**ETL - Design Specification**

**V2.5.7.1**

|  |  |
| --- | --- |
| **Author** |  |
| **Created On** | 18-September-2024 |
| **Reviewed By** | Dr. Srinivasa Vegi |
| **Reviewed On** |  |

**Change Record**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Version No** | **Date** | **Sections Affected** | **Reason/Remarks** | **Author** |
| v1.0 | 22-Feb-2021 |  | Initial version | Bhagyaram S |
| v2.0 | 29-Sep-2021 |  | Update | Bhagyaram S |
| v2.1 | 31-Oct-2022 |  | Update | Bhagyaram S |
| v2.5 | 16-Nov-2022 |  | Update | Bhagyaram S |
| v2.5.5 | 20-Oct-2023 |  | Sep 2023 Release | Bhagyaram S |
| v2.5.7 | 05-Apr-2023 |  | Mar 2024 Release | Bhagyaram S |
| V2.5.7.1 | 12-Jun-2023 |  | June 2024 Release | Bhagyaram S |

# Description

## Purpose

This document describes the software design workflow for the implementation of data extraction from Invoice & PO and AP Files.

## Scope

This describes the software design decisions, data extraction techniques & algorithms and the various processes in the ETL workflow for the implementation of data extraction from Invoice & PO and AP Files.. The intended audience of this document is the developers, designers, and software testers of GPO Health team.

|  |  |  |
| --- | --- | --- |
| **Step** | **Process** | **Description** |
| **1** | **ETL – Extract, Transform & Load** | The overall process comprises of the below 6 stages for structured data: 1. **Extraction** - Extract the data in structured raw format from the various source systems - forms uploaded into the cloud, secure server, directly connected to client base.  2. **Profiling** - The next step uses data profiling tools and techniques to make an initial assessment of the data to understand its overall quality challenges and anomalies.  3. **DataQuality**- Applying various DQ logics on Source data and capture the quantity of Bad data. 4. **Transformation** - Transform the data into the required pre-defined format/schema as recognized by the repository.  6. **Load** – Pass on the transformed data to Mysql Database.  Structured data is directly consumed from the database. |
| **2** | **Audit & Logging** | Proper auditing and Logging is implemented in various stages for measuring quality and performance of ETL code. |
| **5** | **Quality Monitoring** | Data quality is as an ongoing process to track the of data quality over time to avoid data decay, which fundamentally occurs overtime due to changes not being captured or system upgrades. Typically, the ongoing process is to execute the data quality check module on a scheduled basis (ideally after inception of a new vendor/bi-weekly/monthly) to auto-correct the variations based on pre-defined business rules. |

# Work flow

## L1 Area to L2 processing - Structured Data Extraction

1. Structured data is extracted from the L1 layer Data ETL workflow – Data Extraction, profiling, transformation and output generation
2. Capturing delta customer name and Load date between l1 table and l2 audit table and extracting data from l1 table for each customer and load date wise.
3. Reading metadata details like source and target tables and databases etc., from prod\_context\_l2\_l3 table.
4. Reading L2 queries from Querymetadata\_context\_l2 table one by one and passed to ETL workflow Python code.
5. This Python code will execute queries one by one and update the audit table.

2.2. L2 Area to L3 processing - Structured Data Extraction

1. Structured data is extracted from the L2 layer Data ETL workflow – Data Extraction, profiling, transformation and output generation

2. Capturing delta customer name and Load date between l2 table and l3 audit table and extracting data from l2 table for each customer and load date wise.

3. Reading metadata details like source and target tables and databases etc., from prod\_context\_l2\_l3 table.

4. Reading L2 queries from Querymetadata\_context\_l3 table one by one and passed to ETL workflow Python code.

5. This Python code will execute queries one by one and update the audit table.

2.3. L3 Area to L4 processing - Structured Data Extraction

1. Structured data is extracted from the l3 layer Data ETL workflow – Data Extraction, profiling, transformation and output generation

2. Capturing delta customer name and Load date between l3 table and l4 audit table and extracting data from l2 table for each customer and load date wise.

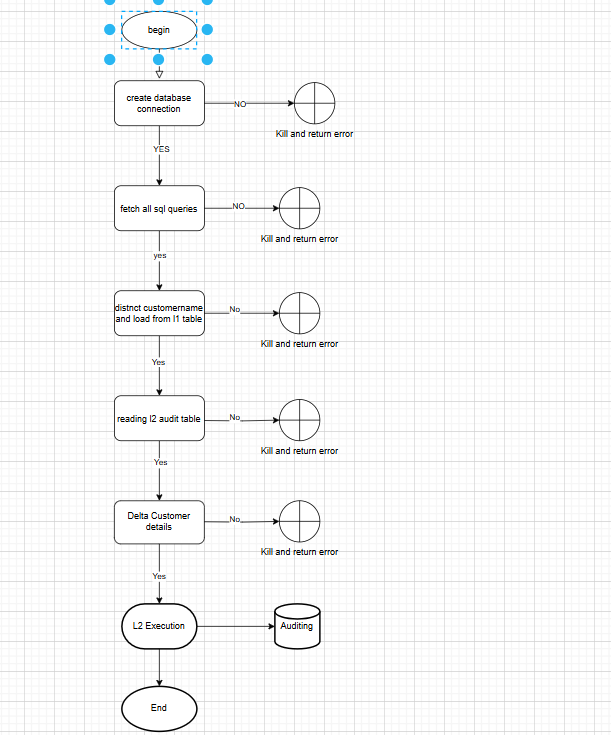
3. Reading metadata details like source and target tables and databases etc., from prod\_context\_l2\_l3 table.

4. Reading L2 queries from Querymetadata\_context\_l3 table one by one and passed to ETL workflow Python code.

5. This Python code will execute queries one by one and update the audit table.

# L1 Area to L2 Area Processing - structured Data Extraction

**FLOW CHART:**



For processing l2 job we must import required libraries. Below are the list of libraries were imported.

**Import necessary libraries:**

* import pandas as pd:
  + Import the pandas library, commonly used for data manipulation and analysis
* from sqlalchemy import create\_engine, text:
  + Import create\_engine for establishing database connections and text for creating SQL text object.
* from sqlalchemy.orm import sessionmaker:
  + Import sessionmaker for creating new session objects to interact with the database.
* from sqlalchemy.exc import SQLAlchemyError:
  + Import SQLAlchemyError for handling exceptions related to SQLAlchemy operations.
* import time:
  + Import the time module for time-related functions (e.g., delays).
* import os:
  + Import the os module for interacting with the operating system (e.g., file paths, environment variables).

## databaseHandler(class)

* This is the initiation of L2 load and defining audit columns.
* In this class definition, below sequence of steps are defined.
* Initiate the self method by connecting to database with URL
* Defining Audit columns and create table.
* Table('l2\_audit\_data', self.metadata,  
   Column('Job\_ID', String(250)),  
   Column('Job\_Name', String(250)),  
   Column('Customer\_Name', String(500)),  
   Column('Flow\_Type', String(250)),  
   Column('Operation\_type', String(120)),  
   Column('Source\_Count', Integer),  
   Column('Target\_count', Integer),  
   Column('New\_rcrd\_count', Integer),  
   Column('update\_rcrd\_count', Integer),  
   Column('Start\_Time', DateTime),  
   Column('End\_Time', DateTime),  
   Column('Status', String(250)),  
   Column('Insert\_By', String(250)),  
   Column('Error\_desc', String(500)),  
   Column('L1\_load\_date', Date),  
   Column('L2\_load\_date', DateTime))

## insertL2AuditEntry

* **Try** to perform the following:
  + Print the message: "Inserting L2 audit entry: " followed by the value of l2AuditEntry.
  + Insert the l2AuditEntry into the database table l2AuditTable.
  + Execute the insertion using the current database session.
  + Commit the transaction to save the changes.
  + Print the message: "L2 audit entry committed."
* **Catch** any exceptions that occur:
  + Print the message: "Error inserting L2 audit record: " followed by the error message.
  + Rollback the transaction to undo any uncommitted changes.

**Function close(self)**

* Print the message: "Closing the session and connection..."
* Close the active database session.
* Close the active database connection

## create\_connection.

* **Description:** This function creates a database connection using the databaseHandler class.
* **Input:** dbUrl – The database URL required to establish a connection.
* **Try** to perform the following:
  + Instantiate an object of databaseHandler class with dbUrl as the argument.
  + Print the message: "Database connection established."
  + **Return** the database handler object.
* **Catch** an SQLAlchemyError if it occurs:
  + Print the message: "Error creating database connection: " followed by the error message.
  + **Raise** the error again to notify of the failure.

## read\_metadata(l2MetadataTable, l2metadb, engine)

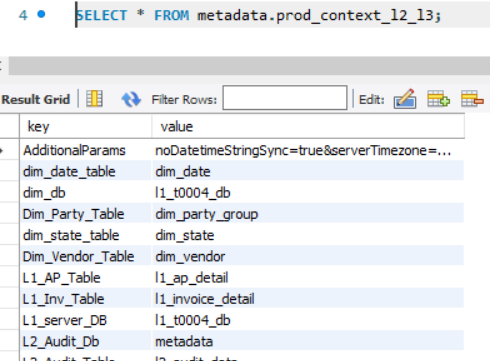
* + - Input:
* l2MetadataTable: Name of the metadata table.
* l2metadb: Database name where the metadata table resides.
* engine: Database engine used for querying.
* **Try** to perform the following:
  + Create a query string l2\_metaquery:
    - "***SELECT \* FROM {l2metadb}.{l2MetadataTable}***" to select all data from the specified table.
* Use pandas.read\_sql\_query to execute the SQL query on the engine and store the result in df\_metadata.
* Print the message: "Metadata successfully retrieved from the database."
* Convert the df\_metadata dataframe to a dictionary, setting the key column as the dictionary key and the value column as the value, then return this dictionary.
* **Catch** an SQLAlchemyError if it occurs:
  + Print the message: "Error reading metadata: " followed by the error message.
  + Raise\*\* the error again to notify of the failure.

Eg: {‘ dim\_date\_table’:’ dim\_date’,’ dim\_db’:’ l1\_t0004\_db’}

**Table Structures:**

* **prod\_context\_l2\_l3:**

|  |  |
| --- | --- |
| Column Name | Datatpe |
| key | String |
| value | String |

* **Sample Data**
* 

**Example:**

**Python:**

l2MetadataTable = "metadata"

l2metadb = "l2db"

engine = some\_database\_engine

**Output Example 1:**

Metadata successfully retrieved from the database.

**Returned Value:**

{

'key1': 'value1',

'key2': 'value2',

'key3': 'value3'

}

**Input Example 2 (with error):**

**python**

l2MetadataTable = "non\_existent\_table"

l2metadb = "l2db"

engine = some\_database\_engine

**Output Example 2 (error case):**

Error reading metadata: Table 'l2db.non\_existent\_table' does not exist (or other SQLAlchemyError message)

**Returned Value:**

The error is raised, terminating the function

## 5. fetch\_all\_sql\_queries(session, query\_meta\_table, ctxArea)

* Input:
  + session: The database session used to execute SQL queries.
  + query\_meta\_table: Name of the table containing SQL query metadata.
  + ctxArea: A filter criterion for the ctxarea column in the query\_meta\_table.
* Try to perform the following:
  + Execute the following SQL query using the provided session:

**Sql**

***SELECT ctxval, remarks FROM {query\_meta\_table} WHERE ctxarea='{ctxArea}' ORDER BY ctxkey***

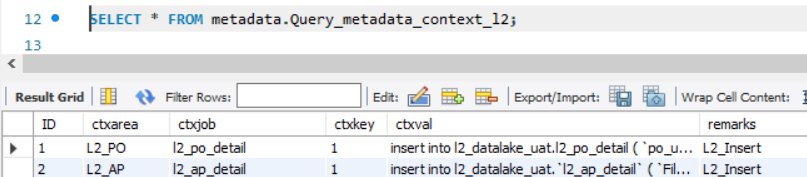
* Store the query results in result.
* Print the message: "SQL queries successfully fetched from the metadata table."
* Return all rows from the query result as a list of tuples using result.fetchall().
* **Catch** an SQLAlchemyError if it occurs:
  + - * Print the message: "Error fetching SQL queries: " followed by the error message.
      * Raise the error again to notify of the failure

**Table Structures:**

* **Query\_meta\_table:** Query\_metadata\_context\_l2

|  |  |
| --- | --- |
| Column Name | Datatpe |
| ID | int (primary key) |
| Ctxarea | varchar(100) |
| Ctxjob | varchar(100) |
| Ctxkey | varchar(100) |
| Ctxval | mediumtext, |
| Remarks | varchar(255) |

* **Sample data:**



**Input Example 1:**

python

session = some\_database\_session

query\_meta\_table = "query\_metadata"

ctxArea = "L2\_invoice"

**Output Example 1:**

SQL queries successfully fetched from the metadata table.

**Returned Value:**

Python:

[

('SELECT \* FROM finance\_table', 'Query for finance-related data'),

('SELECT balance FROM account\_table', 'Query for account balances'),

]

**Input Example 2 (with error):**

Python:

session = some\_database\_session

query\_meta\_table = "non\_existent\_table"

ctxArea = "L2\_invoice"

**Output Example 2 (error case):**

Error fetching SQL queries: Table 'non\_existent\_table' does not exist (or other SQLAlchemyError message)

**Returned Value:**

The error is raised, terminating the function.

## 6. Function read\_l1\_inv\_table(l1InvTable, l1dbname, engine):

* **Input**:
  + - * `l1InvTable`: Name of the L1 Invoice table.
      * `l1dbname`: Name of the database where the table resides.
      * `engine`: Database engine used for querying.
* **Try** to perform the following:
  + Construct a SQL query string to retrieve distinct values for `invhdrNameCustomer` (renamed as `Customer\_Name`) and `load\_date` (renamed as `L1\_load\_date`) from the `l1InvTable` in the specified database `l1dbname`.
  + Use `pandas.read\_sql\_query` to execute the SQL query on the `engine` and store the result in a dataframe `df`.
  + Print the message: "L1 Invoice table data successfully retrieved."
  + Return the dataframe `df`.
* **Catch** an `SQLAlchemyError` if it occurs:
  + Print the message: "Error reading L1 Invoice table: " followed by the error message.
  + Raise the error again to notify of the failure.
* **Input Example 1:**
  + python
  + l1InvTable = "invoice\_table"
  + l1dbname = "invoicedb"
  + engine = some\_database\_engine
* **Output Example 1:**

L1 Invoice table data successfully retrieved.

* **Returned Value:**

| Customer\_Name | L1\_load\_date |

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

| ABC Corp | 2024-09-20 |

| XYZ Inc. | 2024-09-21 |

| DEF Ltd. | 2024-09-22 |

* **Input Example 2 (with error):**
* Python:

l1InvTable = "non\_existent\_table"

l1dbname = "invoicedb"

engine = some\_database\_engine

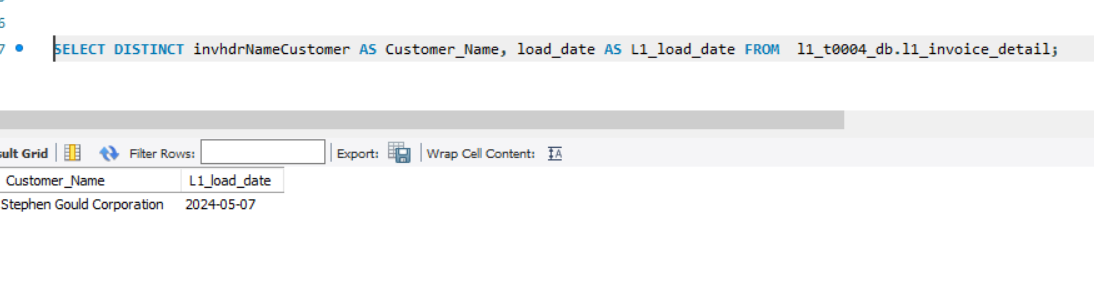
* **Output Example 2 (error case):**

Error reading L1 Invoice table: Table 'invoicedb.non\_existent\_table' does not exist (or other SQLAlchemyError message)

* **Returned Value:**

The error is raised, terminating the function.

**Example:**



## 7. Function read\_l2\_audit\_table(l2AuditTable, l2dbname, engine):

* **Input:**
  + l2AuditTable`: Name of the L2 Audit table.
  + `l2dbname`: Name of the database where the table resides.
  + `engine`: Database engine used for querying.
* **Try** to perform the following:
  + Construct a SQL query string to retrieve distinct values for `Customer\_Name` and `L1\_load\_date` from the `l2AuditTable` in the specified database `l2dbname`.
  + Use `pandas.read\_sql\_query` to execute the SQL query on the `engine` and store the result in a dataframe `df`.
  + Print the message: "L2 Audit table data successfully retrieved."
  + Return\*\* the dataframe `df`.
* **Catch** an `SQLAlchemyError` if it occurs:
  + Print the message: "Error reading L2 Audit table: " followed by the error message.
  + Raise the error again to notify of the failure.
* **Input Example 1:**
  + python
  + l2AuditTable = " audit\_table "
  + l2dbname = "invoicedb"
  + engine = some\_database\_engine
* **Output Example 1:**

L2 Audit table data successfully retrieved.

* **Returned Value:**

| Customer\_Name | L1\_load\_date |

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

| ABC Corp | 2024-09-20 |

| XYZ Inc. | 2024-09-21 |

| DEF Ltd. | 2024-09-22 |

* **Input Example 2 (with error):**
* Python:

l2AuditTable = "non\_existent\_table"

l2dbname = "invoicedb"

engine = some\_database\_engine

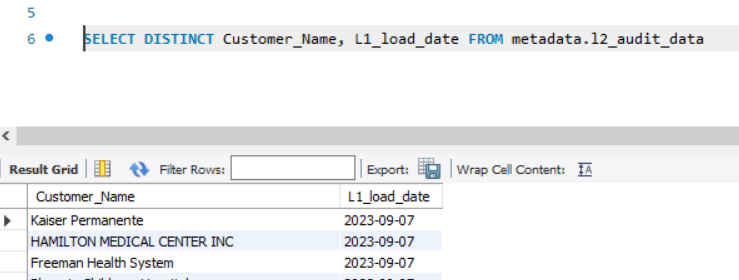
* **Output Example 2 (error case):**

Error reading L2 audit table: Table 'invoicedb.non\_existent\_table' does not exist (or other SQLAlchemyError message)

* **Returned Value:**

The error is raised, terminating the function.

**Example:**



## 8.Function `delta\_cust\_details(df1, df2)`

* **Input:**
  + `df1`: DataFrame representing L1 data.
  + `df2`: DataFrame representing L2 data.
* **Try** to perform the following:
  + Merge `df1` and `df2` on the columns `Customer\_Name` and `L1\_load\_date` using a left join. Use the parameter `indicator=True` to indicate which rows exist only in `df1`, only in `df2`, or both.
  + Filter the merged DataFrame to keep only the rows where `\_merge` is `'left\_only'` (i.e., rows present in `df1` but not in `df2`).
  + Drop the `\_merge` column from the resulting DataFrame `delta\_df`.
  + Print the message: "Delta between L1 and L2 data successfully calculated."
  + Return the DataFrame `delta\_df` containing the differences.
* **Catch** any generic `Exception` that occurs:
  + Print the message: "Error calculating delta: " followed by the error message.
  + Raise\*\* the error again to notify of the failure.

**Input Example 1:**

**python**

* df1 = pd.DataFrame({

'Customer\_Name': ['ABC Corp', 'XYZ Inc.', 'DEF Ltd.'],

'L1\_load\_date': ['2024-09-20', '2024-09-21', '2024-09-22']

})

* df2 = pd.DataFrame({

'Customer\_Name': ['XYZ Inc.', 'DEF Ltd.'],

'L1\_load\_date': ['2024-09-21', '2024-09-22']

})

**Output Example 1:**

Delta between L1 and L2 data successfully calculated.

**Returned Value:**

| Customer\_Name | L1\_load\_date |

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

| ABC Corp | 2024-09-20 |

**Input Example 2 (with error):**

**python**

df1 = "invalid\_dataframe"

df2 = pd.DataFrame({

'Customer\_Name': ['XYZ Inc.', 'DEF Ltd.'],

'L1\_load\_date': ['2024-09-21', '2024-09-22']

})

**Output Example 2 (error case):**

Error calculating delta: Can only merge DataFrame objects, but received invalid input (or other error message)

**Returned Value:**

The error is raised, terminating the function.

## 9. Function `l2Audit\_cnt(srcDb, targetDb, srcTable, targetTable, Customer\_Name, LoadDate, engine)`

* **Input:**
  + `srcDb`: Name of the source database.
  + `targetDb`: Name of the target database.
  + `srcTable`: Name of the source table.
  + `targetTable`: Name of the target table.
  + `Customer\_Name`: The customer name to filter by.
  + `LoadDate`: The load date to filter by.
  + `engine`: Database engine used for querying.
* **Formulate the source query (`src\_qury`):**
  + Create a SQL query to count the number of rows in the `srcTable` from the `srcDb` where `invhdrNameCustomer` matches `Customer\_Name` and `load\_date` matches `LoadDate`.
* **Execute the source query:**
  + Use `pandas.read\_sql\_query` to run `src\_qury` on the `engine` and store the result in `tsrc\_cnt`.
  + Extract the count from the first row and first column of `tsrc\_cnt`, and store it in `src\_cnt`.
* **Formulate the target query (`trgt\_qury`):**
  + Create a SQL query to count the number of rows in the `targetTable` from the `targetDb` where `invhdrNameCustomer` matches `Customer\_Name` and `load\_date` matches `LoadDate`.
* **Execute the target query:**
  + Use `pandas.read\_sql\_query` to run `trgt\_qury` on the `engine` and store the result in `ttrgt\_cnt`.
  + Extract the count from the first row and first column of `ttrgt\_cnt`, and store it in `trgt\_cnt`.
* Return `src\_cnt` and `trgt\_cnt` as the counts from the source and target tables, respectively.

## 10. Function `execute\_sql(session, sql\_query, variables=None)`

* **Input:**
  + `session`: The database session used to execute SQL queries.
  + `sql\_query`: A string containing the SQL query to be executed.
  + `variables`: An optional dictionary of variables to bind to the SQL query.
* **Try** to perform the following:
  + Convert the `sql\_query` string to a SQLAlchemy text object called `query`.
  + Execute the `query` using the session, passing in `variables` (or an empty dictionary if `variables` is `None`).
  + Commit the transaction to save changes to the database.
  + Return the tuple `('SUCCESS', None)` indicating the query executed successfully.
* **Catch** an `SQLAlchemyError` if it occurs:
  + Rollback the session to undo any uncommitted changes.
  + Convert the error to a string and store it in `error\_mesg`.
  + Return the tuple `('FAIL', error\_mesg)` indicating the query failed, along with the error message.

**Input Example 1:**

**python**

session = some\_database\_session

sql\_query = "INSERT INTO users (username, email) VALUES (:username, :email)"

variables = {'username': 'john\_doe', 'email': 'john@example.com'}

**Output Example 1:**

('SUCCESS', None)

**Input Example 2 (with error):**

**python**

session = some\_database\_session

sql\_query = "INSERT INTO users (username, email) VALUES (:username, :email)"

variables = {'username': 'jane\_doe', 'email': 'jane@example.com'}

**Output Example 2 (error case):**

**python**

# Assume the table does not allow duplicates and 'jane\_doe' already exists

('FAIL', 'UNIQUE constraint failed: users.username')

11. Function `l2Execution()`

* **Description**:
  + Main execution function that integrates the database handler and SQL operations.
* **Try** to perform the following:
  + Create a database session using the handler:
  + Set `dbUrl` to connect to the database.
  + Call `create\_connection(dbUrl)` to create `db\_handler`.
* **Read metadata and parameters:**
  + Set `l2MetadataTable` and `meta\_db\_name`.
  + Call `read\_metadata(l2MetadataTable, meta\_db\_name, db\_handler.engine)` to get `meta\_l2\_keys`.
  + Extract values from `meta\_l2\_keys` into appropriate variables.
* **Fetch all SQL queries:**
  + Call `fetch\_all\_sql\_queries(db\_handler.session, query\_meta\_table, ctxArea)` to retrieve `sql\_queries`.
* **Read data from L1 Invoice and L2 Audit tables:**
  + Call `read\_l1\_inv\_table(l1InvTable, l1db\_name, db\_handler.engine)` to get `df1`.
  + Call `read\_l2\_audit\_table(l2AuditTable, l2\_auditdb, db\_handler.engine)` to get `df2`.
* **Calculate delta:**
  + Call `delta\_cust\_details(df1, df2)` to get `df\_delta`.
  + Print `df\_delta`.
* **For each row in `df\_delta`:**
  + Extract `context\_customer` and `context\_l1\_date` into `variables`.
  + For each SQL query in `sql\_queries`:
    - Get the SQL query string.
    - Call `execute\_sql(db\_handler.session, sql\_query, variables)` to execute the query.
    - Call `l2Audit\_cnt(srcDb, targetDb, l1InvTable, targetTable, customer\_name, L1\_load\_date, db\_handler.engine)` to get source and target counts.
    - Create an `audit\_entry` dictionary with relevant details.
    - Call `db\_handler.insertL2AuditEntry(audit\_entry)` to insert the audit entry.
    - Close the database connection using `db\_handler.close()`.
* **Catch** any generic `Exception` that occurs:
  + Print the message: "Error in main execution: " followed by the error message.
  + Raise the error again to notify of the failure.
* **Input Example 1:**

**Python**

# Assuming the following metadata and database configurations

srcDb = "l1\_db"

targetDb = "l2\_db"

l2MetadataTable = "prod\_context\_l2\_l3\_python"

meta\_db\_name = "stg\_tbl"

l1InvTable = "l1\_invoices"

l2AuditTable = "l2\_audit"

* **Output Example 1:**

Delta between L1 and L2 data successfully calculated.

# DataFrame output for df\_delta

Customer\_Name L1\_load\_date

ABC Corp 2024-09-20

XYZ Inc. 2024-09-21

* **Input Example 2 (with error):**

python

# Invalid database connection or missing tables

srcDb = "invalid\_db"

targetDb = "invalid\_db"

* **Output Example 2 (error case):**

Error in main execution: Error connecting to the database (or other specific error message)

* **Returned Value:**

The error is raised, terminating the function.

12. if \_\_name\_\_ == "\_\_main\_\_":

* This is main program execution.
* If the script is executed as the main program the it call the function `l2Execution()`.
* There are no direct inputs for this part of the code; it relies on the execution of `l2Execution()` which handles its own input.

# L2 Area to L3 Area Processing - structured Data Extraction

## 

For processing l3 job we must import required libraries. Below are the list of libraries were imported.

**Import necessary libraries:**

* import pandas as pd:
  + Import the pandas library, commonly used for data manipulation and analysis
* from sqlalchemy import create\_engine, text:
  + Import create\_engine for establishing database connections and text for creating SQL text object.
* from sqlalchemy.orm import sessionmaker:
  + Import sessionmaker for creating new session objects to interact with the database.
* from sqlalchemy.exc import SQLAlchemyError:
  + Import SQLAlchemyError for handling exceptions related to SQLAlchemy operations.
* import time:
  + Import the time module for time-related functions (e.g., delays).
* import os:
  + Import the os module for interacting with the operating system (e.g., file paths, environment variables).

## create\_connection.

* **Description:** This function creates a database connection using the databaseHandler class.
* **Input:** dbUrl – The database URL required to establish a connection.
* **Try** to perform the following:
  + Instantiate an object of databaseHandler class with dbUrl as the argument.
  + Print the message: "Database connection established."
  + **Return** the database handler object.
* **Catch** an SQLAlchemyError if it occurs:
  + Print the message: "Error creating database connection: " followed by the error message.
* **Raise** the error again to notify of the failure.

## read\_metadata

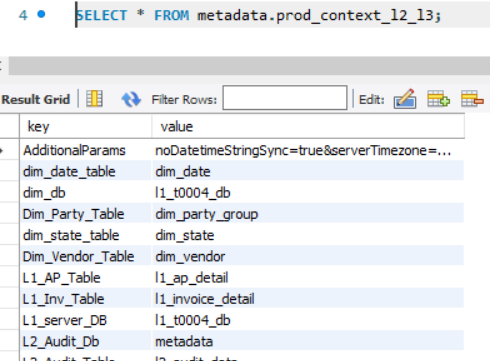
* + - Input:
* L3MetadataTable: Name of the metadata table.
* L3metadb: Database name where the metadata table resides.
* engine: Database engine used for querying.
* **Try** to perform the following:
  + Create a query string l2\_metaquery:
    - "***SELECT \* FROM {l3metadb}.{l3MetadataTable}***" to select all data from the specified table.
* Use pandas.read\_sql\_query to execute the SQL query on the engine and store the result in df\_metadata.
* Print the message: "Metadata successfully retrieved from the database."
* Convert the df\_metadata dataframe to a dictionary, setting the key column as the dictionary key and the value column as the value, then return this dictionary.
* **Catch** an SQLAlchemyError if it occurs:
  + Print the message: "Error reading metadata: " followed by the error message.
  + Raise\*\* the error again to notify of the failure.

Eg: {‘ dim\_date\_table’:’ dim\_date’,’ dim\_db’:’ l1\_t0004\_db’}

**Table Structures:**

* **prod\_context\_l2\_l3:**

|  |  |
| --- | --- |
| Column Name | Datatpe |
| key | String |
| value | String |

* **Sample Data**
* 

**Example:**

**Python:**

L3MetadataTable = "metadata"

L3metadb = "l3db"

engine = some\_database\_engine

**Output Example 1:**

Metadata successfully retrieved from the database.

**Returned Value:**

{

'key1': 'value1',

'key2': 'value2',

'key3': 'value3'

}

**Input Example 2 (with error):**

**python**

l3MetadataTable = "non\_existent\_table"

l3metadb = "l3db"

engine = some\_database\_engine

**Output Example 2 (error case):**

Error reading metadata: Table 'l2db.non\_existent\_table' does not exist (or other SQLAlchemyError message)

**Returned Value:**

The error is raised, terminating the function

## fetch\_all\_sql\_queries(session, query\_meta\_table, ctxArea)

* Input:
  + session: The database session used to execute SQL queries.
  + query\_meta\_table: Name of the table containing SQL query metadata.
  + ctxArea: A filter criterion for the ctxarea column in the query\_meta\_table.
* Try to perform the following:
  + Execute the following SQL query using the provided session:

**Sql**

***SELECT ctxval, remarks FROM {query\_meta\_table} WHERE ctxarea='{ctxArea}' ORDER BY ctxkey***

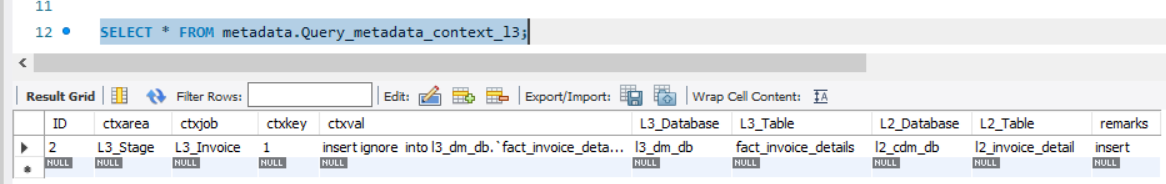
* Store the query results in result.
* Print the message: "SQL queries successfully fetched from the metadata table."
* Return all rows from the query result as a list of tuples using result.fetchall().
* **Catch** an SQLAlchemyError if it occurs:
  + - * Print the message: "Error fetching SQL queries: " followed by the error message.
      * Raise the error again to notify of the failure

**Table Structures:**

* **Query\_meta\_table:** Query\_metadata\_context\_l2

|  |  |
| --- | --- |
| Column Name | Datatpe |
| ID | int (primary key) |
| Ctxarea | varchar(100) |
| Ctxjob | varchar(100) |
| Ctxkey | varchar(100) |
| Ctxval | mediumtext, |
| L3\_Database | varchar(100) |
| L3\_Table | varchar(100) |
| L2\_Database | varchar(100) |
| L2\_Table | varchar(100) |
| Remarks | varchar(255) |

* **Sample data:**



**Input Example 1:**

python

session = some\_database\_session

query\_meta\_table = "query\_metadata"

ctxArea = "L3\_invoice"

**Output Example 1:**

SQL queries successfully fetched from the metadata table.

**Returned Value:**

Python:

[

('SELECT \* FROM finance\_table', 'Query for finance-related data'),

('SELECT balance FROM account\_table', 'Query for account balances'),

]

**Input Example 2 (with error):**

Python:

session = some\_database\_session

query\_meta\_table = "non\_existent\_table"

ctxArea = "L3\_invoice"

**Output Example 2 (error case):**

Error fetching SQL queries: Table 'non\_existent\_table' does not exist (or other SQLAlchemyError message)

**Returned Value:**

The error is raised, terminating the function.

## 6. Function `read\_l2\_audit\_table(l2AuditTable, l2dbname, engine)`:

* **Input**:
  + - * ` l2AuditTable `: Name of the L2 audit table.
      * ` l2dbname `: Name of the database where the table resides.
      * `engine`: Database engine used for querying.
* **Try** to perform the following:
  + Construct the SQL query (`query`) to select the uppercased `Customer\_Name` and cast `L2\_load\_date` as a date, grouping by these columns..
  + Execute the query using `pd.read\_sql\_query()` and store the result in `df`.
  + Print the message: "L2 Audit table data successfully retrieved."
  + Return the dataframe `df`.
* **Catch** an `SQLAlchemyError` if it occurs:
  + Print the message: "Error reading L2 Invoice table: " followed by the error message.
  + Raise the error again to notify of the failure.
* **Input Example 1:**

**Python:**

* + l2AuditTable = "l2\_audit"
  + l2dbname = "audit\_db"
  + engine = some\_database\_engine
* **Output Example 1:**

L2 Audit table data successfully retrieved.

* **Returned Value:**

| Customer\_Name | L2\_load\_date |

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

| ABC Corp | 2024-09-20 |

| XYZ Inc. | 2024-09-21 |

| DEF Ltd. | 2024-09-22 |

* **Input Example 2 (with error):**
* Python:

l1InvTable = "non\_existent\_table"

l1dbname = "invoicedb"

engine =some\_database\_engine

* **Output Example 2 (error case):**
* Error reading L2 Invoice table: (specific error message)
* **Returned Value:**

The error is raised, terminating the function.

## 7. Function `read\_l3\_audit\_table(l3AuditTable, l3dbname, engine)`:

* **Input:**
  + l3AuditTable `: Name of the L3 Audit table.
  + `l32dbname`: Name of the database where the table resides.
  + `engine`: Database engine used for querying.
* **Try** to perform the following:
  + Construct the SQL query (`query`) to select the uppercased `Customer\_Name` and cast `L2\_load\_date` as a date, grouping by these columns.
  + Execute the query using `pd.read\_sql\_query()` and store the result in `df`.
  + Print the message: "L3 Audit table data successfully retrieved."
  + Return the dataframe `df`.
* **Catch** an `SQLAlchemyError` if it occurs:
  + Print the message: "Error reading L3 Audit table: " followed by the error message.
  + Raise the error again to notify of the failure.
* **Input Example 1:**
  + python
  + l3AuditTable = " l3\_audit "
  + l3dbname = "auditdb"
  + engine = some\_database\_engine
* **Output Example 1:**

L3 Audit table data successfully retrieved.

* **Returned Value:**

| Customer\_Name | L2\_load\_date |

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

| ABC Corp | 2024-09-20 |

| XYZ Inc. | 2024-09-21 |

| DEF Ltd. | 2024-09-22 |

* **Input Example 2 (with error):**
* Python:

L3AuditTable = "non\_existent\_table"

L3dbname = "auditdb"

engine = some\_database\_engine

* **Output Example 2 (error case):**

Error reading L3 audit table: Table auditdb.non\_existent\_table' does not exist (or other SQLAlchemyError message)

* **Returned Value:**

The error is raised, terminating the function.

## 8.Function `delta\_cust\_details(df1, df2)`

* **Input:**
  + `df1`: DataFrame representing L2 audit.
  + `df2`: DataFrame representing L3 audit.
* **Try** to perform the following:
  + Merge `df1` and `df2` on the columns `Customer\_Name` and `L2\_load\_date` using a left join, and include an indicator for merge results. Store the result in `merge\_df`.
  + Filter the merged DataFrame to keep only the rows where `\_merge` is `'left\_only'` (i.e., rows present in `df1` but not in `df2`).
  + Drop the `\_merge` column from the resulting DataFrame `delta\_df`.
  + Print the message: "Delta between L2 and L3 data successfully calculated."
  + Return the DataFrame `delta\_df` containing the differences.
* **Catch** any generic `Exception` that occurs:
  + Print the message: "Error calculating delta: " followed by the error message.
  + Raise the error again to notify of the failure.

**Input Example 1:**

**python**

* df1 = pd.DataFrame({

'Customer\_Name': ['ABC Corp', 'XYZ Inc', 'DEF Ltd'],

'L2\_load\_date': ['2024-09-20', '2024-09-21', '2024-09-22']

})

* df2 = pd.DataFrame({

'Customer\_Name': 'ABC Corp', 'DEF Ltd'],

'L2\_load\_date': ['2024-09-20', '2024-09-22']

})

**Output Example 1:**

Delta between L2 and L3 data successfully calculated.

**Returned Value:**

Customer\_Name L2\_load\_date

0 XYZ Inc 2024-09-21

**Input Example 2 (with error):**

**python**

df1 = "invalid\_dataframe"

df2 = pd.DataFrame({

'Customer\_Name': ['XYZ Inc.', 'DEF Ltd.'],

'L1\_load\_date': ['2024-09-21', '2024-09-22']

})

**Output Example 2 (error case):**

Error calculating delta: Can only merge DataFrame objects, but received invalid input (or other error message)

**Returned Value:**

The error is raised, terminating the function.

## 9. Function `l3Audit\_cnt(srcDb, targetDb, srcTable, targetTable, Customer\_Name, LoadDate, l3\_dimdb, partty\_group\_table, engine)`

* **Input:**
  + srcDb`: Source database name.
  + targetDb`: Target database name.
  + `srcTable`: Source table name.
  + `targetTable`: Target table name.
  + `Customer\_Name`: Name of the customer to filter records.
  + `LoadDate`: Date to filter records.
  + `l3\_dimdb`: Dimension database name.
  + `partty\_group\_table`: Name of the party group table.
  + `engine`: Database engine used for executing queries.
* **Formulate the source query (`src\_qury`):**
* Construct the SQL query `src\_qury` to count records in the source table where `invhdrNameCustomer` matches `Customer\_Name` and `load\_date` matches `LoadDate`.
* **Execute the source query:**
  + Use `pandas.read\_sql\_query` to run `src\_qury` on the `engine` and store the result in `tsrc\_cnt`.
  + Extract the count from the first row and first column of `tsrc\_cnt`, and store it in `src\_cnt`.
* **Formulate the target query (`trgt\_qury`):**
  + Create a SQL query to count the number of rows in the `targetTable` from the `targetDb` where `invhdrNameCustomer` matches `Customer\_Name` and `load\_date` matches `LoadDate`.
* **Execute the target query:**
  + Construct the SQL query `party\_query` to select `PARTY\_GROUP\_KEY` from `partty\_group\_table` where `PARTY\_GROUP\_NAME` matches the uppercase version of `Customer\_Name`
  + Execute `party\_query` using `pd.read\_sql\_query()` and store the result in `tparty\_key`.
  + Extract the `PARTY\_GROUP\_KEY` from `tparty\_key` and store it in `partty\_group\_key`.
  + Construct the SQL query `trgt\_qury` to count records in the target table where `PARTY\_GROUP\_KEY` matches `partty\_group\_key` and `LOAD\_DATE` matches `LoadDate`.
  + Use `pandas.read\_sql\_query` to run `trgt\_qury` on the `engine` and store the result in `ttrgt\_cnt`.
  + Extract the count from the first row and first column of `ttrgt\_cnt`, and store it in `trgt\_cnt`.
* Return `src\_cnt` and `trgt\_cnt` as the counts from the source and target tables, respectively.

## 10. Function `execute\_sql(session, sql\_query, variables=None)`

* **Input:**
  + `session`: The database session used to execute SQL queries.
  + `sql\_query`: A string containing the SQL query to be executed.
  + `variables`: An optional dictionary of variables to bind to the SQL query.
* **Try** to perform the following:
  + Convert the `sql\_query` string to a SQLAlchemy text object called `query`.
  + Execute the `query` using the session, passing in `variables` (or an empty dictionary if `variables` is `None`).
  + Commit the transaction to save changes to the database.
  + Return the tuple `('SUCCESS', None)` indicating the query executed successfully.
* **Catch** an `SQLAlchemyError` if it occurs:
  + Rollback the session to undo any uncommitted changes.
  + Convert the error to a string and store it in `error\_mesg`.
  + Return the tuple `('FAIL', error\_mesg)` indicating the query failed, along with the error message.

**Input Example 1:**

**python**

session = some\_database\_session

sql\_query = "INSERT INTO users (username, email) VALUES (:username, :email)"

variables = {'username': 'john\_doe', 'email': 'john@example.com'}

**Output Example 1:**

('SUCCESS', None)

**Input Example 2 (with error):**

**python**

session = some\_database\_session

sql\_query = "INSERT INTO users (username, email) VALUES (:username, :email)"

variables = {'username': 'jane\_doe', 'email': 'jane@example.com'}

**Output Example 2 (error case):**

**python**

# Assume the table does not allow duplicates and 'jane\_doe' already exists

('FAIL', 'UNIQUE constraint failed: users.username')

11.Function `insert\_audit\_record(session, audit\_table, exc\_id, ctxarea, job\_name, customer\_name, operation\_type, source\_count, target\_count, start\_time, end\_time, status, insert\_by, error\_desc, l1\_load\_date, l2\_load\_date, l3\_load\_date)`

* Input:
* `session`: The database session used for executing the SQL query.
* `audit\_table`: The name of the audit table where records will be inserted.
* `exc\_id`: Unique execution identifier.
* `ctxarea`: Context area of the operation.
* `job\_name`: Name of the job being executed.
* `customer\_name`: Name of the customer related to the audit.
* `operation\_type`: Type of operation being logged.
* `source\_count`: Number of records from the source.
* `target\_count`: Number of records in the target.
* `start\_time`: Start time of the operation.
* `end\_time`: End time of the operation.
* `status`: Status of the operation (e.g., SUCCESS or FAIL).
* `insert\_by`: User who performed the insert operation.
* `error\_desc`: Description of any error encountered.
* `l1\_load\_date`: Load date for L1 data.
* `l2\_load\_date`: Load date for L2 data.
* `l3\_load\_date`: Load date for L3 data.
* **Try** to perform the following:
* Construct the SQL `INSERT` query to insert the audit record into `audit\_table`.
* Execute the query using `session.execute()` with the provided parameters.
* Commit the transaction using `session.commit()`.
* Print the message: "Audit record inserted for exc\_id: " followed by `exc\_id`.
* **Catch** an `SQLAlchemyError` if it occurs:
* Print the message: "Error inserting audit record: " followed by the error message.
* Rollback the session using `session.rollback()`.
* **Input Example:**
  + session = some\_database\_session
  + audit\_table = "audit\_log"
  + exc\_id = "12345"
  + ctxarea = "data\_processing"
  + job\_name = "L3\_Audit\_Job"
  + customer\_name = "John Doe"
  + operation\_type = "Insert"
  + source\_count = 50
  + target\_count = 45
  + start\_time = datetime.now()
  + end\_time = datetime.now()
  + status = "SUCCESS"
  + insert\_by = "admin\_user"
  + error\_desc = ""
  + l1\_load\_date = "20240920"
  + l2\_load\_date = "20240921"
  + l3\_load\_date = "20240922"
* **Output Example 1 (successful insertion):**

Audit record inserted for exc\_id: 12345

12. Function `l3Execution()`

* **Description**:
  + Main execution function that integrates the database handler and SQL operations.
* **Try** to perform the following:
  + Create a database connection using `create\_connection()`, storing the results in `session` and `engine`.
  + Set the `l3MetadataTable` and `meta\_db\_name` to appropriate values.
  + Call `read\_metadata()` to retrieve metadata keys into `meta\_l3\_keys`.
  + Extract values from `meta\_l3\_keys` into appropriate variables.
* **Fetch all SQL queries:**
  + Call `fetch\_all\_sql\_queries(db\_handler.session, query\_meta\_table, ctxArea)` to retrieve `sql\_queries`.
* Read data from L2 audit andRead L2 and L3 audit tables into `df1` and `df2` using `read\_l2\_audit\_table()` and `read\_l3\_audit\_table()`.
* **Calculate delta:**
  + Call `delta\_cust\_details(df1, df2)` to get `df\_delta`.
  + Print `df\_delta`.
* **For each row in `df\_delta`:**
  + Set `variables` with `Customer\_Name` and `L2\_load\_date`.
  + Extract `customer\_name` and `L2\_load\_date`.
  + For each SQL query in `sql\_queries`:
    - Extract `sql\_query` and `operation\_type`.
    - Print the SQL query and operation type.
    - Determine `required\_variables` for the current SQL query.
    - Call `l3Audit\_cnt(srcDb, targetDb, l1InvTable, targetTable, customer\_name, L1\_load\_date, db\_handler.engine)` to get source and target counts.
    - Create an `audit\_entry` dictionary with relevant details.
    - Print the variables to be executed.
    - Close the database connection using `db\_handler.close()`.
    - Generate a unique execution ID `exc\_id`.
    - Record the `audit\_start\_time`.
    - Execute the SQL query using `execute\_sql()` and store the result in `status` and `error\_desc`.
    - Retrieve source and target counts using `l3Audit\_cnt()`.
    - Record the `audit\_end\_time`.
    - Insert the audit record using `insert\_audit\_record()`.
* **Catch** any generic `Exception` that occurs:
  + Print the error message: "Error during L3 Execution: " followed by the error details and raise the exception.
  + Raise the error again to notify of the failure.
* **Input Example 1:**

**Python**

# Assuming the following metadata and database configurations

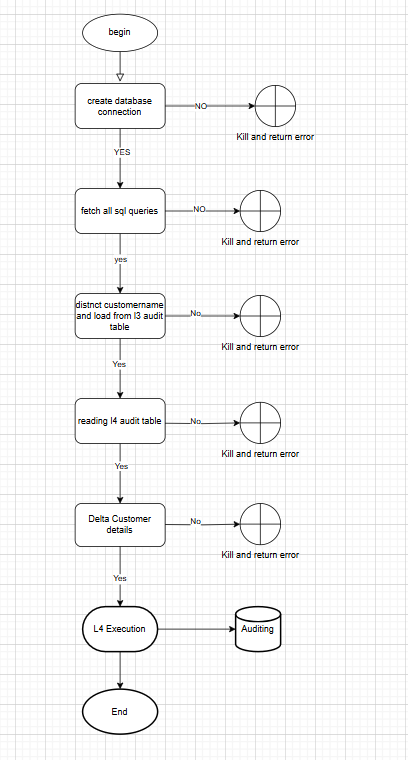
* + session = some\_database\_session
  + engine = some\_database\_engine
  + l3MetadataTable = "prod\_context\_l2\_l3\_python"
  + meta\_db\_name = "stg\_tbl"
* **Output Example 1:**
  + Customer\_Name L2\_load\_date
  + John Doe 20240921
* **Output Example 2 (error case):**

Error during L3 Execution: (specific error message)

13. if \_\_name\_\_ == "\_\_main\_\_":

* This is main program execution.
* If the script is executed as the main program the it call the function `l3Execution()`.
* There are no direct inputs for this part of the code; it relies on the execution of `l3Execution()` which handles its own input.

# L3 Area to L4 Area Processing - structured Data Extraction



For processing l4 job we must import required libraries. Below are the list of libraries were imported.

**Import necessary libraries:**

* import pandas as pd:
  + Import the pandas library, commonly used for data manipulation and analysis
* from sqlalchemy import create\_engine, text:
  + Import create\_engine for establishing database connections and text for creating SQL text object.
* from sqlalchemy.orm import sessionmaker:
  + Import sessionmaker for creating new session objects to interact with the database.
* from sqlalchemy.exc import SQLAlchemyError:
  + Import SQLAlchemyError for handling exceptions related to SQLAlchemy operations.
* import time:
  + Import the time module for time-related functions (e.g., delays).
* import os:
  + Import the os module for interacting with the operating system (e.g., file paths, environment variables).

## create\_connection.

* **Description:** This function creates a database connection using the databaseHandler class.
* **Input:** dbUrl – The database URL required to establish a connection.
* **Try** to perform the following:
  + Instantiate an object of databaseHandler class with dbUrl as the argument.
  + Print the message: "Database connection established."
  + **Return** the database handler object.
* **Catch** an SQLAlchemyError if it occurs:
  + Print the message: "Error creating database connection: " followed by the error message.
* **Raise** the error again to notify of the failure.

## read\_metadata(l4MetadataTable, l4metadb, engine):

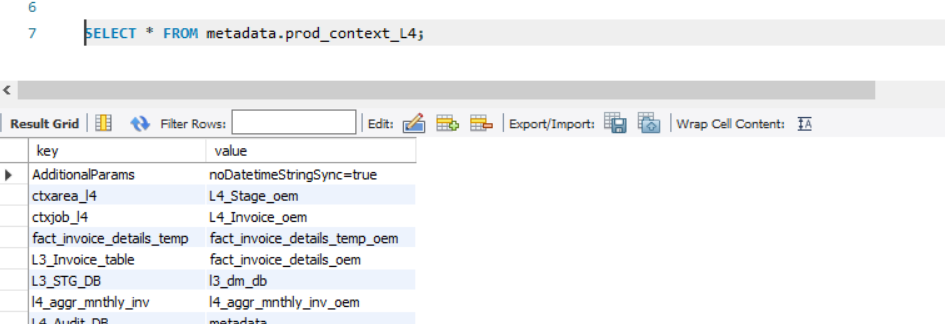
* + - Input:
* L4MetadataTable: Name of the metadata table.
* L4metadb: Database name where the metadata table resides.
* engine: Database engine used for querying.
* **Try** to perform the following:
  + Create a query string l4\_metaquery:
    - "***SELECT \* FROM {l4metadb}.{l4MetadataTable}***" to select all data from the specified table.
* Use pandas.read\_sql\_query to execute the SQL query on the engine and store the result in df\_metadata.
* Print the message: "Metadata successfully retrieved from the database."
* Convert the df\_metadata dataframe to a dictionary, setting the key column as the dictionary key and the value column as the value, then return this dictionary.
* **Catch** an SQLAlchemyError if it occurs:
  + Print the message: "Error reading metadata: " followed by the error message.
  + Raise the error again to notify of the failure.

Eg: {‘ dim\_date\_table’:’ dim\_date’,’ dim\_db’:’ l1\_t0004\_db’}

**Table Structures:**

* **prod\_context\_l4:**

|  |  |
| --- | --- |
| Column Name | Datatpe |
| key | String |
| value | String |

* **Sample Data**
* 

**Example:**

**Python:**

L4MetadataTable = " prod\_context\_L4"

L4metadb = " metadata"

engine = some\_database\_engine

**Output Example 1:**

Metadata successfully retrieved from the database.

**Returned Value:**

{

'key1': 'value1',

'key2': 'value2',

'key3': 'value3'

}

**Input Example 2 (with error):**

**python**

L4MetadataTable = "non\_existent\_table"

L4metadb = "metadata"

engine = some\_database\_engine

**Output Example 2 (error case):**

Error reading metadata: Table 'metadata.non\_existent\_table' does not exist (or other SQLAlchemyError message)

**Returned Value:**

The error is raised, terminating the function

## fetch\_all\_sql\_queries(session, query\_meta\_table, ctxArea)

* Input:
  + session: The database session used to execute SQL queries.
  + query\_meta\_table: Name of the table containing SQL query metadata.
  + ctxArea: A filter criterion for the ctxarea column in the query\_meta\_table.
* Try to perform the following:
  + Execute the following SQL query using the provided session:

**Sql**

***SELECT ctxval, remarks FROM {query\_meta\_table} WHERE ctxarea='{ctxArea}' ORDER BY ctxkey***

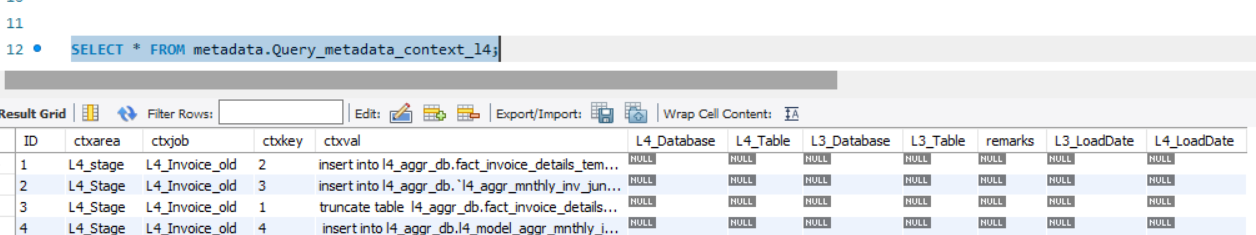
* Store the query results in result.
* Print the message: "SQL queries successfully fetched from the metadata table."
* Return all rows from the query result as a list of tuples using result.fetchall().
* **Catch** an SQLAlchemyError if it occurs:
  + - * Print the message: "Error fetching SQL queries: " followed by the error message.
      * Raise the error again to notify of the failure

**Table Structures:**

* **Query\_meta\_table:** Query\_metadata\_context\_l4

|  |  |
| --- | --- |
| Column Name | Datatpe |
| ID | int (primary key) |
| Ctxarea | varchar(100) |
| Ctxjob | varchar(100) |
| Ctxkey | varchar(100) |
| Ctxval | mediumtext, |
| L4\_Database | varchar(100) |
| L4\_Table | varchar(100) |
| L3\_Database | varchar(100) |
| L3\_Table | varchar(100) |
| Remarks | varchar(255) |
| L3\_LoadDate | varchar(100) |
| L4\_LoadDate | varchar(100) |

* **Sample data:**



**Input Example 1:**

python

session = some\_database\_session

query\_meta\_table = "query\_metadata"

ctxArea = "L4\_invoice"

**Output Example 1:**

SQL queries successfully fetched from the metadata table.

**Returned Value:**

Python:

[

('SELECT \* FROM finance\_table', 'Query for finance-related data'),

('SELECT balance FROM account\_table', 'Query for account balances'),

]

**Input Example 2 (with error):**

Python:

session = some\_database\_session

query\_meta\_table = "non\_existent\_table"

ctxArea = "L4\_invoice"

**Output Example 2 (error case):**

Error fetching SQL queries: Table 'non\_existent\_table' does not exist (or other SQLAlchemyError message)

**Returned Value:**

The error is raised, terminating the function.

## 6. Function read\_l3\_inv\_table(l3InvTable, l3dbname, engine):

* **Input**:
  + - * ` l3InvTable `: Name of the L2 audit table.
      * ` l3dbname `: Name of the database where the table resides.
      * `engine`: Database engine used for querying.
* **Try** to perform the following:
  + Construct the SQL query (`query`) to ***SELECT distinct party\_group\_key,cast(load\_date as date)as load\_date FROM {l3dbname}.{l3InvTable} group by party\_group\_key,cast(load\_date as date)***
  + Execute the query using `pd.read\_sql\_query()` and store the result in `df`.
  + Print the message: " L3 Invoice table data successfully retrieved."
  + Return the dataframe `df`.
* **Catch** an `SQLAlchemyError` if it occurs:
  + Print the message: " Error reading L3 Invoice table: " followed by the error message.
  + Raise the error again to notify of the failure.
* **Input Example 1:**

**Python:**

* + l3InvTable = “fact\_invoice\_details”
  + l3dbname = " l3\_dm\_db "
  + engine = some\_database\_engine
* **Output Example 1:**

L3 Invoice table data successfully retrieved.

* **Returned Value:**

| Customer\_Name | L3\_load\_date |

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

| ABC Corp | 2024-09-20 |

| XYZ Inc. | 2024-09-21 |

| DEF Ltd. | 2024-09-22 |

* **Input Example 2 (with error):**
* Python:

l3InvTable= "non\_existent\_table"

l3dbname = " l3\_dm\_db "

engine =some\_database\_engine

* **Output Example 2 (error case):**
* Error reading L3 Invoice table: (specific error message)
* **Returned Value:**

The error is raised, terminating the function.

## 7. Function ` read\_l4\_audit\_table(l4AuditTable, l4dbname, engine)`:

* **Input:**
  + l4AuditTable `: Name of the L3 Audit table.
  + l4dbname `: Name of the database where the table resides.
  + `engine`: Database engine used for querying.
* **Try** to perform the following:
  + Construct the SQL query (`query`) to “***SELECT distinct party\_group\_key,cast(L3\_Load\_Date as date) as load\_date FROM {l4dbname}.{l4AuditTable} group by party\_group\_key,cast(L3\_Load\_Date as date)”.***
  + Execute the query using `pd.read\_sql\_query()` and store the result in `df`.
  + Print the message: "L4 Audit table data successfully retrieved."
  + Return the dataframe `df`.
* **Catch** an `SQLAlchemyError` if it occurs:
  + Print the message: "Error reading L4 Audit table: " followed by the error message.
  + Raise the error again to notify of the failure.
* **Input Example 1:**
  + python
  + l4AuditTable = " l4\_audit "
  + l4dbname = "auditdb"
  + engine = some\_database\_engine
* **Output Example 1:**

L4 Audit table data successfully retrieved.

* **Returned Value:**

| Customer\_Name | L2\_load\_date |

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

| ABC Corp | 2024-09-20 |

| XYZ Inc. | 2024-09-21 |

| DEF Ltd. | 2024-09-22 |

* **Input Example 2 (with error):**
* Python:

L4AuditTable = "non\_existent\_table"

L4dbname = "auditdb"

engine = some\_database\_engine

* **Output Example 2 (error case):**

Error reading L4 audit table: Table auditdb.non\_existent\_table' does not exist (or other SQLAlchemyError message)

* **Returned Value:**

The error is raised, terminating the function.

## 8.Function `delta\_cust\_details(df1, df2)`

* **Input:**
  + `df1`: DataFrame representing L2 audit.
  + `df2`: DataFrame representing L3 audit.
* **Try** to perform the following:
  + Merge `df1` and `df2` on the columns `Customer\_Name` and `L3\_load\_date` using a left join, and include an indicator for merge results. Store the result in `merge\_df`.
  + Filter the merged DataFrame to keep only the rows where `\_merge` is `'left\_only'` (i.e., rows present in `df1` but not in `df2`).
  + Drop the `\_merge` column from the resulting DataFrame `delta\_df`.
  + Print the message: "Delta between L3 and L4 data successfully calculated."
  + Return the DataFrame `delta\_df` containing the differences.
* **Catch** any generic `Exception` that occurs:
  + Print the message: "Error calculating delta: " followed by the error message.
  + Raise the error again to notify of the failure.

**Input Example 1:**

**python**

* df1 = pd.DataFrame({

'Customer\_Name': ['ABC Corp', 'XYZ Inc', 'DEF Ltd'],

'L3\_load\_date': ['2024-09-20', '2024-09-21', '2024-09-22']

})

* df2 = pd.DataFrame({

'Customer\_Name': 'ABC Corp', 'DEF Ltd'],

'L3\_load\_date': ['2024-09-20', '2024-09-22']

})

**Output Example 1:**

Delta between L3 and L4 data successfully calculated.

**Returned Value:**

Customer\_Name L3\_load\_date

0 XYZ Inc 2024-09-21

**Input Example 2 (with error):**

**python**

df1 = "invalid\_dataframe"

df2 = pd.DataFrame({

'Customer\_Name': ['XYZ Inc.', 'DEF Ltd.'],

'L3\_load\_date': ['2024-09-21', '2024-09-22']

})

**Output Example 2 (error case):**

Error calculating delta: Can only merge DataFrame objects, but received invalid input (or other error message)

**Returned Value:**

The error is raised, terminating the function.

## 9. Function `l4Audit\_cnt(srcDb,targetDb,srcTable,targetTable,PARTY\_GROUP\_KEY,LoadDate,engine):`

* **Input:**
  + srcDb`: Source database name.
  + targetDb`: Target database name.
  + `srcTable`: Source table name.
  + `targetTable`: Target table name.
  + ` PARTY\_GROUP\_KEY `: Key of the customer to filter records.
  + `LoadDate`: Date to filter records.
  + `engine`: Database engine used for executing queries.
* **Formulate the source query (`src\_qury`):**
* Construct the SQL query `src\_qury` to count records in the source table where ` PARTY\_GROUP\_KEY ` matches ` PARTY\_GROUP\_KEY ` and `load\_date` matches `LoadDate`.
* **Execute the source query:**
  + Use `pandas.read\_sql\_query` to run `src\_qury` on the `engine` and store the result in `tsrc\_cnt`.
  + Extract the count from the first row and first column of `tsrc\_cnt`, and store it in `src\_cnt`.
* **Formulate the target query (`trgt\_qury`):**
  + Create a SQL query to count the number of rows in the `targetTable` from the `targetDb` where `invhdrNameCustomer` matches `Customer\_Name` and `load\_date` matches `LoadDate`.
* **Execute the target query:**
  + Construct the SQL query `party\_query` to select `PARTY\_GROUP\_KEY` from `partty\_group\_table` where `PARTY\_GROUP\_NAME` matches the uppercase version of `Customer\_Name`
  + Execute `party\_query` using `pd.read\_sql\_query()` and store the result in `tparty\_key`.
  + Extract the `PARTY\_GROUP\_KEY` from `tparty\_key` and store it in `partty\_group\_key`.
  + Construct the SQL query `trgt\_qury` to count records in the target table where `PARTY\_GROUP\_KEY` matches `partty\_group\_key` and `LOAD\_DATE` matches `LoadDate`.
  + Use `pandas.read\_sql\_query` to run `trgt\_qury` on the `engine` and store the result in `ttrgt\_cnt`.
  + Extract the count from the first row and first column of `ttrgt\_cnt`, and store it in `trgt\_cnt`.
* Return `src\_cnt` and `trgt\_cnt` as the counts from the source and target tables, respectively.

## 10. Function `execute\_sql(session, sql\_query, variables=None)`

* **Input:**
  + `session`: The database session used to execute SQL queries.
  + `sql\_query`: A string containing the SQL query to be executed.
  + `variables`: An optional dictionary of variables to bind to the SQL query.
* **Try** to perform the following:
  + Convert the `sql\_query` string to a SQLAlchemy text object called `query`.
  + Execute the `query` using the session, passing in `variables` (or an empty dictionary if `variables` is `None`).
  + Commit the transaction to save changes to the database.
  + Return the tuple `('SUCCESS', None)` indicating the query executed successfully.
* **Catch** an `SQLAlchemyError` if it occurs:
  + Rollback the session to undo any uncommitted changes.
  + Convert the error to a string and store it in `error\_mesg`.
  + Return the tuple `('FAIL', error\_mesg)` indicating the query failed, along with the error message.

**Input Example 1:**

**python**

session = some\_database\_session

sql\_query = "INSERT INTO users (username, email) VALUES (:username, :email)"

variables = {'username': 'john\_doe', 'email': 'john@example.com'}

**Output Example 1:**

('SUCCESS', None)

**Input Example 2 (with error):**

**python**

session = some\_database\_session

sql\_query = "INSERT INTO users (username, email) VALUES (:username, :email)"

variables = {'username': 'jane\_doe', 'email': 'jane@example.com'}

**Output Example 2 (error case):**

**python**

# Assume the table does not allow duplicates and 'jane\_doe' already exists

('FAIL', 'UNIQUE constraint failed: users.username')

11.Function `insert\_audit\_record(session, audit\_table, exc\_id, ctxarea, job\_name, customer\_name, operation\_type, source\_count, target\_count, start\_time, end\_time, status, insert\_by, error\_desc, l1\_load\_date, l2\_load\_date, l3\_load\_date)`

* Input:
* `session`: The database session used for executing the SQL query.
* `audit\_table`: The name of the audit table where records will be inserted.
* `exc\_id`: Unique execution identifier.
* `ctxarea`: Context area of the operation.
* `job\_name`: Name of the job being executed.
* `customer\_name`: Name of the customer related to the audit.
* `operation\_type`: Type of operation being logged.
* `source\_count`: Number of records from the source.
* `target\_count`: Number of records in the target.
* `start\_time`: Start time of the operation.
* `end\_time`: End time of the operation.
* `status`: Status of the operation (e.g., SUCCESS or FAIL).
* `insert\_by`: User who performed the insert operation.
* `error\_desc`: Description of any error encountered.
* `l1\_load\_date`: Load date for L1 data.
* `l2\_load\_date`: Load date for L2 data.
* `l3\_load\_date`: Load date for L3 data.
* `l4\_load\_date`: Load date for L3 data.
* **Try** to perform the following:
* Construct the SQL `INSERT` query to insert the audit record into `audit\_table`.
* Execute the query using `session.execute()` with the provided parameters.
* Commit the transaction using `session.commit()`.
* Print the message: "Audit record inserted for exc\_id: " followed by `exc\_id`.
* **Catch** an `SQLAlchemyError` if it occurs:
* Print the message: "Error inserting audit record: " followed by the error message.
* Rollback the session using `session.rollback()`.
* **Input Example:**
  + session = some\_database\_session
  + audit\_table = "audit\_log"
  + exc\_id = "12345"
  + ctxarea = "data\_processing"
  + job\_name = "L3\_Audit\_Job"
  + customer\_name = "John Doe"
  + operation\_type = "Insert"
  + source\_count = 50
  + target\_count = 45
  + start\_time = datetime.now()
  + end\_time = datetime.now()
  + status = "SUCCESS"
  + insert\_by = "admin\_user"
  + error\_desc = ""
  + l1\_load\_date = "20240920"
  + l2\_load\_date = "20240921"
  + l3\_load\_date = "20240922"
  + l4\_load\_date = "20240922"
* **Output Example 1 (successful insertion):**

Audit record inserted for exc\_id: 12345

12. Function `l4Execution()`

* **Description**:
  + Main execution function that integrates the database handler and SQL operations.
* **Try** to perform the following:
  + Create a database connection using `create\_connection()`, storing the results in `session` and `engine`.
  + Set the `l4MetadataTable` and `meta\_db\_name` to appropriate values.
  + Call `read\_metadata()` to retrieve metadata keys into `meta\_l4\_keys`.
  + Extract values from `meta\_l4\_keys` into appropriate variables.
* **Fetch all SQL queries:**
  + Call `fetch\_all\_sql\_queries(db\_handler.session, query\_meta\_table, ctxArea)` to retrieve `sql\_queries`.
* Read data from L3 table and L3 audit tables into `df1` and `df2` using `read\_l3\_inv\_table()` and `read\_l4\_audit\_table()`.
* **Calculate delta:**
  + Call `delta\_cust\_details(df1, df2)` to get `df\_delta`.
  + Print `df\_delta`.
* **For each row in `df\_delta`:**
  + Set `variables` with `Customer\_Name` and `L3\_load\_date`.
  + Extract `customer\_name` and `L3\_load\_date`.
  + For each SQL query in `sql\_queries`:
    - Extract `sql\_query` and `operation\_type`.
    - Print the SQL query and operation type.
    - Determine `required\_variables` for the current SQL query.
    - Call `l4Audit\_cnt(srcDb, targetDb, l1InvTable, targetTable, customer\_name, L1\_load\_date, db\_handler.engine)` to get source and target counts.
    - Create an `audit\_entry` dictionary with relevant details.
    - Print the variables to be executed.
    - Close the database connection using `db\_handler.close()`.
    - Generate a unique execution ID `exc\_id`.
    - Record the `audit\_start\_time`.
    - Execute the SQL query using `execute\_sql()` and store the result in `status` and `error\_desc`.
    - Retrieve source and target counts using `l4Audit\_cnt()`.
    - Record the `audit\_end\_time`.
    - Insert the audit record using `insert\_audit\_record()`.
* **Catch** any generic `Exception` that occurs:
  + Print the error message: "Error during L4 Execution: " followed by the error details and raise the exception.
  + Raise the error again to notify of the failure.
* **Input Example 1:**

**Python**

# Assuming the following metadata and database configurations

* + session = some\_database\_session
  + engine = some\_database\_engine
  + l4MetadataTable = "prod\_context\_l4"
  + meta\_db\_name = "metadata"
* **Output Example 1:**
  + Customer\_Name L3\_load\_date
  + John Doe 20240921
* **Output Example 2 (error case):**

Error during L4 Execution: (specific error message)

13. if \_\_name\_\_ == "\_\_main\_\_":

* This is main program execution.
* If the script is executed as the main program the it call the function `l4Execution()`.
* There are no direct inputs for this part of the code; it relies on the execution of `l4Execution()` which handles its own input.