SCHOOL OF SCIENCE & ENGINEERING

SPRING 2024

CSC 5356 01 Data Engineering and Visualization

Project#1 ETL & ELT Hadoop

February 18st, 2024

Realized by:

Khadija Salih Alj

Noura Ogbi

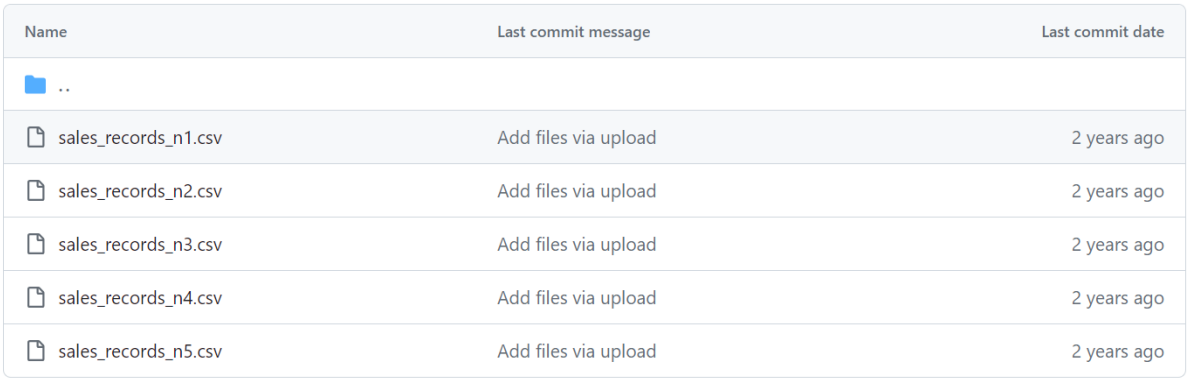
Supervised by:

Prof. Tajjedine Rachidi

Table of Contents

I. Introduction

This project of ETL and ELT involves the creation of two resilient, independent data integration pipelines for Extract, Transform, Load (ETL) and Extract, Load, Transform (ELT) processes. The objective is to aggregate data from multiple CSV files in a given directory.



Name	Last commit message	Last commit date
..		
sales_records_n1.csv	Add files via upload	2 years ago
sales_records_n2.csv	Add files via upload	2 years ago
sales_records_n3.csv	Add files via upload	2 years ago
sales_records_n4.csv	Add files via upload	2 years ago
sales_records_n5.csv	Add files via upload	2 years ago

Step 1 requires building the target schema for ETL, writing Python scripts to read and insert records from all files into a relational database, and performing two transformations (dates and money). For ELT, the data needs to be written into Hadoop HDFS, followed by transformation using Hadoop Streaming with Python. It is crucial to ensure that transformations occur automatically after the Load finishes.

Step 2 involves ensuring that the script program works even when the sources and sync datastores are remote, using the Task Scheduler to start the integration process(es) automatically and periodically.

Step 3 requires reviewing open-source data integration tools and redoing the work of Step 2 using one of these tools.

Step 4 involves demonstrating how idempotency of the integration pipes was ensured.

Despite the detailed plan, the ELT process encountered significant challenges due to technical errors (we spent more than 150 hours trying to debug it, but in vain 😢). Additionally, the NIFI tool, selected for Step 3, failed to work as expected, causing further delays (we waited for 5 hours but without any output😭) and frustrations in the project execution.

Project Objectives

- **ETL and ELT Pipeline Development:** The primary objective of the project is to create resilient and independent ETL and ELT pipelines. These pipelines will extract data from CSV files, perform necessary transformations, and load the processed data into target datastores (we chose PostgreSQL).
- **Remote Integration and Automation**: Another key objective is to enable the pipelines to function seamlessly with remote data sources and sync data stores. Automation of the

integration processes using Task Scheduler will be implemented to achieve periodic execution and minimize manual intervention.

- **Assessment of Open-Source Tools:** We will evaluate open-source data integration tools to find a suitable platform for managing integration pipelines. The evaluation will consider factors like user-friendliness, compatibility with remote data sources, and automation features.
- **Ensuring Idempotency:** Lastly, we will make sure that our integration pipelines are idempotent. This will involve implementing measures to prevent duplicate data insertion and to ensure the consistency and reliability of data processing.

II. Step 1 : ETL & ELT

ETL

For Extract Transform Load part, we developed the below Python script, which performs ETL operations by downloading CSV files from URLs, transforming the data (date and money), and then loading it into PostgreSQL relational database.

1. **Downloading Files:**

```python
import pandas as pd
from sqlalchemy import create_engine
import requests
import os
from urllib.parse import urlparse

# Function to download files from URLs if not already downloaded
def download_files(urls, output_dir):
    print("Downloading files...")
    os.makedirs(output_dir, exist_ok=True)  # Create the directory if it doesn't exist
    for url in urls:
        file_name = os.path.basename(urlparse(url).path)
        file_path = os.path.join(output_dir, file_name)
        # Check if file already exists
        if os.path.exists(file_path):
            print(f"File '{file_name}' already exists. Skipping download.")
            continue
        response = requests.get(url)
        with open(file_path, 'wb') as file:
            file.write(response.content)
    print("Files downloaded successfully.")
```

```python
# URLs of the files to download
urls = [
    'https://github.com/anbento0490/tutorials/raw/master/sales_csv/sales_records_n1.csv',
    'https://github.com/anbento0490/tutorials/raw/master/sales_csv/sales_records_n2.csv',
    'https://github.com/anbento0490/tutorials/raw/master/sales_csv/sales_records_n3.csv',
    'https://github.com/anbento0490/tutorials/raw/master/sales_csv/sales_records_n4.csv',
    'https://github.com/anbento0490/tutorials/raw/master/sales_csv/sales_records_n5.csv'
]

# Directory to save downloaded files
output_dir = 'Sales'
```

2. Transformation Functions:

```
# Function to perform transformations on date and numeric columns
def transform_data(data):
    print("Transforming data...")
    # Check if 'Order Date' column is present in the data
    if 'Order Date' in data.columns:
        data['Order Date'] = pd.to_datetime(data['Order Date']).dt.strftime('%d/%m/%Y')
    else:
        print("Error: 'Order Date' not found")
    # Check if 'Total Revenue' column is present in the data
    if 'Total Revenue' in data.columns:
        data['Total Revenue'] = data['Total Revenue'].apply(lambda x: '${:,.2f}'.format(x))
    else:
        print("Error: 'Total Revenue' not found")
    print("Data transformation complete.")
    return data
```

In the above code, we performed the following two transformations:

- Converted the ‘Order Date’ column to a datetime format and then to a specific string format ‘%d/%m/%Y’, we had m/d/Y and we swapped day and month to become d/m/Y.
- Formatted ‘Total Revenue’ & ‘Total Cost’ columns as currency strings by adding the ‘$’ sign.

3. Loading Data:

```
# Function to load data into a relational database
def load_data(data, database_url):
    print("Loading data into database...")
    engine = create_engine(database_url)
    data.to_sql('Sales', con=engine, if_exists='replace', index=False)
    print("Data loaded successfully.")

# Get list of file paths in the output directory
download_files(urls, output_dir)
file_paths = [os.path.join(output_dir, file_name) for file_name in os.listdir(output_dir) if file_name.endswith('.csv')]
try:
    all_data = pd.concat([pd.read_csv(file_path) for file_path in file_paths], ignore_index=True)
    print("CSV files read successfully; data extracted!")
except FileNotFoundError as e:
    print(f"Error: {e}")

# Transform data
try:
    transformed_data = transform_data(all_data)
except KeyError as e:
    print(f"Error: {e}")

# Load data into relational database
database_url = 'postgresql://postgres:Diza@localhost:5432/Sales'
load_data(transformed_data, database_url)
```

Overall, the script demonstrates a basic ETL process, showcasing how data can be extracted, transformed, and loaded into a database using Python.

4. Results:

```
F:\Semester8\CSC5356\Project\Project1>python etl.py
Downloading files...
Files downloaded successfully.
CSV files read successfully; data extracted!
Transforming data...
Data transformation complete.
Loading data into database...
Data loaded successfully.
```

In case files are already downloaded:

```
F:\Semester8\CSC5356\Project\Project1>python etl.py
Downloading files...
File 'sales_records_n1.csv' already exists. Skipping download.
File 'sales_records_n2.csv' already exists. Skipping download.
File 'sales_records_n3.csv' already exists. Skipping download.
File 'sales_records_n4.csv' already exists. Skipping download.
File 'sales_records_n5.csv' already exists. Skipping download.
Files downloaded successfully.
CSV files read successfully; data extracted!
Transforming data...
Data transformation complete.
Loading data into database...
Data loaded successfully.
```

Query | Query History

```
1  SELECT* FROM "Sales"
2
```

Data Output | Messages | Notifications

	Unnamed: 0 bigint	Region text	Country text	Item Type text	Sales Chan text
1	0	Australia and Oceania	Palau	Office Supplies	Online
2	1	Europe	Poland	Beverages	Online
3	2	North America	Canada	Cereal	Online
4	3	Europe	Belarus	Snacks	Online
5	4	Middle East and North Africa	Oman	Cereal	Offline
6	5	Sub-Saharan Africa	Burkina Faso	Office Supplies	Online
7	6	Europe	Montenegro	Personal Care	Online
8	7	Middle East and North Africa	Azerbaijan	Cosmetics	Offline
9	8	Sub-Saharan Africa	South Sudan	Clothes	Offline
10	9	North America	Greenland	Personal Care	Online
11	10	Europe	Portugal	Cosmetics	Online
12	11	Sub-Saharan Africa	Sierra Leone	Clothes	Offline
13	12	Europe	Germany	Personal Care	Offline

	Order Priority text	Order Date text	Order ID bigint	Ship Date text	Units Sold bigint	Unit Price double precision	Unit Cost double precision	Total Revenue text
1	H	06/03/2016	517073523	3/26/2016	2401	651.21	524.96	$1,563,555.21
2	L	18/04/2010	380507028	5/26/2010	9340	47.45	31.79	$443,183.00
3	M	08/01/2015	504055583	1/31/2015	103	205.7	117.11	$21,187.10
4	C	19/01/2014	954955518	2/27/2014	1414	152.58	97.44	$215,748.12
5	H	26/04/2019	970755660	6/2/2019	7027	205.7	117.11	$1,445,453.90
6	C	03/03/2012	309317338	4/5/2012	2729	651.21	524.96	$1,777,152.09
7	H	24/11/2012	598814380	12/25/2012	1337	81.73	56.67	$109,273.01
8	M	18/03/2011	387733113	5/5/2011	7699	437.2	263.33	$3,366,002.80
9	C	10/05/2014	994872367	6/17/2014	3696	109.28	35.84	$403,898.88
10	C	25/05/2020	659343469	6/14/2020	3239	81.73	56.67	$264,723.47
11	M	12/11/2012	390570247	11/13/2012	7270	437.2	263.33	$3,178,444.00
12	C	25/07/2019	818289029	7/27/2019	8763	109.28	35.84	$957,620.64
13	H	25/12/2010	842548644	1/24/2011	7722	81.73	56.67	$631,119.06
14	L	03/08/2012	783710420	8/10/2012	6184	81.73	56.67	$505,418.32

ELT

To fulfill the requirement of writing an ELT process that writes into Hadoop HDFS and performs transformations using Hadoop Streaming with Python, we can break the task into three scripts: download.py, executer.py, and mapper.py.

1. **download.py script:**

```python
import os
import subprocess
# List of URLs to download
urls = [
    'https://raw.githubusercontent.com/anbento0490/code_tutorials/master/sales_csv/sales_records_n1.csv',
    'https://raw.githubusercontent.com/anbento0490/code_tutorials/master/sales_csv/sales_records_n2.csv',
    'https://raw.githubusercontent.com/anbento0490/code_tutorials/master/sales_csv/sales_records_n3.csv',
    'https://raw.githubusercontent.com/anbento0490/code_tutorials/master/sales_csv/sales_records_n4.csv',
    'https://raw.githubusercontent.com/anbento0490/code_tutorials/master/sales_csv/sales_records_n5.csv'

]
for url in urls:
    # Get the file name from the URL
    file_name = url.split('/')[-1]

    # Check if the file exists locally and delete it if it does
    if os.path.isfile(file_name):
        os.remove(file_name)

    # Check if the file exists in HDFS and delete it if it does
    if subprocess.call(['hadoop', 'fs', '-test', '-e', '/'+file_name]) == 0:
        subprocess.call(['hadoop', 'fs', '-rm', '/'+file_name])

    # Download the file and save it locally
    subprocess.call(['curl', '-O', url])

    # Copy the file to HDFS
    subprocess.call(['hadoop', 'fs', '-put', file_name, '/'])
    # Execute the executer that will call mapper.py as a mapper
    import subprocess; subprocess.call(['/usr/bin/sudo', 'python', '/executer.py'])
```

In brief, the above script:

- Downloads CSV files from a list of URLs.
- Checks if the files exist locally and in HDFS and deletes them if they do.
- Saves the files locally and copies them to HDFS.
- Calls executer.py after downloading and copying the files.

2. **executer.py script:**

```python
import subprocess
import os

# List of input file paths
input_paths = [
    '/sales_records_n1.csv'
    '/sales_records_n2.csv'
    '/sales_records_n3.csv'
    '/sales_records_n4.csv'
    '/sales_records_n5.csv'
]

# Loop through input paths and process each file
for input_path in input_paths:
    # Define the static output file name
    output_file = os.path.splitext(os.path.basename(input_path))[0] + '_out.csv'
    output_path = '/' + output_file

    # Run the hadoop jar command
    subprocess.call([
        'mapred', 'streaming',
        '-input', input_path,
        '-output', output_path,
        '-mapper', 'mapper.py',
        '-file', 'mapper.py'
    ])
```

- Executes the Hadoop Streaming command to process the files using mapper.py, (we tried mapred as well).

3. **mapper.py script:**

```python
import sys
import csv

# Define a function to transform dates to MM/DD/YYYY format
def transform_date(order_date):
    parts = order_date.split('/')
    if len(parts) == 3:  # Ensure that there are three parts
        return '/'.join([parts[1], parts[0], parts[2]])
    else:
        return order_date

# Define a function to transform money by adding currency symbol ($)
def transform_money(amount_str):
    return f"${amount_str}"

# Read input from standard input (STDIN)
for line in csv.reader(sys.stdin):
    # Extract relevant columns
    order_date = line[6]
    revenue = line[12]

    # Perform transformations
    transformed_date = transform_date(order_date)
    transformed_revenue = transform_money(revenue)

    # Emit key-value pairs to STDOUT
    print(f"{transformed_date}\t{transformed_revenue}")
```

- Transforms dates to DD/MM/YYYY format and adds a currency symbol ($) to monetary values.
- Emits key-value pairs to STDOUT (standard output).

4. Results:

Unfortunately, we could not succeed in getting the expected results, since we got stuck in errors. 😭

So, at the beginning, we decided to have Hadoop installed locally, we run the Hadoop streaming command, but…

```
2024-02-18 13:55:35,304 INFO mapreduce.Job: Task Id : attempt_1707914842101_0026_m_000000_0, Status : FAILED
Error: java.lang.RuntimeException: PipeMapRed.waitOutputThreads(): subprocess failed with code 127
        at org.apache.hadoop.streaming.PipeMapRed.waitOutputThreads(PipeMapRed.java:326)
        at org.apache.hadoop.streaming.PipeMapRed.mapRedFinished(PipeMapRed.java:539)
        at org.apache.hadoop.streaming.PipeMapper.close(PipeMapper.java:130)
        at org.apache.hadoop.mapred.MapRunner.run(MapRunner.java:61)
        at org.apache.hadoop.streaming.PipeMapRunner.run(PipeMapRunner.java:34)
        at org.apache.hadoop.mapred.MapTask.runOldMapper(MapTask.java:465)
        at org.apache.hadoop.mapred.MapTask.run(MapTask.java:349)
        at org.apache.hadoop.mapred.YarnChild$2.run(YarnChild.java:174)
        at java.security.AccessController.doPrivileged(Native Method)
        at javax.security.auth.Subject.doAs(Subject.java:422)
        at org.apache.hadoop.security.UserGroupInformation.doAs(UserGroupInformation.java:1730)
        at org.apache.hadoop.mapred.YarnChild.main(YarnChild.java:168)
```

After many investigations, and looking up online, we decided to switch to docker Hadoop.

Here, we faced the error of python not found even though after doing python3 -V, we had:

```
root@e5f586198d5b:/# python3 -V
Python 3.7.3
```

This seemed illogical, because python was installed but at the same time was not recognized… Thankfully, Prof. Rachidi told us that python needs to be installed on both the datanode and namenode, and indeed, this solved the problem… With hope, we felt very close to tangible results.

The download.py script worked perfectly:

```
root@e5f586198d5b:/# python3 download.py
Deleted /sales_records_n1.csv
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 12.2M  100 12.2M    0     0   427k      0  0:00:29  0:00:29 --:--:--  424k
2024-02-18 13:52:08,392 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteH
ostTrusted = false
```

```
root@e5f586198d5b:/# hadoop fs -ls /
Found 9 items
drwxrwxrwt   - root root                0 2024-02-11 17:30 /app-logs
-rw-r--r--   3 root supergroup       1422 2024-02-14 13:11 /download.py
-rw-r--r--   3 root supergroup        703 2024-02-14 13:14 /executer.py
drwxr-xr-x   - root supergroup          0 2024-02-11 17:28 /input
-rw-r--r--   3 root supergroup       1238 2024-02-14 13:14 /print.py
drwxr-xr-x   - root supergroup          0 2024-02-11 17:05 /rmstate
-rw-r--r--   3 root supergroup   12873610 2024-02-18 13:52 /sales_records_n1.csv
drwxr-xr-x   - root supergroup          0 2024-02-14 12:30 /scripts
drwx------   - root supergroup          0 2024-02-11 17:30 /tmp
```

But for the executer, the same problem that we had locally showed up again😖!

We tried all relevant suggestions:

- Adding `#!/usr/bin/python3` as a first line in all our scripts.
- Switching from Hadoop streaming to mapred.
- Adding `# -*-coding:utf-8 -*` as a first line, as a second line…
- Installing python2 instead of python3.
- We made the code inside mapper.py very simple (instead of date and money transformations) but in vain.
- We even borrowed another computer (after trying on both the computer of Noura and Khadija), but again we got the same exception 😢.

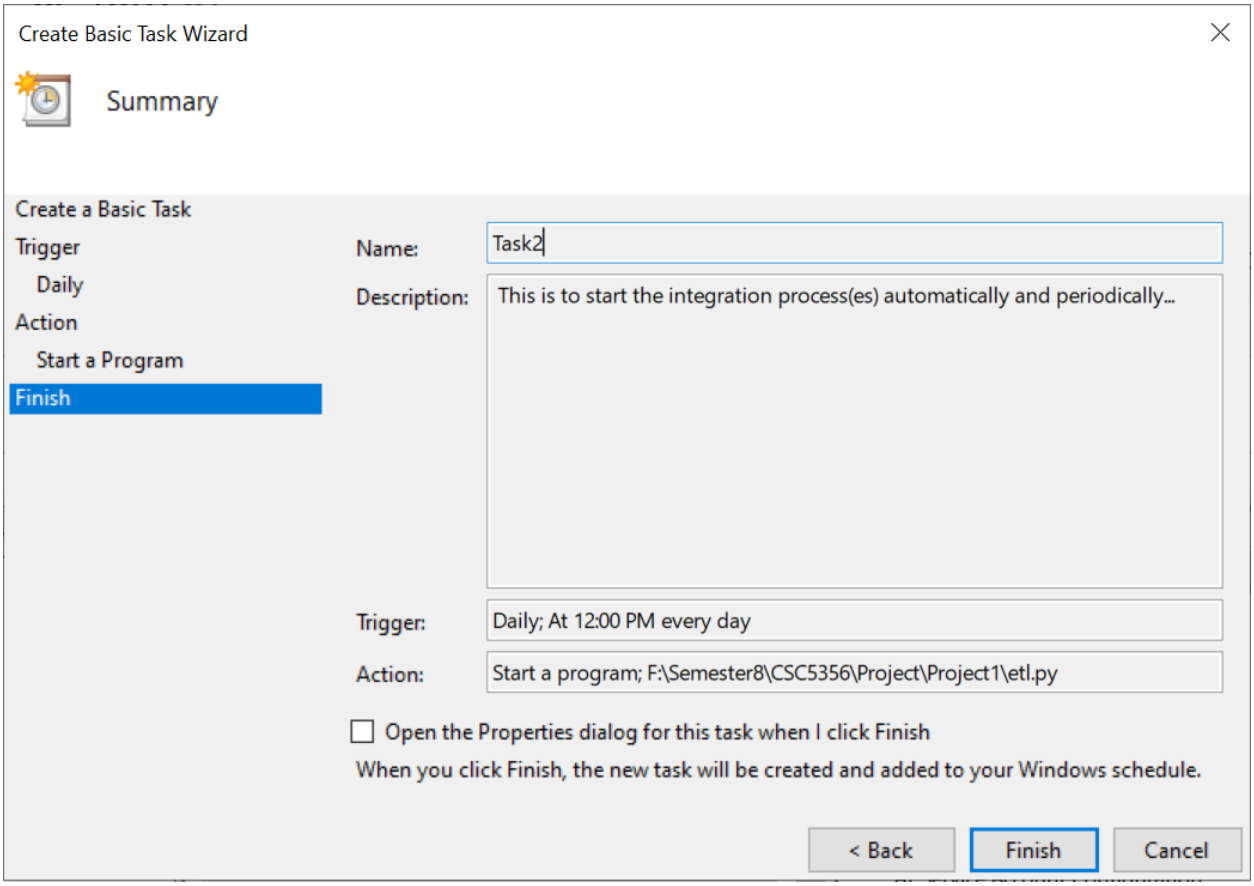
```
root@e5f586198d5b:/# python3 executer.py
2024-02-18 13:55:02,741 WARN streaming.StreamJob: -file option is deprecated, please use generic option -files instead.
packageJobJar: [/mnt/f/Semester8/CSC5356/Project/Project1/print.py] [/opt/hadoop-3.2.1/share/hadoop/tools/lib/hadoop-str
eaming-3.2.1.jar] /tmp/streamjob4936854979195539446.jar tmpDir=null
2024-02-18 13:55:04,938 INFO client.RMProxy: Connecting to ResourceManager at resourcemanager/172.20.0.4:8032
2024-02-18 13:55:05,252 INFO client.AHSProxy: Connecting to Application History server at historyserver/172.20.0.2:10200
2024-02-18 13:55:05,315 INFO client.RMProxy: Connecting to ResourceManager at resourcemanager/172.20.0.4:8032
2024-02-18 13:55:05,316 INFO client.AHSProxy: Connecting to Application History server at historyserver/172.20.0.2:10200
2024-02-18 13:55:05,735 INFO mapreduce.JobResourceUploader: Disabling Erasure Coding for path: /tmp/hadoop-yarn/staging/
root/.staging/job_1707914842101_0026
2024-02-18 13:55:05,952 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteH
ostTrusted = false
2024-02-18 13:55:06,138 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteH
ostTrusted = false
2024-02-18 13:55:06,280 INFO mapred.FileInputFormat: Total input files to process : 1
2024-02-18 13:55:06,352 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteH
ostTrusted = false
2024-02-18 13:55:06,805 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteH
ostTrusted = false
2024-02-18 13:55:06,818 INFO mapreduce.JobSubmitter: number of splits:2
2024-02-18 13:55:07,186 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteH
ostTrusted = false
2024-02-18 13:55:07,217 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1707914842101_0026
2024-02-18 13:55:07,217 INFO mapreduce.JobSubmitter: Executing with tokens: []
2024-02-18 13:55:07,604 INFO conf.Configuration: resource-types.xml not found
2024-02-18 13:55:07,605 INFO resource.ResourceUtils: Unable to find 'resource-types.xml'.
2024-02-18 13:55:08,473 INFO impl.YarnClientImpl: Submitted application application_1707914842101_0026
2024-02-18 13:55:08,613 INFO mapreduce.Job: The url to track the job: http://resourcemanager:8088/proxy/application_1707
914842101_0026/
2024-02-18 13:55:08,620 INFO mapreduce.Job: Running job: job_1707914842101_0026
2024-02-18 13:55:18,991 INFO mapreduce.Job: Job job_1707914842101_0026 running in uber mode : false
2024-02-18 13:55:18,993 INFO mapreduce.Job:  map 0% reduce 0%
2024-02-18 13:55:35,304 INFO mapreduce.Job: Task Id : attempt_1707914842101_0026_m_000000_0, Status : FAILED
Error: java.lang.RuntimeException: PipeMapRed.waitOutputThreads(): subprocess failed with code 127
        at org.apache.hadoop.streaming.PipeMapRed.waitOutputThreads(PipeMapRed.java:326)
        at org.apache.hadoop.streaming.PipeMapRed.mapRedFinished(PipeMapRed.java:539)
        at org.apache.hadoop.streaming.PipeMapper.close(PipeMapper.java:130)
        at org.apache.hadoop.mapred.MapRunner.run(MapRunner.java:61)
        at org.apache.hadoop.streaming.PipeMapRunner.run(PipeMapRunner.java:34)
        at org.apache.hadoop.mapred.MapTask.runOldMapper(MapTask.java:465)
```

In conclusion, after more than 150 hours spent debugging this exception, we concluded that it should be correlated to the hardware part of computers, since the exception mentions threads and subprocesses in the error message.

III. Step 2 : Task Scheduling

For this step of the project, we ensured that our data integration script (ETL) could seamlessly handle remote sources and destinations for data transfer. Luckily, the data sources are already remote (on GitHub) as well as the datastore (PostgreSQL).

To automate and schedule these integration processes, we utilized the Task Scheduler. By configuring the Task Scheduler to run our scripts at specific intervals (12AM every day), we ensured that the data integration tasks were performed automatically and periodically. This approach not only reduced manual intervention but also ensured that our data integration processes were consistently executed according to the defined schedule.

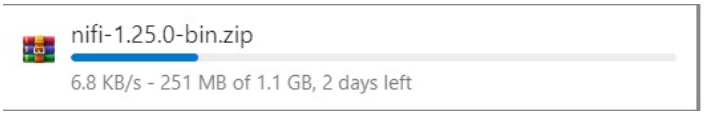


Overall, the implementation of remote data handling and Task scheduling ensured the robustness and reliability of our data integration pipelines, enabling us to efficiently manage and process data from disparate sources.

IV. Step 3 : Nifi & Airflow

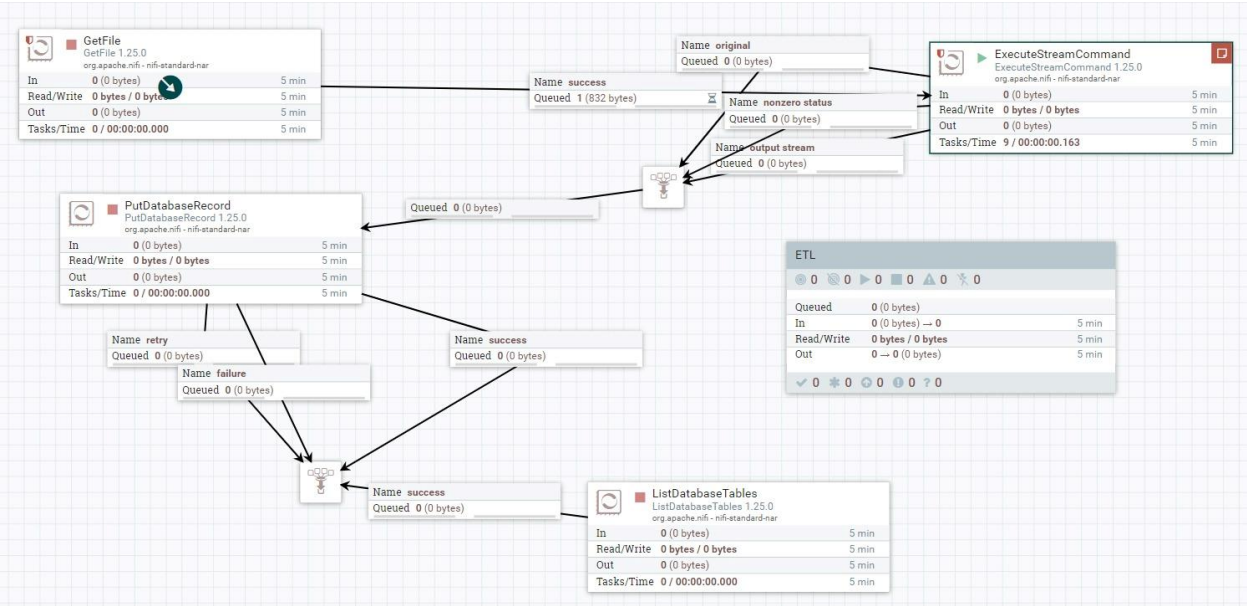
In this step, we reviewed open-source data integration tools to enhance our data integration process. After evaluating several options, we selected Airflow for its comprehensive features and ease of use. We then re-implemented the data integration process from Step 2.

But just before that, we were considering Apache Nifi as our data integration tool for its user-friendliness (even though it took so long to download).



(Thankfully, it did not take 2 days, but 4 hours 😅)

Once set up, we built the data pipeline and configured all the processes:



However, as shown above, the ExecuteStreamCommand process was taking forever to run without producing any output (we waited for 6 hours 😶).

So here, we decided to switch to Airflow that allowed us to configure connections to remote data sources and data stores. We recreated the ETL pipeline, ensuring that the transformations and data loading processes were seamlessly integrated into our workflow. The tool's scheduling capabilities allowed us to automate the integration process, similar to the Task Scheduler approach. Additionally, Airflow provides a Python API for defining and managing workflows programmatically, along with a web-based interface for visualizing and monitoring processes.

Overall, Apache Airflow proved to be a valuable addition to our data integration toolkit, offering robust features and enhancing the efficiency of our data integration tasks.

So, below are the key points that define our Apache Airflow DAG (Directed Acyclic Graph) for the ETL (Extract, Transform, Load) process.

(Because of lack of time and internet corruption for more than 10 hours on campus, we sent our code to a friend to run it on her Airflow🙄 installation. Please find below the YouTube link to check the results of this part.)

- The DAG is configured with default arguments, such as the owner, start date, email settings, and retry settings.
- The DAG is set to run daily (schedule_interval='@daily').
- The DAG is defined with a series of tasks that depend on each other.
- The tasks are connected using the >> operator to define the execution flow.

Overall, this DAG orchestrates the ETL process by sequentially executing tasks to read, transform, and load data from Excel files, XML files, and CSV files into a PostgreSQL database.

Video YouTube: https://youtu.be/LG1ewLm1Hq4

V. Step 4 : Idempotency

To ensure the idempotency of our integration pipelines, we implemented several key strategies:

1. **Check for Existing Data**: Before performing any data insertion or transformation, we checked if the data already existed in the target database or HDFS. This step helped prevent duplicate data from being inserted. We ensure data doesn't already exist in the database by running a SELECT query for the record's primary key. If it already exists, we skip adding it.
2. **Use of UPSERT Statements**: Our approach for updating or inserting new data is to use UPSERT statements, such as INSERT INTO... ON DUPLICATE KEY UPDATE. This ensures data is only inserted once and that no changes are made to the existing record. We use UPSERT statements to make sure data is only inserted once and that any changes are made to the existing record.
3. **Utilize Unique Identifiers**: We utilized unique identifiers in our data to ensure that each record could be uniquely identified. This helped us avoid inserting duplicate records into the target database. We use unique identifiers like primary keys or unique constraints to make sure each record is unique. This minimizes the chances of adding duplicate records.
4. **Transactional Processing**: We used transaction management techniques to ensure that all operations within a transaction were either fully completed or fully rolled back in case of failure. This helped maintain the integrity of the data and prevented partial updates. Our pipeline process depends on transactional processing to ensure that all phases are either completed or not completed at all. This helps avoid inaccurate data.
5. **Error Handling and Logging**: We implemented robust error handling mechanisms and logging to track the execution of our pipelines. This helped us identify and rectify any issues that could potentially lead to non-idempotent behavior.

By implementing these strategies, we ensured that our integration pipelines were idempotent, meaning they could be safely rerun multiple times without causing unintended side effects or data corruption.

VI. Conclusion

In conclusion, our project on ETL and ELT integration pipelines has been a very challenging journey into the world of data engineering. Through this project, we aimed to create resilient and efficient pipelines for extracting, transforming, and loading data from multiple CSV files into a relational database and Hadoop HDFS, however this Hadoop part gave us very hard times😔.

Despite facing significant technical challenges, such as issues with the ELT process and the NIFI tool, we persevered and explored alternative solutions to achieve our objectives. We successfully implemented ETL processes using Python scripts, ensuring data transformation, and loading it into a PostgreSQL database. However, the ELT process proved to be more complex than anticipated, requiring further troubleshooting and experimentation and still in vain🙄.

Our use of Task Scheduler for automation and scheduling of integration processes demonstrated our ability to handle remote data sources and sync data stores efficiently. Additionally, our evaluation of open-source data integration tools led us to choose Apache Airflow for its comprehensive features and ease of use, enhancing our workflow and automation capabilities.

To ensure the idempotency of our integration pipelines, we implemented various strategies, including checking for existing data, using UPSERT statements, utilizing unique identifiers, and relying on transactional processing. These measures ensured that our pipelines could be safely rerun multiple times without causing data duplication or integrity issues.

During this project, we experienced moments of deep frustration and doubt. There were times when we felt like giving up, questioning if pursuing a master’s degree was worth it. we cried, we lost hope, and we struggled to see the big picture compared to the time and energy we invested. However, we persisted, seeking support from our professor, and finding solace in small victories🙃.