Project Report Part II: Vaccines

Group 21:

Ananov Georgy - 890980

Le Duong - 894834

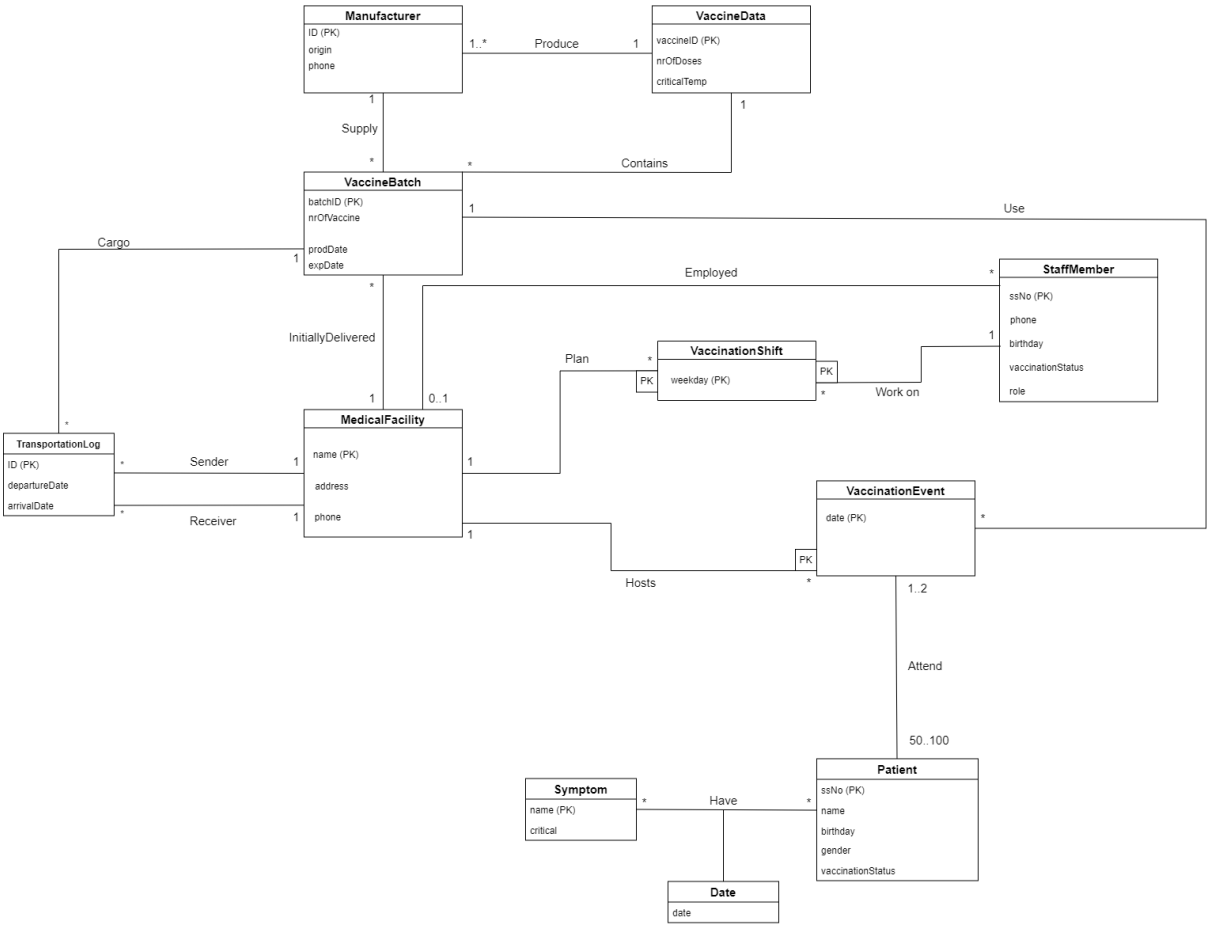
Nguyen Tam - 906968

Pham Hieu - 895118

Database

1. Improved UML & Relational Schema

Taking into account both the feedback part 1 and the problems arising in the process of building a real database from the theoretical relational schema, we made significant changes in the UML as well as the relational schema. Below is the revised UML (a PDF with old and new UML for better comparison is also attached at the end of the report).



Most of the changes in the UML is deleting the foreign keys in a class. Other important changes in the UML includes:

- Association “Contain” is added between class VaccineData and VaccineBatch to add the vaccineID to the vaccine batches.
- Association “Cargo” is added between class VaccineBatch and TransportationLog to add the vaccine batch ID to the transportation logs.
- Class VaccinationShift is changed into a PK class, and it receives the PK from class MedicalFacility and StaffMember. This modification removes a lot of redundancies relating to class VaccinationShift when creating tables in a database. It also allows class VaccinationShift to be more efficient and useful in terms of connecting MedicalFacility and StaffMember and storing information about vaccination shifts according to the data.
- Association “Employed” is added between class MedicalFacility and StaffMember to store the information of where each staff works (adapting to the data).
- Class VaccinationEvent is changed into a PK class, and it receives the PK from class MedicalFacility to store the information about date and location of each vaccination event.

Along with the changes in UML diagram, the relational schema is also modified to be more efficient when creating tables in the real database:

- We create a relation for each class in the UML diagram.
- For many-one associations, we also decided to combine the relations by adding the PK of the one-side to the relation of the many-side as a foreign key. The relation combination effectively reduces the number of relations in the schema from 21 to 14. This change also makes things much easier for us when creating and managing the tables in the database later on.
- Only many-many associations have separate relations.

Below is the final relational schema:

```
VaccineData(vaccineID, nrOfDoses, tempMin, tempMax)
Manufacturer(ID, origin, phone, vaccineID)
MedicalFacility(name, address, phone)
VaccineBatch(batchID, amount, manufDate, expDate, manufID,
vaccineID, initialReceiver)
TransportationLog(ID, departureDate, arrivalDate, batchID,
senderName, receiverName)
StaffMember(ssNo, name, phone, birthday, vaccinationStatus,
role, employer)
VaccinationShift(location, weekday, worker)
VaccinationEvent(date, location, batchID)
Patient (ssNo, name, birthday, gender)
Symptom (name, critical)

Diagnosed (patient, symptom, date)
Attend(date, location, patient)
```

2. Database Creation & Populating

2.1. Database Creation

To create the database in PostgreSQL, we create 2 files: one in SQL, one in Python.

- The SQL file (table_creation.sql) contains the queries to create the tables in the database according to the relational schema above. In addition to the CREATE TABLE queries, we also added 2 DOMAINs, which is used for checking the weekday in VaccinationShift and the gender in Patient.
- The Python file (table_creation.py) is in charge of connecting to the PostgreSQL database and executing the table_creation.sql file.
- Note: when we executed the table_creation.sql file, we faced the error with creating table Attend. It turned out that the table was using both the date and the location from VaccinationEvent as the foreign keys, and when writing the SQL queries, we had to clarify those foreign keys as a tuple, not two single foreign key declarations. This case was mentioned in the quiz in lecture week 2, but it seems we only really learn once we do it ourselves.

2.2. Database Populating

To populate the database in PostgreSQL, we use one Python file (table_population.py). The Python file is used for connecting to PostgreSQL, and populating the tables in the database using pandas library. CSV files in the folder /data/CSVs are used as the inputs to create the dataframes in pandas, and then these dataframes are converted into SQL tables with the pandas function to_sql(). These CSV files contained cleaned data from the provided Excel file, and the information in the files (column names, orders, etc,.) are also modified to fit the tables in the database (read more about data cleaning and preprocessing below).

Data Cleaning

We took data entirely from different sheets in the CSV file. Since most of the data were in a good condition with no missing information, we did not have to clean them too intensively.

First we took all the sheets and read them into dataframes using the pandas function read_csv(). Here is an example with the vaccine type sheet:

```
df_vaccine_type  =  pd.read_excel('data/vaccine-distribution-data.xlsx',
sheet_name="VaccineType")
```

For all sheets, we deleted all the empty columns that were in the file themselves or might be generated when the file is being exposed to usage. An example could be seen in how we work with the vaccine data sheet:

```
vaccine_data = df_vaccine_type[[col for col in df_vaccine_type if not
col.startswith('Unnamed:')]]
```

Then for different tables we took the needed columns, renamed and rearranged them so that they were corresponding to the table schema that we had created. For example, we look at the staff member dataframe:

```
staffMember.columns = ['ssno', 'name', 'birthday', 'phone', 'role',
'vaccinationstatus', 'employer']
staffMember = staffMember.reindex(columns = ['ssno', 'name', 'phone',
'birthday', 'vaccinationstatus', 'role', 'employer'])
```

For all columns that were supposed to be dates, we turned them into the correct data type using the to_datetime function in pandas.Consequently, this will generate NaNs when the text is syntactically incorrect (not in date forms such as YYYY-MM-DD) or when it is an invalid date (such that 2021-02-29). When there were rows with NaNs generated by this function, we deleted them. While there can be other choices such as setting them to some constant values or ignoring them, we believed the date information should be crucial in a medical system, and getting a wrong date would prove to be detrimental to the patient as well as medical staff, so we made the decision to delete them altogether.. An instance for this practice can be seen in the vaccination event dataframe:

```
vaccination_event['date'] =
pd.to_datetime(vaccination_event['date'],errors='coerce')
vaccination_event = vaccination_event.dropna(axis=0, subset=['date'])
```

On a more specific note, in the CSV file the data for the transportation logs lack the ID that we had set to be the primary key for the respective table. We thus have to add them to the dataframe as follow:

```
transportationLog['id'] = transportationLog.index
```

We also noticed that when tables and relations are created using PostgreSQL, their names and attributes are changed into lowercase, which would be extremely important since the names being called in the queries are case sensitive. We specifically changed all tables' and columns' names to lowercase to not cause troubles when reading them.

After checking that there were no further missing values and ill-formatted data, we created different CSV files for different dataframes. This proved to help smooth the process of creating tables in PostgreSQL.

SQL Queries

Query 1

```
SELECT ssno,
       staffmember.name,
       staffmember.phone,
       role,
       vaccinationstatus,
       medicalfacility.name AS location
 FROM  staffmember,
       medicalfacility,
       vaccinationevent,
       vaccinationshift,
       (SELECT(CAST(to_char(to_date('2021-05-10', 'YYYY-MM-DD'), 'Day')
AS CHAR(10))) AS weekday) AS day
 WHERE staffmember.employer = medicalfacility.name AND
       medicalfacility.name = vaccinationevent.location AND
       vaccinationevent.date = '2021-05-10' AND
       vaccinationshift.worker = staffmember.ssno AND
       vaccinationshift.weekday = day.weekday;
```

The query

```
     ssno      |       name        |    phone     |  role  | vaccinationstatus |          location
---------------+-------------------+--------------+--------+-------------------+----------------------------
 19920802-4854 | Kaden Tromp       | 044-624-1591 | nurse  |                 1 | Tapiola Health Center
 19740919-7140 | Deon Hoppe        | 040-399-1121 | nurse  |                 0 | Tapiola Health Center
 19940615-4448 | Jordy Hilpert     | 044-506-1982 | doctor |                 1 | Tapiola Health Center
 19630812-6581 | Jazlyn Schneider  | 040-868-2528 | nurse  |                 1 | Sanomala Vaccination Point
 19771003-5988 | Samir Hills       | 040-093-0059 | nurse  |                 1 | Sanomala Vaccination Point
 19880817-8027 | Haylie Wintheiser | 050-448-8894 | nurse  |                 1 | MyyrmΣki Energia Areena
 19820218-5928 | Elena Bartell     | 041-938-9451 | nurse  |                 1 | MyyrmΣki Energia Areena
 19720223-1761 | Alfreda Champlin  | 041-631-1851 | nurse  |                 1 | MyyrmΣki Energia Areena
(8 rows)
```

The query result

For this first query, the idea is to convert 2021-05-10 into weekdays and check from the ‘vaccinationshift’ table to get who is in charge of that shift for each hospital. The most interesting part is the conversion. Since we don’t have any way to query the weekdays of a day, we have to convert it by hand. First, we change the string ‘2021-05-10’ into date format using to_date, and then using the function to_char to get what weekdays it is, finally, we cast that weekdays into char(10) so we can compared it with the ‘weekday’ attribute of ‘vaccinationshift’. After we have the subquery for converting date into weekday, it is straightforward to do the rest.

Query 2

```
SELECT ssno,
       staffmember.name
  FROM staffmember,
```

```
       vaccinationshift,
       medicalfacility
 WHERE vaccinationshift.worker = staffmember.ssno AND
       vaccinationshift.location = medicalfacility.name AND
       UPPER(medicalfacility.address) LIKE '%HELSINKI%' AND -- add
UPPER METHOD to check for all case-sensitive possibilities
       vaccinationshift.weekday = 'Wednesday' AND
       staffmember.role = 'doctor';
```

The query

```
     ssno      |       name
---------------+------------------
 19740731-5488 | Rosalia Simonis
 19750726-4531 | Shaylee Kris
 19751212-3265 | Hilbert Purdy
 19760102-8374 | Elnora Greenholt
(4 rows)
```

The query result

For the second query, we selected the social security number, and the name of the staff members from tables staffMember, vaccinationShift, and medicalfacility. We then use the conditions as above to join the table. The first two conditions are used to make sure that all three tables are merged correctly, and the last three are to make the query do as in description.

This approach makes sense because the address will have the city name in the last position, so a **LIKE** condition can be used to check whether the city is correct or not. In our data, we have decided a domain for the weekday and role of a staff member, so a simple check as in the query is sufficient.

Query 3

For this problem, we separate them into two different parts: one is to find all the current location of the vaccine batches and the last location in the transportation log, and the other is to find all rows where difference exists between them. The respective queries for the two parts are query 3.1 and query 3.2.

Query 3.1

```
SELECT currentState.batchID,
       currentState.receivername AS lastknownLocation,
       vaccinationbatch.initialReceiver AS currentLocation
  FROM (
           SELECT t.batchid,
                  t.lastdate,
                  m.receivername
             FROM (
                      SELECT batchID,
```

```
                        MAX(arrivaldate) AS lastdate
                    FROM TransportationLog
                   GROUP BY batchID
                )
                AS t
                JOIN
                (
                    SELECT batchid,
                           arrivaldate,
                           receivername
                      FROM TransportationLog
                )
                AS m ON t.batchid = m.batchid AND
                        t.lastdate = m.arrivaldate
      )
      AS currentState,
      vaccinationbatch
WHERE currentState.batchid = vaccinationbatch.batchid;
```

The query

For the third query, we need to create some subqueries for the workarounds. First, we create a subquery to find the last known date of the batch, this is relation ‘t’ in the query. We simply group the TransportationLog according to the batchID, and select the latest date possible. We then join ‘t’ with a relation ‘m’ (‘m’ being the three columns batchid, arrivaldate, and receivername from TransportationLog table) so that we can get the table ‘currentState’ that records the last known date and location of the batchID. The results can be seen in the next table.

batchid	lastknownlocation	currentlocation
B01	Sanomala Vaccination Point	Sanomala Vaccination Point
B02	Sanomala Vaccination Point	Messukeskus
B03	MyyrmΣki Energia Areena	MyyrmΣki Energia Areena
B04	Malmi	Malmi
B06	MyyrmΣki Energia Areena	Iso Omena Vaccination Point
B07	MyyrmΣki Energia Areena	MyyrmΣki Energia Areena
B08	Tapiola Health Center	Tapiola Health Center
B12	Sanomala Vaccination Point	Sanomala Vaccination Point
B13	Iso Omena Vaccination Point	Iso Omena Vaccination Point
B15	Malmi	Malmi
B16	Tapiola Health Center	Tapiola Health Center
B17	MyyrmΣki Energia Areena	MyyrmΣki Energia Areena
B18	Tapiola Health Center	Tapiola Health Center
B21	Iso Omena Vaccination Point	Iso Omena Vaccination Point
B22	MyyrmΣki Energia Areena	MyyrmΣki Energia Areena
B23	Sanomala Vaccination Point	Sanomala Vaccination Point
B24	Malmi	Malmi
B25	Malmi	Malmi
B27	MyyrmΣki Energia Areena	MyyrmΣki Energia Areena
B28	Iso Omena Vaccination Point	Iso Omena Vaccination Point
B29	Sanomala Vaccination Point	MyyrmΣki Energia Areena
B30	Iso Omena Vaccination Point	Iso Omena Vaccination Point

(22 rows)

The query result

Query 3.2

```
SELECT currentState.batchID,
       currentState.receivername AS lastknownLocation,
       vaccinationbatch.initialReceiver AS currentLocation,
       medicalfacility.phone
  FROM (
           SELECT t.batchid,
                  t.lastdate,
                  m.receivername
             FROM (
                      SELECT batchID,
                             MAX(arrivaldate) AS lastdate
                        FROM TransportationLog
                       GROUP BY batchID
                  )
                  AS t
                  JOIN
                  (
                      SELECT batchid,
                             arrivaldate,
                             receivername
                        FROM TransportationLog
```

```
                )
                AS m ON t.batchid = m.batchid AND
                        t.lastdate = m.arrivaldate
        )
        AS currentState,
        vaccinationbatch,
        medicalfacility
 WHERE currentState.batchid = vaccinationbatch.batchid AND
        currentState.receiverName != vaccinationbatch.initialreceiver
AND
        vaccinationbatch.initialreceiver = medicalfacility.name;
```

The query

```
 batchid |     lastknownlocation      |       currentlocation       |    phone
---------+----------------------------+-----------------------------+--------------
 B02     | Sanomala Vaccination Point | Messukeskus                 | 093-101-0024
 B06     | MyyrmΣki Energia Areena    | Iso Omena Vaccination Point | 098-163-4500
 B29     | Sanomala Vaccination Point | MyyrmΣki Energia Areena     | 093-104-5930
(3 rows)
```

The query result

For the query 3.2, we do it in a similar way with 3.1 with one additional detail: in the outermost query, we only selected batches that have current location different from their last known location. This is done simply by taking an inequality checking between the location name in the subquery and the location name in the ‘batchid’ relation.

Query 4:

```
SELECT criticalpatient.patient,
       vaccinationevent.batchid,
       vaccinationbatch.vaccineid,
       vaccinationevent.date,
       vaccinationevent.location
  FROM vaccinationevent,
       (
           SELECT patient.ssno AS patient
             FROM patient,
                  diagnosed,
                  symptom
            WHERE diagnosed.date > '2021-05-10' AND
                  patient.ssno = diagnosed.patient AND
                  diagnosed.symptom = symptom.name AND
                  symptom.critical = 1
       )
       AS criticalpatient,
```

```
       vaccinationBatch,
       attend
 WHERE criticalpatient.patient = attend.patient AND
       attend.date = vaccinationevent.date AND
       attend.location = vaccinationevent.location AND
       vaccinationevent.batchid = vaccinationbatch.batchid;
```

The query

```
 patient | batchid | vaccineid | date | location
---------+---------+-----------+------+----------
(0 rows)
```

The query result

To do this query, we use a subquery to get all the patients that are diagnosed with critical symptoms later than 2021-05-10. This is done by matching the tables ‘patient’, ‘diagnosed’, and ‘symptom’, and taking all the patients that have been diagnosed later than 2021-05-10 and the critical value of ‘symptom’ is 1. When we get that information, all we have to do is matching the patient’s social security number with the vaccination event they attend and the vaccine used in that event.

One detail worth mentioning is the result of the query does not have any rows at all. At first, this might seem strange, but after an inspection of the provided diagnosed data, we found out that there are only 4 patients that have critical symptoms, and all of them are diagnosed before the required date. Thus, the result of the query makes sense here.

Query 5

```
CREATE VIEW patientvaccinationstatus (
    ssno,
    name,
    birthday,
    gender,
    vaccinationstatus
)
AS
    (SELECT patient.ssno,
           patient.name,
           patient.birthday,
           patient.gender,
           (0.5 * (COUNT(attend.date) - 1 + ABS(COUNT(attend.date) - 1)
) )
     FROM patient,
           attend
     WHERE attend.patient = patient.ssNo
     GROUP BY patient.ssno)
     UNION
```

```
    (SELECT patient.ssno,
          patient.name,
          patient.birthday,
          patient.gender,
          0.0
    FROM patient
    WHERE patient.ssNo NOT IN (SELECT patient FROM attend))
```

The query

Run query to check the View (SELECT * FROM patientvaccinationstatus;)

```
    ssno     |         name          | birthday   | gender | vaccinationstatus
-------------+-----------------------+------------+--------+-------------------
 871118-242U | Dr. Victor Armstrong  | 1987-11-18 | M      |               0.0
 830908-9826 | Ana Ward              | 1983-09-08 | F      |               0.0
 930508-413K | Mr. Reid Little II    | 1993-05-08 | M      |               0.0
 790503-394M | Kathlyn Moore         | 1979-05-03 | F      |               0.0
 070218-9109 | Angelina White        | 2007-02-18 | F      |               0.0
 730927-7319 | Mr. Gunner O'Hara MD  | 1973-09-27 | M      |               0.0
 710327-313B | Prof. Ewell Conn      | 1971-03-27 | M      |               0.0
 741222-8947 | Devon Nicolas         | 1974-12-22 | M      |               0.0
 150419-7928 | Ona Crona III         | 2015-04-19 | M      |               0.0
 880810-358W | Braxton Hane          | 1988-08-10 | M      |               1.0
(10 rows)
```

The query result

Since the resulting view has over **150 rows**, I only take the first 10 rows of the view to show how the result looks like.

For this query, we can roughly divide it into two main parts: before the UNION and after the UNION. The part before the union is the part where we find all the patients that attend at least one vaccination and give them a vaccination status. To get the vaccination status of a patient, we use the function 0.5*(COUNT(attend.date) - 1 + ABS(COUNT(attend.date) - 1)). This function will find the maximum between COUNT(attend.date) - 1 and 0. And since we assume that the maximum number of doses for all vaccines is 2, this function will create the vaccination status as described in the task description.

The second part is where we find all the patients that have not attended any vaccinations. This is done simply by selecting all the patients that are not present in the ‘attend’ table, and assign to all of them a vaccination status of 0. When we have all two parts, we union them to obtain the required view.

Query 6

```
SELECT sumfortype.location,
       vaccinetype,
       typesum,
       totalsum
  FROM (
          SELECT vaccinationbatch.initialreceiver AS location,
```

```
                vaccinationbatch.vaccineid AS vaccinetype,
                SUM(vaccinationbatch.amount) AS typesum
            FROM vaccinationbatch
            GROUP BY vaccinationbatch.initialreceiver,
                     vaccinationbatch.vaccineid
    )
    AS sumfortype
    INNER JOIN
    (
        SELECT vaccinationbatch.initialreceiver AS location,
               SUM(vaccinationbatch.amount) AS totalsum
          FROM vaccinationbatch
         GROUP BY vaccinationbatch.initialreceiver
    )
    AS totalsum ON sumfortype.location = totalsum.location
ORDER BY sumfortype.location, vaccinetype;
```

The query

location	vaccinetype	typesum	totalsum
Iso Omena Vaccination Point	V01	10	65
Iso Omena Vaccination Point	V02	30	65
Iso Omena Vaccination Point	V03	25	65
Malmi	V01	20	65
Malmi	V02	30	65
Malmi	V03	15	65
Messukeskus	V01	30	120
Messukeskus	V02	75	120
Messukeskus	V03	15	120
MyyrmΣki Energia Areena	V01	30	85
MyyrmΣki Energia Areena	V02	30	85
MyyrmΣki Energia Areena	V03	25	85
Sanomala Vaccination Point	V01	10	40
Sanomala Vaccination Point	V02	30	40
Tapiola Health Center	V01	10	55
Tapiola Health Center	V02	45	55

(16 rows)

The query result

For this query, we need two subqueries, one to calculate the sum of each dose in each facility, and one to calculate the total sum of vaccine doses in each facility. The first subquery is done by grouping the vaccine batch using ‘initialreceiver’ and ‘vaccineid’, and if we sum the amount of vaccine in each batch, we can get the sum of each vaccine type at each facility. The second query is similar, except that this time we group the vaccine batch using only the receiver to get the total amount of doses. After we have those two subqueries, we can join them and select the required columns.

The result of this query consists of 4 columns, the location, ‘vaccintype’, ‘typesum’, and ‘totalsum’. ‘Location’ and ‘vaccintype’ are quite self-explanatory, the ‘typesum’ is the total number of doses of a particular ‘vaccinetype’ in a medical facility, while the total sum is the

total number of ‘vaccinedoses’ in a medical facility. Because of the way the relation is structured, there can be some redundancy in the ‘totalsum’ attribute. Namely, if the location of the medical facility is the same, then the total sum is also the same. But to preserve the readability and simplicity of the query, we will leave the result in this form.

Note: We assume that each time a vaccine batch is used in a vaccination event, the ‘amount’ attribute in ‘vaccinebatch’ is updated, meaning that the ‘amount’ attribute in the excel file is the *current* amount stored in that hospital and we only calculate the sum based on that.

Query 7

```
SELECT vaccinatedpatients.vaccinetype AS vaccinetype,
       withsymptom.symptom AS symptom,
       ROUND( (CAST (withsymptom.numberofpatient AS DECIMAL) /
vaccinatedpatients.numberofpatient), 3) AS frequency
  FROM (
           SELECT vaccinationbatch.vaccineid AS vaccinetype,
                  COUNT(patient.ssNo) AS numberofpatient
             FROM vaccinationevent,
                  vaccinationbatch,
                  attend,
                  patient
            WHERE vaccinationevent.batchid = vaccinationbatch.batchid
AND
                  vaccinationevent.date = attend.date AND
                  vaccinationevent.location = attend.location AND
                  attend.patient = patient.ssno
            GROUP BY vaccinationbatch.vaccineid
       )
       AS vaccinatedpatients
       INNER JOIN
       (
           SELECT vaccinationbatch.vaccineid AS vaccinetype,
                  symptom.name AS symptom,
                  COUNT(patient.ssno) AS numberofpatient
             FROM vaccinationevent,
                  vaccinationbatch,
                  attend,
                  patient,
                  symptom,
                  diagnosed
            WHERE vaccinationevent.batchid = vaccinationbatch.batchid
AND
                  vaccinationevent.date = attend.date AND
```

```
                vaccinationevent.location = attend.location AND
                attend.patient = patient.ssno AND
                patient.ssno = diagnosed.patient AND
                diagnosed.symptom = symptom.name AND
                vaccinationevent.date <= diagnosed.date
            GROUP BY vaccinationbatch.vaccineid,
                symptom.name
        )
        AS withsymptom ON vaccinatedpatients.vaccinetype =
withsymptom.vaccinetype
  ORDER BY vaccinetype, symptom;
```

The query

vaccinetype	symptom	frequency
V01	blurring of vision	0.029
V01	diarrhea	0.029
V01	fatigue	0.029
V01	feelings of illness	0.029
V01	fever	0.086
V01	headache	0.200
V01	high fever	0.057
V01	inflammation near injection	0.029
V01	itchiness near injection	0.114
V01	joint pain	0.171
V01	muscle ache	0.229
V01	nausea	0.114
V01	warmth near injection	0.086
V02	chills	0.037
V02	fatigue	0.037
V02	feelings of illness	0.148
V02	fever	0.074
V02	headache	0.037
V02	high fever	0.037
V02	joint pain	0.148
V02	lymfadenopathy	0.074
V02	muscle ache	0.185
V02	nausea	0.074
V02	vomiting	0.037
V03	anaphylaxia	0.027
V03	chest pain	0.027
V03	diarrhea	0.081
V03	fatigue	0.027
V03	fever	0.081
V03	headache	0.108
V03	high fever	0.027
V03	inflammation near injection	0.027
V03	joint pain	0.054
V03	muscle ache	0.081
V03	pain near injection	0.027

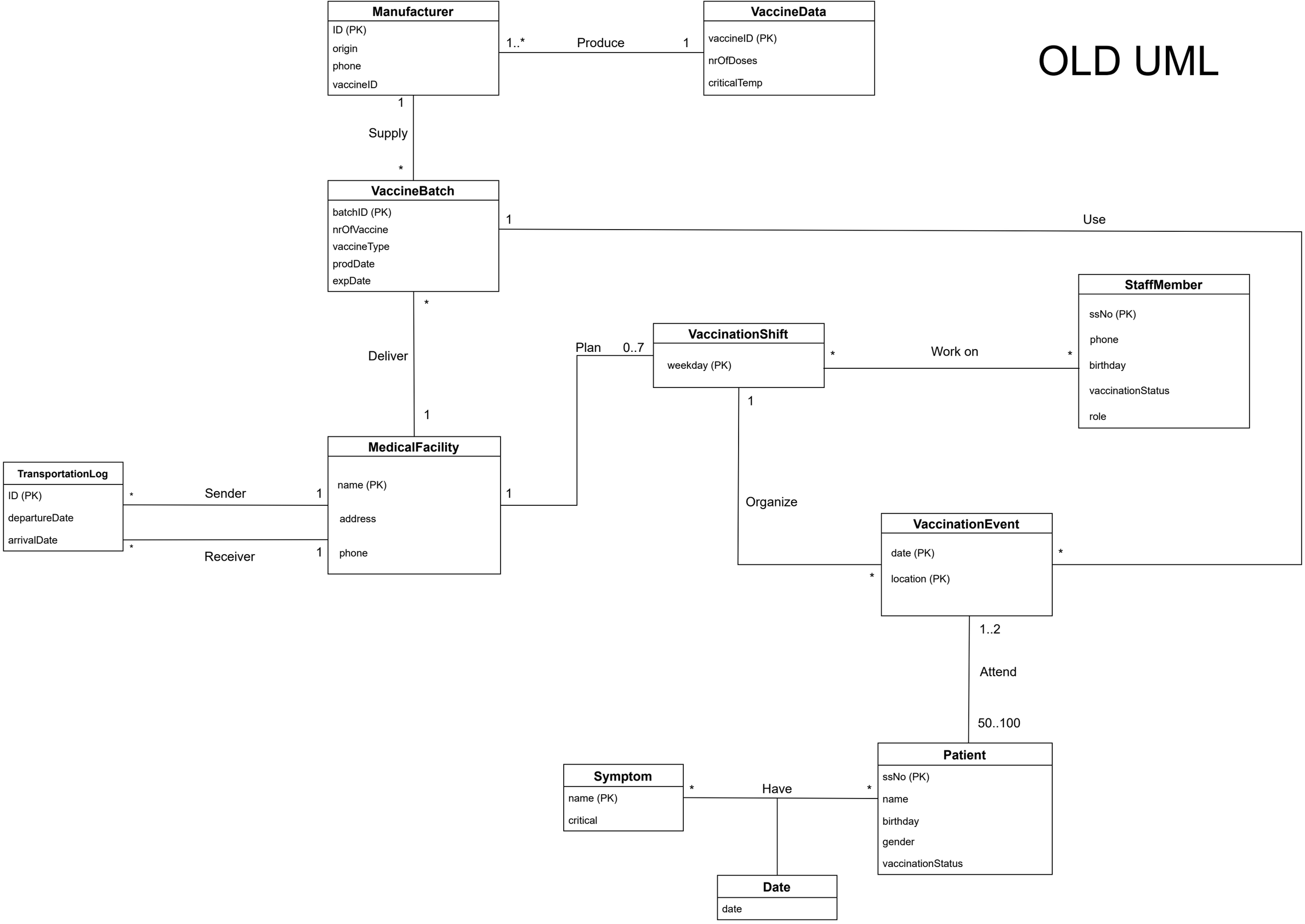
(35 rows)

The query result

Similar to query 6, in this query we created 2 subqueries and inner join them. The first subquery is to get the total number of patients who got vaccinated for each vaccine type. This is done by matching the ‘vaccinationevent’, ‘vaccinationbatch’, ‘attend’, and ‘patient’, grouping them using ‘vaccinationid’ that are stored in 'vaccinationbatch’ and summing the number of patients for each vaccine type. The second subquery is used to get the patients who are diagnosed with a certain symptom after getting treated with the vaccine. The query is quite similar to the first subquery, except that we have to choose only the patient who are diagnosed with a symptom after they attend a vaccination, and we group by the vaccine type and the symptom. After we have the two subqueries, we can inner join them and divide the number of patients that have a symptom after getting a vaccine dose to the total number of patients who got vaccinated using that vaccine type, we also round the results to 3 characters after the decimal.

Note: For the second query, we choose that the symptoms are considered to be caused by the vaccine even if the patient is diagnosed on the same day they receive the vaccine. This makes sense because it would be better if we are pessimistic about the vaccine rather than ignoring some potential dangerous symptoms that are caused by the vaccine.

OLD UML



NEW UML