Databases for Data Science

CS-A1155

Final Project Deliverable

Designing a Volunteer Matching System (VMS) with the Finnish Red Cross (FRC)

Submitted by Group 15:

Aleksi Alatalo

Hoang Xuan Gia Khanh

Le Hong Phuc

Napat Borvornpadungkitti

Introduction

Volunteering plays a vital role in addressing community needs and fostering an environment of mutual aid and cooperation. However, connecting volunteers with tasks suitable for their skill sets can be challenging at times. Traditional methods of volunteer matching are often time-consuming and inefficient, leading to suboptimal allocation of volunteer resources.

As part of the course Databases for Data Science, our four-member team—consisting of Aleksi Alatalo, Hoang Xuan Gia Khanh, Le Hong Phuc, and Napat Borvornpadungkitti—attempted to design a robust database system that supports effective volunteer matching for the Finnish Red Cross (FRC). Our work in Parts I and II of the project includes creating UML diagrams and relational schemas, implementing SQL statements, performing data analysis with Python, and exploring how predictive analysis can be utilized to enhance the volunteer matching system.

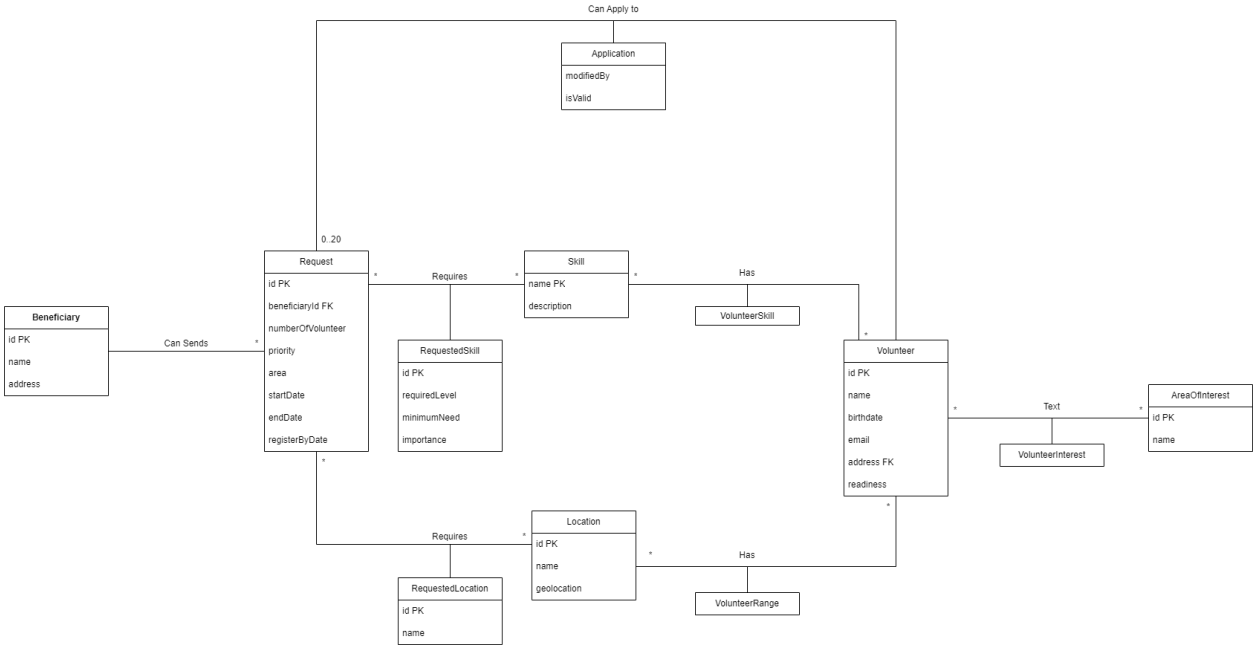
We hope that through this project, we were able to leverage knowledge and skills gained from the course to deliver a comprehensive solution that enhances the Finnish Red Cross's ability to efficiently respond to needs within our community.

Summary of Parts I and II

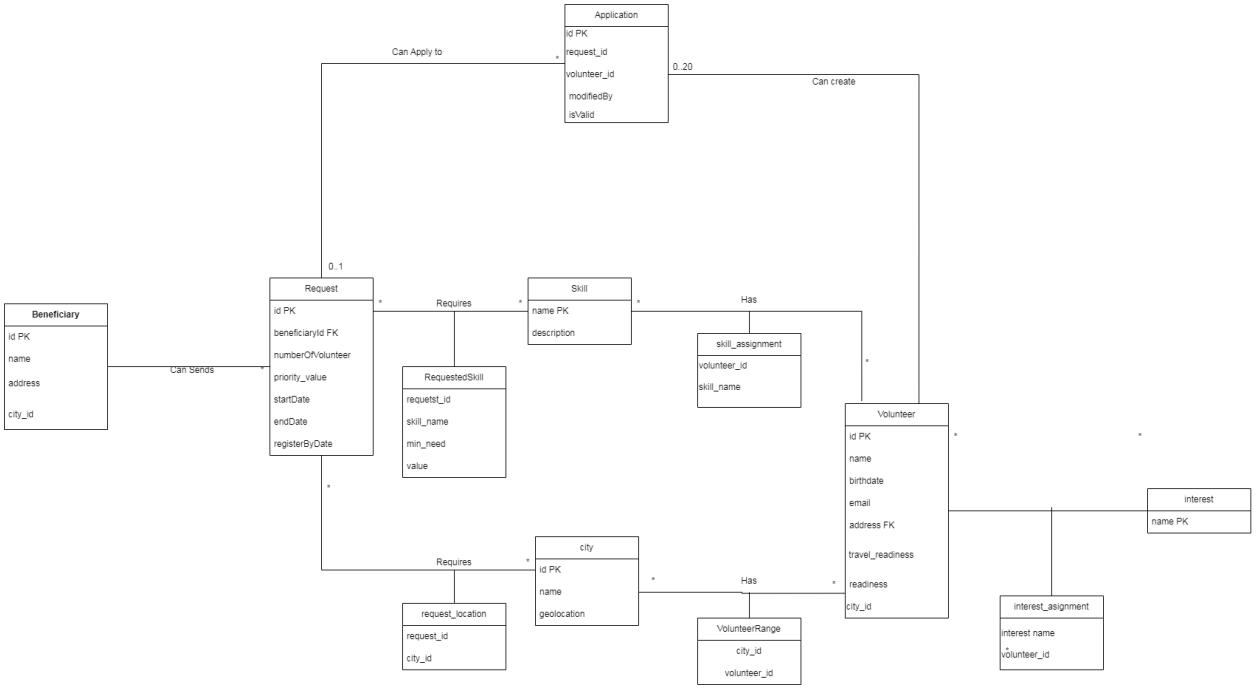
UML Diagrams

We represented components in the volunteering system using the Unified Modeling Language, also known as UML. In Part I of the project, our UML diagram was constructed based on our interpretations of the details provided in the Project Description page of MyCourses. One such interpretation was that each request skill could be identified with an ID, in addition to the ID of the request and skill name. On the other hand, the modified diagram was designed based on our understanding of the data with which we populated our database in Part II. Since the primary keys for most of the modified classes are named "id", we decided to rename such keys in the associations that connect those relations to clarify which classes they are from. For example, classes "request" and "city" have a many-to-many association, so their primary keys will be in "request_location". However, as both primary keys are called "id", they were renamed to "request_id" and "city_id" in the association class to avoid confusion.

Original UML:



Modified UML:



Relational Schemas

In Part I, we created a relational schema based on the original UML diagram. However, due to the given data, we changed our approach when populating the database in Part II. The resulting differences in the relational schemas can be seen below:

Original Classes:

- Beneficiary(ID, name, address)
- Request(ID ,beneficiaryID, numberOfVolunteer, priority, areaOfInterest, startDate, endDate, registerByDate)
- Skill(name, description)
- Volunteer(ID, name, birthdate, email, address, readiness)
- AreaOfInterest(ID, name)
- Location(ID, name, geolocation)

Original Association classes:

- Applications(ID, volunteerID, requestID, modifiedBy, isValid)
- RequestSkill(ID, requestID, name, requiredLevel, minimumNeed, importance)
- RequestLocation(ID, requestID, locationID, name)
- VolunteerSkill(volunteerID, name)
- VolunteerInterest(volunteerID, interestID)
- VolunteerRange(volunteerID, locationID)

Modified Classes:

- Beneficiary(id, name, address, city_id)
- City(name, id, geolocation)
- Interest(name)
- Request(id, title, beneficiary_id, number_of_volunteer, priority_value, start_date, end_date, register_by_date)
- Skill(name, description)
- Volunteer(id, birthdate, city_id, name, email, address, travel_readiness)

Modified Association Classes:

- Interest_assignment(interest_name, volunteer_id)

- Request_location(request_id, city_id)
- Request_skill(request_id, skill_name, min_need, value)
- Skill_assignment(volunteer_id, skill_name)
- Volunteer_applicaton(id, request_id, volunteer_id, modified, is_valid)
- Volunteer_range(volunteer_id, city_id)

The main difference between the two schemas is the names of the classes and attributes. Additionally, some association classes in both schemas that have the same purpose have different attributes as their primary keys.

BCNF and Anomalies

BCNF is defined as either X -> Y being a trivial functional dependency or X being a superkey. We'll go through the relations above and show that they're BCNF:

Beneficiary: id-> name, address, city_id (only functional dependency has id as superkey meaning the relation is in BCNF)
City: id, ->name, geolocation (same as previous)
Interest: name (name is a superkey, meaning it is in BCNF)
Request:id -> title, beneficiary_id, number_of_volunteer, priority_value, start_date, end_date, register_by_date (same as beneficiary and city)
Skill: name -> description (only functional dependency has superkey as key = in BCNF)
Volunteer: id -> birthdate, city_id, name, email, address, travel_readiness (same as beneficiary)
Interest_assignment: interest_name, volunteer_id (relation consists of supekey, meaning it is in BCNF)
Request_location: request_id, city_id (same as previous)
Request_skill: request_id, skill_name -> min_need, value (the combined attributes form the superkey, this relation is in BCNF)
Skill_assignment: volunteer_id, skill_name (same as reques_location)
Volunteer_applicaton: id -> request_id, volunteer_id, modified, is_valid (same as beneficiary)
Volunteer_range: volunteer_id, city_id (same as skill_assignment)

We've shown that all the relations in the database are in BCNF. Being in BCNF is advantageous for the database since it helps manage anomalies. Update anomalies occur when the same data is repeated in multiple rows, and changes are made in some but not all instances. Since each attribute is stored only once, updating a single tuple in the table is enough.
when a delete anomaly occurs it means that you cannot delete data from the table without having to delete the entire record. Since the relation is in BCNF, deleting a tuple in one table does not directly affect data in other tables, but it does render it somewhat useless, meaning that deletion anomalies exist in the database. Insertion anomalies exist as well, since you can't for example add a skill to a volunteer that doesn't exist.

Queries

We started out thinking we can just follow exactly the instructions that we were given, and that each question will have one definitive answer. As we approached each question, we realized that there might be several ways to carry out each question. Therefore, instead of trying to figure out what exactly is the intended way each query should be carried out, we made our own assumptions for them and wrote the query accordingly. Here are our design processes and assumptions for the SQL statements:

Exercise A.1: The starting date and the end date is joined with the title using concat

Exercise A.2: The exercise is done under the assumption that for each request each valid applicant's id is displayed next to the number of skills said applicant has that match with the request. The count(concat(rs.request_id, ' ', sa.volunteer_id)) is used to count how many skills the applicant has that match with the request

Exercise A.3: The query returns the request id, name of the skill requested, and the number of missing volunteers for that requested skill. The number of missing number of volunteers, named missing_number, was obtained by first grouping by the request Id, requested skill's name, and

requested skill's minimum need and the subtracting the number of distinct volunteers who have applied to the request and possess the requested skill from the minimum need of that requested skill.

Exercise A.4: Assumption: I don't know what is meant by closest date so I assume that the closest date must have been after the database is established, therefore after the data is available so that's why I ordered it from the latest day down

Exercise A.5: This query returns the volunteer ID and every request within the range with at least 2 matching skills or no skills required. This tasks was accomplished through the union of 2 subqueries:

- The first one searches for requests within each volunteer's range that have at least 2 matching skills and
- The second one finds requests within each volunteer's range that do not require skills.

For the first subquery, we joined the tables volunteer_range, skill_assignment, request_location, and request_skill, and counted the number of distinct skill names which were grouped by the volunteer and request Ids. For the second subquery, we joined volunteer_range to request_location, and returned the request Ids that do not appear in the request_skill table for each volunteer Id, as that would mean the requests do not require any skill.

Exercise A.6: This query returns volunteer Ids, the Ids of requests that are still available for registration and whose title matches the volunteer's interest and, and the request title. To compare the interest names to the titles of requests, the two column names were modified:
For the request titles, the word "need" and space between each word were removed with the substring and replace functions, respectively, before the upper function was used to convert all lowercase letters to their uppercase counterparts.
For the interest names, only the upper function was used to convert lowercase letters to uppercase. To ensure that the requests are still available for register, we used the current_date function to obtain the current date and compared it with the register-by dates.

Exercise A.7: The first 3 lines return the results when some of the applicant's range does not match with the request location, SOME but not ALL. This is when the subquery comes into action, by using NOT IN, it will filter out the ones that have the range which match with request from the original 3 query lines

Exercise A.8: Pretty much the same as the question

Exercise A.9 [Free choice]: The query retrieves the data of volunteers who do not have applications.
Justification: This helps identify inactive volunteers so that the FRC organization can encourage them to participate in volunteer work.

Exercise A.10 [Free choice]: The query searches up the beneficiaries, the skills they have requested for, and the number of requests for each skill.
Justification: This helps identify the skills that each beneficiary needs most frequently, enabling the organization to better tailor support towards them.

Exercise A.11 [Free choice]: The query returns the number of applications that each volunteer has.
Justification: This helps notify the organization which volunteers are the most active, which gives said volunteers recognition for their work and may encourage other volunteers to be more active.

Exercise A.12 [Free choice]: For each city the query returns the request in those cities from most prioritized to least. This is to help the city authorities allocate resource and easier management of volunteering activities

Exercise B.A.1: This view lists next to each beneficiary the averages of the following: number of volunteers applied, age of volunteers, and number of volunteers needed per request. The number of applicants per request was obtained using the subquery 'na' where distinct application Ids were counted after being grouped by the request Id. This subquery was then joined to request,

volunteer_application, and volunteer and averaging with the function avg after grouping by beneficiary Id. The age of the volunteer was obtained with the age and date_part functions before being averaged with the function avg. The average number of volunteers was obtained by using the function avg on r.number_of_volunteers which we assumed to be the average number of volunteers needed. We had also tried the method of creating a subquery where we grouped request_skill by request Id and summed the minimum need of each skill together before we averaged it in the main query. However, upon reading the project description, we came to believe that averaging r.number_of_volunteers was the correct action.

Exercise B.A.2 [Free choice]: Taking inspiration from A.2, the query returns for each volunteer the valid request that they applied to how many skills they have that match that request and the request is in their range, ordered by most matching skills to least. This is to help volunteers find which request they are most suited for/helpful in.

Exercise B.B.1: There are three criteria to be fulfilled in order for an ID to be valid:

- I checked for the first one by using the regex '_____'. While this works, I think using LENGTH() makes the code more readable.
- For the 2nd criteria, I use SUBSTRING() to extract the 7th character and check if it is in the valid separators array, which contains all of the valid separators.
- For the last one, I store all of the valid control characters in an array and check if the one in the id is correct via its index in said array.

Exercise B.B.2: For this one, instead of using the provided formula, the new number of volunteers is calculated by adding the previous one with the difference between the new and previous min_need. The 2 formulas should be equivalent.

Exercise B.C.1: Since the specification is phrased so badly and even contradicts itself, I basically followed what I can understand from the specification and ignored the contradiction. However, I created it as a function that takes in a request_id and has a transaction inside it. This is because the specification said to create a transaction for "a request". I interpreted this as a request inputted by the user.

Exercise B.C.2 [Free choice]: This transaction registers a new volunteer, assigns their skills, and sets their range. I think this transaction should be implemented because registering a new volunteer is a crucial step that will be needed in the future. In production, this transaction can take in the volunteers' inputs as parameters for their skills and ranges.

Predictive Analytics [Optional Addition]

To effectively predict vulnerabilities and demand, it would be essential to gather more comprehensive data to analyze. We might start developing a predictive model based on the data like shown below.

With the data in the database as is, some predictions could be made to foresee the amount of volunteers and requests in a given city. The model wouldn't be that far from just graphing the data on plots. so, we could scrape through social media to gather more data. This could include a table that contains numbers that indicate the likeliness of more or less requests being sent from a certain area, based off of for example news of natural disasters. After cleaning up the data, we could also identify and create other relevant features, such as calculating the availability of volunteers based on their travel readiness and previous commitments or prioritizing requests based on the number of volunteers needed, start date, and priority value. Next, we could use machine learning algorithms to build models that can predict future trends. These models might vary from different tasks. For example, a regression model may be used to predict the number of volunteers needed for future requests. On the other hand, a classification model may be used to categorize requests based on urgency and required skills.

The predictive models could be used to forecast demand based on events happening in the world and identify potential hotspots for either demand of volunteers or volunteer availability. For example, if a model predicts a high demand for volunteers with medical skills in a particular city, the FRC can proactively allocate resources and prepare volunteers in advance. This could help ensure a more efficient and timely response to crises.

Reflection

Role division

Part I:

- Le Hong Phuc: UML

- Napat Borvornpadungkitti: Relational schema
- Hoang Xuan Gia Khanh: non-trivial functional dependencies
- Aleksi Alatalo: BCNF

Part II:

- Le Hong Phuc: Triggers and Transactions
- Napat Borvornpadungkitti and Hoang Xuan Gia Khanh: Queries and Views
- Aleksi Alatalo: Analysis

Performance and scalability

The database itself performs quite well. Only when there is significant strain on the servers it is hosted on do the queries take longer. The database can perform its most critical tasks, meaning the allocation of volunteers quickly. Since the load on the database doesn't peak that high during use, but instead is focused on more computationally expensive queries, it would be better to initially scale vertically rather than horizontally.

Conclusion

The primary goal of this project was to utilize skills and knowledge gained from the course to design a volunteer matching system to streamline the process of connecting volunteers with suitable opportunities.

In Part I, our team designed UML diagrams and created relational schemas for the matching system we had in mind, as well as evaluated the nontrivial functional dependencies and BCNF. In Part II, we attempted to realize our designs. We created a database, populated with the provided data, and implemented the required SQL statements as well as ones we believe will support our system's ability to streamline the process of allocating volunteers to appropriate causes. Additionally, we also explored how predictive analysis can be leveraged to improve the system by forecasting vulnerabilities, so that measures can quickly be developed to cope with such problems.

Overall, it was an honor to be able to contribute to a cause greater than ourselves. We learnt some valuable lessons from the course and hope that our efforts on this project helped the Finnish Red Cross manage their data and smoothly carry out their missions.