MCO1: A Query Processing Technical Report

Bryan Alvarez[1], Lance Orendain[2], Sarah Jumilla[3], Sehyun Park[4]

College of Computer Studies, De La Salle University

[1]bryan_kahlil_g_alvarez@dlsu.edu.ph, [2]lance_galahad_m_orendain@dlsu.edu.ph, [3]sarah_jumilla@dlsu.edu.ph, [4]park_park@dlsu.edu.ph

Abstract:

This paper explores the optimization of data warehouses and query processing, bridging theoretical knowledge and practical skills. Our focus extends to formulating optimized queries and crafting an original data warehouse schema. The integration of Extract, Transform, Load (ETL) scripts and the implementation of an Online Analytical Processing (OLAP) Application further highlight the practical applications of our approach. The project serves as a test of our group's proficiency in query formulation, index utilization, and database restructuring. This real-life application demonstrates the effective organization and optimization of data, leading to improved insights and data management.

Keywords: Optimization, Data warehouse, Query processing, Extract, Transform, Load (ETL) scripts, Online Analytical Processing (OLAP) Application

1. Introduction

The Appointment dataset, comprising records from the SeriousMD startup company, serves as a source of information containing appointments, doctors, clinics, and patient data. To harness the analytical potential within this dataset, a data warehouse is essential. According to IBM, a data warehouse is a centralized, consistent data store that facilitates data analysis, mining, and supports artificial intelligence and machine learning [5].

With the intention of constructing a comprehensive data warehouse, this project focuses on designing a dimensional model in a star schema format. The fact table, centered around appointments, connects to dimension tables such as px, clinics, and doctors. This design choice is informed by the need for efficient Online Analytical Processing (OLAP) operations, specifically slice and dice, achievable by joining multiple fact and dimension tables [5].

The Extract, Transform, Load (ETL) script details the cleaning process of the SeriousMD datasets, involving Python for data cleaning and Apache Nifi for ETL processes. The cleaned data is then loaded into a MySQL data warehouse, utilizing a refined ETL pipeline that addresses initial bottlenecks and enhances processing speed.

The subsequent section delves into the OLAP application, implemented through Python Jupyter Notebook/Google Colab, showcasing visual dashboards. SQL queries play a crucial role in this application, requiring optimization for efficient performance. Query processing and optimization strategies, including indexing, partitioning, and parallel processing, are discussed as crucial elements in enhancing the overall system performance.

2. Data Warehouse

Data warehouse, as defined by International Business Machines (IBM) is a system that aggregates data from different sources into a single, central, consistent data store to support data analysis, data mining, artificial intelligence (AI), and machine learning [5].

The data sets which the group will process have been derived from SeriousMD start-up company. SeriousMD data sets includes for different csv files, namely appointments, doctors, clinics, and px. Each data contains a unique id to be used as a primary id, and appointments data includes the id from different data to be used as a foreign key. These factors will be later determined in designing the dimensional model for the data warehouse.

The dimensional model the group has designed to follow a star schema format (Figure 1). The design of the star schema modeling includes a fact table in the middle connected towards a different dimension table surrounding it. Dimension table and fact table each have one is-to many relationships. As mentioned above, each of the SeriousMD data sets includes a primary key in the data, and appointments data has access towards them to be used as a foreign key. Additionally, the large file size of each of the data sets may cause a delay in data loading time when designed incorrectly. Compared to snowflake schema, star schema has relatively faster data loading, and is best suited for OLAP operations such as slice and dice, as it can be achieved by joining two or more fact and dimension tables together. With all of the reasonings considered, the group will be utilizing a star schema design for the data warehouse.

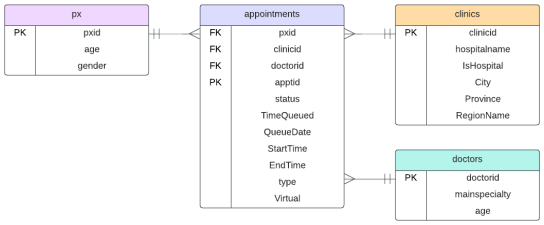

Figure 1. Data Warehouse Dimensional Model

The fact table of the data warehouse is appointments, it has a primary key of its own and three foreign keys which links the fact table to other dimensional tables. The hierarchy of this table flows from QueueDate, TimeQueued, StartTime, and EndTime, which can be used to perform OLAP operations such as roll up and drill down. The dimension tables connect to the fact tables are px, clinics, and doctors. All of the dimension tables include a primary key which is linked to the fact table. This setup can be used to implement OLAP operations such as slice and dice by simply joining multiple dimension tables with a fact table.

3. ETL Script

3.1. Data Cleaning

To clean the data, we have decided to use python. In cleaning px.csv, we first removed rows with NaN. Then rows that have the same pxid(or duplicates) as previous rows were removed. In the column “gender”, if it is not MALE or FEMALE, the row will be removed. The data type of age was changed from int to float since it was easier to use. Then if the age is a negative value or is greater than 100, the row was also removed.

```
import pandas as pd

df = pd.read_csv("/content/px.csv", engine='python', encoding='utf-8', error_bad_lines=False)
df = df.dropna(subset=['age','pxid','gender']) # Removes NaN
df = df.drop_duplicates(subset=['pxid']) # Removes duplicate pxid
df = df[df['gender'].isin(['MALE', 'FEMALE'])] # Removes any value not male or female
df['age'] = pd.to_numeric(df['age'], errors='coerce') # Changes dtype of age to float64
df = df[(df['age'] >= 0) & (df['age'] <= 100)] # Removes if age is negative or greater than 100
df.to_csv('cleaned_px.csv', index=False) # Save changes
```

Figure 2. Cleaning PX Dataset

To clean the doctors dataset, we installed fuzzywuzzy to be able to use the extractOne function. It is used to find the best match for a given ‘value’ within a list of ‘choices’ using fuzzy string matching. In cleaning doctors.csv, the group first removed rows with no values in the columns ‘age’ and ‘mainspecialty’. Newline characters from ‘mainspecialty’ were also removed to be uniform with others. If the string length for the doctorid is less than or equal to 25, it is removed. If the mainspecialty’s string length is greater than or equal to 2 or null, it is loaded into the dataset. If the age is between 18-100, the data will be loaded because doctors can only be 18 years old or above. Rows with both null or invalid characters in mainspecialty and age are removed. Duplicate rows were removed. Commas were replaced with “&” and are converted to uppercase letters. Numeric characters from mainspecialty were removed. Special signs were replaced with “AND”. Strings with “&” attached to a string were replaced with “ AND “ with space to detach it from the string. Values with 2 or fewer characters were renamed to “UNSPECIFIED”. Quotation marks were removed so that the data will be uniform. Rows with emails in their mainspecialty columns were removed. Rows with values ``N/A, na, NA, non, none, NOT” in their mainspecialty are removed if there is no age. Rows with repeated characters in their mainspecialty were also removed.

The use of fuzzy wuzzy was used in def correct_spelling(value, choices). This function was used to alter similar or misspelled words into one for consistency. If the words matched 90% of the choices, it will be replaced.

Then the group created a specific list to remove rows where mainspecialty is not a specialty for doctors:

1. SOFTWARE DEVELOPER
2. SOLSE
3. ADASD
4. SDAF
5. SUPER SAIYAN
6. FPOGSF AND PSMFMF AND PSUOG
7. KDSJFLKFDSJG
8. SDFSDA
9. CFVHGBJNK
10. DOCTOR PAYTON
11. UNSPECIFIED
12. ASDASD
13. ADASD
14. SASA
15. WQEQWE

The group also created a list for specific spelling instructions in case the fuzzy wuzzy function was not applied to these:

1. 'SURGUY', 'SURGERY
2. 'SIRGERY', 'SURGERY'
3. 'SUPORT', 'SUPPORT'
4. 'ASESMENT', 'ASSESSMENT'
5. 'GALBLADER', 'GALLBLADDER'
6. 'WELNES', 'WELLNESS'
7. 'SESIONS', 'SESSIONS'
8. 'STRES', 'STRESS'
9. 'SICKNES', 'SICKNESS'
10. 'KNE', 'KNEE'
11. 'GENRAL', 'GENERAL'
12. 'INTENAL', 'INTERNAL'

The group also created a correct_spelling function to correct some other misspelled words in the mainspecialty.

Overall, doctors.csv is not yet clean even with all the preprocessing and cleaning the group did on the dataset. The dataset is full of outliers and the only viable way of cleaning it is to manually clean it.

```
import pandas as pd
import re
!pip install fuzzywuzzy
from fuzzywuzzy import process

# Read the CSV file into a DataFrame
df_doctors = pd.read_csv('/content/doctors.csv' , engine='python', encoding = "ISO-8859-1", error_bad_lines=False)

# Remove rows with no age and mainspecialty
df_doctors = df_doctors.dropna(subset=['age', 'mainspecialty'])

# Remove newline characters from 'mainspecialty' column
df_doctors['mainspecialty'] = df_doctors['mainspecialty'].replace('\n', '', regex=True)

# Only load data if doctorid's string length is > 25
df_doctors = df_doctors[df_doctors['doctorid'].str.len() > 25]

# Only load data if mainspecialty's string length is >= 2 or null
df_doctors = df_doctors[(df_doctors['mainspecialty'].str.len() >= 2) | df_doctors['mainspecialty'].isnull()]

# Only load data if age is between 18 - 100
df_doctors = df_doctors[(df_doctors['age'] >= 18) & (df_doctors['age'] <= 100)]

# Remove rows with both null or invalid char mainspecialty and age
df_doctors = df_doctors.dropna(subset=['mainspecialty'], how='all')
```

Figure 3. Snippet 1 of Cleaning doctors.csv

```
df_doctors = df_doctors.dropna(subset=['age'], how='all')

# Drop duplicate data
df_doctors = df_doctors.drop_duplicates()

# Replace commas with "&" and convert to all caps
df_doctors['mainspecialty'] = df_doctors['mainspecialty'].str.replace(',', '&')

# All Caps for Consistency in mainspecialty
df_doctors['mainspecialty'] = df_doctors['mainspecialty'].str.upper()

# Remove numeric characters from mainspecialty
df_doctors['mainspecialty'] = df_doctors['mainspecialty'].apply(lambda x: re.sub(r'\d', '', str(x)))

# Replace special signs with "AND"
df_doctors['mainspecialty'] = df_doctors['mainspecialty'].replace({'&': 'AND', '/': 'AND', '+': 'AND', '-': ' ', ':': ' ', ')': ' '})

# Replace "&" that is attached to a character and convert to all caps
df_doctors['mainspecialty'] = df_doctors['mainspecialty'].str.replace(r'&([A-Za-z])', r'\1 AND ').str.upper()

# Replace values with 2 or fewer characters to "unspecified"
df_doctors.loc[df_doctors['mainspecialty'].str.len() <= 2, 'mainspecialty'] = 'UNSPECIFIED'

# Remove quotation marks
df_doctors['mainspecialty'] = df_doctors['mainspecialty'].str.replace('"', '')
```

Figure 4. Snippet 2 of Cleaning doctors.csv

```
# Remove rows with emails in the 'mainspecialty' column
df_doctors = df_doctors[~df_doctors['mainspecialty'].str.contains(r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b', regex=True)]

# Remove rows with values "N/A, na, NA, non, none, NOT" in mainspecialty if there is no age
na_values = ['N/A', 'na', 'NA', 'non', 'none', 'NOT']
df_doctors = df_doctors[~((df_doctors['mainspecialty'].str.lower().isin(na_values)) & df_doctors['age'].isnull())]

# Replace '&' with ' AND ' in 'mainspecialty' column
df_doctors['mainspecialty'] = df_doctors['mainspecialty'].str.replace('&', ' AND ')

# Remove rows with repeated characters in mainspecialty
df_doctors['mainspecialty'] = df_doctors['mainspecialty'].apply(lambda x: re.sub(r'(.)\1+', r'\1', str(x)))

# Alter similar or misspelled word into one for consistency
def correct_spelling(value, choices): # Alters similar or misspelled word into one for consistency (e.g. Adicti0n -> Addiction)
                                      # Who knows if this function actually works
    result, score = process.extractOne(value, choices)
    return result if score >= 90 else value # If word matched 90% of the choices, replace it

# Remove rows where 'mainspecialty' is not a specialty for doctors
df_doctors = df_doctors[(df_doctors['mainspecialty'].str.upper() != 'SOFTWARE DEVELOPER') & (df_doctors['mainspecialty'].str.upper() != 'SOLSE') & (df_doctors['mainspecialty'].str.upper()
& (df_doctors['mainspecialty'].str.upper() != 'SDAF') & (df_doctors['mainspecialty'].str.upper() != 'SUPER SAIYAN') & (df_doctors['mainspecialty'].str.upper() != 'FPOGSF AND PSMFMF AND PSU
& (df_doctors['mainspecialty'].str.upper() != 'SDFSDA') & (df_doctors['mainspecialty'].str.upper() != 'KDSJFLKFDSJG') & (df_doctors['mainspecialty'].str.upper() != 'CFVHGBJNK') & (df_docto
& (df_doctors['mainspecialty'].str.upper() != 'ASDASD') & (df_doctors['mainspecialty'].str.upper() != 'ADASD') & (df_doctors['mainspecialty'].str.upper() != 'SASA') & (df_doctors['mainspec

df_doctors['mainspecialty'] = df_doctors['mainspecialty'].str.replace('SURGUY', 'SURGERY', case=False)
df_doctors['mainspecialty'] = df_doctors['mainspecialty'].str.replace('SIRGERY', 'SURGERY', case=False)
df_doctors['mainspecialty'] = df_doctors['mainspecialty'].str.replace('SUPORT', 'SUPPORT', case=False)
df_doctors['mainspecialty'] = df_doctors['mainspecialty'].str.replace('ASESMENT', 'ASSESSMENT', case=False)
df_doctors['mainspecialty'] = df_doctors['mainspecialty'].str.replace('GALBLADER', 'GALLBLADDER', case=False)
df_doctors['mainspecialty'] = df_doctors['mainspecialty'].str.replace('WELNES', 'WELLNESS', case=False)
```

Figure 5. Snippet 3 of Cleaning doctors.csv

```
df_doctors['mainspecialty'] = df_doctors['mainspecialty'].str.replace('SESIONS', 'SESSIONS', case=False)
df_doctors['mainspecialty'] = df_doctors['mainspecialty'].str.replace('STRES', 'STRESS', case=False)
df_doctors['mainspecialty'] = df_doctors['mainspecialty'].str.replace('SICKNES', 'SICKNESS', case=False)
df_doctors['mainspecialty'] = df_doctors['mainspecialty'].str.replace('KNE', 'KNEE', case=False)
df_doctors['mainspecialty'] = df_doctors['mainspecialty'].str.replace('GENRAL', 'GENERAL', case=False)
df_doctors['mainspecialty'] = df_doctors['mainspecialty'].str.replace('INTENAL', 'INTERNAL', case=False)

df_doctors['mainspecialty'] = df_doctors['mainspecialty'].apply(correct_spelling, choices=['ADDICTION', 'AESTHETIC', 'ANESTHESIOLOGY','APPLIED', 'ASSESSMENT', 'CARDIOLOGY',
                                                 'CHIROPRACTOR', 'DERMITOLOGY','ENDOCRINOLOGY', 'GYNE','PRACTITIONER', 'SURGERY', 'SUPPORT', 'UROLOGY'])
# Save the cleaned DataFrame to a new CSV file
df_doctors.to_csv('/content/cleaned_doctors.csv', index=False)
```

Figure 6. Snippet 4 of Cleaning doctors.csv

In the graph below, we can see that there are thousands of main specialties after cleaning. It was hard to group main specialties with acronyms and its counterpart meanings ex.: ‘OB GYN’, ‘OBSTETRICS AND GYNECOLOGY’, and ‘OB-GYN’. There are lots of variations that were made in the column of mainspecialty for each field.

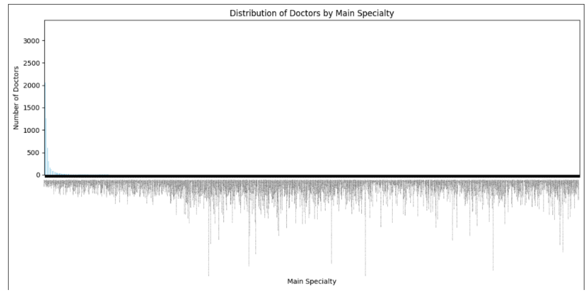

Figure 18. Bar Graph of doctors.csv after cleaning

In cleaning appointments.csv, the group had to upload the csv file into google drive in order to upload it into google colab since the raw uncleaned file size was about 2GB. The cleaned_doctors.csv file was also read in order to just get the rows in appointments.csv wherein the doctorid exists in cleaned_doctors.csv. Then ‘QueueData’, ‘TimeQueued’, ‘StartTime’, and ‘EndTime’ were converted into datetime format into month, date, year, time. QueueDate does not need time so time was removed in the conversion.

```
import pandas as pd
# Link your gdrive and upload the appointments.csv there
# Change the directory according to your google drive
df = pd.read_csv("/content/drive/MyDrive/STADVDB/appointments.csv", engine='python', encoding = "ISO-8859-1", error_bad_lines=False)
df_doctors = pd.read_csv("/content/cleaned_doctors.csv", engine='python', encoding = "ISO-8859-1", error_bad_lines=False)
# Only get the rows where doctorid exists in cleaned_doctors csv
df = df[df['doctorid'].isin(df_doctors['doctorid'])]

# Automatically converts the field into datetime format
df['QueueDate'] = pd.to_datetime(df['QueueDate'], errors='coerce', infer_datetime_format=True)
df['TimeQueued'] = pd.to_datetime(df['TimeQueued'], errors='coerce', infer_datetime_format=True)
df['StartTime'] = pd.to_datetime(df['StartTime'], errors='coerce', infer_datetime_format=True)
df['EndTime'] = pd.to_datetime(df['EndTime'], errors='coerce', infer_datetime_format=True)

# Changes the formatting of the datetime
df['QueueDate'] = df['QueueDate'].dt.strftime('%m/%d/%Y') # QueueDate doesn't need time, so remove time
df['TimeQueued'] = df['TimeQueued'].dt.strftime('%m/%d/%Y %I:%M:%S %p') # Others, keep the time
df['StartTime'] = df['StartTime'].dt.strftime('%m/%d/%Y %I:%M:%S %p')
df['EndTime'] = df['EndTime'].dt.strftime('%m/%d/%Y %I:%M:%S %p')

df.to_csv('cleaned_appointments.csv', index=False) # Save changes
```

Figure 7. Cleaning appointments.csv

In cleaning clinics.csv, rows with missing values in important columns: 'clinicid', 'IsHospital', 'City', 'Province', and 'RegionName' were dropped. Duplicate rows based on the ‘clinicid’ column were removed.

```
import pandas as pd

# Assuming the file path for clinic.csv
clinics_file_path = '/content/clinics.csv'

# Read the CSV file into a DataFrame with an alternative encoding
df_clinics = pd.read_csv(clinics_file_path, encoding='latin-1')

# Drop rows with missing values in important columns
df_clinics = df_clinics.dropna(subset=['clinicid', 'IsHospital', 'City', 'Province', 'RegionName'])

# Drop duplicate rows based on the 'clinicid' column
df_clinics = df_clinics.drop_duplicates(subset=['clinicid'])

# Save the cleaned DataFrame to a new CSV file
df_clinics.to_csv('/content/cleaned_clinics.csv', index=False)
```

Figure 8. cleaning clinics.csv

3.2. Apache Nifi

A subsequent step after cleaning the SeriousMD datasets is to utilize Apache Nifi to perform extraction, transformation, and loading (ETL) of the dataset from csv files into the MySQL data warehouse.

Apache NiFi is an open-source data integration and automation tool designed to facilitate the flow of data between disparate systems. NiFi provides a wide range of processors, connectors, and processors for integrating with different data systems and services.

The NIFI version chosen to be worked with is version 1.24, as it supports Java versions as low as version 8 as mentioned by the official documentation [1].

Since all of the SeriousMD datasets are following the csv file format, the initial flow created to perform ETL process was as follows.

Listing 1. Initial ETL Pipeline to transfer data

GetFile -> SplitRecord -> PutDatabaseRecord

There are three unique processors used for the initial ETL pipeline, each handling different stages of the file transfer process. The GetFile processor would retrieve the files within the path specified by the user and allow the downstream processor to retrieve them as a flowfile. In this pipeline, the SplitRecord processor was simply used to convert the flowfile file type from csv to JSON using the default controllers, record reader *CSVReader* and record writer *JsonRecordSetWriter* provided by the Apache Nifi. Finally PutDatabaseRecord would read the JSON flowfile with the Nifi-provided *JsonTreeReader* controller and perform *INSERT* keyword into the data warehouse table users have specified in the controller found at Database Connection Pooling Service.

Unfortunately, there were multiple problems with the initial ETL pipeline. Firstly, Apache Nifi has started sending out an error statement: *Unable to write flowfile content to content repository container default due to archive file size constraints*. Upon further research, this was caused by having a low number of Records Per Split in the SplitRecord processor, This would cause Apache Nifi to create a large number of small flowfiles, causing increased process overhead which ultimately results in memory restriction [2].

The simplest solution to address this problem was to increase the number of Records Per Split to reduce the number of flowfiles. Our team has increased the amount of Records Per Split from 1 into 300 to group the flowfiles into a bigger file and have ceased the problem. In addition to increasing the number of Records Per Split, our team has found that by disabling the *nifi.content.repository.archive.enabled* found in the nifi.properties file, Apache Nifi would disregard the files sent to the content repository due to the previously mentioned problem [3]. This allowed Apache Nifi to continuously work on the pipeline while disregarding the errors found by the previous mistakes.

The second problem our team has encountered was the speed of the ETL pipeline transactions. This problem wasn't caused by the malfunctioning processors, but rather by the structure of the ETL pipeline the group had created. We have noticed that there was a bottleneck in the flowfile coming from SplitRecord to PutDatabaseRecord, as there was only a single route to transact a large amount of data. This problem was overlooked by the team during the structuring of the initial pipeline as the team has never utilized a large dataset with Apache Nifi before.

With this, a new ETL pipeline had to be designed to address the extreme bottleneck found at the end of the pipeline. The idea the group had come up with was to assign a unique id or value in each of the flowfile and distribute them on separate PutDatabaseRecord processors in accordance with their id. With further research, the group figured that the UpdateAttribute processor can be utilized to assign a specific property to each flowfiles [4].

UpdateAttribute process has a configuration to store state locally and we can utilize the local variable inside the processor to assign an unique id to the flowfiles (Figure 9.).

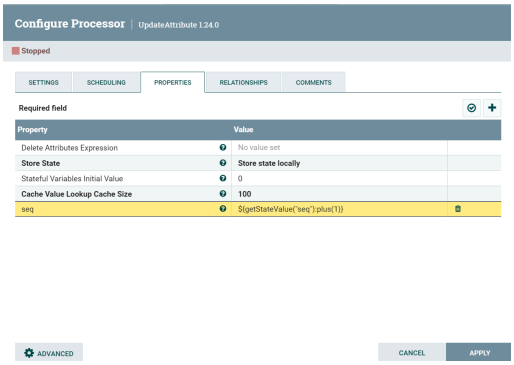

Figure 9. UpdateAttribute Processor Configuration

The group has created a new property *seq* in UpdateAttribute processor which utilizes the query

${getStateValue("seq"):plus(1)}

This query would retrieve the current value of the *seq* property and increment its value by 1. This value will be used in the downstream processor to segregate the flowfile into different destinations.

The downstream processor of UpdateAttribute the group has utilized is RoutesOnAttribute. This processor allows the user to create a routing condition for the upcoming flowfile and redirect them accordingly (Figure 10.).

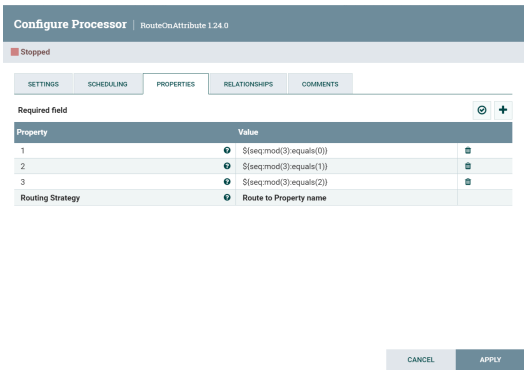

Figure 10. RoutesOnAttribute Processor Configuration

The group has configured the processor to have three different routes, and utilized the following query to segregate them.

${seq:mod(3):equals(x)}

This query would apply a mod function which will mod the value of the *seq* property assigned to each flowfile by 3. The value x found in the query x ranges from 0 to 2 to segregate the flowfile depending on their modded values. Each segregated flowfile would then flow into the final processor PutDatabaseRecord which will perform *INSERT* query into the specified MySQL schema table.

This processor would allow creation of concurrent processing of flowfile in the ETL pipeline, and the amount of concurrency can be increased by simply increasing the number of routes in the processor.

With this, the final ETL pipeline will look as follows (Figure 11, Listing 2)

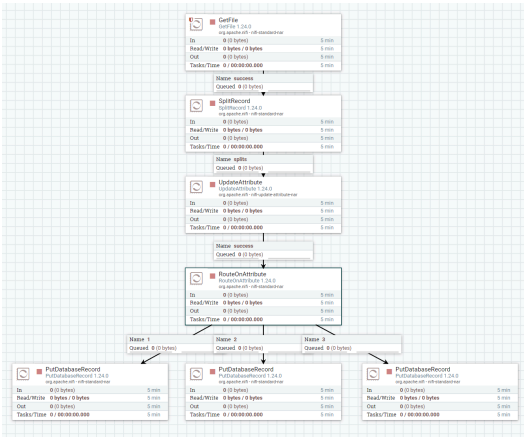

Figure 11. Final ETL Pipeline

Listing 2. Final ETL Pipeline to transfer data

GetFile -> SplitRecord -> UpdateAttribute -> RouteOnAttribute -> PutDatabaseRecord (x3)

4. OLAP Application

The application serves as a platform designed for the purpose of delivering analytical reports on the state of medical care in the Philippines, based solely on the *SeriousMD* dataset. The information that it generates can be observed and used as reference when making decisions regarding the accessibility and quality of health services within the country.

The first query involves a display of the number of clinics by location, either in a given city, province, or region. This can assist researchers in identifying which areas are heavily equipped with medical resources and which areas lack access to it. This makes use of a roll-up operation, enabling users to switch views between the number of clinics in a city, which has the highest level of granularity, to province, and then to region, which has the lowest level of granularity. Within the application, users can interact with the dropdown menu and select the location type they wish to explore. Upon selection, a visual representation of it is displayed via a bar chart, which would help users to easily discern differences in the data (Figure 12). In *SQL*, it uses an aggregate function *COUNT()* in calculating the number of clinics present in the dataset. Additionally, it makes use of a *CASE WHEN* statement, which is a conditional expression that aids in directing which data to load based on the location type that the user has selected.

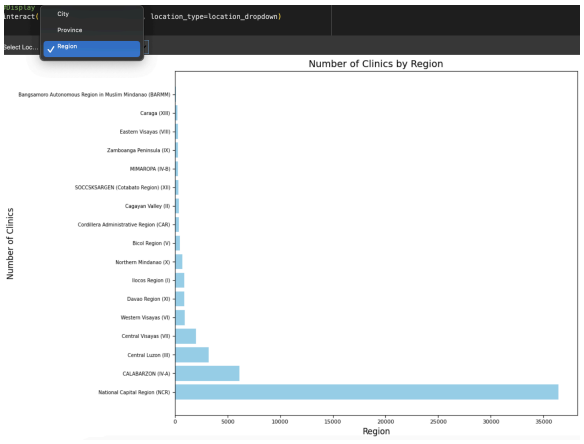

Figure 12. Visualization for Number of Clinics by Location

The second query concerns the distribution of appointments among doctors and among the main specialties of doctors.

This helps pinpoint doctors and specialties that are in demand based on the number of appointments they have received. When inspecting the top doctors, we can consider that these are the most sought-after by patients, whether or not this is due to great service or other reasons. When inspecting the top specialties, users are able to discover which specialties handle a high number of patients and as a result, researchers may determine that these departments need to hire more staff. This uses a drill-down operation as users can view from doctors to the doctor's main specialty, obtaining a more detailed level of information. Within the application, users can change between the two visualizations, with bar charts chosen for their intuitive display of data (Figure 13). Its equivalent *SQL* statement uses a *JOIN* between the *appointments* table and the *doctors* table and connects through the *doctorid* and uses an aggregate function to count the number of appointments per doctor and doctor's main specialty.

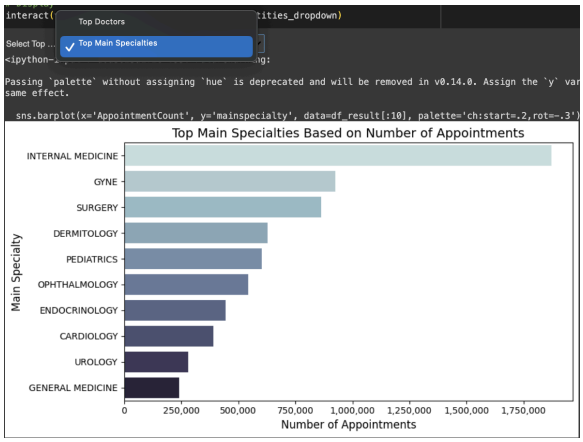

Figure 13. Visualization for Top Category Based on Number of Appointments

The third query is for evaluating the virtual and non-virtual appointments throughout the years. This facilitates observing the growth of appointments that are held virtually or online over time, gaining insight into the use of a new way of accessing medical care. This uses a slice operation to obtain data from the appointments table and where the Virtual dimension is set as *'True*,' acquiring only relevant and needed data information. The data is displayed in a line graph since it is a useful tool for demonstrating trends in both types of appointments data (Figure 14). This, written in *SQL*, involves the use of a *WHERE* clause to specify either slicing virtual or non-virtual appointments data and a *GROUP BY* clause for the years that is needed for the comparison.

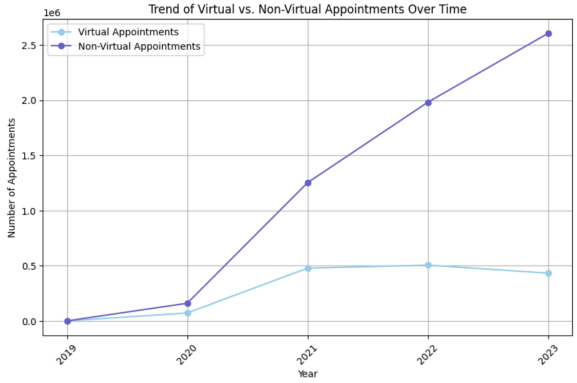

Figure 14. Visualization for Virtual vs. Non-Virtual Appointments Over Time

The fourth query is related to the third query, but it examines the Virtual dimension set as *'True'* and another dimension, which is the *hospitalname* that is set depending on the user's selection. Accessing this data is beneficial as this allows users from particular hospitals to analyze more focused data that they may find pertinent. By indicating the hospital, they are able to see the growth of Virtual appointments throughout the years in their own clinic. As described, it uses a dice operation, as it operates with two specific dimensions, generating a subcube. Within the application, users can browse through specific clinic data by hovering and selecting through the dropdown menu the clinic they wish to review. The movement of the data points is then displayed in a line graph, making it simple to identify any patterns (Figure 15). In *SQL*, it uses a *COUNT()* aggregate function and indicates its specification through the *WHERE* clause in order to get the smaller piece of data that will be examined.

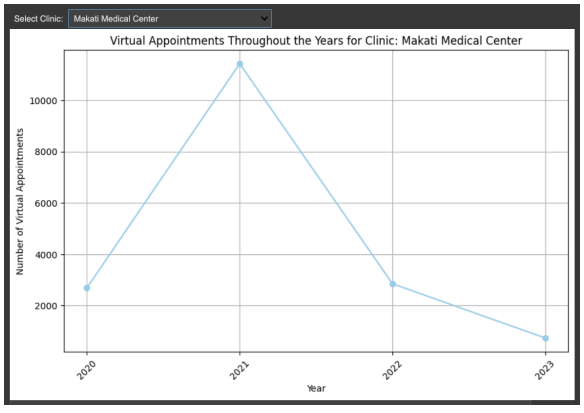

Figure 15. Visualization for Visual Appointments Throughout the Years for a Specific Clinic

5. Query Processing and Optimization

Query optimization refers to the process of improving the performance and efficiency of SQL queries, which are used to retrieve and manipulate data from a relational database management system (RDBMS) [6]. Utilizing a query optimization may result in significant performance amplification when used properly. With better performance, an application will be able to calculate results faster and enhance user experience.

There are several query optimization strategies which the developers can utilize. *Indexing* creates a separate data structure which allows the queries to retrieve and locate data in a more efficient way. Indexing is best utilized when performing GROUP BY or ORDER BY clauses, as data structure created through indexing reduces the time it takes for sorting and grouping operations by allowing the query to search for conditional data faster. *Partitioning* is a technique where a large amount of data is divided into a smaller group of data. Partitioning improves management and maintenance of data, while also allowing segregation of data.

Both indexing and partitioning was utilized when formulating the OLAP application, and specific details are as follows:

During the performance of ETL while utilizing an apache NIFI, the initial design the group created encountered a slow ETL transaction speed as there was a large single flowfile which was causing a bottleneck at the end of the pipeline. To resolve the problem, one of the solutions applied to the ETL pipeline was to partition the data set into a bigger "split" and allow different processors to handle each segregated data separately (Figure. 16).

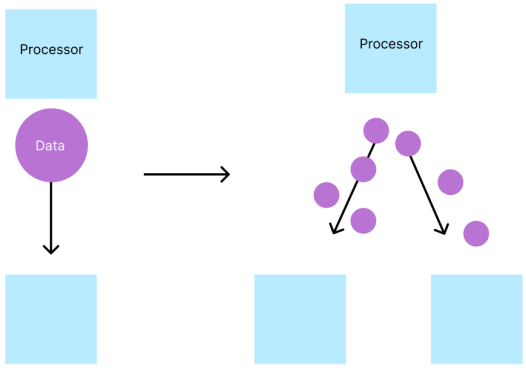

Figure 16. Visualization of Partitioning in ETL pipeline

This has allowed the ETL pipeline to perform at a significantly faster speed and was able to resolve the memory problem an application was experiencing.

In the later segment of the OLAP application, indexing optimization technique was utilized to decrease the query execution time of data visualization in python jupyter. One of the visualizations our group has proposed is a query *for top doctors based on the number of appointments* with the following SQL queries.

Listing 3. query for top doctors based on the number of appointments

```
SELECT
   d.doctorid,
   d.mainspecialty,
   COUNT(a.apptid) AS AppointmentCount
FROM
   appointments a
JOIN
   doctors d ON a.doctorid = d.doctorid
GROUP BY
   d.doctorid,
   d.mainspecialty
ORDER BY
   AppointmentCount DESC;
```

The following set of query involves costly operations such as SUM, JOIN, and GROUP BY. To reduce the query execution time of the above query, indexing can be utilized to create a separate data structure which the query can use to perform the statements faster. This will allow the query to not go through the entire rows, but rather just the selected portions.

Listing 4. Indexing Statements

```
CREATE INDEX IF NOT EXISTS
  idx_appointments_doctorid
ON
  appointments(doctorid);

CREATE INDEX IF NOT EXISTS
  idx_doctors_doctorid_mainspecialty
ON
  doctors(doctorid, mainspecialty);
```

The query above is used to create a two separate index for the JOIN conditions. The first query simply creates an index with doctorid sourced from the appointments table, while the second query creates an index with both doctorid and mainspecialty sourced from doctors table. With the existence of an index in an appropriate column, when the

SQL query is called in the OLAP application, it will automatically utilize the index created to retrieve and perform the statement faster (Figure. 17).

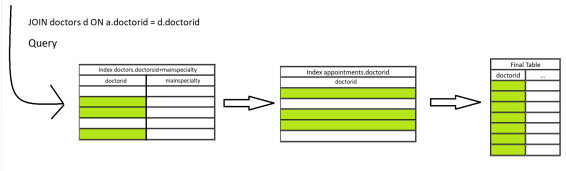

Figure 17. Visualization of Index with Inner Join

The two indexes created will be applied in both JOIN and GROUP BY statements found in Listing 3. It was more efficient to create an index that has both doctorid and mainspecialty, as two of these columns are being used in GROUP BY, and the query can utilize these columns extracted for JOIN statements as well. Creating three different indexes each having their own column was relatively inefficient and slower.

6. Result and Analysis

To evaluate the OLAP application, two types of testing were performed - functional and performance testing. The functional testing verifies that each component of the application is working as designed and is able to meet the standard for functionality. Additionally, functionality testing aids in detecting potential defects which can be addressed early in the development process, leading to a more reliable and high-quality software product.

Performance testing is important to ensure the responsiveness and speed of the overall application. This type of testing evaluates how well an application performs in terms of speed, scalability, and stability, especially when subjected to varying levels of load or stress. Performance testing was conducted to significantly reduce the amount of time it takes to execute a query, and overall increase the user experience of the application.

6.1 Functional Testing

The functional testing was conducted throughout the different phases of developing the OLAP application. When creating and designing an ETL pipeline through Apache Nifi, the group had to ensure that the pipeline is properly directing the flow files into the data warehouse schema.

To apply functional testing for ETL pipeline, the group has utilized a *Run Once* feature found in Apache Nifi. Run once feature allows its user to instruct each processor to perform just a single instance of processing allowing processor such as UpdateAttribute to retrieve just a single flow file from its upstream processor. We have utilized this feature to examine the flow of the ETL pipeline with a small amount of data, allowing the group to easily debug any flaws when faced with a problem.

Upon verifying the correctness of the ETL pipeline design and its functionality, the group proceeded to test the structure of the pipeline with bigger data. When conducting a test with a bigger data, few errors which weren't present during the test cases with a small data were present such as the memory problem - this was further elaborated in section 3.2 of the paper.

The group was able to locate and eliminate such errors on the functionality of the ETL pipeline, and have ensured that the design and implementation of ETL pipeline through Apache Nifi works as intended.

The functional testing for the outputs from the queries involving OLAP operations were conducted through cross-validation between two different environments, MySQL and Jupyter notebook. The same queries were run through both as the group verified the consistency of the outputs. If a query produced identical and expected results in both environments, it confirms that the data retrieval and aggregation is accurate.

6.2 Performance Testing

Performance testing is an important aspect when developing an application, ensuring that the proposed application or a system meets its performance requirements under various conditions. Our testing focuses on how well the OLAP queries generate reports under different conditions.

Our group has performed a performance testing to ensure that each OLAP operation runs under 5 seconds. Performance testing of a query was conducted using the indexing optimization method we have applied to the queries, and the results are as follows.

6.2.1 Performance of Queries using Indices

The query that will undergo Performance Testing in this paper is as follows:

```
SELECT
   d.doctorid,
   d.mainspecialty,
   COUNT(a.apptid) AS AppointmentCount
FROM
   appointments a
JOIN
   doctors d ON a.doctorid = d.doctorid
GROUP BY
   d.doctorid,
   d.mainspecialty
```

ORDER BY
AppointmentCount DESC;

The query above is a part of the OLAP application which we have mentioned above, used to generate reports on top doctors based on number of appointments in relation to their main specialty.

There are a considerable amount of heavy statements in the query, some examples being JOIN and GROUP BY. These statements generally cause a long execution time, and must be optimized to increase the performance of an application.

To optimize the performance of the query, two indices were created with the following queries:

CREATE INDEX idx_appointments_doctorid
ON appointments(doctorid);

CREATE INDEX idx_doctors_doctorid_mainspecialty
ON doctors(doctorid, mainspecialty);

The first index *idx_appointments_doctorid* is an index with just doctorid from the appointments table. While the second index *idx_doctors_doctorid_mainspecialty* involves two columns: doctorid and mainspecialty from the doctors table.

Initially, the group had created three separate indices, segregating the three columns used for JOIN and GROUP BY statements, namely; appointments.doctorid, doctors.doctorid, and doctors.mainspecialty. This segregation of indices did not affect the query execution time significantly, due to inefficient usage of indices as the query had to go through multiple indices to produce a result. The group later realized that the GROUP BY statement uses both doctors.doctorid and doctors.mainspecialty, and this index can be utilized in JOIN statements without having to create a new index for doctors.doctorid. Although the index included a doctors.mainspecialty column which is not used for the JOIN statements, the slight inefficiency was overlooked by the significant reduction of the query execution time.

Table 1. shows the performance results of the queries after optimizing the code with indices, with over 10 seconds reduced the optimization shows significant effect in reducing the query execution time in generating reports.

Table 1. Performance Results of Queries

	Trial #1	Trial #2	Trial #3	Avg
Before Optimization	11.7709 seconds	11.9590 seconds	11.9237 seconds	11.8845 seconds
Initial Optimization	8.5437 seconds	8.9351 seconds	8.9680 seconds	8.8156 seconds
Final Optimization	1.5339 seconds	1.5202 seconds	1.5242 seconds	1.5261 seconds

7. Conclusion

The completion of the project required the group to maximize their knowledge of database management and query operation. Query optimization and knowledge about ETL process tools were essential skills needed to complete the project.

A huge chunk of the work was cleaning up the data that came with the project. The data needed a lot of cleaning up as it contained, irrelevancies, duplicates, and was highly disorganized. Multiple steps were done to try and clean the data, and the group ultimately decided to use python and fuzzywuzzy to clean and extract data from the csv files. This process was difficult as the file was huge and the group had to resort to sometimes manually cleaning the file. Fuzzy wuzzy was used for its extractOne function to detect spelling errors, which in turn, replaced the data if it had a 90% or greater similarity.

After the data had been thoroughly cleaned, the next step for the group was to use Apache NiFi to load the data into our data warehouse in MySQL, the group then proceeded to run into problems such as Java incompatibility and the speed at which the data was being processed and transferred. To address this, the group proceeded to redesign the ETL pipeline in order to remove the bottleneck and further optimize the process. This is when the group realized that our initial thought processes on how to go about the project were inherently wrong, there were a lot of bumps that needed to be addressed and the overall structure of how we set up our ETL tools were primitive when compared to the process we used when queries were further optimized. The project also made us realize that a working process can always be improved further, and that efficiency in multiple avenues of database management is a necessary skill.

The bulk of the project heavily relied on the group being able to optimize everything, this ranged from data and query to ETL pipelines and database storehouses, the project was a tedious task that required the group to continuously improve and question their line of thinking. Initially, the group had experienced

hardships in getting the data clean, this setback made us realize that the project was not straightforward, it required us to pursue a path that was not explicitly stated and rely on our creativity and effort to ascertain that the work we were making was getting better overall. In the end, we were able to push through and realize the importance of using everything taught; indexing, optimization, database management, and everything else in between.

8. References

[1] Apache Nifi. Documentation for 2.0.0-M1 - Apache Nifi. *Official Documentation*. November 22, 2023. [Online]. Available: https://nifi.apache.org/documentation/v2/

[2] B. Bende. "Apache Nifi - OutofMemory Error", *StackOverflow*, July 29, 2019. [Online]. Available: https://stackoverflow.com/questions/38653745/apache-nifi-outofmemory-error-gc-overhead-limit-exceeded-on-splittext-process

[3] M. Hajibaba, "NiFi unable to write flowfile content to content repository error", *StackOverflow.* June 27, 2022. [Online]. Available: https://stackoverflow.com/questions/72774274/nifi-unable-to-write-flowfile-content-to-content-repository

[4] M. Guru. "How to re-route flows into 4 different processors", *Cloudera.* March 06, 2018. [Online]. Available: https://community.cloudera.com/t5/Support-Questions/How-to-re-route-flow-files-to-4-Different-processors/td-p/233627

[5] IBM. "What is a data warehouse?", *International Business Machines.* n.d. [Online]. Available: https://www.ibm.com/topics/data-warehouse

[6] A. Roy. "What is SQL Optimization? How to do it?", *Linkedin.* May 23, 2023. [Online]. Available: https://www.linkedin.com/pulse/what-sql-optimization-how-do-amit-roy/

9. Contribution

Alvarez - Abstract, Introduction, Jupyter Notebook Data cleaning, Data Visualization

Orendain - Conclusion

Jumilla - Functional Testing, Data Visualization, OLAP Application

Park -Query processing and Optimization, Data Cleaning, ETL Script, Performance testing, Jupyter Notebook Data Cleaning