

Project #1: ETL and ELT integration pipelines (Data Engineering & Software Engineering Master Course project)

Project statement:

In this project you are required to write 2 resilient simple independent data integration (ETL and ELT) pipelines that aggregate data from multiple .xlsx files in this given directory. In the directory we have 5 .xlsx files of the following format:

Region	Country	Item Type	Sales Channel	Order Priority	Order Date
Australia and Oceania	Palau	Office Supplies	Online	H	3/6/2016
Europe	Poland	Beverages	Online	L	4/18/2010
North America	Canada	Cereal	Online	M	1/8/2015

Technology Stack:

- OS: Windows 11 Pro
- Programming Language: Python
- Database for ETL: PostgreSQL
- ETL Tool: Python Libraries (Pandas for data manipulation, SQLAlchemy or PyODBC for database interaction)
- Data Transformation: Python (using libraries like Pandas, NumPy for date and money transformations)
- Hadoop Distribution: Apache Hadoop (for ELT pipeline and Hadoop Streaming)
- Hadoop File System: HDFS (Hadoop 3.3.0 Windows Distributed File System)
- Job Scheduler: Windows Task Scheduler (for automating the integration process on Windows)
- Source Control: Git (for version control of scripts and configurations)
- Text Editor/IDE: Visual Studio Code (for writing and testing Python scripts)
- Data Integration Tool (open-source): Apache Airflow (for redoing the work in Step 3 with an open-source tool)

Step 1:

- You will need to build the target schema for ETL, write scripts (in python) to read and insert records from all the files into a relational database.
- You will need to operate at least 2 transformations (dates and money).

Adnane El Amrani, Ayoub Maimmadi

The script we provided is a two-part Python-based ETL (Extract, Transform, Load) process designed to read data from Excel files and insert the transformed records into a PostgreSQL database.

Let's break down how each part of the script contributes to building the target schema and handling the ETL process:

Part 1: Database Schema Creation and Connection

This part focuses on establishing a connection with the PostgreSQL database and creating the necessary table schema to hold the data.

Database Connection:

Establishes a connection to a PostgreSQL database using `psycopg2`.

The `connect_to_database` function handles the database connection process.

Table Creation:

The `create_table` function creates a table `sales_records` in the database if it doesn't exist.

Defines the schema for the table, including data types for each column.

Part 2: Data Extraction, Transformation, and Loading

This part handles reading Excel files, transforming the data, and loading it into the database.

File Processing:

Reads Excel files from a specified directory.

Ensures each file is processed once, using flag files (`.processing` and `.processed`).

Data Transformation:

Transforms date formats and formats monetary values.

The `transform_date` and `transform_money` functions are used to convert date strings and format monetary values, respectively.

Data Insertion into Database:

The `insert_dataframe_to_db` function inserts transformed data into the database.

Ensures that duplicate records are not inserted, using the `record_exists` function to check if a record with the same Order ID already exists in the database.

Handling Failures and Restarts:

If a file processing fails, the `.processing` flag file is deleted, allowing for reprocessing in subsequent runs.

Adnane El Amrani, Ayoub Maimmadi

Provides a mechanism for retrying database connections with exponential backoff (in `create_db_connection` function).

Automation and Atomicity:

The script automates the ETL process from reading files to loading data into the database.

Atomicity is partially handled in the database insertions – if an error occurs during the insertion of records, the transaction is rolled back, preventing partial data writes.

- For the ELT you need to write into Hadoop HDFS, then write the Transformation using Hadoop Streaming with Python. You will need to operate at least 2 transformations (dates and money).

In our script, the MapReduce process in Hadoop is implemented using Hadoop Streaming.

Loading Data to HDFS (Step 1):

The script first checks if the local directory with data (`local_path_data`) exists and contains files. If the directory exists and is not empty, it checks whether the HDFS path (`hdfs_path_data`) exists. If the HDFS path does not exist, it uses the `hadoop fs -copyFromLocal` command to load data from the local file system to HDFS.

Loading the Mapper Script to HDFS (Step 2):

The script checks if the local file containing the mapper code (`local_file_mapper`) exists. If the file exists and is not already present in HDFS, it uses the `hadoop fs -put` command to load the mapper script to HDFS.

Executing Hadoop Streaming (Step 3):

This is where the MapReduce process is initiated using Hadoop Streaming.

- The `streaming_command` includes a call to `hadoop jar` with the Hadoop Streaming jar file. This jar file is essential for running streaming jobs.
- The `-files` option specifies the location of the mapper script in HDFS.
- The `-mapper` option indicates the command to run for the map phase, which in this case is `python mapper.py`. This tells Hadoop Streaming to use Python to execute the mapper script.
- The `-input` option specifies the location of the input data in HDFS.
- The `-output` option specifies where the output of the MapReduce job should be stored in HDFS.

Adnane El Amrani, Ayoub Maimmadi

- In this process, the mapper script (mapper.py) is applied to each input file in the /sales_data/* directory in HDFS.
- The mapper script processes each line of the input files and outputs the result. Since there is no reducer script specified, the output of the mapper becomes the final output of the job, which is stored in /output-sales-data on HDFS.

Hadoop Streaming allows us to write map and reduce functions in languages other than Java (like Python in our case).

It uses standard input and output to pass data between ourscript and the Hadoop framework. The mapper reads data from standard input, processes it, and writes the output to standard output, where Hadoop picks it up and manages the shuffling and sorting of data before it's passed to the reducer (if a reducer is specified).

In our script, only a mapper is used, so the processing is map-only.

The mapper.py script in step 3 of ourHadoop Streaming job serves as the mapper component of the MapReduce process.

It reads data line by line from standard input, applies specific transformations to each line, and writes the results to standard output. Here's a breakdown of its role and functionality:

Read Input Data:

The script reads data from standard input (sys.stdin), which Hadoop Streaming feeds it line by line. Each line represents a record from our input data set, typically a line from a CSV file.

Process Each Line:

The process_line function is called for each line of the input. It splits the line into columns using a comma (,) as the delimiter. It then checks if the line has 15 columns to proceed with the processing.

Apply Transformations:

Capitalize Region Name: The capitalize_region_all_upper function transforms the region name to uppercase.

Convert Date Format: The convert_date_format function is used to change the date format from 'MM/DD/YYYY' to 'YYYY-MM-DD'. It's applied to the 'Order Date' and 'Ship Date' columns.

Round Units and Convert Monetary Values to Integers: The int_ function rounds the units sold and converts monetary values (like price, cost, revenue, etc.) to integer values. It's applied to the 'Units Sold' column and all monetary columns.

If any conversion fails (like an invalid date format or non-numeric value where a number is expected), a default error value (like 'Invalid-Date' or 'Invalid-Units') is returned.

Output Processed Data:

The transformed data is joined back into a single comma-separated line and printed to standard output.

Hadoop Streaming then captures this output and handles the shuffling and sorting before writing it to the specified output path in HDFS.

Role in MapReduce:

In the MapReduce process, this script is the "map" part. It processes each input record independently and transforms the data as per the defined logic.

Since a reducer is not specified in our command, the output of the mapper script becomes the final output of the job.

The script essentially modifies and formats the data according to the specific transformation rules, making it suitable for further analysis or processing in subsequent stages of our data pipeline.

- Ensure that transformations follow directly and automatically after the load finishes.

In the script we provided, the transformations implemented in mapper.py do not follow directly and automatically after the data loading step (step_1). Instead, the transformations are executed during the third step (step_3), which is the Hadoop Streaming process.

Here's the sequence of events:

Step 1 - Loading Data to HDFS (step_1):

This step checks for the existence of data in the local path and loads it into the Hadoop Distributed File System (HDFS) if it's not already there.

Once this step is completed, the script moves to the next step, but no data transformation occurs at this stage.

Step 2 - Loading Mapper to HDFS (step_2):

This step involves transferring the mapper script (mapper.py) from the local filesystem to HDFS. Just like step 1, this step does not perform any data transformation. Its primary function is to ensure that the mapper script is available in HDFS for the Hadoop Streaming process.

Step 3 - Execute Hadoop Streaming (step_3):

This is the critical step where data transformation occurs.

The streaming_command triggers the Hadoop Streaming process, which utilizes the mapper script (mapper.py) to process and transform the data loaded in step 1.

During this process, Hadoop streams the data from the /sales_data/* directory in HDFS to the mapper script. The mapper script then processes each line of the input data as per the logic defined

within it (like converting date formats, capitalizing region names, etc.), and outputs the transformed data.

The output from this process is then stored in the /output-sales-data directory on HDFS.

In summary, the actual data transformation as defined in mapper.py occurs only in step 3, i.e., during the Hadoop Streaming process. Steps 1 and 2 are preparatory steps that ensure the necessary data and scripts are in place in HDFS for the Hadoop Streaming job.

Step 2:

- Ensure that our script program works when the sources and the sync datastores are remote and use Task Manager services to start the integration process(es) automatically and periodically.

The script performs an ETL process by fetching Excel files from a remote URL, transforming their data, and then loading it into a remote PostgreSQL database.

The script is designed to work with sources and sync datastores that are remote, but there are some considerations and modifications to be made for its usage with Task Manager services for automation:

Remote Data Sources:

The script fetches data from remote Excel files hosted on GitHub. It uses the requests library to download the files and pandas to read their content.

This approach works well for remote data sources as long as the URLs are accessible and the data format is consistent.

Remote Database Connection:

The script connects to a PostgreSQL database using credentials provided in the script. This implies that the database is remotely hosted and the script can connect to it as long as the correct credentials and network permissions are set up.

Automation with Task Manager Services:

To automate this script using services like Windows Task Scheduler or cron jobs on a Unix-based system, you need to ensure the script is executable without manual intervention.

Ensure all dependencies (Python, pandas, pycopg2, etc.) are installed on the system where the Task Manager service runs.

Error handling should be robust to avoid script failure without notice. Consider adding logging and notifications for critical failures.

Scheduling the Script:

When configuring the script to run periodically through a Task Manager, we can set the frequency of execution according to our data refresh requirements.

Be mindful of the time it takes to process files and load data to avoid overlapping runs if the script is scheduled to run frequently.

Handling Failures:

The script has basic error handling but lacks a comprehensive mechanism to handle partial failures (e.g., successful data download but failure in database insertion).

Consider adding more detailed logging and possibly a retry mechanism for database insertions.

Security Considerations:

The database credentials are hardcoded in the script, which is not a secure practice, especially for automated scripts. Consider using environment variables or a secure vault system to store sensitive information.

Script Execution Environment:

The script will run on the environment where the Task Manager service is set up. Ensure that this environment has network access to both the GitHub repository and the PostgreSQL database.

Step 3:

- Redo the same work of **Step 2** using one of the Apache Airflow integration tool.

Apache Airflow is an open-source platform used for orchestrating complex computational workflows, data processing pipelines, and automating scripts. It is widely used in the field of data engineering and has become a standard for workflow orchestration.

Core Components

DAGs (Directed Acyclic Graphs):

The primary structure in Airflow is a DAG, which represents a collection of tasks and their dependencies. Each DAG defines a workflow, where tasks are nodes and dependencies are directed edges in a graph. The 'acyclic' part means the graph has no loops, so it must have a finite number of executions.

Tasks:

A task represents a unit of work within a DAG. Tasks are defined by operators in Airflow, which are Python classes designed for different types of work. Examples include PythonOperator (for Python scripts), BashOperator (for bash commands), or SqlOperator (for SQL queries).

Operators:

Operators are templates for tasks. They define what is to be done in a task. Airflow comes with a variety of built-in operators and also allows for the creation of custom operators.

Scheduler:

The Airflow Scheduler monitors all tasks and DAGs, triggers the task instances once their dependencies are complete, and manages the allocation of resources for executing these tasks.

Web Server/UI:

Airflow provides a web-based user interface for managing and monitoring DAGs and tasks. Through the UI, users can view DAGs' structures, task progress, scheduling, and execution logs.

Metadata Database:

Airflow uses a metadata database to store state and configuration information. This database contains information about the DAGs' structures, schedule intervals, task instances, and execution history.

Workflow Execution

DAG Definition:

A user writes a DAG script in Python to define tasks and their dependencies. This script is placed in the Airflow DAGs directory.

DAG Parsing:

The Airflow Scheduler periodically parses the scripts in the DAG directory to build DAG objects.

Scheduling:

Based on the scheduling parameters defined in the DAG (like start date, end date, and frequency), the Scheduler decides when to execute the tasks.

Task Execution:

Once the dependencies of a task are met and the task is scheduled, the Scheduler pushes the task for execution. Tasks can be executed on the same machine as the Scheduler or on a cluster of worker nodes.

Monitoring and Logging:

During and after the execution of tasks, Airflow records logs and monitors the progress. These logs are available via the UI for troubleshooting and auditing.

Key Features

- Extensibility: Airflow is designed to be easily extended with custom operators, executors, and hooks.
- Scalability: It can scale to handle a large number of tasks and workflows.
- Flexibility: Allows dynamic generation of DAGs and supports various integrations with data sources, storage, and other external systems.
- Community Support: As an open-source project, it has strong community support and a wealth of contributed plugins and integrations.

This code is an Apache Airflow script used to define and manage an ELT (Extract, Load, Transform) pipeline with Hadoop. Apache Airflow is an open-source tool for orchestrating complex computational workflows and data processing. Here's a breakdown of what this specific script does:

Import Necessary Modules:

The script imports modules like datetime, subprocess, and Airflow-specific modules (DAG and PythonOperator).

Define Helper Functions:

run_hadoop_command(): Executes a given Hadoop command using the subprocess module. It handles various exceptions to catch errors that might occur during the execution of the command.

step_1(): Checks if a specific HDFS (Hadoop Distributed File System) path exists. If not, it copies data from a local path to the HDFS path.

step_2(): Similar to step_1, but it's used to upload a mapper script to HDFS.

step_3(): Executes a Hadoop streaming job using a jar file and the mapper script loaded in the previous step. The job processes data in the HDFS path and outputs the results to another HDFS path.

run_elt_automated(): Runs an external Python script (ELT_automated.py) which could contain additional ELT logic or steps.

Set Default Arguments for the DAG:

These arguments are common parameters that you want to apply to the entire pipeline (or DAG - Directed Acyclic Graph). It includes configurations like owner, start date, email alerts on failure and retry, number of retries, and delay between retries.

Define the DAG:

- A DAG is defined with a unique identifier ('HADOOP_ELT_pipeline'), default arguments (default_args), a description, and a schedule interval (how often the pipeline should run).
- Create PythonOperator Tasks:
 - Each step (step_1, step_2, step_3, and run_elt_automated) is wrapped in a PythonOperator. This operator allows us to execute a Python callable (function) as part of an Airflow DAG.
 - Each task is given a unique task_id and is associated with the DAG defined earlier.

- Define Task Dependencies:
- The >> operator is used to set the order of the tasks. Here, task_1 (load data to HDFS) must be completed before task_2 (copy mapper to HDFS) can start, and so on. This sets up the flow of the pipeline.
- Execution Flow:
- When this DAG is triggered (based on its schedule or manually), Airflow executes task_1, task_2, task_3, and task_4 in sequence. Each task performs its defined function, such as loading data, executing a Hadoop job, or running an external Python script for further ELT processes.

Step 4:

- Show how you ensured the idempotency of our integration pipes, give solid arguments.
- The script we provided demonstrates several key principles to ensure idempotency in data integration pipelines. Idempotency in data engineering means that even if an operation (like a data load or transformation) is performed multiple times, the end result remains consistent and unchanged after the first successful operation. Let's break down how this script achieves idempotency:
- Check for Existing Records: The record_exists function checks if a record with a specific order ID already exists in the database. This prevents duplicate entries in case the script is rerun. This approach is crucial for idempotency because it ensures that the same data isn't inserted more than once.
- File Processing Flags: The script uses .processed and .processing flags for each file. Before processing a file, it checks if a .processed flag exists, indicating that the file has already been successfully processed. If a .processing flag is found, it implies that the file may have been partially processed in a previous run and handles it accordingly. This mechanism ensures that each file is processed exactly once, even if the script is interrupted or rerun.
- Database Transactions: The insert_dataframe_to_db function uses transactions (commit and rollback) while inserting data into the database. If an error occurs during the insertion of a batch of records, the transaction is rolled back, preventing partial and inconsistent writes.
- File-Level Processing: Each file is processed independently. This modular approach means that the failure to process one file does not impact the processing of others.
- Logging and Error Handling: Comprehensive logging and error handling allow for monitoring the process and quickly addressing issues. This is essential for maintaining idempotency as it provides visibility into the pipeline's operations and helps identify if a process was completed successfully or not.

- **Data Transformations:** Data transformations (like date and money format conversions) are applied consistently to each record. This ensures uniformity in the data regardless of how many times it is processed.
- **Retry Mechanism for Database Connection:** The `create_db_connection` function implements a retry mechanism with exponential backoff. This ensures that temporary issues with database connectivity do not result in incomplete or failed data processing.
- **The script we've provided outlines an Extract, Load, and Transform (ELT) process using Hadoop, with a focus on ensuring idempotency in the integration pipeline. Idempotency in this context means that regardless of how many times you run the integration process, the end state of the system remains consistent without unintended duplication or alteration. Let's analyze how your script ensures idempotency:**

Checking for Existence before Actions (Steps 1 and 2):

In both `step_1` and `step_2`, before copying data to Hadoop Distributed File System (HDFS), the script checks if the target path already exists using `hadoop fs -test`.

If the target path exists, it skips the copying process. This prevents duplicate data loading into HDFS, ensuring that the same state is maintained regardless of multiple executions.

Logging and Error Handling:

The use of logging provides clear visibility into the pipeline's operations and its current state, which is crucial for maintaining idempotency. If something goes wrong, it's logged, and the system can be checked against these logs.

Error handling in the `run_hadoop_command` function, with retries and delays, allows the script to handle transient errors without failing the entire process.

Retries with Delay (`run_hadoop_command`):

In case of command failure, the script retries the command with a specified delay. This ensures that temporary issues (like network glitches) don't result in a failed process, which could lead to inconsistency if the process is rerun.

Return Codes and Conditional Logic:

The script uses the return code from the subprocess commands to determine the next steps. This use of conditional logic based on the success or failure of previous steps adds to ensuring that the process doesn't perform redundant operations.

Skipping Steps Based on Preconditions:

Before proceeding with each step, the script checks certain preconditions (like the existence of a local file or directory). If these conditions are not met, it skips the step. This approach prevents errors and inconsistencies that could arise from trying to process non-existent files or directories. To further enhance the idempotency and robustness of your pipeline, consider the following:

Atomic Operations: Where possible, make sure that operations are atomic. If an operation fails, it should fully revert to its previous state.

Deliverables: (in a .zip file)

- Scripts ETL and ELT
- Task Manager configuration table
- A video of a demonstration including failures and restart of integration when one source isn't available. Make sure to handle fault tolerance, automation, and atomicity.

In the context of data engineering, key concepts like handling failures and restarts, fault tolerance, automation, and atomicity are crucial for ensuring the reliability, efficiency, and consistency of data processes. Let's define each concept:

Failures and Restarts of Integration:

Failures: In data engineering, failures refer to interruptions or errors in data processing pipelines. These can be due to various reasons such as network issues, hardware failures, data inconsistencies, or software bugs.

Restarts: Restart mechanisms are strategies to resume data processing after a failure. Effective restarts aim to continue processing from the point of failure, minimizing data loss and redundant processing.

This involves tracking progress, saving states, and having the ability to reload and process only the unprocessed data.

Fault Tolerance:

Fault tolerance in data engineering is the system's ability to continue functioning correctly in the event of partial failures. It involves designing systems that can handle and recover from errors without causing data corruption or significant downtime. Techniques include data replication, checkpointing, redundant systems, and error handling mechanisms.

Automation:

Automation in data engineering refers to the use of technology to perform data processes with minimal human intervention. It involves creating workflows that can extract, load, transform, and manage data automatically. Automation is key for scalability, consistency, and reducing manual errors. It can include scheduling jobs, monitoring data flows, and automatically responding to certain conditions or thresholds.

Atomicity:

Atomicity, often part of the ACID properties (Atomicity, Consistency, Isolation, Durability) of database transactions, refers to completing data transactions in an "all-or-nothing" manner. It

means that a transaction or a process step either fully happens or doesn't happen at all. This ensures data integrity by preventing partial updates that could lead to inconsistent states.

- The script we've provided demonstrates several practices that contribute to fault tolerance, automation, and atomicity in the context of ETL (Extract, Transform, Load) processes. Let's analyze how these concepts are handled:

Fault Tolerance

Retry Mechanism for Database Connection: The script employs a retry mechanism with exponential backoff when establishing a database connection (`create_db_connection` function). This approach enhances fault tolerance by allowing the script to handle temporary network issues or database downtimes gracefully.

Error Handling and Logging: Throughout the script, errors are caught, logged, and handled appropriately. This ensures that the script can tolerate and recover from unexpected issues without crashing or causing data corruption.

File Processing Flags: By using `.processing` and `.processed` flags, the script can handle partial processing of files. If a file's processing is interrupted (e.g., due to a system crash or power failure), the script can resume processing from where it left off, thereby tolerating and recovering from failures.

Automation:

Scheduled Execution: While the provided script doesn't directly implement scheduling, it can easily be integrated into a scheduling tool like Apache Airflow or cron jobs. This would automate the ETL process, allowing it to run at specified intervals without manual intervention.

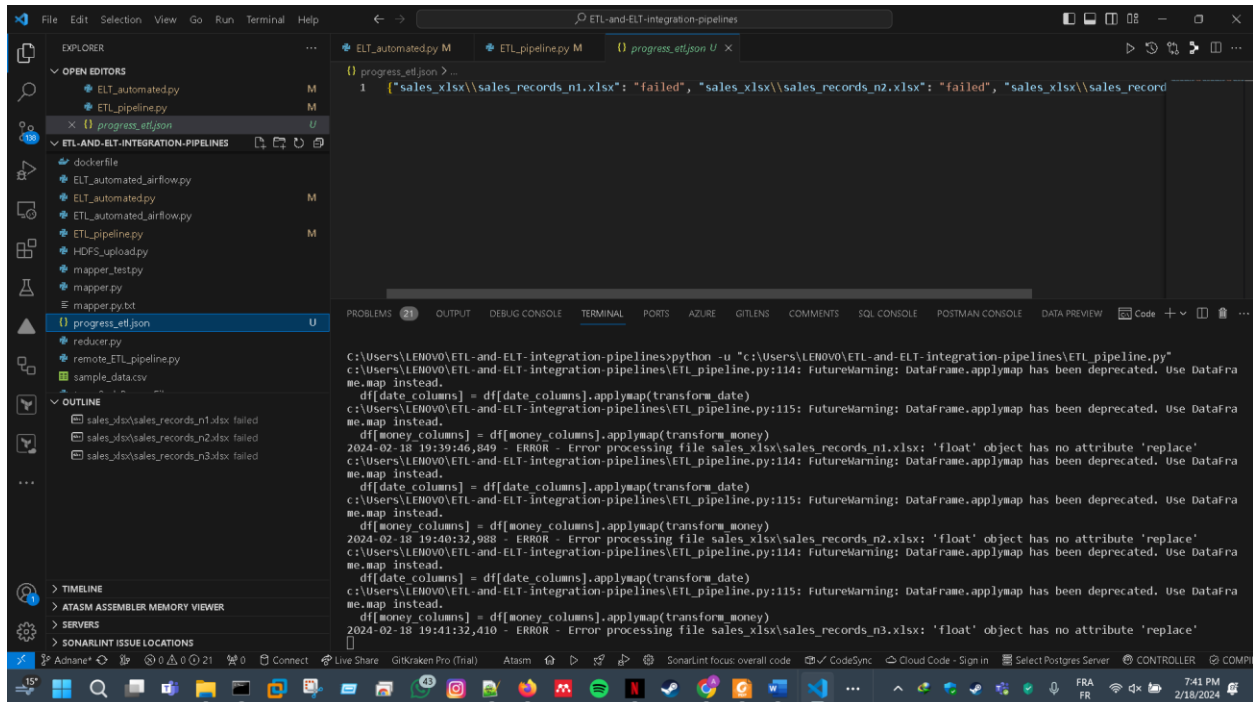
Independence from Source Availability: The script processes files already downloaded to a local directory. If the data source is temporarily unavailable, the script can still process previously downloaded files, making the ETL process less dependent on real-time data source availability.

Atomicity:

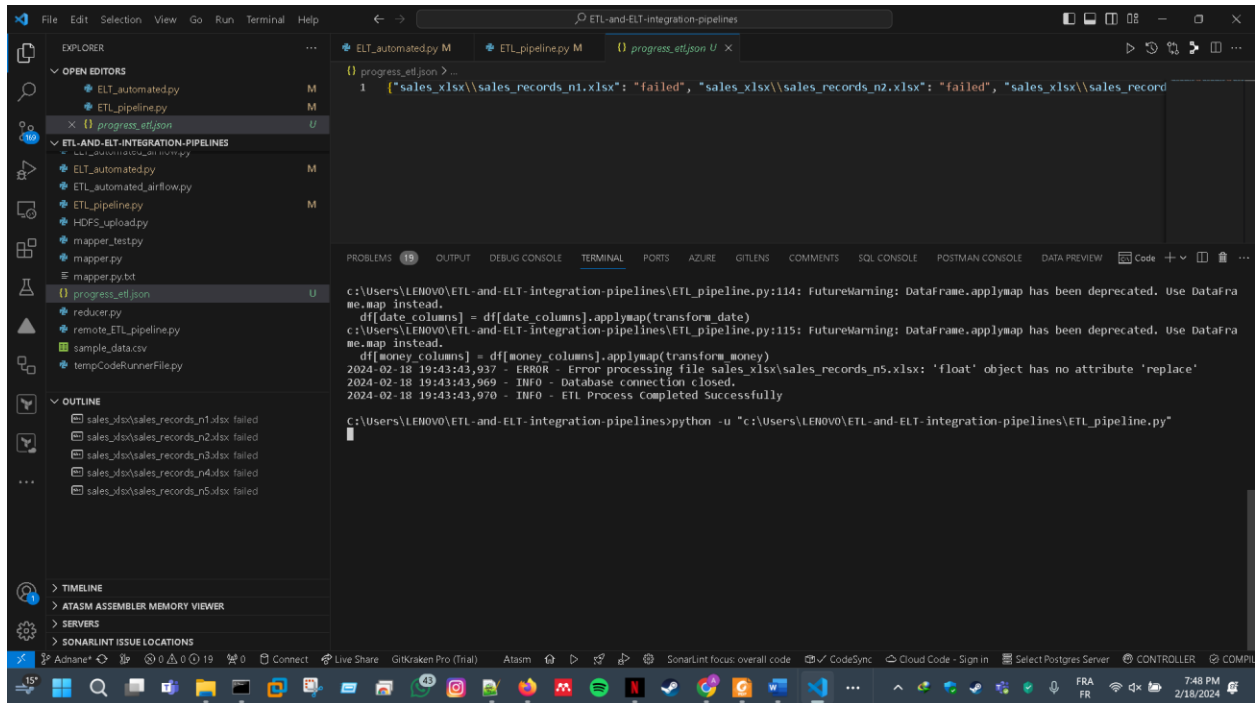
Database Transactions: The script uses database transactions when inserting records. If an error occurs during the insertion of a batch of records, the entire transaction is rolled back, ensuring atomicity. This means that the database operations are all-or-nothing, preventing partial updates that could lead to data inconsistencies.

All-or-Nothing File Processing: Each file is fully processed or not processed at all, as indicated by the `.processed` flag. This approach ensures that the data from each file is either completely integrated into the database or left untouched, adhering to the principle of atomicity.

Record-Level Checks: Before inserting each record, the script checks if the record already exists in the database. This ensures that each record is treated as an atomic unit, avoiding partial or duplicate entries.



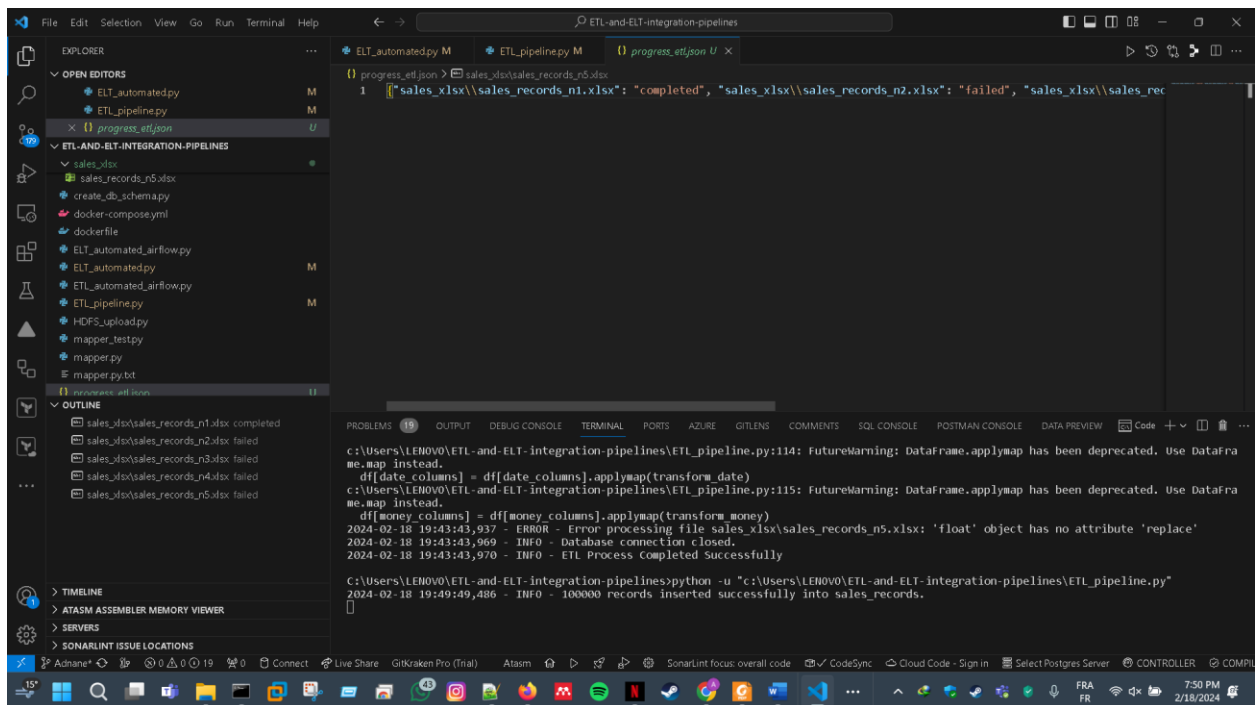
Running again to correct the failures based on the current state of the .json file :



```
progress_etl.json > ...
1 [{"sales_xlsx\\sales_records_n1.xlsx": "failed", "sales_xlsx\\sales_records_n2.xlsx": "failed", "sales_xlsx\\sales_record...
```

```
c:\Users\LEMOVO\ETL-and-ELT-integration-pipelines\ETL_pipeline.py:114: FutureWarning: DataFrame.applymap has been deprecated. Use DataFra
me.map instead.
df[date_columns] = df[date_columns].applymap(transform_date)
c:\Users\LEMOVO\ETL-and-ELT-integration-pipelines\ETL_pipeline.py:115: FutureWarning: DataFrame.applymap has been deprecated. Use DataFra
me.map instead.
df[money_columns] = df[money_columns].applymap(transform_money)
2024-02-18 19:43:43,937 - ERROR - Error processing file sales_xlsx\\sales_records_n5.xlsx: 'float' object has no attribute 'replace'
2024-02-18 19:43:43,969 - INFO - Database connection closed.
2024-02-18 19:43:43,970 - INFO - ETL Process Completed Successfully

c:\Users\LEMOVO\ETL-and-ELT-integration-pipelines>python -u "c:\Users\LEMOVO\ETL-and-ELT-integration-pipelines\ETL_pipeline.py"
```

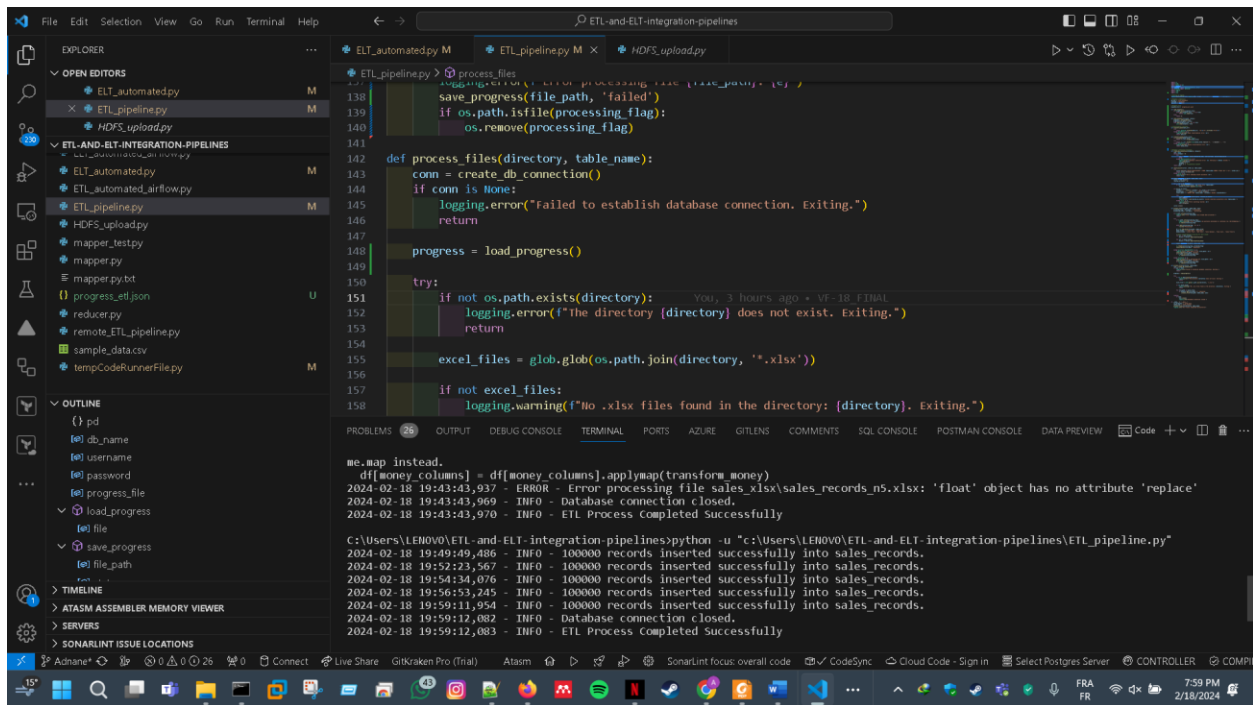


```
progress_etl.json > sales_xlsx\\sales_records_n5.xlsx
1 [{"sales_xlsx\\sales_records_n1.xlsx": "completed", "sales_xlsx\\sales_records_n2.xlsx": "failed", "sales_xlsx\\sales_rec...
```

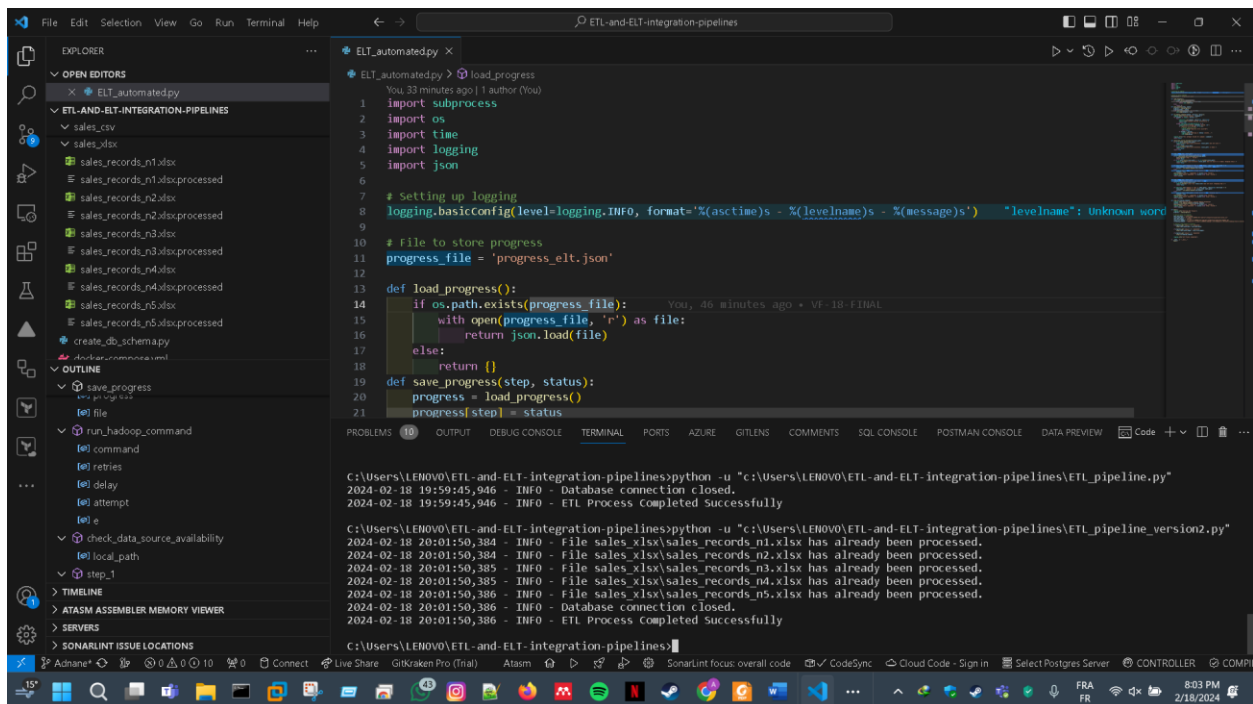
```
c:\Users\LEMOVO\ETL-and-ELT-integration-pipelines\ETL_pipeline.py:114: FutureWarning: DataFrame.applymap has been deprecated. Use DataFra
me.map instead.
df[date_columns] = df[date_columns].applymap(transform_date)
c:\Users\LEMOVO\ETL-and-ELT-integration-pipelines\ETL_pipeline.py:115: FutureWarning: DataFrame.applymap has been deprecated. Use DataFra
me.map instead.
df[money_columns] = df[money_columns].applymap(transform_money)
2024-02-18 19:43:43,937 - ERROR - Error processing file sales_xlsx\\sales_records_n5.xlsx: 'float' object has no attribute 'replace'
2024-02-18 19:43:43,969 - INFO - Database connection closed.
2024-02-18 19:43:43,970 - INFO - ETL Process Completed Successfully

c:\Users\LEMOVO\ETL-and-ELT-integration-pipelines>python -u "c:\Users\LEMOVO\ETL-and-ELT-integration-pipelines\ETL_pipeline.py"
2024-02-18 19:49:49,486 - INFO - 100000 records inserted successfully into sales_records.
```


Adnane El Amrani, Ayoub Maimmadi



```
{ "sales_xlsx\\sales_records_n1.xlsx":      "completed",      "sales_xlsx\\sales_records_n2.xlsx":
"completed",      "sales_xlsx\\sales_records_n3.xlsx":      "completed",
"sales_xlsx\\sales_records_n4.xlsx":      "completed",      "sales_xlsx\\sales_records_n5.xlsx":
"completed" }
```



- The ELT_automated.py script is designed to automate an ELT (Extract, Load, Transform) process using Hadoop. Let's analyze how it handles failures, restarts, fault tolerance, automation, and atomicity:

Handling Failures and Restarts:

Failure Handling:

Each Hadoop command is executed within the run_hadoop_command function, which includes a retry mechanism. If a command fails (detected by a subprocess.CalledProcessError), the script will wait for a specified delay (default is 5 seconds) and then retry the command. This process will repeat up to a set number of retries (default is 3).

In each step (step_1, step_2, step_3), the success of the operation is checked. If the operation is unsuccessful (i.e., if run_hadoop_command returns False), the script logs the failure and saves the progress as 'failed' for that step using the save_progress function.

Restart Handling:

The script uses a JSON file (progress.json) to track the progress of each step. The save_progress function updates this file after each step with the status 'completed' or 'failed'.

When the script starts (in the main function), it first loads the existing progress from progress.json using the load_progress function. Based on this progress, it determines which steps have been completed successfully.

If a step is marked as 'completed' in the progress file, the script skips that step and proceeds to the next one. This way, if the script is restarted after a failure, it will resume from the last uncompleted step rather than starting from the beginning.

Fault Tolerance:

The script includes a retry mechanism (run_hadoop_command) that attempts to run a command multiple times (default 3 retries) with a delay between retries. This can handle transient failures like network issues.

Logging is used to record successes and failures, which aids in troubleshooting.

Automation:

The script automates the ELT process in three main steps: loading data to HDFS, loading the mapper script to HDFS, and executing the Hadoop Streaming job.

Each step is methodically executed in sequence, ensuring that each process step is completed before moving to the next.

Atomicity:

The script checks for the existence of directories and files before performing operations, preventing partial loads or overwrites.

If a step fails, the script does not proceed to the next step, maintaining atomicity at a step level.

Adnane El Amrani, Ayoub Maimmadi

- The run_hadoop_command function includes an example of handling a specific type of error, which could be customized based on the types of errors you expect.
- A new function check_data_source_availability is added to verify if the local directory exists and is not empty before proceeding with loading data in step_1.
- The step_1 function is modified to use this new check, and the progress is updated accordingly.

Project ILOs:

Be able to write scripts to integrate data from plain data sources/ text with focus on fault tolerance, automation, and atomicity.