CLOUDERA

Cloudera Professional Services

BNI
CDP Cluster

Saketh Gadde
Solutions Architect

TABLE OF CONTENTS

Revision History

Version	Author	Description	Date
1.0	Saketh Gadde	Document creation	16 August 2024

CLOUDERA

IMPORTANT NOTICE

Cloudera, Inc.
395 Page Mill Road
Palo Alto, CA
94306
info@cloudera.com
US:
1-888-789-1488
Intl: 1-650-362-0488
www.cloudera.com

Release Information
Version: 1.0 Date: 13/06/2022

Engagement Summary

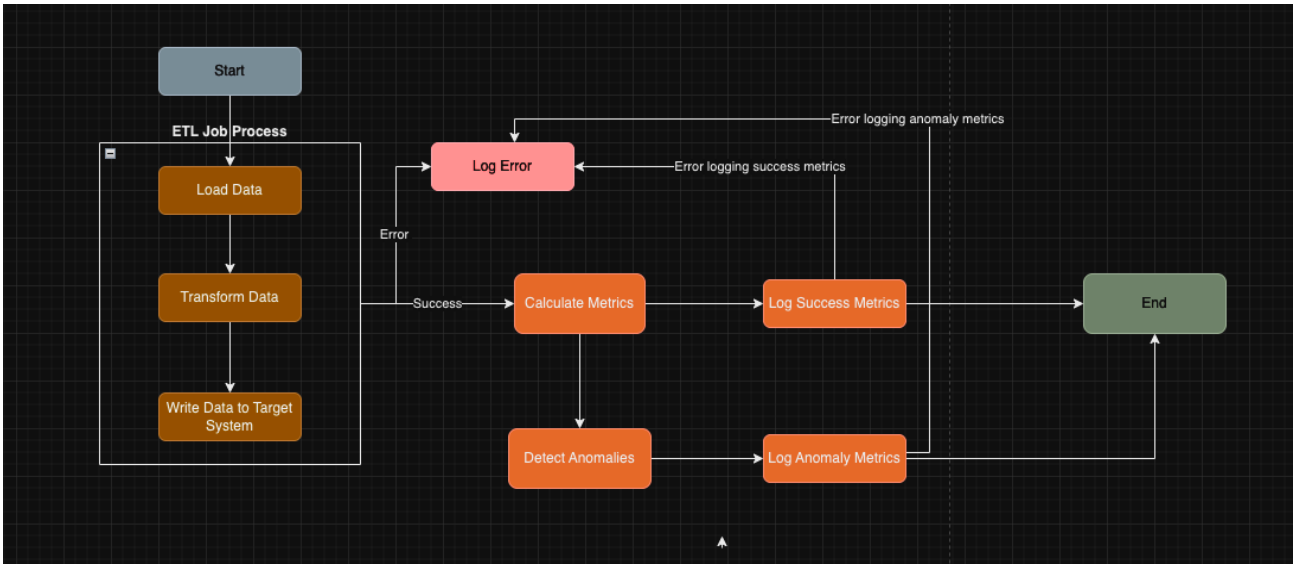
This document consists of the Data Loading framework guidelines for BNI where Cloudera Professional Services represented by Saketh Gadde built a logging framework to capture success, error and anomaly metrics. The framework has been successfully tested on BNI's CDP environment.

Requirements

This document provides comprehensive guidelines for understanding and effectively utilizing the Data Loading framework, with a focus on two critical components: ETL_job.py and etl_logger.py. The framework is meticulously designed to ensure efficient ETL (Extract, Transform, Load) operations, while also offering robust mechanisms for monitoring data quality and managing anomalies.

1.1 Purpose and Objectives

The primary goal of this framework is to establish a logging mechanism that captures detailed success, error, and anomaly metrics during the execution of ETL jobs. By doing so, the framework not only facilitates smooth data processing but also provides valuable insights into the performance and integrity of each job. This ensures that any issues or deviations are promptly identified and addressed.



1.2 Key Tables Overview

The framework relies on three essential tables to track and log various aspects of the ETL process:

1. Success Table

Purpose: Records metrics and details for jobs that complete successfully.
Contents: This table includes information such as job execution time, the number of records processed, and any other relevant success indicators that help in assessing the performance of the job.
Structure:
process_name (string): Tracks the name of the ETL process or job that was executed.
business_date (date): Captures the business date for which the ETL job was run, usually representing the data's effective date.
out_db_name (string): Logs the name of the output database where the data is loaded after processing.
out_object_name (string): Captures the name of the output object (e.g., table) where the processed data is stored.
rows_inserted (int): Tracks the number of rows successfully inserted into the output table during the ETL process.
rows_updated (int): Logs the number of rows that were updated in the output table during the ETL process.
rows_deleted (int): Captures the number of rows that were deleted from the output table during the ETL process.
run_date (date): The date on which the ETL job was executed.
start_ts (timestamp): Records the timestamp when the ETL job started.
end_ts (timestamp): Logs the timestamp when the ETL job is completed.
update_date (date): Captures the date when the success metrics were last updated.
update_user (string): Tracks the user who last updated the success metrics.
update_ts (timestamp): Logs the timestamp when the success metrics were last updated.

2. Error Table

Purpose: Logs metrics and details when a job fails to execute correctly.
Contents: It captures error messages, failure codes, and other diagnostic information that can be used to troubleshoot and resolve the issues encountered during the ETL process.
Structure:
logger_name (string): Tracks the name of the logging process that encountered an error during the ETL job execution.

error_msg (string): Logs the error message that was generated when the ETL process failed.
update_date (date): Captures the date when the error log was last updated, indicating when the error was logged or modified.
update_user (string): Tracks the user who last updated the error log, providing accountability for changes made to the error records.
update_ts (timestamp): Logs the exact timestamp when the error log was last updated, providing a precise audit trail of changes.

3. Anomaly Table

Purpose: Tracks and logs metrics whenever anomalies are detected during the ETL process.
Contents: The table contains details about the nature of the anomalies, affected data records, and any threshold breaches that triggered the anomaly detection. This helps in understanding data quality issues and ensures corrective measures can be taken.
Structure:
row_number (int): Provides a unique identifier for each anomaly record, used for ordering or referencing anomalies within the table.
table_name (string): Logs the name of the table in which the anomaly was detected, helping to pinpoint the data source associated with the anomaly.
metric_name (string): Tracks the specific metric that was analyzed for anomalies, such as row counts or sum totals, that triggered the anomaly detection.
metric_flag (string): Captures the anomaly metric is numeric or varchar
metric_numeric (bigint): Logs the numeric value of the metric that was determined to be anomalous, providing insight into the scale of the anomaly.
sample_data (varchar(300)): Contains a sample of the data or records associated with the detected anomaly, offering a snapshot of the issue for further investigation.
as_of_date (date): Captures the date when the anomaly was detected, representing the business or processing date of the data.
is_anomaly (string): Indicates whether the record is confirmed as an anomaly, typically represented as 'Yes' or 'No'.
update_ts (timestamp): Logs the timestamp when the anomaly record was last updated, providing an audit trail for any changes made to the anomaly detection records.

1.3 Process Overview

1. ETL_Job

'ETL_job.py' is the main script responsible for orchestrating the ETL process. Manages the ETL process, from data extraction to loading and metrics calculation. It encompasses the following key functions:

- **Data Extraction:**

The script initiates by loading data from various source systems, including databases, files, and external APIs.

- **Data Transformation:**

After extraction, the script applies necessary transformations to the data, including cleaning, aggregating, and reformatting. This step is crucial for ensuring the data is in the correct format for loading into the target system.

- **Data Validation and Anomaly Detection:**

The script performs validation checks to ensure data quality and integrity. It also detects anomalies, such as unexpected data patterns or values, by comparing the processed data against predefined thresholds and rules.

- **Data Loading:**

Finally, the transformed data is loaded into the target system, such as a Hive table or a database. This step completes the ETL cycle.

- **Logging:**

Throughout the process, ETL_job.py interacts with etl_logger.py to log success, errors, and any detected anomalies into their respective tables.

2. ETL_Logger

'etl_logger.py' is dedicated to handling all logging activities related to the ETL process. It includes the following important functions:

- **Log Success Metrics:**

This function records all relevant success metrics into the Success Table. It is triggered upon the successful completion of an ETL job.

- **Log Error Metrics:**

When an ETL job fails, this function captures and logs all error-related details into the Error Table. This includes the specific error encountered, as well as any additional context that might be useful for debugging.

- **Log Anomaly Metrics:**

If anomalies are detected during the ETL process, this function logs the details into the Anomaly Table. It records the type of anomaly, the affected data, and the rules or thresholds that were breached.

- **Timestamp and Process Name Management:**

etl_logger.py ensures that each log entry is accurately timestamped and associated with the correct process name, facilitating easier tracking and analysis.

3. Anomaly Table

Anomaly detection process can be structured as follows:

1. **Start of Anomaly Detection**
    - Input: Metrics from the ETL job
2. **Retrieve Historical Metrics**
    - Query the quality control/anomaly table for past records
3. **Compare Current vs. Historical Metrics**
    - Numeric comparisons (e.g., sum, average)
    - Categorical comparisons (e.g., product name lists)
4. **Identify Deviations**
    - Apply business rules or thresholds to detect anomalies
5. **Flag and Log Anomalies**
    - Log detected anomalies to a separate table
    - Optionally trigger alerts or notifications
6. **Update Control Table**
    - Save current metrics and detection results for future comparisons
7. **End of Anomaly Detection**

Execution Flow

1. **ETL_Job Execution flow:**

Step 1: Configuration and Initialization

- Set Hive configurations: Define the Hive metastore URI, warehouse directories, and table details.
- Instantiate ETLLogger: Initializes the logging mechanism with the necessary configurations.

Step 2: Argument Parsing (Optional)

- Parse job-specific arguments: (Commented out) This section is intended for parsing any job-specific arguments that might be passed when running the script.

Step 3: Build Spark Session

- Spark session configuration: Initialize a Spark session with configurations that support Hive and allow for dynamic partitioning.

Step 4: Extract Data

- Construct and execute SQL queries: Queries are defined to read data from the source tables in Hive, using filters and transformations to prepare the data for loading.

Step 5: Load Data

- Insert data into the target table: The transformed data is inserted into the target Hive table using spark.sql().

Step 6: Metrics Calculation

- Calculate metrics dynamically: Metrics like amounts_sum, average_approval_amt, and product_name are calculated dynamically based on the DataFrame's schema.

Step 7: Log Metrics

- Log success metrics: The calculated metrics, along with the start and end timestamps, are logged to a Hive table using ETLLogger.

Step 8: Anomaly Detection

- Detect anomalies: The calculated metrics are passed to ETLLogger for anomaly detection, ensuring any deviations from expected values are flagged.

Step 9: Exception Handling

- Error logging: If any exceptions occur during the ETL process, they are logged to an error table for review.

Step 10: Session Termination

- Close Spark session: Ensure that all resources are properly closed after the ETL process is complete.

2. ETL_Logger Execution flow:

Step 1: Initialization

- Spark session setup: Initialize a Spark session specifically for logging, with Hive Warehouse Connector configurations.

Step 2: Logging Success

- Log success metrics: Create a DataFrame for the success metrics and save it to a Hive table (success_table).

Step 3: Logging Errors

- Log errors: Capture and log any errors encountered during the ETL process.

Step 4: Anomaly Detection

- Detect and log anomalies: Compare the calculated metrics to expected values or historical data, logging any significant deviations to an anomaly table.

3. Function Call Order and Functionality

- **'__main__'** function in **'etl_job.py'**:

1. Initializes configurations and ETLLogger.
2. Builds a Spark session.
3. Executes SQL queries for data extraction.
4. Inserts data into the target table.
5. Calculates dynamic metrics.
6. Calls log_success and detect_anomalies functions from ETLLogger.
7. Handles exceptions and closes the Spark session.

- ETLLogger class in etl_logger.py:

1. **'__init__':** Sets up the Spark session for logging.
2. **'log_success':** Logs success metrics to the Hive table.
3. **'log_error':** Logs error details.
4. **'detect_anomalies':** Analyzes metrics for anomalies.

This structured approach helps ensure that each component and function of the framework is executed in the correct order, with clear responsibilities and error handling throughout the process.

Anomaly Detection Process

The anomaly detection logic in your framework is designed to identify and handle potential data inconsistencies or unusual patterns in the data processed by your ETL (Extract, Transform, Load) job. Here's a detailed breakdown of how the anomaly detection works in the etl_logger.py file:

Step-by-Step Process of Anomaly Detection:

1. **Invocation of detect_anomalies Method:**
   - After the ETL job completes and metrics are calculated, the method etl_logger.detect_anomalies(rpt_table, metrics, end_time) is invoked.
   - This method is responsible for detecting anomalies based on the metrics calculated from the ETL process.
2. **Parameters Passed:**
   - **rpt_table**: This is the table name from which the data was processed.
   - **metrics**: A dictionary containing calculated metrics (e.g., sum, average, row counts, and specific lists like product names) based on the transformed data.
   - **end_time**: The timestamp when the ETL job finished processing.
3. **Querying Historical Data:**
   - The method queries historical data from a quality control or anomaly tracking table, based on the same report table (rpt_table). This query retrieves previous records for comparison.
   - The retrieval of historical data is essential for comparing current metrics with past metrics to identify any significant deviations.
4. **Comparison Logic:**
   - The method compares the newly calculated metrics against the historical metrics retrieved from the control table.
   - For numeric metrics, the comparison might involve checking if the current sum or average significantly deviates from historical values (e.g., percentage change thresholds).
   - For categorical or list metrics (e.g., product_name), the method could compare the length of the current set of values to detect any unusual entries or missing expected values.
5. **Identifying Anomalies:**
   - If the comparison reveals significant deviations or unexpected changes, the method flags these as anomalies.
   - The exact criteria for detecting anomalies depend on the business rules or statistical thresholds defined within the method.
6. **Logging or Handling Anomalies:**
   - Once anomalies are detected, the method may log these anomalies to a separate table or raise alerts.
   - This logging ensures that the data quality team or stakeholders are informed about potential issues in the processed data.
   - The framework could also include automated handling of detected anomalies, such as triggering additional validation steps or notifying responsible teams.
7. **Updating Control Table:**
   - The method might update the control table with the new metrics and anomaly detection results, ensuring that the most recent data is included for future comparisons.
   - This step helps in maintaining a history of metrics and anomalies for ongoing monitoring and continuous improvement of the ETL process.

Limitations:

- Currently, you can only have one varchar column metric for a single job process.

Conclusion

By integrating the capabilities of ETL_job.py and etl_logger.py, this Data Loading framework provides a robust solution for managing ETL processes. It not only ensures that data is processed and loaded efficiently but also offers comprehensive tools for tracking performance and detecting potential issues. This structured approach to logging success, errors, and anomalies empowers users to maintain high data quality standards and quickly respond to any challenges that arise during the ETL lifecycle.