MCO1: OLAP APPLICATION

De Leon, Francis Zaccharie
francis_zaccharie_deleon@dlsu.edu.ph
De La Salle University
Manila, Philippines

Dela Cruz, Frances Julianne
frances_julianne_delacruz@dlsu.edu.ph
De La Salle University
Manila, Philippines

Gamboa, Mikkel Dominic
mikkel_gamboa@dlsu.edu.ph
De La Salle University
Manila, Philippines

ABSTRACT

This paper focuses on query processing in a data warehouse which involves exploring the steps in building a data warehouse, setting up an ETL script, developing an OLAP application, and the application of query processing and optimization strategies. Data wrangling, OLAP transformations, and the evaluation of the impact of query optimization were performed on the SeriousMD dataset.

KEYWORDS

OLAP, MySQL, Data Visualization, Query Optimization

ACM Reference Format:
De Leon, Francis Zaccharie, Dela Cruz, Frances Julianne, and Gamboa, Mikkel Dominic. 2024. MCO1: OLAP APPLICATION. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The SeriousMD dataset is a collection of information about doctors, clinics, patients, and appointments. As far as the group's knowledge, this data set is dummy data and only mimics real-world data. The given dataset is very unclean and noisy; some rows are skewed and some tables can't be initially imported. The data set has the tables: (1) *doctors* which have 60,000 rows and contain the doctors' age, specialty, and ID (2) *clinics* which have the clinics' pertinent location, clinic ID, and information on whether it is in a hospital or not (3) *patients* which has 6 million rows and contains the patients' age and gender (4) *appointments* which have about 10 million rows and contains pertinent information about an appointment such as the doctor ID, patient ID, clinic ID, time, status, and what type of appointment. Each ID present is a 32-length hex character. For the database, they are represented as *VARCHAR(32).*

The ETL tool used by the group is Apache's Nifi Flow and Python with Pandas and SQLAlchemy framework. Since each tool has its strengths and weaknesses, the group has frequently switched to using Nifi and Python for the data cleaning section.

The data warehouse (DW) follows a snowflake schema with the appointments being the fact table, The other tables from the database were turned into a branching dimension table. The appointment table references the ID of the dimension tables.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

The OLAP application was developed using Tableau, a visual analytical tool that can connect to various data sources like MySQL. With its assistance, the application can generate reports and charts relating to the performance of the clinics in a location, duration, and appointment status. Moreover, the diversity in the specialties between two given hospitals is generated as well. [2]

2 DATA WAREHOUSE

The DW used MySQL as the database software. A snowflake model was used with a fact table and dimension tables branching out. Shown in Figure 1 is a zoomed-out picture of the model.

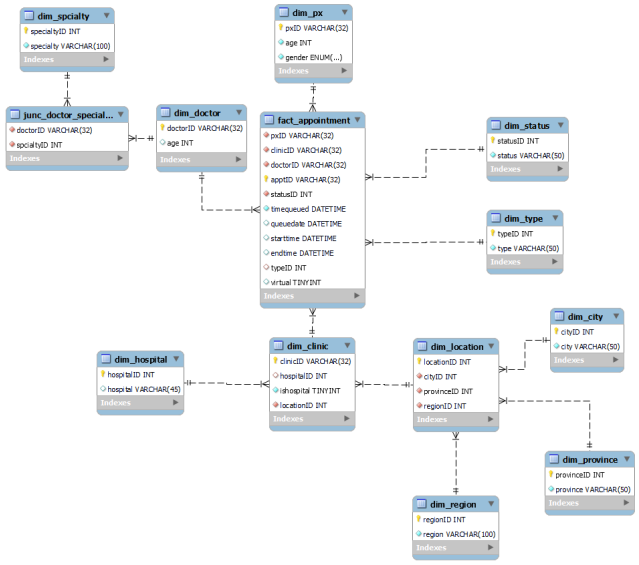


Figure 1: Data Warehouse Schema

The appointments table was used as a fact table. By definition, a fact table stores quantitative and foreign key information. The appointments table stores the foreign keys from all the other tables. With the appointments table as the fact table, the group can join all the other tables with the appointment tables and make an analysis.

Shown in Figure **??** are the fields used in the appointment table

The IDs represent the foreign keys used from the dimension tables. Status tells what that appointment status is ('Complete', 'NoShow', 'Cancel', 'Serving', 'Queued', 'Skip') while the time queued is the time the patient has queued up for the appointment. The queuedate specifies the date that the patient queued up. The starttime tells when the start of the appointment is while the endtime tells when it ends. The type indicated what type of appointment it is ('Consultation', 'Inpatient'). While the virtual says if it will be held online or not. All the IDs, status, type, and timequeued are

NOT NULL as these are vital information for the fact table. The IDs except for the apptID are set to NOT UNIQUE as these will repeat (e.g. the same patient sets up an appointment again with the same doctor).

The clinics, px, and doctors were the ones used as dimension tables as they represent a dimension from the appointments table. These dimension table acts as a support for the appointment table. They supply all the necessary foreign keys and information needed by the fact table.

The px dimension table is straight forward since it is a one-to-one copy of the px table from the database.

The pxID serves as the primary key of the dimension table while the age and gender of the patient are listed. The pxID serves as the foreign key for the fact table.

The clinics dimension table needed to branch out on location and the list of hospital. Shown in Figure **??** is the cluster of dimension table used to represent the clinics

The dim_clinic is the highest level for the dimension tables. It has clinicID, hospitalID, ishospital, and locationID. The dim_hospital acts as a dimension table for the list of all hospitals. This table serves as the foreign key for all the available hospital names. The dim_location serves as the dimension of location. It has sub-dimensions: region, province, and city.

The doctors dimension table has an interesting case. From the original database, the doctors' ID, age, and specialty can be found as the fields. Some doctors may have listed no specialty in the specialty field while some have listed more than 1. So the number of specialties per doctor is 0 to many. To represent this in the DW, a junction table for the doctors and the specialties was created. There is a separate table for doctors and a table for all the possible specialties. These two tables are connected through a junction table.

The junction table references a doctor ID and a specialty ID, linking and forming a relationship with a doctor and a particular specialty. When a doctorID is listed in the junction table multiple times, then that doctor has multiple specialties. When the doctorID of a particular doctor is not listed in the junction table, then that doctor does not have any specialties.

Since the creation of the DW is similar to what the group has already done in their CCINFOM, not much problem was encountered except for one. The group had a difficult time thinking on how they could represent the 0 to many specialties of the doctors. They initially thought of creating multiple fields of specialties with the maximum number of specialties that a single doctor could have as the number of fields to be added. Let us say that in the doctors table, it was found that the maximum amount of specialty a doctor has is 5. This would mean that the group needed 5 fields of specialties for the doctors table. But, that would be space inefficient since there would be many values in the table that would be left unused. This could make a bad table design. Through googling, the group found the solution [3] of using a junction table.

3 ETL SCRIPT

Before transferring the data from the database to the DW, the CSV files must first be loaded into the database. Initial loading from CSV to database was not straightforward as the CSV files were very unclean that the loading software was not able to successfully interpret and load all the data. This implies that a little data cleaning is required for the data to be loaded in the database. For this, the Nifi software was used.

The doctors CSV was the first table to be imported. A GetFile processor was used to fetch the CSV file, which is then fed into a SplitText which splits all the rows into single rows and multiple flow files and then into a ConvertRecord where CSV flow files were converted into JSON. Which is then fed into a JSON to SQL converter. This SQL will be executed in the database server. The process was failing at Convert Record. With inspection, the group has found out that there were some '\n' characters in the specialty field. A person may have inputted Ob '\n' Gyn where the '\n' indicates another specialty. This meant that the csv has unexpected LF. The SplitText interprets this as another line instead of one continuous line. The ConverRecord will not know what to do with this broken data so is throws it into failure.

To fix this problem, a ReplaceText processor was used. All unexpected '\n' will be deleted. Since there is still '\n' at the end of each row, the unexpected '\n' should be pinpointed. To do this, the regex (?<!")'\n'(?!") and (?<="[a-zA-Z0-9_]*)'\n'(?!") was used. For a brief explanation, this regex indicates to find a '\n' character that is not in between a double quote. To give context, the CSV file puts a double quote for each value. (e.g. "doctorID", "20", "internal medicine"'\n'"doctorID2", "21", "General Medicine") With that example, the only valid "\n' is the one between a double quotes. From the example of "Ob '\n' Gyn", the second regex was used. This regex finds a "\n'' between a double quote and 0 to make text. If the character proceeding that "\n'' is not a double quote, then that "\n'' is returned by the regex. With this two regex, the unexpected "\n'' was removed in the CSV file. The pipeline for this is as follows GetFile -> ReplaceText -> SplitText -> ConvertRecord -> ConvertJSONToSQL -> PutSQL

The clinics were straightforward to import as it was imported to the database successfully with no problems.

The Nifi was fast in processing these CSV files. But it gets really slow as the volume of data increases. When the number of rows reaches hundreds of thousands to millions, it could take a whole day. To process things a bit faster, concurrency was used. This functionality allows processors to process multiple flow files simultaneously making it faster. Unfortunately, because it was set to process too much flow files, Nifi often crashes. The number of concurrency processors must be balanced.

While loading the patients with the same pipeline, it was sending some flow files into failure after PutSQL. The flow files in the failure queue had the values INSERT INTO px VALUES (?, ?, ?). This was difficult to debug as I did not know which values caused the error. The group looked around online for a better debugging method and found the LogAttributes. It was their first time looking into logs and making sense of it, With the logs, the group was able to pinpoint the exact problem in the CSV file. It contained 2 header rows. The other header row was hiding in the middle of the CSV file. The PutSQL was expecting an integer data type for age but got "age" instead. They manually did a search function in Notepad++ found the duplicate row and deleted it. The CSV file was fed into the pipeline and it was successfully imported into the database.

The appointments CSV was large. It consumes 2GB of memory and has about 9 million rows. It would take about a whole day to

import the clean data. Due to the time constraint of the project, the group does not have a whole day for this. They decided to first trim down the appointments CSV and then load it into the database. To trim it down, the appointments was loaded into Python pandas dataframe. The cleaned px, doctors, and clinics was exported as CSV files through Nifi's PutFile processor. These CSVs were also loaded into Python pandas. The use of pandas will be explained later in detail. It was found that not all pxID in the appointments table is referencing a foreign key from the px table. This means that this is an invalid record. To remove every invalid references, a join operation was done where the px and the appointments table were inner joined by the pxID

Listing 1: Join Operation on px and appointments

```
main_df = pd.merge(appointments_df,
    patients_df['pxid'], on='pxid', how='
    inner')
```

This reduced it to 522,345 rows.

The same procedure was done with clinics and doctors however, every reference is valid. No rows were dropped.

Since importing the appointments CSV into pandas did not result to any error, this implies that no data was skewed. There are no unexpected '\n' characters present.

The group has already limited time importing the reduced appointments and Nifi is slow when it comes to importing. Because of this, the group has used SQLs LOAD FILE SQL script. This loaded all the appointments table in the database in about 3 minutes.

Below is the SQL LOAD Script used

Listing 2: SQL Script to Load File

```
SET GLOBAL local_infile=1;
LOAD DATA LOCAL INFILE 'path_to_csvfile.csv'
INTO TABLE appointments
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\r\n'
IGNORE 1 ROWS;
```

The number of rows is checked in the SQL server and it reflected the 522,345 rows in the CSV file.

To import the data from the database to the data warehouse, the tables, appointments, doctors, and px were processed through Python because some of the group members were more comfortable with using Python as their ETL tool. With the short amount of time given, having to learn a new tool and transform a large volume of data is not prudent. For the rest of the tables, dim_clinics, dim_city, dim_province, dim_region, dim_hospital, dim_location, they were processed through Apache NIFI. Apache NIFI was used due to familiarity of the technology and its feature to use processors that are relevant to the task at hand.

For Python, the pandas framework was used to manipulate tables. To get and send data to MySQL, SQLAlchemy framework was used as this was compatible with the pandas framework. With python, the ETL process is as follows: Start a connection to the DB and DW with SQLAlchemy, import the table from the db to a pandas dataframe, do ETL on the dataframe, export the dataframe to the DW

To use the frameworks, the following imports are executed:

Listing 3: Python Imports

```
import pandas as pd
from sqlalchemy import create_engine
from sqlalchemy import URL
from enum import Enum
import sqlalchemy
```

All ETL process has the same connection, import, and export process.

To start a connection the following Python code is used:

Listing 4: Start Python Connection

```
url_object = URL.create(
   drivername='mysql',
   username='user',
   password='password',
   host='localhost',
   port=3306,
   database='seriousmd'
)
MySqlEngine = create_engine(url_object)
```

Where the username and password are the MySQL server's login credentials. Since the database is in the local machine, the host is set to localhost.

As per SQLAlchemy, this is only an engine, not a connection. To create a connection, an initial engine must be running. For a specific database, a separate engine must be running. This implies that a separate engine for the database and DW is needed.

To get data of a whole table from the database, the following Python code is used:

Listing 5: Get All Data from Table

```
seriousmd_conn = MySqlEngine.connect()
appointments_df = pd.read_sql(
   sql="""SELECT * FROM <table>""",
   con=seriousmd_conn,
   }
)
seriousmd_conn.close()
```

The SQL parameter is the input SQL sent to the database server. Since all data will be needed in the table the asterisk was used. The <table> indicates the desired table that would be imported.

To send a pandas Dataframe to the DW, the following Python code is used:

Listing 6: Send a pandas Dataframe to the DW

```
mdwarehouse_conn = MySqlEngineDW.connect()
rows_affected = appointments_df.to_sql(
   name='<table name>',
   con=mdwarehouse_conn,
   if_exists='append',
   index=False,
   dtype={
       <field name>: <data type>
```

```
    },
    chunksize=5000,
    method='multi'
)
print('Rows affected:' + str(rows_affected))
mdwarehouse_conn.close()
```

Where the <table name> is the target table to export data to. The <field name> is a field within the <table name> and the <data type> is the specified data type for that field. In this way, the data types being exported to the DW are being constrained.

The chunksize and method affects how fast the transfer is. The chunksize indicates how many rows will it insert into the target database at a time. While the method tells SQLAlchemy to INSERT VALUES into a one line statement. The group made a mistake in setting the default value for this parameter making the function send all the rows at once. With small amounts of data, this would be fine. But with millions of rows, the Python kernel could crash due to memory issues.

The protocols that was followed to do ETL is as follow: 1. Check for incorrect data types 2. Check for duplicate values 3. Check for multiple representations 4. Check for missing and default values 5. Check for inconsistent format

To check for incorrect data types, the method Dataframe.infos() was used which displayed all the fields and their data types.

3.1 Appointments

Duplicate rows were checked using the code

```
appointments_df[appointments_df.duplicated()
    ]
```

This returned rows that are similar to each other in all fields. To drop these rows, the index of the rows were recorded and then dropped

```
duplicated_index = appointments_df[
    appointments_df.duplicated()].index
appointments_df = appointments_df.drop(index
    =duplicated_index)
```

This reduced the number of rows to 320,139.

For checking multiple representations, the Series.unique() method was used to check all the unique values in a given field. The status, type, and virtual fields were checked.

```
appointments_df['status'].unique()
appointments_df['type'].unique()
appointments_df['Virtual'].unique()
```

The status and type returned no anomalies, however virtual returned True, False, and ''. A blank is not ideal. It could be better represented by a null value. To replace the null value, the Series.replace method was used.

```
appointments_df['Virtual'].replace(
    to_replace=r'^\s*$', value=np.nan, regex
    =True, inplace=True)
```

The regex,

```
^\s*
```

was used to search through the blank values and was replaced with NaN. For checking missing values, the Dataframe.infos() were used. No missing values except for queuedate, starttime, and endtime. These rows were not dropped as the group thinks these values optional and by design.

For checking inconsistent formats for all the date fields, the method Series.str.match was used.

```
list_of_valid_format_starttime =
    appointments_df['StartTime'].astype(str)
    .str.match(r'^\d{4}-\d{2}-\d{2} \d{2}:\d
    {2}:\d{2}$')
```

The regex value,

```
list_of_valid_format_starttime = ^\d{4}-\d
    {2}-\d{2} \d{2}:\d{2}:\d{2}$
```

matches each line of the format yyyy-MM-dd hh:mm:ss format. The method returns a series of boolean where each index represents True or False whether the particular index follows the regex value. When the negation ( ) of this series is passed into a dataframe parameter, it shows which values are not following this format.

```
appointments_df[~
    list_of_valid_format_starttime]['
    StartTime'].unique()
```

The returned value is NaT. Which is a NaN equivalent for dates. This implies that the only value that does not follow this format is the NaN value. So no value in the field is formatted incorrectly.

This is done in all date fields and all were formatted correctly.

Lastly. Before sending the Dataframe to the DW, the field names are renamed to match the field name in the DW.

```
appointments_df.rename(
   columns={
       'pxid': 'pxID',
       'clinicid': 'clinicID',
       'doctorid': 'doctorID',
       'apptid': 'apptID',
       'TimeQueued': 'timequeued',
       'QueueDate': 'queuedate',
       'StartTime': 'starttime',
       'EndTime': 'endtime',
       'Virtual': 'virtual'
   },
   copy=False,
   inplace=True,
   errors='raise'
)
```

While sending the data, a foreign key error was raised. This is because the clinics, px, and doctors were not yet populated. The appointments has no reference for the doctorID, pxID, and clinicID.

Once these 3 dimension tables were imported, the appointment table was successfully exported to the DW.

3.2 Patients

Duplicates were checked using the code

```
patients_df[patients_df.duplicated()]
```

With a similar code from appointments, these duplicates were dropped. Checking for duplicate pxIDs is also needed since there could be duplicate pxIDs left

```
patients_df[patients_df['pxid'].duplicated()
    ]
```

Which returned one row,

```
FBA46EA3EF7CCD4F3551C22272FE865F,4,MALE
```

After checking the ID using,

```
patients_df[patients_df['pxid'] == '
    FBA46EA3EF7CCD4F3551C22272FE865F']
```

It returned 2 rows

```
FBA46EA3EF7CCD4F3551C22272FE865F,42,MALE
FBA46EA3EF7CCD4F3551C22272FE865F,4,MALE
```

Since older people tend to go to the hospital, the age of 4 was considered a typo, thus the second duplicate row was dropped.

For checking multiple representations, the gender field was investigated using Series.unique() which only returned MALE and FEMALE

For checking missing and default values, the Series.isna() is used

```
patients_df[patients_df['gender'].isna()]
```

This was all done in all three fields and all returned 0 rows.

For checking incorrect formats, the pxid is the only field susceptible for that. Each value in the field should be 32 characters long and should only have the values 0-9 and A-F as this represents a hex value. The Series.str.match() method was used again with the regex value

```
^[a-fA-F0-9]{32}$
```

This returned 0 rows which implies all rows comply with the id format.

While investigating the values in the age field, the group has found that there are age below 0. These rows were dropped using the code

```
invalid_age_patients = patients_df[
    patients_df['age'] < 0].index
patients_df.drop(index=invalid_age_patients,
     inplace=True)
patients_df.reset_index(drop=True, inplace=
    True)
```

Some patients were also above 100. Since the group did not know if these were legitimate or not, it was best not to drop these data. Afterall, some people have surprisingly lived a long time.

Before sending the Dataframe to the DW, the field names are renamed to match the field name in the DW.

```
patients_df.rename(
    columns={
        'pxid': 'pxID',
    },
    copy=False,
    inplace=True,
    errors='raise'
)
```

3.3 Doctors

Duplicate values were checked and returned 0 rows. And all columns have consistent formats.

Some doctors have an age that is NaN. However, these were not dropped. They were left as is since some doctors may have purposefully left it blank. Interpolation methods such as finding the average of the whole age and filling all the NaN with the average will heavily skew the data. So, these were not dropped. Some ages are also very high such as 1048. These high values were dropped using a similar code presented in the patients data cleaning.

Multiple representation is an interesting case. In the mainspecialty field, some doctors have inputted inconsistent values. (e.g. some doctors have inputted internal medicine while some INTERNAL MEDICINE while others inputted IM).

The DW has 3 tables: specialties, junction table, and doctors table. The first course of action is to get the list of all specialties available in the data. Using the Series.unique() the group has listed out and interpreted all the specialties which are: Cardiology, Dentistry, Dermatology, Endocrinology, Family Medicine, Gastrology, General Medicine ,Gynecology, Internal Medicine, Obstetrics, Ophthalmology, Orthopedy, Pediatrics, Psychiatry, Pulmonology, Radiology, Surgery, Urology. Now that there is a list of all the specialties, the group will now associate a doctor with a specialty through the junction table.

To do this, a copy of the doctors dataframe is set for manipulation

Listing 7: A copy of the doctor's dataframe is instantiated

```
doctors_df_cpy = doctors_df.copy()
```

In that copy, different regex values are tested and see which mainspecialties will be displayed

Listing 8: Internal Medicine is being searched through the dataframe

```
regex_val = r'^Internal␣Medicine$'
doctors_df_cpy[doctors_df_cpy['mainspecialty
    '].str.contains(pat=regex_val, case=
    False, na=False)]['mainspecialty'].
    value_counts()
```

In this example, the regex Internal Medicine was tested with false case sensitivity, which returned the values Internal Medicine, Internal medicine, internal medicine, INTERNAL MEDICINE, internal Medicine, Internal MEdicine, Internal MEDICINE.

The doctors who listed their mainspecialties as any of the given results will be associated with Internal Medicine through the junction table using this code

Listing 9: Doctors that matched the search is being appended in the junction table

```
temp_df = doctors_df_cpy[doctors_df_cpy['
    mainspecialty'].str.contains(pat=
    regex_val, case=False, na=False)]['
    doctorid'].copy()
temp_df = pd.DataFrame(temp_df)
temp_df.insert(loc=1, column='mainspecialty'
    , value='Internal Medicine')


junc_doctor_specialty = pd.concat([
    junc_doctor_specialty, temp_df],
    ignore_index=True)

drop_rows = doctors_df_cpy[doctors_df_cpy['
    mainspecialty'].str.contains(pat=
    regex_val, case=False, na=False)].index
doctors_df_cpy = doctors_df_cpy.drop(index=
    drop_rows)
```

The temp_df serves as the df to catch all the rows that have the doctors with those main specialties. This temp_df will be concatenated to the junc_doctor_specialty dataframe. The doctors that were already listed will be dropped in the copy of the doctors dataframe.

The dataframe will then be searched again using another keyword. This time, 'IM' was used as a regex value.

Listing 10: The regex ÎM$ was used to search through the dataframe

```
regex_val = r'^IM$'
doctors_df_cpy[doctors_df_cpy['mainspecialty
    '].str.contains(pat=regex_val, case=
    False, na=False)]['mainspecialty'].
    value_counts()
```

Which returned the values IM and im.

The doctors who listed any of these will be associated with Internal Medicine in the junction table using the same code as above. These values will be dropped in the copy dataframe once appended to the junction table.

After Internal Medicine is General Medicine. A fresh new doctors dataframe copy is needed since the copy had some dropped values. A fresh start is needed

Listing 11: A copy of the doctor's dataframe is instantiated

```
doctors_df_cpy = doctors_df.copy()
```

The same procedures were followed for appending the data to the junction table.

With this procedure, when a doctor listed out multiple specialties such as 'Surgery / IM', that doctor would have the Surgery and Internal Medicine associated with them.

This was repeated in all the specialties and was sent out to the DW.

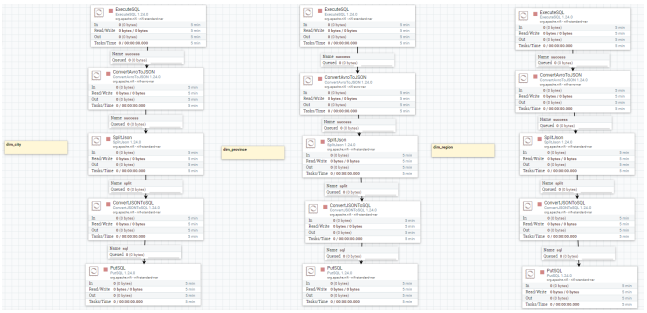


Figure 2: ETL Pipeline to populate `dim_region`, `dim_province`, and `dim_city`

3.4 Clinics

In order to populate the dim_clinics table, the following tables were populated first in order to have a reference for the foreign keys present in the table: the dim_city, dim_province, dim_region, and dim_hospital. In Figure 2, the pipeline follows a general process of extracting the relevant fields from the source dataset, converting to JSON File, splitting the JSON File, converting from JSON to SQL and then putting the SQL.

Listing 12: SQL Code To Extract From dim_city

```
SELECT  City as city,
        @row_number := @row_number + 1 AS
            cityID
FROM (  SELECT DISTINCT City
        FROM clinics
        WHERE City != ""
     )  AS subquery;
```

In Listing 12, the SQL Script below was inputted into the ExecuteSQL Processor to extract the data. In which the row number was generated as the identifiers of the tables for the data warehouse schema. As the dim_hospital, dim_city table, dim_province table and dim_region table only have the identifier and their corresponding location value, the AVRO file was converted to JSON, split, and converted into an SQL statement to be inserted into the data warehouse.

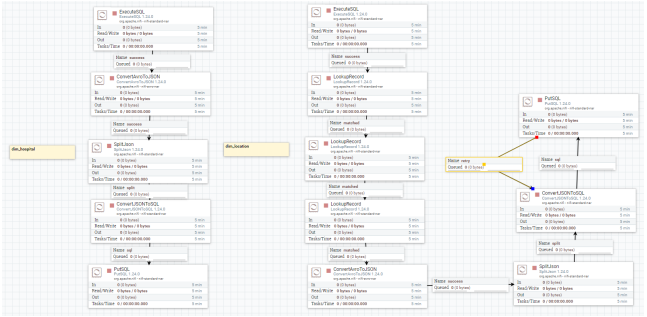


Figure 3: ETL Pipeline to populate `dim_location`

The dim_location was then populated by the same process but with a LookUpRecord Processor. In 3, one processor is needed for

each reference to a foreign key. The processor works by using the data extracted from the ExecuteSQL and looks it up in an existing table, which in this case are the dim_region, dim_province, and dim_city tables.

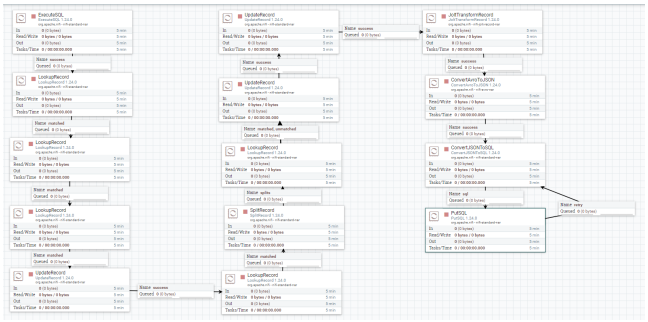


Figure 4: ETL Pipeline to populate `dim_clinic`

With the references now in place, the dim_clinic can now be populated. The clinicID, hospitalname, ishospital, city, province, and region was extracted into an Avro using the ExecuteSql processor from the source dataset. In order to reference the locationID, LookUpRecord processors were used again to reference the dim_city, dim_province and dim_region. Afterwards, a dummy column was created by using the UpdateRecord to concatenate the identifiers referenced for it to be used as a reference to get the locationID. A dummy table was also created which contains the locationID and the concatenated location string from the identifiers. The LookUpRecord processor was then used to reference the concatenated strings and extract the locationID. The record was then split in order to reference the hospitalID from the dim_hospital as some values did not have a hospitalID. Once the hospitalID was extracted, the UpdateRecord processor was used to convert the String value of a boolean into a Binary value. The following functions were used:

Listing 13: Replace "False" to "0"

```
replaceRegex( /ishospital, 'False', 0 )
```

Listing 14: Replace "True" to "1"

```
replaceRegex( /ishospital, 'True', 1 )
```

Since it was read as a String, the JoltTransformRecord was used to convert it into an integer using the following specifications:

Listing 15: JoltTransform Specification

```
[
  {
    "operation": "modify-overwrite-beta",
    "spec": {
      "ishospital": "=toInteger"
    }
  },
  {
    "operation": "default",
    "spec": {}
  }
]
```

During the loading of the data into the data warehouse, the main constraints that were considered are the data types, the relationships between the tables and the newly generated identifiers for the tables. These were all addressed through various methods such as processors that transformed the data, splitting the tables and using lookups and for python, enforcing the data types when exporting the data.

3.5 Updating the DW

In python, for the DW to be updated automatically, a scheduler is set to run all the ETL processes. Jupyter Notebook has a native scheduled run. First, Jupyter Notebook should be running. Then, go to the directory where the ETL scripts are located. Then, create a notebook job. Click on the calendar. Then click Run on a schedule and pick the desired interval. The group chose once a week expressed in CRON. This will then run the ETL scripts once a week. While in Nifi, the ExecuteSQL processors were set to run at a periodic rate as well using the properties option.

4 OLAP APPLICATION

The application serves as a dashboard to monitor the performance of the clinics and hospitals. It takes into account the number of appointment bookings they have, the general performances of clinics in a specific location, the number of appointments that are booked throughout a duration and the diversity of the doctors' specialties available between hospitals.

The application allows analysis on appointment booking to support strategic planning on optimizing scheduling and resource allocations. It also allows location-based and date-based performance analysis in order to aid in identifying trends and patterns. Moreover, it allows analysis on the common specialties to assist in strategies involving collaborations and sharing of resources.

The application generates the following reports: (1) Total Number of Appointments per clinic; (2) Total Number of Appointments per clinic, per status; (3) Total Number of Appointments per clinics, in a specific region; (4) Total Number of Appointments per clinics, in a specific province; (5) Total Number of Appointments per clinics, in a specific city; (6) The most common doctor specialties between two hospitals; (7) Top number of Appointments per year; (8) Top number of Appointments per year, per quarter; (9) Top number of Appointments per year, per quarter, per month.

4.1 Total Number of Appointments per clinic, per status

This report generates the total number of appointments of a clinic per status. In the application, stacked vertical bar charts were used. This is so that the DRILLDOWN/ROLLUP could be properly represented. The summation of the appointments per status would be the total height of the individual bar while the different sections/colors of the bar would correspond to the number of the appointments per status.

In Listing 16, the ROLLUP statement was used to get the total number of appointments in a clinic regardless of the status. The

GROUP BY statement was used in order to aggregate the clinic and allow the use of the COUNT function to get the total appointments.

Listing 16: Total Number of Appointments per clinic, per status

```
SELECT  c.clinicID,
        f.status,
        COUNT(c.clinicID) as numAppointments
FROM    fact_appointment as f
        JOIN dim_clinic as c
        ON f.clinicID = c.clinicID
GROUP BY c.clinicID, f.status WITH ROLLUP;
```

By using those statements, it was now possible to drilldown and get the number of appointments booked per status. And due to the ROLLUP, it was also possible to get the summation of the total appointments per status, as seen in Figure 5

clinicID	status	numAppointme...
00A61BA99599C66D87F80D1DE9EBD899	Cancel	7
00A61BA99599C66D87F80D1DE9EBD899	Complete	36
00A61BA99599C66D87F80D1DE9EBD899	NoShow	1
00A61BA99599C66D87F80D1DE9EBD899	Queued	9
00A61BA99599C66D87F80D1DE9EBD899	NULL	53
049A632EFBA268D49B6A861759A4268A	Cancel	251
049A632EFBA268D49B6A861759A4268A	Complete	3927
049A632EFBA268D49B6A861759A4268A	NoShow	118
049A632EFBA268D49B6A861759A4268A	Queued	131
049A632EFBA268D49B6A861759A4268A	Serving	44
049A632EFBA268D49B6A861759A4268A	NULL	4471
060C36E1C116CC67408A8F5A5935D5C1	Complete	145
060C36E1C116CC67408A8F5A5935D5C1	Queued	16
060C36E1C116CC67408A8F5A5935D5C1	Serving	4
060C36E1C116CC67408A8F5A5935D5C1	NULL	165
093562AEE4B200F3C39FDD15462F2F9B	NoShow	1
093562AEE4B200F3C39FDD15462F2F9B	Queued	209
093562AEE4B200F3C39FDD15462F2F9B	Serving	1
093562AEE4B200F3C39FDD15462F2F9B	NULL	211
0B17C3DF4005268C403589C52E300809	Complete	6
0B17C3DF4005268C403589C52E300809	Queued	2
0B17C3DF4005268C403589C52E300809	NULL	8

Figure 5: Sample Output for Query #2

4.2 Total Number of Appointments per clinic in X location

This report generates the total number of appointments as well but it varies in the location. Possible values for the location include: Region, Province and City. Horizontal bar charts were used in order to represent the total number of appointments. This is because only a few variables are involved in the report.

The query performs a SLICE operation on the data. Listing 17 is the general format on which the queries involving locations follow. In which it first uses the GROUP BY function to allow aggregation on the clinicID and the specified location. Afterwards a specific location value is defined, in which the data would only show the data corresponding to that single dimension.

Listing 17: Total Number of Appointments per clinic in X location

```
SELECT  c.clinicID,
        c.region,
        COUNT(c.clinicID) as numAppointments
FROM    fact_appointment as f
        JOIN dim_clinic as c
        ON f.clinicID = c.clinicID
WHERE   c.region = 'CALABARZON (IV-A)'
GROUP BY c.clinicID, c.region;
```

In the OLAP Application, the location value is a parameter in which the user would have to define.

clinicID	region	numAppointme...
0B17C3DF4005268C403589C52E300809	CALABARZON (IV-A)	8
197E342C9FBAD962DBE6D3322F723CA6	CALABARZON (IV-A)	1394
50CF0FE63E0FF857E1C9D01D827267CA	CALABARZON (IV-A)	32
54F9DCAFFEB95F402AA2AC051B02C24B	CALABARZON (IV-A)	64
5B8E9841E87FB8FC590434F5D933C92C	CALABARZON (IV-A)	24
A73305D5BA2857F26BD6EF46E3FBEAE5	CALABARZON (IV-A)	342
C31475EB51378905878A207F72D1CDAE	CALABARZON (IV-A)	7
D7012F5714B8A6A970F5BD34806267C6	CALABARZON (IV-A)	1
E614F646836AAED9F89CE58E837E2310	CALABARZON (IV-A)	1240

Figure 6: Sample Output for Query #3

4.3 Between 2 hospitals, what are the most common doctor specialties

This report generates the common doctor specialties found between two hospitals. It demonstrates the DICE operation as it chooses two dimensions, in this case it is the two hospitals. The report uses a highlighted chart in order to emphasize which specialties have a greater value than the others. In Listing 18, the GROUP BY statement was used to aggregated on the hospitals and specialties in order to calculate the total specialties of the doctors of a hospital, which is made possible by using the COUNT function.

Listing 18: Most common doctor specialties between two hospitals

```
SELECT  h.hospital,
        ds.specialty,
        COUNT(ds.specialty) as
            specialtyCount
FROM    dim_specialty as ds
        JOIN junc_doctor_specialty as jds
        ON ds.specialtyID = jds.specialtyID
        JOIN dim_doctor as d
        ON jds.doctorID = d.doctorID
        JOIN fact_appointment as f
        ON d.doctorID = f.doctorID
        JOIN dim_clinic as c
        ON f.clinicID = c.clinicID
        JOIN dim_hospital as h
        ON c.hospitalID = h.hospitalID
WHERE   h.hospital = "Makati Medical Center"
        OR h.hospital = "The Medical City"
GROUP BY h.hospital, ds.specialty
ORDER BY specialtyCount DESC, ds.specialty;
```

4.4 Top Number of Appointments throughout a Date Duration

This report generates the number of appointments throughout a date value. Possible date values include: per year, per quarter, and per month. Three line charts were used here in which when a specific year is selected in the line chart for the year, it would show the quarters of the year selected. And when a quarter is

hospital	specialty	specialtyCou...	
Makati Medical Center	Internal Medicine	3580	
Makati Medical Center	Dermatology	3131	
Makati Medical Center	Gastrology	2203	
The Medical City	Gynecology	1119	
The Medical City	Obstetrics	1119	
The Medical City	Pediatrics	641	
Makati Medical Center	Orthopedy	46	
Makati Medical Center	Surgery	46	
The Medical City	Pulmonology	12	

Figure 7: Sample Output for Query #6

selected in the line chart for the quarter, the months of that quarter are shown. In Listing 19, Listing 20, and Listing 21, the GROUP BY statement was used on the targeted date value. This allows the COUNT function to perform an aggregate on the number of appointments on the targeted date value. The query displays the Drill-down by using the GROUP BY on the quarter or the month. Along with this, the ROLLUP functions are also used as the drill-down operation is performed in order to see the summation of the sales regardless of the date value selected.

Listing 19: Top Number of Appointments per year

```
SELECT  YEAR(queuedate) as appointmentYear,
        COUNT(apptID) as appointmentCount
FROM    fact_appointment
GROUP BY appointmentYear
ORDER BY appointmentCount DESC;
```

Listing 20: Top Number of Appointments per year, per quarter

```
SELECT  YEAR(queuedate) as appointmentYear,
        QUARTER(queuedate) as
            appointmentQuarter,
        COUNT(apptID) as appointmentCount
FROM    fact_appointment
GROUP BY appointmentYear, appointmentQuarter
    WITH ROLLUP
ORDER BY appointmentCount DESC;
```

Listing 21: Top Number of Appointments per year, per quarter, per month

```
SELECT  YEAR(queuedate) as appointmentYear,
        QUARTER(queuedate) as
            appointmentQuarter,
        MONTH(queuedate) as appointmentMonth
            ,
        COUNT(apptID) as appointmentCount
FROM    fact_appointment
GROUP BY appointmentYear, appointmentQuarter
    , appointmentMonth WITH ROLLUP
ORDER BY appointmentCount DESC;
```

However, due to the limitation of the Tableau application when referencing a Custom SQL query that uses MySQL in-built Date functions. The OLAP Application had to perform the corresponding date functions on its own.

appointmentYear	appointmentCount
2019	53195
2020	48554
2021	41618
2022	41233
2023	39586
2018	32206
2017	27382
2016	8024
2024	4469
2015	4406
2010	3662
2009	3205

Figure 8: Sample Output for Query #7

5 QUERY PROCESSING AND OPTIMIZATION

Query optimization is the process of "configuring, tuning, and measuring performance, at several levels [5]." These levels include the database level and the hardware level. Optimization is necessary when creating these databases because inefficient queries create slow response times which lead to slow applications.

At the database level, optimization can be accomplished through the following: creating secondary indexes, using relational algebra, and restructuring the database through normalization and denormalization. On the other hand, at the hardware level, having newer processors, disk drives, and memory chips tends to provide faster and more efficient results.

In this research, we focused on creating secondary indexes and restructuring the database at the database level, while comparing different generations of Macbooks as the basis for optimization at the hardware level.

5.1 Database Level: Secondary Indexes

The basis for determining which attribute would receive a secondary index was based on the attributes found in the *GROUP BY* statement [5]. As seen in Listing **??**, the two attributes found in the *GROUP BY* statement are *c.clinicID* and *r.region*. Since *c.clinicID* is a *primary key*, it already has its own indexes and no longer needs any more. Conversely, *r.region* is not a *primary key* so it would be appropriate to create a *secondary index* for that attribute. Therefore, a *secondary index* is created for the *dim_region* table labeled *regionIndexes* for the *region* attribute.

5.2 Database Level: Database Redesign

The database design originally followed the snowflake schema. This was done to keep the integrity of the data within the schema. However, in order to optimize the processing speed of the design, the tables in the schema were denormalized into a star schema as seen in Figure 9. This allowed fewer *JOIN* statements to be used thereby optimizing the queries in theory.

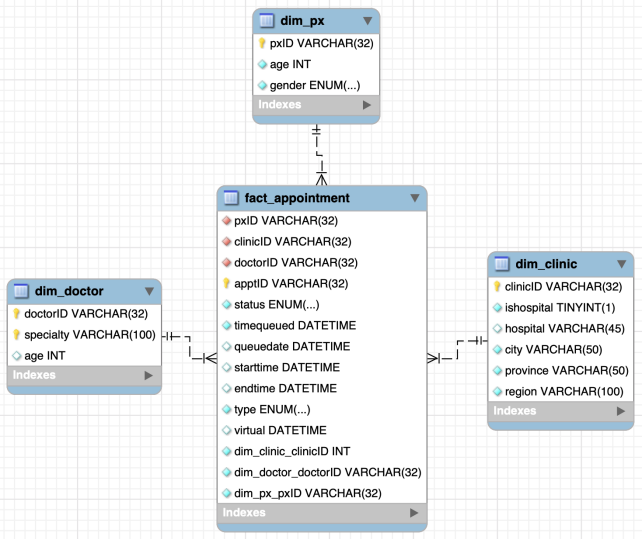


Figure 9: Star Schema of Database

5.3 Hardware Level

The two laptops used are a Macbook 12" that was released in early 2016, and a Macbook Pro 14" that was released in late 2021. The data in Table 1 shows the differences between the two laptops in terms of their hardware specifications.

Table 1: Comparison between Macbook 2016 and Macbook Pro 2021

	Macbook 2016	Macbook Pro 2021
Processor Core	2 Cores	10 Cores
Processor Type	Intel Core m5	Apple M1 Pro
Processor Speed	1.2 GHz	3.2 GHz
RAM Type	8 GB	16 GB
RAM Speed	LDDR3 SDRAM	Unified Memory
Video Card	Intel HD Graphics 515	16 Cores

6 RESULTS AND ANALYSIS

Two forms of testing were carried out in order to assess the OLAP application: function testing and performance testing. Function testing is employed to validate the integrity of the resulting data, whether it is accurate or not. Meanwhile, performance testing is employed to determine the speed and efficiency of the application in displaying the results.

6.1 Function Testing

Function testing aims to determine whether the software is generating the correct results. In this research, the test was applied to the ETL process and the OLAP operations.

6.1.1 ETL Process. To test whether there were some errors in loading the CSV files to the database, the number of rows was checked. The Database was queried with the number of rows it has while also checking how many rows the CSV file has. If it is equal, then all rows were imported successfully into the database.

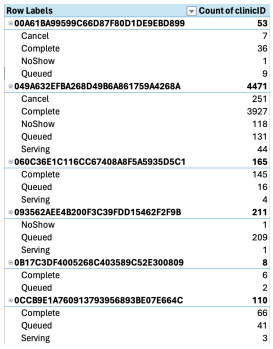
For the transferring of data from the database to the DW, the number of rows was also checked. While doing the ETL operations, *Dataframe.infos()* was constantly executed which provides the number of rows a dataframe has. If the latest *Dataframe.infos()* matches the number of rows in the DW, then all relevant data was transferred successfully.

While also doing the ETL process, the Dataframe was checked using the *Dataframe.head()* every alteration of it to see if the desired manipulation was correct.

Due to the table constraints set in the DW, python should throw an error if an export invalidates any of the set constraints and would stop the export process. This happened in the appointments table where there was a foreign key error. Since all errors were resolved, this means that all tests have passed

For Nifi, if an operation has failed it would reroute to the failure queue. By the end of the ETL, since no flow files were in the failure section and the rows all matched, the ETL process passed testing.

6.1.2 OLAP Operations. To validate the results of the OLAP operations, Microsoft Excel was utilized. In Excel, the different CSV files were first uploaded in order to create a Power Query. Then, *JOIN* operations were performed to merge the tables together. The resulting data was then used to create a pivot table wherein the results were displayed according to the query. Different queries were used as test cases to validate the integrity of the data. For the first example as seen in Figure 10, the results from Query #2 are the same in both Excel and MySQL. Moreover, for the second example as seen in Figure 11, the results from Query #3 are the same in both Excel and MySQL.



Row Labels	Count of clinicID
00A61BA99599C66D87F80D1DE9EBD899	53
Cancel	7
Complete	36
NoShow	1
Queued	9
049A632EFBA268D49B6A861759A4268A	4471
Cancel	251
Complete	3927
NoShow	118
Queued	131
Serving	44
060C36E1C116CC67408A8F5A5935D5C1	165
Complete	145
Queued	16
Serving	4
093562AEE4B200F3C39FDD15462F2F9B	211
NoShow	1
Queued	209
Serving	1
0B17C3DF4005268C403589C52E300809	8
Complete	6
Queued	2
0CCB9E1A7609137939568893BE07E664C	110
Complete	66
Queued	41
Serving	3

(a) Query # 2 in Excel

clinicID	status	numAppointme...
00A61BA99599C66D87F80D1DE9EBD899	Cancel	7
00A61BA99599C66D87F80D1DE9EBD899	Complete	36
00A61BA99599C66D87F80D1DE9EBD899	NoShow	1
00A61BA99599C66D87F80D1DE9EBD899	Queued	9
00A61BA99599C66D87F80D1DE9EBD899	NULL	53
049A632EFBA268D49B6A861759A4268A	Cancel	251
049A632EFBA268D49B6A861759A4268A	Complete	3927
049A632EFBA268D49B6A861759A4268A	NoShow	118
049A632EFBA268D49B6A861759A4268A	Queued	131
049A632EFBA268D49B6A861759A4268A	Serving	44
049A632EFBA268D49B6A861759A4268A	NULL	4471
060C36E1C116CC67408A8F5A5935D5C1	Complete	145
060C36E1C116CC67408A8F5A5935D5C1	Queued	16
060C36E1C116CC67408A8F5A5935D5C1	Serving	4
060C36E1C116CC67408A8F5A5935D5C1	NULL	165
093562AEE4B200F3C39FDD15462F2F9B	NoShow	1
093562AEE4B200F3C39FDD15462F2F9B	Queued	209
093562AEE4B200F3C39FDD15462F2F9B	Serving	1
093562AEE4B200F3C39FDD15462F2F9B	NULL	211
0B17C3DF4005268C403589C52E300809	Complete	6
0B17C3DF4005268C403589C52E300809	Queued	2
0B17C3DF4005268C403589C52E300809	NULL	8
0CCB9E1A7609137939568893BE07E664C	Complete	66
0CCB9E1A7609137939568893BE07E664C	Queued	41
0CCB9E1A7609137939568893BE07E664C	Serving	3
0CCB9E1A7609137939568893BE07E664C	NULL	110

(b) Query # 2 in SQL

Figure 10: Results of Query #2

Row Labels	Count of clinicID
0B17C3DF4005268C403589C52E300809	8
CALABARZON (IV-A)	8
197E342C9FBAD962DBE6D3322F723CA6	1394
CALABARZON (IV-A)	1394
50CF0FE63E0FF857E1C9D01D827267CA	32
CALABARZON (IV-A)	32
54F9DCAFFEB95F402AA2AC051B02C24B	64
CALABARZON (IV-A)	64
5B8E9841E87FB8FC590434F5D933C92C	24
CALABARZON (IV-A)	24
A73305D5BA2857F26BD6EF46E3FBEAE5	342
CALABARZON (IV-A)	342
C31475EB513789058 78A207F72D1CDAE	7
CALABARZON (IV-A)	7
D7012F57148B8A6A970F5BD34806267C6	1
CALABARZON (IV-A)	1
E614F646836AAED9F89CE58E837E2310	1240
CALABARZON (IV-A)	1240

(a) Query # 3 in Excel

clinicID	region	numAppointme...
0B17C3DF4005268C403589C52E300809	CALABARZON (IV-A)	8
197E342C9FBAD962DBE6D3322F723CA6	CALABARZON (IV-A)	1394
50CF0FE63E0FF857E1C9D01D827267CA	CALABARZON (IV-A)	32
54F9DCAFFEB95F402AA2AC051B02C24B	CALABARZON (IV-A)	64
5B8E9841E87FB8FC590434F5D933C92C	CALABARZON (IV-A)	24
A73305D5BA2857F26BD6EF46E3FBEAE5	CALABARZON (IV-A)	342
C31475EB513789058 78A207F72D1CDAE	CALABARZON (IV-A)	7
D7012F57148B8A6A970F5BD34806267C6	CALABARZON (IV-A)	1
E614F646836AAED9F89CE58E837E2310	CALABARZON (IV-A)	1240

(b) Query # 3 in SQL

Figure 11: Results of Query #3

6.2 Performance Testing

Performance testing aims to evaluate the efficiency of the software by measuring the execution speed of the different queries. In this research, the tests were performed centered on the base level with no optimization strategies, and the 3 different optimizations that were conducted on both the database and hardware level. Each test had 10 run-throughs, and the average of those results is shown in Table X.

Table 2: Average Performance of Queries in Seconds With and Without Optimization

Queries	Base Query	Secondary Index	Denormalization	Hardware
Query #1	0.8994	N/A	0.8703	0.3534
Query #2	1.3033	1.3229	1.2130	0.4390
Query #3	0.2635	0.2410	0.1885	0.1270
Query #4	0.2607	0.2406	0.2007	0.1286
Query #5	0.0036	0.0031	0.1927	0.0054
Query #6	0.8019	0.8181	0.6578	0.2566
Query #7	0.2416	N/A	0.2297	0.1135
Query #8	0.4969	N/A	0.4421	0.1836
Query #9	0.5635	N/A	0.4749	0.1997

6.2.1 Performance of Queries using Indexes. Indexes in databases aim to quickly identify the specific value being searched for [1]. When making *primary keys*, a *primary index* is immediately created for each row based on the *primary key*. However, all the other attributes in the table will not have its own index.

Through this logic, in order to speed up the processing speed of the queries, a *secondary index* was created for every attribute found in the *GROUP BY* statement that does not already have a *primary index*. Although this optimization strategy is theoretically supposed to produce more efficient outputs, based on the results in Table 2, the queries are marginally, if not at all, affected by the strategy. We believe that the results are this way because *secondary indexes* are more effective when there is a *WHERE* statement that is being used over a large quantity of data. Because the queries either don't use a *WHERE* statement or are not being used on a large enough quantity of data, the results make sense. Listings 22 - 26 show the SQL statements to *create*, *show*, and *drop* the secondary indexes on their corresponding queries.

Listing 22: Query #2 Secondary Indexes

```
1 ALTER TABLE fact_appointment ADD INDEX
      statusIndexes(status)
2 DROP INDEX statusIndexes on fact_appointment
3 SHOW INDEX FROM fact_appointment
```

Listing 23: Query #3 Secondary Indexes

```
1 ALTER TABLE dim_region ADD INDEX
      regionIndexes(region)
2 DROP INDEX regionIndexes on dim_region
3 SHOW INDEX FROM dim_region
```

Listing 24: Query #4 Secondary Indexes

```
1 ALTER TABLE dim_province ADD INDEX
      provinceIndexes(province)
2 DROP INDEX provinceIndexes on dim_province
3 SHOW INDEX FROM dim_province
```

Listing 25: Query #5 Secondary Indexes

```
1 ALTER TABLE dim_city ADD INDEX cityIndexes(
      city)
2 DROP INDEX cityIndexes on dim_city
3 SHOW INDEX FROM dim_city
```

Listing 26: Query #6 Secondary Indexes

```
1 ALTER TABLE dim_hospital ADD INDEX
      hospitalIndexes(hospital)
2 DROP INDEX hospitalIndexes on dim_hospital
3 SHOW INDEX FROM dim_hospital
4 ALTER TABLE dim_specialty ADD INDEX
      specialtyIndexes(specialty)
5 DROP INDEX specialtyIndexes on dim_specialty
6 SHOW INDEX FROM dim_specialty
```

The results in Table 2 show that Query #1, and #7-9 are not applicable for *secondary indexes* because of two different reasons. For Query #1, the strategy is not applicable because the *GROUP BY* statement is being employed on *c.clinicID* which is a *primary key* of table *dim_clinic*. Because it is a *primary key*, the attribute would already have its own *primary indexes* and would not benefit from having a *secondary index* on top of that. Secondly, the reason why the strategy is not applicable for Query #7-9 is that it's not possible to create *secondary indexes* using functions like *YEAR()*, *QUARTER()*, and *MONTH()* [4]. One potential workaround would be to separate the values of those functions into their own columns and create *secondary indexes* from there.

6.2.2 Performance of Queries using Star Schema. A star schema is similar to the snowflake schema wherein all the sub-tables of the *dimension tables* in the snowflake schema are denormalized into just one table. This results in one fact table with only one layer of *dimension tables* as seen in Figure 9.

Based on the results in Table 2, we can see that the processing speed of most queries is significantly faster than the base query with no optimizations. This result is as expected because by denormalizing the tables into one, the queries no longer have to spend extra time joining the other tables together and can almost directly find the data instead. When comparing Listing 18 and Listing 27,

the number of *JOIN* statements significantly decreases from 5 statements to only 2.

Listing 27: Denormalized Query #6

```
SELECT c.hospital,
       d.specialty,
       COUNT(d.specialty) as specialtyCount
FROM dim_doctor as d
       JOIN fact_appointment as f
       ON d.doctorID = f.doctorID
       JOIN dim_clinic as c
       ON f.clinicID = c.clinicID
WHERE  c.hospital = "Makati␣Medical␣Center"
            OR c.hospital = "The␣Medical␣
    City"
GROUP BY c.hospital, d.specialty
ORDER BY specialtyCount DESC, d.specialty;
```

However, even if most of the queries are performing as expected, the outlier in these results is Query #5 in Table 2 which slowed down in its efficiency by almost 100%. This may potentially be caused by the query having to look through denormalized data where the *region*, *province*, and *city* are all joined together with the *fact table* while the unoptimized query only needs to join with the *fact table* and the *city dimension table*. A possible reason why this problem arose is that the *location table* was not normalized in the first place. If the table was normalized properly, then the comparison may be more accurate with the expected results.

6.2.3 Performance of Queries in Various Hardware. To test the impact of the hardware specifications on the performance of the queries, the queries were run 10 times on both a Macbook 12" (2016) and a Macbook Pro 14" (2021). As seen in Table 2, this form of optimization may be the most effective with a range of 100% - 200% improvement on the speed of the queries if there are any. This seems to show that while optimizing the queries at a database level is important, failing to optimize at the hardware level will cause significant inefficiencies.

7 CONCLUSION

The paper described the process of building an OLAP application that can generate analytical reports concerning the peformances of the clinics in a given location, duration and appointment status. Moreover it can also generate reports on doctor specialties between two given hospitals. The application utilizes a data warehouse that is populated from the SeriousMD Dataset using ETL tools such as Python and Apache NiFi.

Results showed that the largest impact of query optimization may come from hardware-level optimization. Conversely, based on the queries that were used, secondary indexes have little to no impact on the optimization. Lastly, for queries that focus on data reports, a Star schema is usually faster in producing the appropriate results.

Building and maintaining a data warehouse is essential in keeping data integrity, reliability, and consistency. With the assistance of ETL tools, it keeps the warehouse updated by automating data transformation and validation. It's also recommended to use a data warehouse if the source dataset is very large or the data warehouse accommodates multiple data sources. With the help of OLAP operations, analytical reports can be generated on the source dataset. Those kinds of operations are preferred over OLTP as the goal of the application is to perform analysis on the data rather than to perform CRUD operations. In order for the application to be user-friendly, query optimization, whether it's at the database level or hardware level, is needed, especially since the dataset the application is working on is large.

At the database level, secondary indexes may work best when the *WHERE* statement is being used on a large dataset. Furthermore, if there are a lot of *JOIN* statements in the query, it would be best to denormalize the schema of the Data Warehouse. Lastly, it would be remiss to disregard the hardware level as that may be a major factor in slowing down the application.

The OLAP application can help generate analytical reports that would help the medical field in understanding their strategies concerning resource allocation, hospital locations and trends. It would also assist future developers by providing a documented technical report that would assist them in developing their own databases and analytical applications.

8 DECLARATIONS

8.1 Declaration of Generative AI in Scientific Writing

"During the preparation of this work the author(s) used ChatGPT in order to collect knowledge on Regular Expressions, fix Latex formatting, and correct the grammar of the paper. After using this tool/service, the author(s) reviewed and edited the content as needed and take(s) full responsibility for the content of the publication."

8.2 Contributions

Table 3: Record of Contributions

Member	MCO1	Technical Report
Frances Dela Cruz	ETL Script, OLAP Application	ETL Script, OLAP Application, Conclusion
Francis De Leon	Data Warehouse Design, ETL Script	Introduction, Data Warehouse, ETL Script
Mikkel Gamboa	OLAP Application, Function Testing, Performance Testing	Query Processing and Optimization, Results and Analysis, Conclusion

REFERENCES

[1] 2019. SQL CREATE INDEX Statement. https://www.w3schools.com/sql/sql_create_index.asp
[2] 2019. What is Tableau? https://www.tableau.com/why-tableau/what-is-tableau
[3] markingmyname. 2023. Map Many-to-Many Relationships - Visual Database Tools. https://learn.microsoft.com/en-us/sql/ssms/visual-db-tools/map-many-to-many-relationships-visual-database-tools?view=sql-server-ver16

[4] Yevgeny Simkin. 2014. Does it improve performance to index a date column? https://stackoverflow.com/questions/20938753/does-it-improve-performance-to-index-a-date-column

[5] Michael Widenius and David Axmark. 2024. MySQL :: MySQL 8.0 Reference Manual :: 8 Optimization. https://dev.mysql.com/doc/refman/8.0/en/optimization.html