated:

Valid rows:

valid\_data.csv:

John,Doe,30

Alice,Smith,25

Invalid rows:

rejected\_data.csv:

Jane,Doe

The returned tuple will be:

('data.csv', 2, 2, 1, 'Success')

* 1. **Method Execution Flow**
* **Initialize Validator (`\_\_init\_\_`):**
  + Create the instance and prepare directories for valid and rejected files.
* **Validate Row (`validate\_row`):**
  + Check if a row has the expected number of columns based on the delimiter.
* **Process File (`process\_file`):**
  + Read the file, validate each row, separate valid and invalid rows, write them to respective files, and return a summary of the operation.

1. **Function Name:` fetchColumnsFromMetadata `:**
   1. **Purpose:**Fetches the column names from the metadata of a given database table where the column is active.
   2. **Input**:
      * `dbname`: (string) The name of the database.
      * `tablename`: (string) The name of the table
   3. **Process:**
      * Create an SQL query to select the `COLUMN\_NAME` from the table where `ACTIVE\_FLAG` is 'Y'.
      * Execute the SQL query using `pd.read\_sql\_query`.
      * Convert the result to a list of column names.
      * Return the list of active column names.

**Example**:

python

columns = fetchColumnsFromMetadata('my\_db', 'my\_table')

**Output**:

If the table contains columns with the `ACTIVE\_FLAG = 'Y'`, the output might be:

columns = ['column1', 'column2', 'column3']

1. **Function Name:` fetchDateColumns `:**
   1. **Purpose:**Fetches the column names from the metadata of a given database table where the column is a date column and active.
   2. **Input**:
      * dbname`: (string) The name of the database.
      * `tablename`: (string) The name of the table.
   3. **Process**:
      * Create an SQL query to select the `COLUMN\_NAME` from the table where `ACTIVE\_FLAG` is 'Y' and `DATE\_FLAG` is 'Y'.
      * Execute the SQL query using `pd.read\_sql\_query`.
      * Convert the result to a list of date column names.
      * Return the list of date columns.

**Example**:

python

date\_columns = fetchDateColumns('my\_db', 'my\_table')

**Output**:

If the table contains date columns, the output might be:

date\_columns = ['date\_column1', 'date\_column2']

1. **Function Name:` fetchDateColumns `:**
2. **Purpose**: Fetches the column names from the metadata of a given database table where the column has duplication allowed and is active.
3. **Input**:
   * + `dbname`: (string) The name of the database.
     + `tablename`: (string) The name of the table.
4. **Process**:
   * + Create an SQL query to select the `COLUMN\_NAME` from the table where `ACTIVE\_FLAG` is 'Y' and `DUPLICATION\_FLAG` is 'Y'.
     + Execute the SQL query using `pd.read\_sql\_query`.
     + Convert the result to a list of duplicationallowed column names.
     + Return the list of duplicationallowed columns.

**Example**:

python

dup\_columns = fetchDupColumns('my\_db', 'my\_table')

Output:

If the table contains columns with duplication allowed, the output might be:

dup\_columns = ['dup\_column1', 'dup\_column2']

1. **Function Name:`** fetchNullColumns **`:**
   1. **Purpose:** Fetches the column names from the metadata of a given database table where the column can have null values and is active, sorted by column index.
   2. **Input**:
      * dbname`: (string) The name of the database.
      * `tablename`: (string) The name of the table.
   3. **Process**:
      * Create an SQL query to select the `COLUMN\_NAME` from the table where `ACTIVE\_FLAG` is 'Y' and ` NULL\_FLAG ` is 'Y'.
      * Execute the SQL query using `pd.read\_sql\_query`.
      * Convert the result to a list of nullable column names.
      * Return the list of nullable columns.

**Example**:

```python

null\_columns = fetchNullColumns('my\_db', 'my\_table')

**Output**:

If the table contains columns that allow null values, the output might be:

null\_columns = ['null\_column1', 'null\_column2']

1. **Function Name:`** fetchColumnsLengthchk **`:**
   1. **Purpose:** Fetches the column names and their sizes from the metadata of a table, where the columns are active. Returns a dictionary mapping column names to their sizes.
   2. **Input**:
      * dbname`: (string) The name of the database.
      * `tablename`: (string) The name of the table.
   3. **Process**:
      * Create an SQL query to select the `COLUMN\_NAME` and COLUMN\_SIZE from the table where `ACTIVE\_FLAG` is 'Y' and ` ordered by `COLUMN\_INDEX`.
      * Execute the SQL query using `pd.read\_sql\_query`.
      * Convert the result into a dictionary mapping `COLUMN\_NAME` to `COLUMN\_SIZE`.
      * Return the dictionary

**Example**:

```python

column\_sizes = fetchColumnsLengthchk('my\_db', 'my\_table')

Output:

If the metadata contains information about column sizes, the output might be:

column\_sizes = {'column1': 255, 'column2': 128}

1. **Function Name:`** fetchColumnsDatatypechk **`:**
   1. **Purpose:** Fetches the column names and their data types from the metadata of a table where the columns are active. Returns a dictionary mapping column names to their data types.
   2. **Input**:
      * dbname`: (string) The name of the database.
      * `tablename`: (string) The name of the table.
   3. **Process**:
      * Create an SQL query to select the `COLUMN\_NAME` and DATA\_TYPE from the table where `ACTIVE\_FLAG` is 'Y' and ` ordered by `COLUMN\_INDEX`.
      * Execute the SQL query using `pd.read\_sql\_query`.
      * Convert the result into a dictionary mapping `COLUMN\_NAME` to ` DATA\_TYPE `.
      * Return the dictionary

**Example**:

python

column\_types = fetchColumnsDatatypechk('my\_db', 'my\_table')

Output:

If the metadata contains information about column data types, the output might be:

column\_types = {'column1': 'VARCHAR', 'column2': 'INT'}

**Summary of Functions:**

* + - fetchColumnsFromMetadata`: Fetches active column names.
    - fetchDateColumns`: Fetches active date column names.
    - fetchDupColumns`: Fetches columns with duplication allowed.
    - fetchNullColumns`: Fetches nullable column names, sorted by index.
    - fetchColumnsLengthchk`: Fetches a dictionary of column names and their sizes.
    - fetchColumnsDatatypechk`: Fetches a dictionary of column names and their data types.

These functions are used to retrieve metadata about database columns, filtering them based on certain flags and returning the results in convenient formats like lists or dictionaries.

1. **Function Name:` columnsDefnition `:**
   1. **Purpose:** Fetches the column names and their data types from the metadata for a given database and table..
   2. **Input**:
      * dbname`: (string) The name of the database.
      * `tablename`: (string) The name of the table.
   3. **Process**:
      * Call the `fetchColumnsFromMetadata` function to get the active column names from the table.
      * Call the `fetchColumnsDatatypechk` function to get the data types of the columns.
      * Return the column names and data types.

**Example**:

```python

columns, data\_types = columnsDefnition('invoice\_db', 'invoice\_table')

**Output**:

If the table contains metadata for columns, the output might be:

```python

columns = ['invoice\_id', 'invoice\_date', 'amount']

data\_types = {'invoice\_id': 'VARCHAR', 'invoice\_date': 'DATE', 'amount': 'FLOAT'}

1. **Function Name:` generateHashKey `:**
   1. **Purpose:** Generates a SHA256 hash key for a row of data using a delimiter to combine the row's values.
   2. **Input**:
      * `row`: (List of strings or numbers) The row data that needs to be hashed.
   3. **Process**:
      * Convert the row into a string, joining each element with a delimiter (pipe `|`).
      * Use the `hashlib.sha256` function to generate a hash of the string.
      * Convert the hash to a hexadecimal string and return it.

**Example**:

```python

row = ['123', '20230925', '500.00']

hash\_key = generateHashKey(row)

Output:

The output might be a hash string:

```python

hash\_key = 'b94d27b9934d3e08a52e52d7da7dabfadee7b4aaff3f2c2a348d15b7f6f4e6c4'

1. **Function Name:` generateHashKey `:**
   1. **Purpose:** Reads a source CSV file into a DataFrame, applying column definitions and data types. Additionally, generates a hash key for each row.
   2. **Input**:
      * filePath`: (string) The path of the source file.
      * fileName`: (string) The name of the source file.
      * colums\_read`: (list) The list of column names to be read from the file.
      * data\_types`: (dict) The dictionary of column names and their corresponding data type
   3. **Process**:
      * Read the source file using `pd.read\_csv`, specifying the column names (`usecols`), data types (`dtype`), and other relevant options like delimiter and quote character.
      * Apply the `generateHashKey` function to each row to create a new column, `hashKey`, containing the hash value.
      * Return the resulting DataFrame.

**Example**:

```python

df = readSourceFile('/path/to/source', 'invoice\_data.csv', columns, data\_types)

Output:

If the file contains data like:

invoice\_id | invoice\_date | amount

123 | 20230925 | 500.00

456 | 20230926 | 300.00

```

The output might be a DataFrame with a new `hashKey` column:

```python

df = pd.DataFrame({

'invoice\_id': ['123', '456'],

'invoice\_date': ['20230925', '20230926'],

'amount': [500.00, 300.00],

'hashKey': ['b94d...', 'c4e8...']

})

**Summary of Functions:**

* + `columnsDefnition`: Fetches column names and their data types from metadata.
  + `generateHashKey`: Generates a hash key for a row of data by joining row elements with a delimiter.
  + `readSourceFile`: Reads a CSV file into a DataFrame using specified column names and types, and generates a hash key for each row.

These functions help to define the columns and data types of a table, process a source file, and generate unique hash keys for each row in the file.

1. **Function Name:` date\_Formats\_Vault `:**
   1. **Purpose:** Validates and identifies the format of a given date string. If the date is valid, it returns the detected date format; otherwise, it returns `'unknown date format'`.
   2. **Input**:
      * `input\_Data`: (string or date) The date value to validate and check its format.
   3. **Process**:
      * If `input\_Data` is NaN (not a valid date), return `'null'`.
      * Define a list of possible date formats (`possible\_DateFormats`).
      * Loop through each format in the list:
        1. Try to convert the `input\_Data` to a datetime object using `pd.to\_datetime`.
        2. If successful, return the detected date format.
        3. If it raises an exception (invalid format), continue to the next format.
      * If no formats match, return `'unknown date format'`.

**Example**:

```python

format = date\_Formats\_Vault('20230925')

**Output**:

If the input is a valid date:

```python

format = '%Y%m%d'

If the input has an unrecognized format:

```python

format = 'unknown date format'

1. **Function Name:` dateFormatFinder `:**
   1. **Purpose:** Applies the `date\_Formats\_Vault` function to two date columns from the input DataFrame, writes the date formats to a CSV file, and calculates how many date formats are unknown.
   2. **Input**:
      * `input\_file\_path`: (string) The path of the input CSV file.
      * `output\_file\_Path`: (string) The path where the output CSV will be saved.
      * `dbname`: (string) The database name where metadata is stored.
      * `tablename`: (string) The table name to fetch the metadata.
      * `delimiter`: (string) The delimiter used in the CSV file.
      * `df`: (DataFrame) The DataFrame containing the source data.
   3. **Process**:
      * Fetch the columns containing date values by calling `fetchDateColumns` function with `dbname` and `tablename`.
      * Extract the first and second date columns (assumed to be `date\_column1` and `date\_column2`).
      * Apply the `date\_Formats\_Vault` function to each value in `date\_column1` and `date\_column2` to detect their date formats.
      * Store the detected formats in two new columns: `date\_Format1` and `date\_Format2`.
      * Write the DataFrame to the specified `output\_file\_Path`, including the original date columns and the format columns.
      * Read the resulting CSV into a DataFrame (`df\_Grp`).
      * Count how many values in `date\_Format1` and `date\_Format2` contain `'unknown date format'`.
      * Calculate the total number of rows with bad/unknown date formats.
      * Return the count of bad/unknown date formats.

**Example**:

```python

total\_bad\_data = dateFormatFinder('/path/to/input', '/path/to/output', 'invoice\_db', 'invoice\_table', '|', df)

**Output**:

If the data contains unknown date formats:

```python

total\_bad\_data = 5

This means 5 rows in the file have unrecognized date formats.

**Summary of Functions:**

* `date\_Formats\_Vault`: Checks the format of a date string from a predefined set of date formats.
* `dateFormatFinder`: Applies the date format validation to two date columns in the DataFrame, writes the results to a CSV file, and returns the count of unknown/bad date formats.

These functions help validate date formats in a CSV file, identify bad data, and save the result for further analysis.

1. **Function Name:` dupCheckInv `:**
   1. **Purpose:** This function checks for duplicate rows in a DataFrame based on the `hashKey` column and returns the count of duplicates.
   2. **Input**:
      * `df`: A pandas DataFrame containing a `hashKey` column which is used to identify duplicates.
   3. **Process**:
      * Identify Duplicates:
        1. Use the `duplicated()` method on the `hashKey` column to identify duplicate rows.
        2. Create a new DataFrame `df2` that only contains the duplicate rows based on `hashKey`.

* count Duplicates:
  1. Calculate the number of duplicate rows by getting the length (`len()`) of the `df2` DataFrame.
* Return:
  1. Return the count of duplicate rows (`count\_dup`).

**Example**:

```python

data = {

'hashKey': ['abc123', 'def456', 'abc123', 'ghi789']

}

df = pd.DataFrame(data)

count\_dup = dupCheckInv(df)

Input DataFrame:

| hashKey |

|---------|

| abc123 |

| def456 |

| abc123 |

| ghi789 |

**Output**:

```python

count\_dup = 1

Explanation: One duplicate row (`abc123`) is found based on the `hashKey`.

**Summary:**

* The function `dupCheckInv` checks for duplicates in the `hashKey` column of a DataFrame and returns the number of duplicate rows.

1. **Class: `FileSizeProcessor`**
   1. **Purpose:**This class processes all the files in a given directory, checks their sizes, and returns the size for each file (or 0 for files smaller than 500 KB).
   2. **Attributes:**
      * `dir\_path` (string): The path to the directory where the files are located.
   3. **Method 1: `\_\_init\_\_(self, dir\_path)`**
      * Purpose: Initialize the `FileSizeProcessor` class with the directory path.

* Input:
  + - `dir\_path`: The directory path containing the files.
* Process:
  + - Store the directory path in the class attribute `self.dir\_path`.
  1. **Method 2: `checkFileSize(self, file\_path)`**
     + Purpose: Check the file size and return 0 if the file size is less than 500 KB, otherwise return the actual size.
* Input:
  + - `file\_path`: Path of the file to check.
* Process:
  + - Get the size of the file using `os.path.getsize(file\_path)`.
    - If the file size is greater than 500 KB (500 \* 1024 bytes), return the actual size. Otherwise, return 0.
* Output:
  + - Return 0 if the file size is less than 500 KB, else return the actual file size in bytes.
  1. **Method 3: `processFile(self)`**
     + Purpose: Process all the files in the directory and return their sizes.
     + Process:
  + Initialize an empty list called `results` to store the file sizes.
  + Iterate over all files in the directory using `os.listdir(self.dir\_path)`.
  + For each file, construct its full path using `os.path.join(self.dir\_path, file\_name)`.
  + Check if it is a file (not a directory) using `os.path.isfile(file\_path)`.
  + Call `checkFileSize(file\_path)` to get the file size and append the result to the `results` list.
    - * Output:

Return a list of file sizes (0 for files smaller than 500 KB).

**Example Input and Output:**

Example Directory:

dir\_path = "/path/to/directory"

Files in directory:

1. file1.txt (600 KB)

2. file2.txt (300 KB)

3. file3.txt (700 KB)

**Example Usage:**

```python

processor = FileSizeProcessor("/path/to/directory")

results = processor.processFile()

Expected Output:

[614400, 0, 716800]

Explanation:

file1.txt is 600 KB -> 614400 bytes

file2.txt is 300 KB -> 0 (because it is less than 500 KB)

file3.txt is 700 KB -> 716800 bytes

**Summary**

* The `FileSizeProcessor` class checks file sizes for all files in a directory.
* If a file is smaller than 500 KB, it returns `0`. Otherwise, it returns the actual file size in bytes.

1. **Function Name:` nullCheckInv `:**
   1. **Purpose:** This function checks for rows in a DataFrame where specific columns contain only null values. It returns the count of such rows.
   2. **Input**:
      * `df` (DataFrame): The input DataFrame containing the data to be validated.
      * `qd\_dbname` (string): The name of the database.
      * `dq\_tablename` (string): The name of the table in the database.
   3. **Process**:
      * Fetch Columns with Null Checks:
        1. Call `fetchNullColumns(qd\_dbname, dq\_tablename)` to retrieve the list of column names that need to be checked for null values.
        + Check for Rows with All Null Values:
        1. Use `df[columsCheck].isnull().all(axis=1)` to create a boolean column named `'allCols'`. This column will be `True` for rows where all values in the specified columns are `null`, and `False` otherwise.
        + Filter Rows with All Null Values:
        1. Filter the DataFrame `df` to keep only rows where `'allCols'` is `True` (i.e., rows where all specified columns are null). Store these rows in a new DataFrame `nullCnt`.
        + Count the Null Rows:
        1. Count the number of rows in `nullCnt` using `len(nullCnt)` and store the result in `count\_true`.
        + Return the Result:
        1. Return the count of rows (`count\_true`) where all specified columns are null.
        + Output:
        1. `count\_true` (integer): The count of rows where all specified columns contain null values.

**Example Input and Output:**

Example DataFrame (`df`):

```plaintext

| ID | Column1 | Column2 | Column3 |

|----|---------|---------|---------|

| 1 | 10 | null | 5 |

| 2 | null | null | null |

| 3 | 20 | null | 15 |

| 4 | null | null | null |

Example Database and Table:

- `qd\_dbname`: `"sample\_db"`

- `dq\_tablename`: `"sample\_table"`

Assume `fetchNullColumns(qd\_dbname, dq\_tablename)` returns `['Column1', 'Column2', 'Column3']`.

Function Call:

```python

count\_true = nullCheckInv(df, "sample\_db", "sample\_table"

Expected Output:

```plaintext

count\_true = 2

**Summary:**

- The function checks specific columns for rows where all values are `null`.

- It returns the count of rows where all values in the specified columns are null.

1. **Class: `schemaValidator`**
   1. **Purpose:**The class validates the schema of a file against a schema stored in a database. It fetches the schema from the database, reads the file header, and compares them.
   2. Inputs:
      * `dbUrl` (string): The URL of the database connection.
      * `schemaTable` (string): The name of the table that contains the schema information.
      * `tableName` (string): The specific table name in the schemaTable for which the schema should be fetched.
      * `filePath` (string): The path of the file to be validated.
      * `delimiter` (string): The delimiter used in the file to separate columns (e.g., `','` for CSV).
   3. **Methods:**
   4. **`\_\_init\_\_(self, dbUrl)`:**
      1. Purpose: Initializes the class with a database connection using the provided `dbUrl`.
      2. Arguments:
         1. `dbUrl`: URL of the database
      3. Process:
         1. Creates a database engine and session.
         2. Loads metadata for the schema.
   5. **`getSchemaFromDb(self, schemaTable, tableName)`:**
      1. Purpose: Fetches the schema (column names) from the database for the given table.
      2. Arguments:
         1. `schemaTable`: The table in the database where schema information is stored.
         2. tableName`: The specific table name to fetch the schema for.
      3. Process
         1. Queries the database for the schema of the table.
         2. If a schema is found, it splits the column names by delimiter and returns them.
         3. If no schema is found or an error occurs, returns `None`.
         4. Returns: A list of column names or `None` if an error occurs.
   6. getFileHeader(self, filePath, delimiter)`:
      1. Purpose: Reads the header (first row) of the file to extract column names.
      2. Arguments:
         1. `filePath`: The path of the file to read.
         2. `delimiter`: The delimiter used in the file.
      3. Process:
         1. - Reads the file using pandas with only the header row.
         2. - Extracts the column names and returns them.
         3. - Returns: A list of column names or an empty list if an error occurs.
   7. **`validateSchema(self, schemaTable, tableName, filePath, delimiter)`:**
      1. Purpose: Validates the schema of the file against the schema fetched from the database.
      2. Arguments:
         1. `schemaTable`: The table where schema information is stored.
         2. `tableName`: The table name in the schema table.
         3. `filePath`: The path of the file to validate.
         4. `delimiter`: The delimiter used in the file.
      3. Process:
         1. Fetches the schema from the database using `getSchemaFromDb()`.
         2. Fetches the file header using `getFileHeader()`.
         3. Compares the schema and header using `compareSchemaAndHeader()`.
         4. Returns: A boolean indicating whether the schema matches and a report detailing any discrepancies.
   8. **`compareSchemaAndHeader(self, schema, header)`**
      1. Purpose: Compares the schema from the database with the header of the file.
      2. Arguments:
         1. `schema`: List of column names fetched from the database.
         2. `header`: List of column names from the file.
      3. Process:
         1. Compares if both lists are exactly the same.
         2. If not, identifies missing and extra columns.
         3. Returns a detailed report of the comparison.
         4. Returns: A boolean indicating if the schema matches and a string report.

**Example Input and Output**

**Example Input:**

```python

dbUrl = "mysql://username:password@localhost:3306/mydb"

schemaTable = "table\_schemas"

tableName = "invoices"

filePath = "/path/to/invoice\_data.csv"

delimiter = ","

Example Call:

```python

validator = schemaValidator(dbUrl)

is\_valid, report = validator.validateSchema(schemaTable, tableName, filePath, delimiter)

print(is\_valid)

print(report)

Expected Output:

- If schema matches:

```plaintext

True

Schema matches the file header.

If schema doesn't match:

```plaintext

False

Schema does not match the file header.

Missing columns: column1, column2

Extra columns: columnA

**Summary:**

- This code validates a file's schema against a predefined schema in a database.

- It checks for missing or extra columns and provides detailed reports for any mismatches.

1. **Class: `** **DataQualityChecker `**
   1. **Purpose:**This class is responsible for loading data from a file, validating the data for issues (such as invalid dates, duplicates, and nulls), calculating the percentage of bad data, and reporting the status of the data quality check.
   2. Attributes:
      * `dbname` (string): Database name to connect.
      * `tablename` (string): Name of the table to validate against.
      * `input\_file\_path` (string): Path to the input data file.
      * `output\_file\_path` (string): Path to save output files.
      * `delimiter` (string): Delimiter used in the input file (e.g., comma, pipe).
      * `start\_time` (float): Time when the data quality check started.
      * `fileName` (string): Name of the input file.
   3. **Methods:**
      * **`\_\_init\_\_(self, dbname, tablename, input\_file\_path, output\_file\_path, delimiter)`:**
        1. Purpose: Initializes the `DataQualityChecker` with the provided parameters.
        + Arguments:
        1. `dbname`: Name of the database.
        2. tablename`: Name of the table to validate.
        3. input\_file\_path`: Path of the input data file.
        4. output\_file\_path`: Path for output files.
        5. `delimiter`: Delimiter for the data file.
        6. Process: Sets the class attributes and records the start time.
2. **load\_data(self)`:**
   * Purpose: Loads the data from the specified input file.
   * Process:
     1. Fetches column definitions and data types using `columnsDefnition`.
     2. Reads the source file into a DataFrame using `readSourceFile`.
     3. Counts the number of records in the loaded DataFrame.
     4. Returns: None (updates class attributes).
3. `validate\_data(self)`:
   * Purpose: Validates the loaded data for various quality checks.
   * Process:
     1. Calls `dateFormatFinder` to check for invalid dates and records the count.
     2. Calls `dupCheckInv` to check for duplicates and records the count.
     3. Calls `nullCheckInv` to check for null values and records the count.
     4. Returns: None (updates class attributes).
4. `bad\_data\_quality(self)`:
   * Purpose: Calculates and prints statistics about the bad data quality.
   * Process:
     1. Calculates total bad data by summing invalid counts (dates, duplicates, nulls).
     2. Calculates the percentage of bad data relative to total records.
     3. Prints the count and percentage of bad data.
     4. Returns: None.
5. `get\_status(self)`:
   * Purpose: Returns a summary of the data quality check status.
   * Process:
     1. Records the end time and calculates elapsed time.
     2. Constructs a status dictionary with relevant information (start time, end time, record counts).
     3. Returns: A dictionary with the status of the data quality check.
6. `run(self)`:
   * Purpose: Executes the complete data quality checking process.
   * Process:
     1. Calls `load\_data` to load data from the file.
     2. Calls `validate\_data` to perform data validation.
     3. Calls `bad\_data\_quality` to assess the quality of the data.
     4. Calls `get\_status` to gather the final status report.

* Returns: A status dictionary summarizing the results of the data quality check.

**Example Input and Output:**

Example Input:

```python

dbname = "my\_database"

tablename = "invoices"

input\_file\_path = "/path/to/invoices.csv"

output\_file\_path = "/path/to/output.csv"

delimiter = ","

data\_quality\_checker = DataQualityChecker(dbname, tablename, input\_file\_path, output\_file\_path, delimiter)

status = data\_quality\_checker.run()

**Expected Output:**

- Console Output:

```plaintext

Invalid date count: 5

Duplicate data count: 3

Null data count: 2

Bad data count: 10

Total record count: 100

Bad data percentage: 10.0

Status Dictionary:

```plaintext

{

"start\_date": "2024-09-26 10:00:00",

"end\_date": "2024-09-26 10:01:00",

"source\_file\_count": 100,

"target\_file\_count": 90,

"status": "Completed"

}

**Summary:**

This pseudo code provides a comprehensive overview of the `DataQualityChecker` class, detailing how it initializes, loads data, validates it, calculates bad data metrics, and returns a status summary. The input and output examples illustrate how to use the class in practice.

# **SRC.** data\_profiling**:**

* 1. **Class: `DataProfiler`**
     1. **Purpose:**This class profiles and analyzes data from delimited files, performing data quality checks, generating profiling metrics, and storing the results in a database.
     2. Attributes:
* `filepath` (string): Path to the input data file.
* `file\_type` (string): Type of the file (e.g., 'AP', 'PO', 'Invoice').
* `separator` (string): Delimiter used in the file (default is "|").
* `db\_url` (string): Database connection URL.
* `df` (DataFrame): Data loaded from the input file.
* `engine` (SQLAlchemy engine): Database engine for database operations.
* `file\_name` (string): Name of the input file.
* `column\_mapping` (dictionary): Mapping of columns fetched from the database.
  1. Methods:
     1. `\_\_init\_\_(self, filepath: str, file\_type: str, separator: str = "|", db\_url: str = '')`:
        1. Purpose: Initializes the `DataProfiler` object with the file path, file type, separator, and database URL.
        2. Process:
           1. Loads the data into a DataFrame.
           2. Initializes the database engine if a URL is provided.
           3. Loads column mappings from the database using `load\_column\_mappings`.
  2. `load\_column\_mappings(self) -> Dict[str, Dict[str, str]]`:
     1. Purpose: Loads column mappings from the database.
     2. Process:
        + 1. Executes a SQL query to fetch column mappings for the specified file type.
          2. Organizes the mappings into a dictionary.
          3. Returns: A dictionary containing column mappings for the file type.
  3. `data\_profiling(self) -> pd.DataFrame`:
     1. Purpose: Generates a DataFrame with basic profiling information.
     2. Process:
        + 1. Creates a DataFrame containing the file name, column names, data types, and nullability information.
          2. Returns: DataFrame with profiling information.
  4. `profile\_data(self) -> pd.DataFrame`:
     1. Purpose: Profiles the data and generates summary statistics.
     2. Process:
        + 1. Validates that the file type exists in the column mapping.
          2. Converts specified date columns to datetime format.
          3. Calculates metrics such as total rows, total columns, distinct dates, and total spend.
          4. Returns: DataFrame with profiling results.
  5. `data\_quality\_metrics(self) -> pd.DataFrame`:
     1. Purpose: Computes data quality metrics for each column.
        1. Process:
        2. Iterates through column mappings to calculate null counts and unique counts for each column.
        3. Returns: DataFrame with data quality metrics.
  6. `store\_in\_db(self, df: pd.DataFrame, table\_name: str) -> None`:
     1. Purpose: Stores the given DataFrame in the specified database table.
     2. Process:
        1. Uses the SQLAlchemy engine to save the DataFrame to the database.
        2. Returns: None.
  7. run\_all\_profiles(self) -> Dict[str, Optional[int]]`:
     1. Purpose: Runs all profiling methods and stores results in the database.
     2. Process:
        1. Calls `data\_profiling`, `profile\_data`, and `data\_quality\_metrics`.
        2. Stores results in the database using `store\_in\_db`.
        3. Returns a summary of profiling operations.
     3. Returns: A dictionary with profiling results.
  8. Function: `process\_directory(directory\_path: str, file\_type: str, delimiter: str, db\_url: str) -> Dict[str, Dict[str, Optional[int]]]`
     1. Purpose:Processes all files in a specified directory, applying data profiling and logging results.
     2. Arguments:
        1. `directory\_path` (string): Path to the directory containing the files.
        2. `file\_type` (string): Type of files to process (e.g., 'AP', 'PO', 'Invoice').
        3. `delimiter` (string): Delimiter used in the files.
        4. `db\_url` (string): Database connection URL for storing results.
     3. Returns: A dictionary with profiling results for each processed file.
     4. Process:
        1. Validate that the specified directory exists.
        2. List all files in the directory.
        3. For each file:
           1. Initialize a `DataProfiler` object.
           2. Run the profiling methods and store results.
           3. Log any errors that occur during processing.

**Example Input and Output:**

Example Input:

```python

directory\_path = "/path/to/data"

file\_type = "Invoice"

delimiter = "|"

db\_url = "mysql+pymysql://user:password@localhost/mydatabase"

results = process\_directory(directory\_path, file\_type, delimiter, db\_url)

Expected Output:

Console Output:

```plaintext

Processing file: /path/to/data/invoice\_data\_1.csv

Successfully processed and logged file: /path/to/data/invoice\_data\_1.csv

Processing file: /path/to/data/invoice\_data\_2.csv

Successfully processed and logged file: /path/to/data/invoice\_data\_2.csv

Results Dictionary:

```python

{

"invoice\_data\_1.csv": {

"start\_time": "2024-09-26 10:00:00",

"end\_time": "2024-09-26 10:01:00",

"file\_name": "invoice\_data\_1.csv",

"source\_count": 100,

"target\_count": 100,

"rejection\_count": 0,

"job\_status": "Success"

},

"invoice\_data\_2.csv": {

"start\_time": "2024-09-26 10:05:00",

"end\_time": "2024-09-26 10:06:00",

"file\_name": "invoice\_data\_2.csv",

"source\_count": 150,

"target\_count": 150,

"rejection\_count": 0,

"job\_status": "Success"

}

}

**Summary:**

This pseudo code provides a structured overview of the `DataProfiler` class and the `process\_directory` function, detailing their operations and logic. The input and output examples illustrate how to use the class and function to perform data profiling on files in a directory.