Data Warehouse for vulnerabilities

Overview

This project involves creating a data warehouse for vulnerabilities and changes. It includes the following steps:

1. **Schema and Table Creation**: Creating schemas and tables for storing vulnerabilities and changes data.
2. **ETL for data loading**: Transforming and loading data from a source schema to the data warehouse.
3. **Querying the Data Warehouse**: Running analytical queries for the queries asked
4. **API creation**: API end points creation for the data

Requirements

- Python 3.x
- MySQL
- mysql-connector-python library
- flask library
- pandas library
- json library
- sqlalchemy
- requests
- datetime

Installation

1. **Install Python Dependencies**:

   Bash:
   ```
   pip install mysql-connector-python
   pip install -U Flask
   pip install pandas
   ```

2. **Setup MySQL Database**:
   - Ensure MySQL is installed and running.
   - Create a database named cybercube.

Configuration

Update the MySQL connection parameters in your script:

```
mysql_host = 'localhost'
mysql_user = 'root'
mysql_password = 'admin'
mysql_database = 'cybercube'
```

Organizational Details

Whole ETL has been designed in layers:

- Extraction – Data is extracted from json and inserted in dwextract schema tables.

DB Schema – dwextract
Python file - data_extract.py

- Cleaning – Data is cleaned in this step. (Data validation, Handling Nulls, Handling duplicate records, Setting defaults etc) and after cleaning, data is inserted in dwclean schema tables.

DB Schema - dwclean
Python file - data_clean.py

- Transform – Datetime columns are broken in Date and time. Further, tables are created to perform normalization for the repetitive values.

DB Schema – dwtransform
Python file - data_transform.py

- Hist – At this step historization can be performed, if required or the surrogate keys can be formed. In this case, making a surrogate key was not required since no dimensional data was coming in.

DB Schema – dwcurrent (vulnerabilities) , dwhist(cve_changes)
Python file - data_hist.py

- Conform – At this step all the lookups where we convert repetitive values to codes.

DB Schema - dwconform
Python file - data_conform.py

- Load – This is the final step of the ETL, data is loaded at this stage to the data warehouse.

DB Schema - dw
Python file - data_load.py

After the data is loaded, the given queries are answered and the queries are saved in a file named – cybercube_assignment_queries.sql

Then, for the third part of the assignment, API is created in the python file named – assignment_api.py and there is a document for the API details – Cybercube API Documentation.

NOTE – All the python files have all the ddl scripts (schema creation and table creation) required to run the program.

Code Explanation

Data_extract.py:

Note – start_date can be changed to the year from which we want the data.

Imports and Setup

- **Imports**: The script imports necessary libraries (datetime, requests, json, pandas, sqlalchemy, and mysql.connector) for date manipulation, API requests, data transformation, and database operations.
- **Date Range**: Sets the date range for the data to be fetched (from January 1, 1985, to May 1, 2024) and the maximum days per API call (120 days).
- **Lists Initialization**: Initializes empty lists to store vulnerabilities and CVE changes.

Fetching Data from NVD

1. **fetch_vulnerabilities function**: Fetches vulnerabilities data for a specified date range.
    - Constructs the date strings for the API parameters.
    - Makes API calls in a loop to handle paginated responses.
    - Parses JSON responses and appends vulnerabilities to all_vulnerabilities list.
    - Handles errors and breaks the loop if no data is returned or if there's a failure.
2. **fetch_cve_changes function**: Fetches CVE changes data for a specified date range.
    - Similar to the fetch_vulnerabilities function, it constructs API parameters, handles paginated responses, and stores the data in all_cve_changes list.

Loop Through Date Range

- **Date Range Loop**: Iterates through the date range in chunks of 120 days, calling the fetch_vulnerabilities and fetch_cve_changes functions for each chunk.

Data Extraction

1. **extract_vulnerability_data function**: Extracts relevant details from the fetched vulnerabilities.
    - Iterates through the vulnerabilities, extracting various fields (e.g., CVE ID, description, CVSS scores, impact, etc.).

    - Handles complex fields like configurations and references, converting them to JSON strings.
    - Adds current date as the record added date.
2. **extract_cve_change_data function**: Extracts relevant details from the fetched CVE changes.
    - Iterates through the changes, extracting fields (e.g., CVE ID, event name, change ID, etc.).
    - Adds current date as the record added date.

Data Conversion and Saving

- **DataFrames Creation**: Converts the extracted data into pandas DataFrames.
- **CSV Export**: Saves the DataFrames to CSV files on the specified path.

Database Operations

1. **Database Connection**: Establishes a connection to the MySQL database using mysql.connector.
2. **Schema and Table Creation**: Creates a new schema (dwextract) and defines tables for vulnerabilities and CVE changes.
    - Defines SQL scripts to create these tables with appropriate data types.
    - Truncates the tables if they already exist to ensure fresh data loading.
3. **Data Insertion**: Inserts the data from DataFrames into the MySQL tables.
    - Uses a loop to insert rows individually with error handling.
    - Commits the transaction upon successful insertion or rolls back in case of errors.
    - Closes the database cursor and connection at the end.

This script ensures that the data is accurately fetched, transformed, and loaded into the database, facilitating further analysis and reporting.

Data_clean.py

Import Necessary Libraries

```
import pandas as pd
import re
from IPython.display import display
import mysql.connector
from datetime import date
```

- pandas: For data manipulation and analysis.
- re: For regular expression operations.
- IPython.display.display: For displaying dataframes in Jupyter notebooks.
- mysql.connector: For connecting to a MySQL database.
- datetime.date: For handling date-related operations.

Define MySQL Connection Parameters

```
mysql_host = 'localhost'
mysql_user = 'root'
mysql_password = 'admin'
mysql_database = 'cybercube'
```

- These variables hold the connection parameters for the MySQL database.

Create a Connection to the MySQL Database

```
conn = mysql.connector.connect(host=mysql_host, user=mysql_user,
password=mysql_password, database=mysql_database)
cursor = conn.cursor()
```

- Establishes a connection to the MySQL database and creates a cursor object for executing SQL queries.

Function process_vulnerabilities_table() to Process the Vulnerabilities Table

- Loads data from dwextract.vulnerabilities into a DataFrame.
- Filters the DataFrame based on valid CVE IDs using a regex pattern.
- Fills missing values with defaults and applies additional validations.
- Removes duplicate records.
- Adds the current date to RECORD_ADDED_DATE.
- Inserts the cleaned data into dwclean.vulnerabilities.

Function process_cve_changes_table() processes the cve_changes table:

- Retrieves data from dwextract.cve_changes and loads it into a Pandas DataFrame.
- Defines a regex pattern to match valid CVE IDs and filters the DataFrame to keep only valid entries.
- Sets default values for missing data.
- Ensures CREATED_DATE is of datetime type.
- Displays the filtered DataFrame.
- Removes duplicate records.
- Adds a RECORD_ADDED_DATE column with the current date.
- Inserts the cleaned data into the dwclean.cve_changes table.

Data_transform.py

Imports and Database Connection:

- **pandas**: For data manipulation.
- **re**: For regular expression operations.

- **mysql.connector**: For connecting to the MySQL database.
- **datetime.date**: To get the current date.

extract_platform_product(criteria) function**:**

Uses a regex pattern to extract platform and product information from the CONFIGURATION_CRITERIA field.

create_and_insert_into_separate_table function**:**

Creates a table if it doesn't exist and inserts unique data values from a specified column, avoiding duplicates.

process_vulnerabilities_table function:

- Loads data from dwclean.vulnerabilities into a DataFrame.
- Extracts platform and product information.
- Separates date and time components from timestamp fields.
- Drops the REFERENCE_URLS column.
- Moves the RECORD_ADDED_DATE column to the end.
- Inserts the transformed data into dwtransform.vulnerabilities.
- Calls create_and_insert_into_separate_table to insert distinct values into separate tables for various fields.

process_cve_changes_table function:

- Loads data from dwclean.cve_changes into a DataFrame.
- Separates date and time components from the CREATED_DATE field.
- Moves the RECORD_ADDED_DATE column to the end.
- Inserts the transformed data into dwtransform.cve_changes.
- Calls create_and_insert_into_separate_table to insert distinct values into separate tables for EVENT_NAME and SOURCE_IDENTIFIER.

Data_hist.py

Importing Libraries and Setting Up the Connection

- **mysql.connector**: This library is used to connect to and interact with the MySQL database.
- **datetime**: Imported for handling date and time operations, though not directly used in this script.
- **Database connection parameters**: Defines the connection details (host, user, password, database).
- **Connection and cursor creation**: Establishes a connection to the MySQL database and creates a cursor for executing SQL commands.

Fetch_existing_cve_ids_and_dates(table_name) function:

Fetches and returns a set of existing CVE_ID and CVE_CHANGE_ID pairs from a specified table to avoid duplicate entries.

copy_vulnerabilities_data function:

- Fetches all records from dwtransform.vulnerabilities.
- Creates the dwcurrent schema and the vulnerabilities table if they don't exist.
- Fetches existing CVE_IDs in the dwcurrent.vulnerabilities table to avoid duplicates.
- Inserts new records that are not already in the dwcurrent.vulnerabilities table.
- Commits the transaction to save changes to the database.

copy_cve_changes_data function:

- Fetches all records from dwtransform.cve_changes.
- Creates the dwhist schema and the cve_changes table if they don't exist.
- Fetches existing CVE_ID and CVE_CHANGE_ID pairs in the dwhist.cve_changes table to avoid duplicates.
- Inserts new records that are not already in the dwhist.cve_changes table.
- Commits the transaction to save changes to the database.

Data_conform.py

Defines a function transform_and_insert_vulnerabilities that:

- Fetches all records from dwcurrent.vulnerabilities.
- Transforms each record's real values to codes using lookup tables (like dwtransform.source_identification).
- Collects the transformed records.
- Inserts the transformed records into the dwconform.vulnerabilities table.
- Commits the transaction to the database.

Defines a function transform_and_insert_cve_changes that:

- Fetches all records from dwhist.cve_changes.
- Transforms each record's real values to codes using lookup tables (like dwtransform.source_identification).
- Collects the transformed records.
- Inserts the transformed records into the dwconform.cve_changes table.
- Commits the transaction to the database.

Data_load.py

- **Connects to a MySQL Database**: It connects to a MySQL database using specified connection parameters.

- **Creates Schema and Tables**: It creates a schema named dw and two tables (vulnerabilities and cve_changes) if they don't already exist.
- **Copies Data**: It defines functions to copy data from source tables (dwconform.vulnerabilities and dwconform.cve_changes) to the corresponding target tables in the dw schema.
- **Closes the Database Connection**: After copying the data, it closes the database connection.

Defines a function fetch_existing_cve_ids_and_dates that:

- Fetches all existing CVE_ID and CVE_CHANGE_ID pairs from the specified table (dw.cve_changes in this case).
- Returns these pairs as a set for quick lookup.

Defines a function copy_vulnerabilities_data that:

- Fetches all records from dwconform.vulnerabilities.
- Transforms and prepares the data for insertion.
- Checks if each record already exists in dw.vulnerabilities based on CVE_ID.
- Collects records that don't already exist.
- Inserts the collected records into dw.vulnerabilities in a batch operation.
- Commits the transaction to the database.

Defines a function copy_cve_changes_data that:

- Fetches all records from dwconform.cve_changes.
- Fetches existing CVE_ID and CVE_CHANGE_ID pairs from dw.cve_changes.
- Checks if each record already exists in dw.cve_changes based on the fetched pairs.
- Inserts records that don't already exist into dw.cve_changes.
- Commits the transaction to the database.

Alternative Approaches:

Instead of ETL, ETL can be applied depending on the volume of the data and if raw data needs to be saved as well. In the case of ELT data would be extracted and in the same format would be saved in tables which would be retained.

The surrogate key could have been applied if there were some dimensional data that would be expected to change over a span of time.

Vulnerabilities could be divided into two tables, one would contain only the columns with descriptive information and another table with numeric columns but with this I found out that these numeric values would change over span of time, So diving the table into two, would just complicate the analytical queries.

Future possible works:

- After applying the extract and some basic manipulations, data can be stored in vault tables, which can be used in case of some failures where the data is lost from the final schema then this schema can be used to retrieve the data back
- Parallel Processing: Optimize the data fetching process by using parallel processing or asynchronous requests to reduce the time required to fetch large datasets.
- Since the data seems to be big, the API created to fetch might respond slowly. Further improvement to increase the response speed.
- Dashboard Development: Create interactive dashboards and visualizations to present vulnerability data and trends to stakeholders.
- Data quality rules can be set, which would count the number of records coming in daily, in general, it would not happen that one record is very high and very low on the other.
- Some kind of monitoring can be implemented such as error log system which would in troubleshoot the issues if there would be any failure.