

Otto Vintola
Miro Keimiöniemi
Marius Scleruc
Harsh Rathee

Contents

1	Introduction	1
1.1	Use cases	1
2	Design	2
2.1	UML Diagram	2
2.2	Relational Schema	3
3	Instructions for using the database	4
4	Analysis and potential improvements	5
4.1	Improvements	5
4.2	Scalability	6
4.3	Example runtimes	6
5	Process and group work	8
5.1	Division of roles and tasks	8
5.2	Schedule used when working on the project	8
5.3	Problem-solving as a group	9
6	Conclusion	9

1 Introduction

This is the documentation for the CS-A1155 - Databases For Data Science course's vaccine distribution database group project as implemented by group 15. This document details the purpose, design and various features, as well as potential improvements of the final database and a run through of the process by which it was created.

1.1 Use cases

The purpose of the database is to host data about a vaccine distribution operation in a well-organized manner that makes it easy and simple to manage the related data. This includes efficient storing, accessing, adding, editing, deleting and drawing inferences leading to important insights about the data by way of querying it in an intuitive manner.

The main use cases, as for most databases, are data storage and data analysis. Some potential and already actualized, more specific use cases include tracking the vaccine batches either in terms of their current real-time location, their full path or their manufacturer. This, combined with

tracking patients symptoms after or during vaccination, can be used to detect the possible sources of defects or side effects, such as whether the problem is common among a certain vaccine type or only for the ones produced by a certain manufacturer, or whether a batch gets contaminated or tampered with during transport.

Provided that the data stream is continuous and gets updated at reasonable intervals, the database can therefore be used to track the live progress of the operation and shape it in real time by, for example, tracking potentially faulty batches to find issues in design, manufacturing or transportation and improve each one appropriately, balance healthcare workers' shifts, track patients' and personnel's vaccination statuses and thus follow the progress of the operation or track potential exposures in healthcare environments and therefore assign quarantines.

Afterwards, the database can be used to evaluate the effectiveness of the campaign in terms of, for example, geographical and demographic reach and risks and benefits of each vaccine down to batch level precision, that can also reveal the best, most reliable and worst, most inconsistent manufacturers.

2 Design

Below are the final designs for the UML diagram and the relational schema, with justifications for their design decisions, that were used to create the database tables and can be used to navigate them in writing new queries or formatting new data to be added to the database.

2.1 UML Diagram

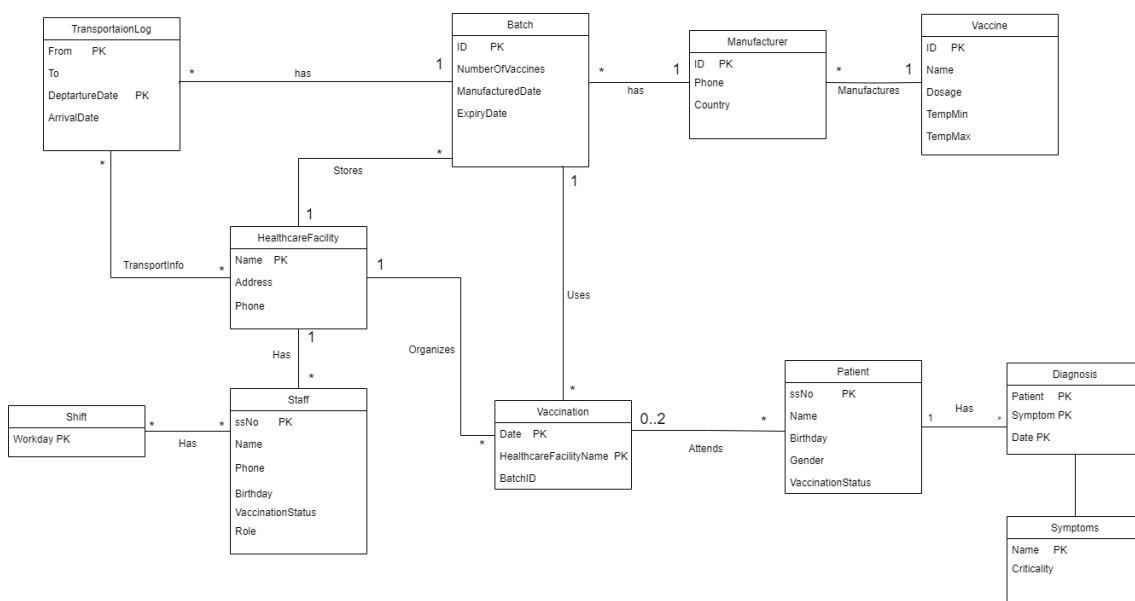


Figure 1

The initial UML model was designed to be as accurate and faithful of a representation as possible with respect to the task assignment description. More importantly, where deviations, assumptions

or varying interpretations were made, they were in service of the three guiding core principles: simplicity, generality and extensibility.

The balance between generality and sufficient specificity manifests mostly in the naming of the classes such as HealthcareFacility, which can contain any type of hospital or clinic and Staff, which includes everyone from nurses to doctors or even cleaners to managers. This implies an assumption about the roles and therefore hierarchy of the staff members being irrelevant, which proved to be the right one for this project as all queries were successful. Similar assumptions about the relative relevance of different types of data with respect to the known and predicted use cases were made for other classes as well. Still, these decision did not lose any information, only made some of it more laborious to access while making others easier to access, which was a worthy trade-off for the given project. Retaining all information was also a priority of ours.

Simplicity was more of a learning process where most of the progress was made in converting the UML diagram to a relational schema and using this process to reflect and continually redraw the UML diagram according to what made sense in the relational schema in terms of the Boyce-Codd Normal Form. This enabled us to get rid of all association classes and keep most classes rather simple with limited attributes and simple inter-class relationships.

Extensibility mostly follows from simplicity and generality but initially we went even a bit too far with it in that we had foregone the expiry date for the Batch relation as we had an expiry time in the Vaccine relation instead, the reasoning being that each vaccine type may have a different expiry time, which could then be used to calculate the date for the particular vaccine if the database were to be extended such that, for example, a batch could contain multiple different types of vaccines and therefore uniform expiry date would be too inflexible. However, as the specification dictated that each batch only contains one type of vaccine, the date was moved back to batch.

Only after getting the data could we properly evaluate our database design and the schema against it. Most of our assumptions turned out to be compatible with the data but some needed revision due to the data being structured a little differently or containing more information than we had prepared for. While it mostly worked, the above decision was made in response along with the addition of a diagnosis class and some changes to a few tables' attributes such as adding a name attribute to vaccine and splitting our initial critical temperature to minimum temperature and maximum temperature, adding an expiry date and manufacturer id to batch, swapping around Vaccine and Manufacturer in the UML, dropping id from Vaccination and adding date to Attends. More about the details of these changes can be read in the table design section of the part 2 project document.

2.2 Relational Schema

Vaccine(id, name, dosage, tempmin, tempmax)

Manufacturer(id, phone, country, vaccineid)

HealthcareFacility(name, address, phone)

Batch(id, numberofvaccines, manufacturerid, manufactureddate, healthcarefacilityname, expiry-date)

Staff(ssno, name, phone, birthday, vaccinationstatus, role, healthcarefacilityname)

Shift(workday, ssno)

TransportationLog(from, to, batchid, departuredate, arrivaldate)

Patient(ssno, name, birthday, gender, vaccinationstatus)

Attends(date, healthcarefacilityname, ssno)

Vaccination(date, healthcarefacilityname, batchid)

Symptoms(name, criticality)

Diagnosis(ssno, symptom, date)

The final relations are almost a one-to-one mapping of the classes present in the UML diagram with only one additional relation that was not directly represented as a UML class, that being the *Attends* relation, that keeps track of the people having attended each vaccination event. This was done to facilitate maximal simplicity for ease of use, enabling extensibility and making the understanding and further development of the database easier and faster.

Their development was very interlinked with the UML diagram such that most other design choices were already mentioned above. The tables of the database were created based on these schema, the only difference being some additional checks on values in addition to the primary keys underlined in these schema and foreign keys, seen as connections between classes in the UML diagram. Again, more about the exact technical details about the evolution of the relational schema can be read about in the table design section of the part 2 project document.

3 Instructions for using the database

The database is created with the createTables.py python file in the code folder that connects to the dbcourse.cs.aalto.fi server - the credentials for which were provided via an email to the group leader - and creates the tables to the postgresSQL database using the tableScript.sql file in the database folder that converts the final relational schema into tables in one-to-one manner with additional constraints for the values.

createTables.py reads the values from vaccine-distribution-data.xlsx in the data folder and updates any changes to the file in the database. The parameter `if_exists='append'` allows adding novel data without rewriting any pre-existing values. Therefore, data can be added by adding it to the excel file or by changing the filepath in the filePath variable to another excel file and running createTables.py but for safety, it will never be replaced, modified or removed by the script, making it safe to run in all scenarios. Moreover, if the user tries to re-run the createTables.py file with the same data, they will get an error, specifying that the data is already stored inside the database - just like we intended.

In all our python scripts in the code folder, the database is accessed via the `psql_conn` variable, that represents a connection to an engine created using an URI consisting of dialect, user, password, host and database, which can all be read from the scripts in this non-security-sensitive project. Using those credentials, the database can be accessed in other ways as well, such as using the psql shell. Queries can be run based on the relational schema or with the assistance of the UML diagram highlighting how the tables are related to each other.

For the final deliverable of the project, our group decided to implement some data visualizations using the streamlit python library. In order to access them, a user has to run `streamlit run ./code/someVisualization.py` from project folder's terminal.

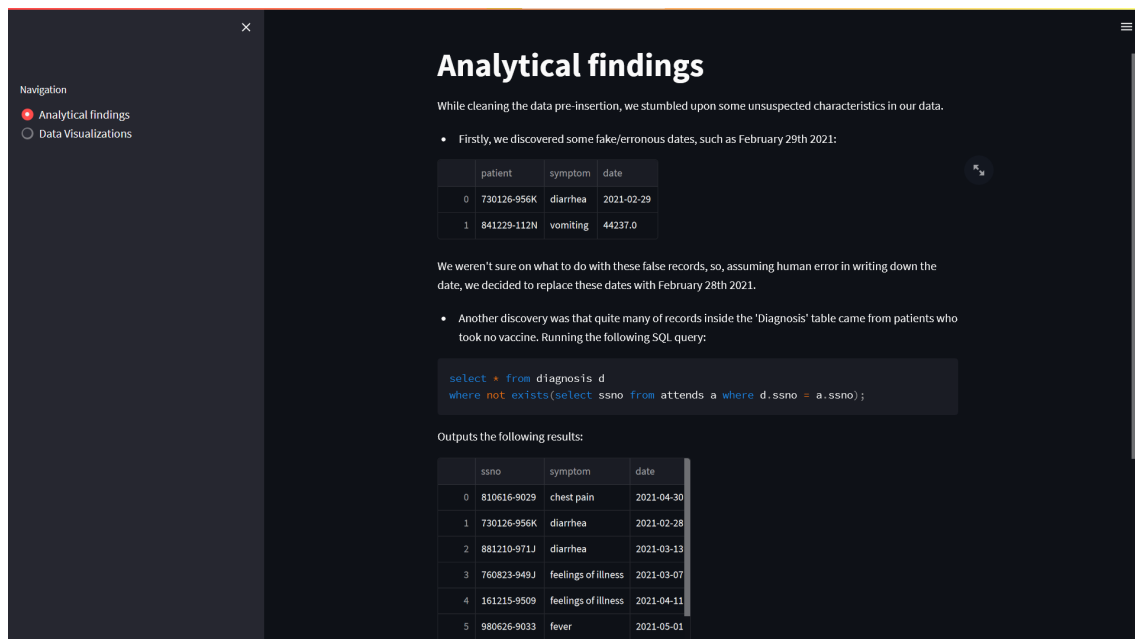


Figure 2: The main page of streamlit web application. It is hosted on localhost:8501.

4 Analysis and potential improvements

4.1 Improvements

4.1.1 (More) Checks and Triggers

To start off the main improvement that would make the workflow a lot easier is to have checks and triggers to make sure that no erroneous data gets inserted. Currently, we format the excel file to fit into our database schema – essentially we clean the data ourselves by manually parsing it. So, we could possibly edit the database such that only correct data goes in. Another addition to this is to add a trigger to check if something has gone massively wrong, in which case we rollback the server to an anchor point.

4.1.2 Backups and recovery

Relative to the previous topic, we could have backups of the server somewhere, to ensure that we do not lose the progress. This would come in use, if something catastrophic were to happen to the database where a simple rollback would not suffice for it – an example would be like a complete wipeout of the server.

4.1.3 Indexes

In addition, we could add indexes to our database. Eventually when more-and-more of the population gets vaccinated, our database will grow in size. This will make it harder to find data from the database since the search space will be larger. This could be relieved by adding indexes to the database. This would make it computationally easier to query the database.

4.1.4 Efficient queries

Another improvement that would be possible is to optimize and streamline query writing with functions. We could do this by implementing functions for common queries, so that we can just call those functions instead of writing the same query over-and-over again. A similar thing related to this is to use query pooling.

4.1.5 Timeout from server

A major issue we noticed in the workflow was having too many connections to the database. This would lead to the members getting timeouts from the server. This was a big issue for us before the deadline for the third part of our project. We were trying to run the queries to find the results but we were getting timeouts and empty results.

4.2 Scalability

In total the number of vaccinations and patients that our database will have to handle is much larger than it is now. Currently, we do not have much data, however, it is good to design the database with scalability in mind.

Using primary keys that correlate to unique constraints in real life is a good step towards scalability in the database, for example, using `ssNo` as a primary key means that it is used to identify a unique person. It is also unique in the real world meaning that two people cannot have the same `ssNo`.

Another way to change the design for scalability is to shard the database in a way that would distribute the data and tables in a more even manner. For example, we could distribute the database into instead of containing the entire Helsinki-Espoo region, we could just do it for Helsinki and Espoo separately, or even break it up for smaller pieces such as a specific hospital. This would limit the amount of data each database would have, but still keep the workload manageable.

Also, as a last note on scalability, we tried to design the database with as few 'ID' attributes as possible. What we mean is, we tried to eliminate redundant 'ID's for relations that already had the necessary attributes to serve as primary keys. For example, initially we had an `eventID` for a vaccination, but eventually realised that we could just use the date and place of the vaccination, since only one hospital/clinic can have only one vaccination per day. This meant we could get rid of the `eventID` attribute and use those two as primary keys. This reduces the workload of the server, increasing the simplicity of the database, since it does not have to interpret unnecessary rows.

4.3 Example runtimes

Here are some example runtimes of queries. Firstly we have a simple example that only uses two different tables and does not use aggregate operations.

```
query = text('''SELECT * FROM patient
                JOIN attends ON patient.ssno = attends.ssno
                where patient.birthday > '2010-01-01';''')

result = pd.read_sql_query(query, psycopg)
print(result)
```

✓ 0.4s

Figure 3: As we can see the runtime is about 0.4s. The query produces 14 rows.

Here we can see a more complicated query that uses the aggregate functions **HAVING**, **GROUP BY** and **COUNT** and also it parses **six** tables.

```
query = text('''SELECT p.name, v.name, b.manufactureddate, COUNT(*) AS num_vaccinations
FROM Patient p
JOIN Attends a ON p.ssNo = a.ssno
JOIN Vaccination va ON a.date = va.date AND a.healthcarefacilityname = va.healthcareFacilityName
JOIN Batch b ON va.batchid = b.id
JOIN manufacturer ON b.manufacturerid = manufacturer.id
JOIN Vaccine v ON manufacturer.vaccineid = v.id
GROUP BY p.name, v.name, b.manufactureddate
HAVING COUNT(*) > 1;''')

result = pd.read_sql_query(query, psycopg)
print(result)
```

✓ 0.6s

Figure 4: It has a runtime of 0.6s and produces only 1 row.

Then we have an even heavier query that uses all of the same aggregate functions, but parses **eight** tables.

```
query = text('''SELECT p.name, v.name, b.manufactureddate, COUNT(*) AS num_vaccinations
FROM Patient p
JOIN Attends a ON p.ssNo = a.ssno
JOIN Vaccination va ON a.date = va.date AND a.healthcarefacilityname = va.healthcareFacilityName
JOIN Batch b ON va.batchid = b.id
JOIN Manufacturer m ON b.manufacturerid = m.id
JOIN Vaccine v ON m.vaccineid = v.id
JOIN Symptoms s ON s.name = p.name
JOIN Diagnosis d ON p.ssNo = d.ssNo AND s.name = d.symptom
WHERE v.dosage = 2
GROUP BY p.name, v.name, b.manufactureddate
HAVING COUNT(*) > 1;''')

result = pd.read_sql_query(query, psycopg)
print(result)
```

✓ 0.7s

Figure 5: It has a runtime of 0.7s and produces no rows.

We can start to notice a trend here, that the more tables and aggregate functions the query

uses, the longer the runtime is, however, since we do not have much data, the amount the runtime increases with more tables is not very significant – it would most likely increase much more per table, if we had more data.

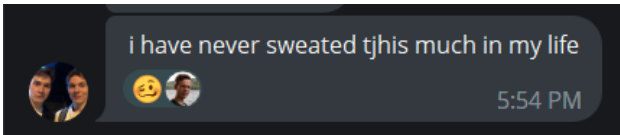
5 Process and group work

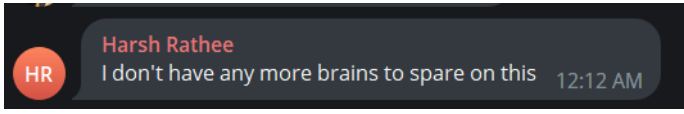
Our process of building the database as a group could be described as quite agile in many ways, in both the good and the bad. Our group had no hierarchy, meaning that persons who were doing well in other courses often took the lead and organized weekly Friday meetings. Often a bulk of the work was already done by then (because of our Thursday exercise sessions), so then we simply split up the rest and polished the final product together. Everyone worked reliably independently and was very involved in the process, and each member took turns leading certain aspects of the project, whether that was making sure everything gets done or gluing it all together. Due to different schedules, the weekly contributions fluctuated somewhat. Nevertheless, overall contributions are approximately equal for each group member.

5.1 Division of roles and tasks

As specified above, our group didn't have neither a specific hierarchy nor assigned individuals for a specific tasks. Everyone was free to voice their preferences, concerns, and doubts. In the end, a small pattern emerged - Miro tended to take most documentation-related responsibility, while others led the coding efforts. Marius got really carried away in week 3 and wrote many of the queries before anyone had even looked at the assignment, Otto handled the submissions and overall coordination of the team and Harsh pointed out all our mistakes consistently and reliably. However, everyone did an equitable job during the project by being involved in each aspect of bringing it about.

5.2 Schedule used when working on the project

Task/Event	Time period
Formation of the group	Week 0 (April 18 th – 24 th)
First meeting to discuss project UML structure	May 3 rd
Received access to the gitlab repository	May 11 th
Commencement of work on turning the UML into BCNF & creation of the relational schema	May 12 th
Realization that the required submission documents had missing parts & scrambling to write them down	May 15 th , 17 : 45
Project part 1 submission	May 15 th 17 : 53 : 51
	May 15 th 17 : 54
First meeting for project part 2	May 19 th
Second meeting for project part 2	May 26 th

Troubles because of constant server time-outs	May 28 th
Aphasia symptoms:	
	May 29 th 00 : 12
Finishing project part 2 & submission	May 29 th morning
Presentation of the Project	May 30 th
First & only meeting for project part 3	June 1 st
Creation of the project deliverable visualization using streamlit	June 6 th
Project part 3 submission	June 12 th
Project deliverable submission	June 16 th

The process was quite seamless and we delivered all parts right on time in quite a polished form. Yet, if there is one thing that our group could have done better, it is time management. We sometimes started working properly only days before the deadline, especially for the first return. Having perhaps not collaborated enough with the entire team in making the very initial UML diagram, we rewrote quite a significant portion of it, perhaps doing a bit more work than was necessary. On the other hand, it was the friction in the process of trying to convert the UML into the relational schema that prompted these changes in the first place. It could have been started earlier but it got done well regardless and so there is really not much to complain.

5.3 Problem-solving as a group

Platforms used for collaboration include Telegram, Gitlab, and in-person meetings in room M235a that had a massive screen for us all to see each others' presentations on what had been done and what still needed to be done. These worked quite seamlessly and we did not even have any major merge conflicts due to well split tasks with clear, self-chosen boundaries on what each one of us would do.

6 Conclusion

In conclusion, we created a well-organized and efficient database that is well-suited for small to mid-size data sets, using the core design principles of simplicity, generality and extensibility, while also making sure that no information would ever be lost. The database is optimized for simple queries such as tracking the vaccine batches and vaccination statuses but can be used relatively easily for more complex queries as well such as identifying possible defects with certain vaccines or batches as well as identifying reliable manufacturers.

Storing, accessing, editing, deleting and analyzing data in order to gain insights and improve the vaccine distribution operation is seamless and intuitive with little redundancy but there is room for improvement in ensuring the integrity of the data via backups and rollback triggers as well as automating the parsing of data prior to inputting it to the database and adding more checks. Additionally, in anticipation for larger data sets, the database could be sharded to distribute the

data more evenly and indexes could be created for faster access along with streamlining query writing by encapsulating the most common queries in functions. The database works well for its current use cases, but, should it be expanded or used with significantly more data, adding these checks, triggers, backups, indexes and sharding would be among the first future improvements.

From the perspective of group work, this project also went quite well. Yes, there were some minor frictions here and there, but we were all able to forget about them in the end by sharing the common goal of getting our project done as efficiently and well as possible.

Overall, the project is a success as we got good answers to all queries of interest and the database is relatively easy to use with great potential for extensibility and scaling. Our group work was just fine and we got everything done on time, even if there were a few close calls due to technical difficulties. Doing the project was very insightful and useful for getting acquainted with the practicalities of creating, maintaining and accessing databases, giving us an excellent foundation for all our future projects.