CS-A1155 - Databases for Data Science
2024

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

COURSE PROJECT

Group 5

Phạm Quang Bách
Roope Kettunen
Meishan Lin
Irmuun Tuguldur

Table of Contents

1 Project Description 2
Part I 3
2 UML Diagram 3
2.1 Assumptions 3
3 From UML model to Relational Schema 4
3.1 Classes 4
3.2 Associations 5
3.3 Reasoning for Relations 5
4 Functional Dependencies 6
4.1 Non-trivial Functional Dependencies 6
4.2 Trivial Functional Dependencies 7
5 Boyce-Codd Normal Form (BCNF) and Redundancy/Anomalies 7
Part II 8
6 Synthetic Data and Queries 8
6.1 Synthetic Data and Queries 8
A. BASIC 8
B. ADVANCED 21
a) Views 21
b) Trigger and Functions 23
c) Transactions 26
d) Analysis 29
6.2 Requirements 39
6.3 Presentation slides link 41
6.4 Group Work Timeline and Reflections 42
Reference 42

1 Project Description

The two main actors of the system are volunteers and beneficiaries:

Beneficiaries have unique IDs, names, and addresses. Any beneficiary can make as many volunteering requests as they need. Requests should include a unique ID, the ID of the beneficiary it was sent by the number of volunteers needed, a priority value to indicate how urgent the request is, the area of interest where the request lies, a start date, an end date, and a register by date. Requests also have criteria for acceptance and application validity. These criteria comprise of request skills and request locations. Each request skill has a request ID and a skill name, and each request location has a request ID and a city ID.

Volunteers on the other hand have unique ID, name, birthdate, email, address, readiness to travel (minutes). A volunteer can choose any combination of areas of interests from the following: CollectDonations, FirstAid, FoodHelp, GuideAndTeach, HelpInCrisis, ImmigrantSupport, OrganiseActivities, PromoteWellbeing, WorkInATeam, WorkInMulticulturalEnvironment, WorkWithElderly, WorkWithYoung. Volunteers can also choose any combination of skills from the following: CommunicationAndMarketing, CookingAndBaking, DigitalCompetence, EventHosting, EventOrganizaton, FinanceAndAccounting, HealthCareOrFirstAid, MeetingPeople, Organizational, PhotographyAndVideo, PublicPerformances, Rescue, TeamGuide, TrainPeople.

Each of these skills have a unique name and a description. Beneficiaries can appraise skills by assigning skills a value, a minimum need, and a value to indicate importance.

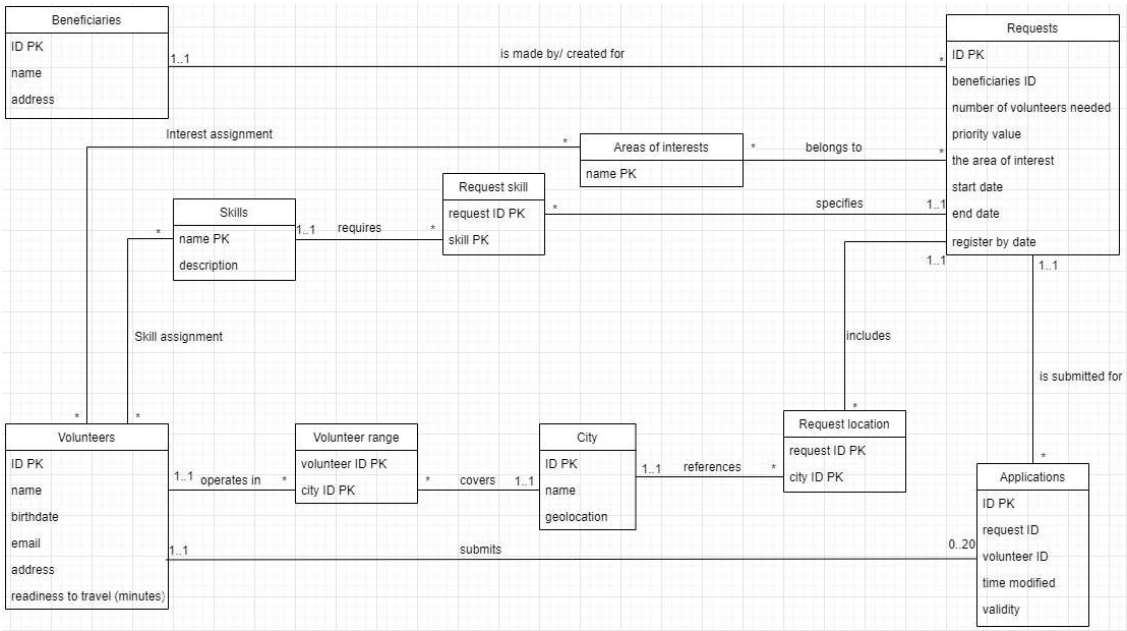
Volunteers can sign up to the system, browse through the volunteering requests, and send up to 20 applications where they apply to the requests. Applications should include a unique ID, the ID of the request it was made to, the ID of the volunteer it was sent by, the time it was modified, and they should indicate whether they are valid or not.

Volunteers operate in ranges. Each volunteer range has a volunteer ID and a city ID. Each city has an ID, a name, and a geolocation [1].

Part I

2 UML Diagram

The following UML model was developed based on the provided description and underlying assumptions:



2.1 ASSUMPTIONS

1. Application validity is represented as a Boolean value, which can be either True or False.
2. A volunteer can be associated with multiple ranges.
3. Priority values are represented as integer number.
4. Request skill "specify" criteria for acceptance: Although the criteria for acceptance are not explicitly shown in the UML, they are implied by this relation. The criteria for acceptance are part of the request, specifying the required skills needed for the request.
5. Criteria for acceptance "includes" Request location: Although the CriteriaForAcceptance are not explicitly shown in the UML anymore, they are implied by this relation. CriteriaForAcceptance was removed due to redundancy, as detailed in the final section of the report. The criteria for acceptance include the locations where the volunteering request can be fulfilled.

6. Assume the request location must select one specific city rather than leaving it empty.
7. Assume volunteers can choose multiple volunteer ranges to operate in, and each volunteer range can only contain 1 city.
8. Assume no more classes need to be added for more information between specific relations, neither consider adding more for part II of this project.

3 From UML model to Relational Schema

3.1 CLASSES

1. Beneficiary (ID, name, address, city_id)
   This class shares the same attributes outlined in the instructions. Given the uniqueness of IDs, we designated it as the primary key for this class.

2. Volunteer (ID, name, birthdate, email, address, travel_readiness)
   This class shares the same attributes outlined in the instructions. Given the uniqueness of IDs, we designated it as the primary key for this class.

3. Request (ID, title, beneficiaries_id, umber_of_volunteers, priority_value, start_date, end_date, register_by_date)
   This class shares the same attributes outlined in the instructions except CriteriaForAcceptance is removed to avoid redundancy. Given the uniqueness of IDs, we designated it as the primary key for this class.

4. Volunteer_ application (ID, request_id, volunteer_id, modified, is_valid)
   This class shares the same attributes outlined in the instructions. Given the uniqueness of IDs, we designated it as the primary key for this class.

5. Interest (name)
   This class shares the same attributes outlined in the instructions. Given that "name" is the only attribute, we designated it as the primary key for this class.

6. Request_location (request_id, city_id)
   This class shared the same attributes outlined in the instructions. Both requestID and cityID are unique, we chose them as the primary keys.

7. Request_skill (request_id, skill_name)
   This class shared the same attributes outlined in the instructions. Both requestID and skill are unique, we chose them both as the primary keys.

8. City (ID, name, geolocation)
   This class shares the same attributes outlined in the instructions. Given the uniqueness of IDs, we designated it as the primary key for this class.

9. Volunteer_range (volunteer_id, city_id)
   This class shared the same attributes outlined in the instructions. Both volunteerID and cityID are unique, we chose them both as the primary keys.

10. Skill (name, description)
    This class shared the same attributes outlined in the instructions. Name is the primary key because each name is unique.

3.2 ASSOCIATIONS

11. Interest_assignment(interest_name, volunteer_id)
    Interest_name: a foreign key linking to the "name" attribute in the "Interest" table, indicating the area of interest associated with a request.
    Volunteer_id: a foreign key linking to the "ID" attribute in the "Volunteer" table, indicating the specific volunteer associated with an area of interest.

12. Skill_assignment(skill_name, volunteer_id)
    Skill_name: a foreign key linking to the "name" attribute in the "Skill" table, indicating the skills associated with a request.
    Volunteer_id: a foreign key linking to the "ID" attribute in the "Volunteer" table, indicating the specific volunteer associated with an area of interest.

3.3 REASONING FOR RELATIONS

1. Request location "references" City: Many-one association. One request location specifies one city it is associated with, but one city can be referred by arbitrarily many Request location from different applications.
2. Requests "includes" Request location: One-many association. One Request location can be included in one Request, but one Request can include arbitrarily many Request locations.
3. Requests "specifies" Request skill: One-many association. One Request skill can be specified in one Requests, but Requests can specify arbitrarily many Request skills.
4. Applications "is submitted for" Requests: Many-one association. One Application entity is related to one specific volunteering request, indicating which request the application is for, but one Request can have arbitrarily many applications.

5. Areas of Interests "belongs to" Requests: Many-many association. One Requests can include many Areas of interests and one areas of interests can belongs to many different Requests.
6. Volunteer range "covers" City: One-many association. Each Volunteer range can only cover 1 city. However, each city can belong to an arbitrarily amount of Volunteer ranges.
7. Request skill "requires" Skills: Many-one association. Each entry in the "Request skill" table specifies a particular skill needed for the request, but one Skill can be selected by Request skill arbitrarily many times.
8. Volunteers "submits" Applications: One-many association. Each application is linked to one volunteer who submits it; however, one volunteer can submit up to 20 applications at the same time.
9. Request "is made by"/ "is created for" Beneficiary: Many-one association. Each request is associated with one beneficiary who either creates the request or is the recipient of the services provided by the request, but one beneficiary can create arbitrarily many Requests.
10. Volunteers "operates in" Volunteer range: Many-one association. Volunteers specify arbitrary many Volunteer ranges they can operate in. And each volunteer range only belongs to one volunteer and specifies only one city.
11. RequestAndAreasOfInterest (<u>interestName</u>, <u>requestID</u>): Many-many association between requests and areas of interest. The "Requests" entity can be associated with multiple areas of interest because a volunteering request might cover various areas. Similarly, an "Area of Interests" can be associated with multiple requests since different requests can fall under the same area of interest.

4 Functional Dependencies

4.1 NON-TRIVIAL FUNCTIONAL DEPENDENCIES

All non-trivial functional dependencies are enumerated below:

1. Beneficiary (<u>ID</u>, name, address, city_id)

   ID → name, address, city_id

2. Volunteer (<u>ID</u>, name, birthdate, email, address, travel_readiness)

   ID → name, birthdate, email, address, travel_readiness

3. Request (<u>ID</u>, title, beneficiaries_id, umber_of_volunteers, priority_value, start_date, end_date, register_by_date)

   ID → title, beneficiaries_id, umber_of_volunteers, priority_value, start_date, end_date, register_by_date

4. Volunteer_ application (ID, request_id, volunteer_id, modified, is_valid)

   ID → request_id, volunteer_id, modified, is_valid

5. City (ID, name, geolocation)

   ID → name, geolocation

6. Skill (name, description)

   name → description

4.2 TRIVIAL FUNCTIONAL DEPENDENCIES

For clarification, trivial functional dependencies are enumerated separately along with their respective justifications:

7. Interest (name)

   It is a trivial relation as it only contains the single attribute name.

8. Request_location (request_id, city_id)
9. Request_skill (request_id, skill_name)
10. VolunteerRange (volunteer_id, city_id)

For the three tables above (RequestLocation, RequestSkill, and VolunteerRange), since it consists of two primary keys, they merge to form a composite primary key. As this resulting key is singular (results in a single primary key for the whole table) the resulting relation is a trivial relation. The same is true for the following relation below (RequestAndAreasOfInterest) that represents matching areas of interest of beneficiaries and their request ID.

11. Interest_assignment(interest_name, volunteer_id)
12. Skill_assignment(skill_name, volunteer_id)

5 Boyce-Codd Normal Form (BCNF) and Redundancy/Anomalies

All the relations listed above are in Boyce-Codd Normal Form. Each of the relations with one primary key has an ID that acts as a determinant for all other columns, so further decompositions are unnecessary.

In the UML and relations, AreaOfInterest only have one column that is "name" and is only referenced by Requests. This makes it possible to combine this table into Requests without any loss of data, meaning it is redundant. However, we believe that this redundancy is justifiable since it allows the database to be more robust with more specific and complex areas of interest in the future if the need arises.

Another anomaly is that the CriteriaForAcceptance have been combined completely with the request class, despite being outlined by the original description. This is due to the fact that what CriteriaForAcceptance contains, RequestLocation and RequestSkill, both independently contains a reference to the requestID, making the class redundant.

Part II

6 Synthetic Data and Queries

6.1 SYNTHETIC DATA AND QUERIES

To create the table and run all our queries, we use python with pandas and sqlalchemy. Run main.py to create the tables and run query for part A, and a), b), c) of B.

The query for part A is in Part_A.sql

The query for part B a, b is in Part_B_a)_b).sql

The query for part B c is in Part_B_c).sql

A. BASIC

1. For each request, include the starting date and the end date in the title.

Sql code:

```sql
SELECT
  REPLACE(title, ' needed', '') AS request,
  start_date AS "start date",
  end_date AS "end date"
FROM
  request;
```

Interpretation & explanations:

We selected title, removed the "needed" with the REPLACE command and return both the start date and end date.

We attempted to eliminate the numerical indexes from all Part A outputs by making adjustments to main.py and altering the SQL code. However, regrettably, those indexes persist.

Query results:

```
              request  start date    end date
0       work in team  2024-07-25  2024-07-28
1    work with young  2022-05-31  2022-06-01
2    guide and teach  2024-07-22  2024-07-27
3    guide and teach  2024-09-25  2024-09-25
4  work with elderly  2021-03-01  2021-03-07
```

2. For each request, find volunteers whose skill assignments match the requesting skills. List these volunteers from those with the most matching skills to those with the least (even 0 matching skills). Only consider volunteers who applied to the request and have a valid application.

Sql code:

```sql
SELECT
  request_id,
  volunteer_id,
  SUM(matching_skills) AS m
FROM
  (
    SELECT
      rs.request_id AS request_id,
      sa.volunteer_id AS volunteer_id,
      COUNT(DISTINCT sa.skill_name) AS matching_skills
    FROM
      skill_assignment sa
      JOIN request_skill rs
        ON rs.skill_name = sa.skill_name
      JOIN volunteer_application va
        ON rs.request_id = va.request_id AND va.volunteer_id = sa.volunteer_id
    WHERE
      va.is_valid
    GROUP BY
      rs.request_id,
      sa.volunteer_id
    UNION
    SELECT
      r.id AS request_id,
      va.volunteer_id AS volunteer_id,
      0 AS matching_skills
    FROM
      request r
      JOIN volunteer_application va
        ON r.id = va.request_id
    WHERE
      va.is_valid
  ) rv
GROUP BY
  request_id,
  volunteer_id
ORDER BY
  request_id,
  m DESC,
  volunteer_id;
```

Interpretation & explanations:

First, the amount of matching skill in each application is Selected with the second SELECT command from the top with the COUNT. Then, it is combined (UNION) with a list of all applications, with the matching skills column all set to 0. This is to account for the application that has no required skill which does not appear in the first SELECT. The combined table is finally selected again by summing to remove the duplicates (Summing with 0 does not change the number of matching skills) and order accordingly.

** We tried to change “m” on the first row into “matching skills” so that the query results are clearer and more readable, but for some reason it was not working the way we want. Therefore, we leave “m” as it is. **

Query results:

```
   request_id volunteer_id    m
0           1  230283-963X  3.0
1           1  011074-9149  1.0
2           1  160903A941P  1.0
3           1  211074-9401  1.0
4           1  211099-910H  1.0
```

3. For each request, show the missing number of volunteers needed per skill (minimum needed of that skill). Assume a volunteer fulfills the need for all the skills they possess.

Sql code:

```sql
SELECT
  rs.request_id,
  rs.skill_name,
  rs.min_need - COUNT(va.volunteer_id) AS missing_volunteers
FROM
  request_skill rs
  JOIN volunteer_application va
    ON rs.request_id = va.request_id
  JOIN skill_assignment sa
    ON va.volunteer_id = sa.volunteer_id AND rs.skill_name = sa.skill_name
WHERE
  va.is_valid
GROUP BY
  rs.request_id,
  rs.skill_name,
  rs.min_need
HAVING
  rs.min_need - COUNT(va.volunteer_id) > 0
ORDER BY
  rs.request_id,
  rs.skill_name;
```

Interpretation & explanations:

We joined request_skill table with volunteer_application and skill_assignments to get the total COUNT of volunteer grouped by request_id and skills_name. Then to get the missing numbers, we just subtract the COUNT from the minimum requirements and pick rows with more than 0 volunteer missing. The results are ordered by request_id and skill_name to make the output organized and easy to read.

Query results:

	request_id	skill_name	missing_volunteers
0	1	CommunicationAndMarketing	2
1	1	EventHosting	1
2	1	EventOrganization	1
3	2	CookingAndBaking	2
4	2	PublicPerformances	1

4. Sort requests and the beneficiaries who made them by the highest number of priority (request's priority value) and the closest 'register by date'.

Sql code:

```sql
SELECT
  id,
  beneficiary_id,
  priority_value,
  register_by_date
FROM
  request r
ORDER BY
  r.priority_value DESC,
  r.register_by_date ASC;
```

Interpretation & explanations:

We just picked all the required columns from the request table, ordered by priority value by descending and register_by_date be ascending.

Query results:

	id	beneficiary_id	priority_value	register_by_date
0	369	4	5	2019-12-26
1	127	4	5	2020-01-03
2	92	2	5	2020-01-29
3	133	3	5	2020-03-07
4	224	8	5	2020-09-11

5. For each volunteer, list requests that are within their volunteer range and match at least 2 of their skills (also include requests that don't require any skills).

Sql code:

```sql
SELECT
  rv.volunteer_id,
  rv.request_id,
  SUM(matching_skills) AS m
FROM
  (
    SELECT DISTINCT
      rv.volunteer_id,
      rv.request_id,
      rv.matching_skills
    FROM
      ((
        SELECT
          rs.request_id AS request_id,
          sa.volunteer_id AS volunteer_id,
          COUNT(DISTINCT sa.skill_name) AS matching_skills
        FROM
          skill_assignment sa
          JOIN request_skill rs
            ON rs.skill_name = sa.skill_name
        GROUP BY
          rs.request_id, sa.volunteer_id
        HAVING
          COUNT(DISTINCT sa.skill_name) >= 2 )
        UNION
        SELECT
          s.request_id AS request_id,
          v.id AS volunteer_id,
          0 AS matching_skills
        FROM
          volunteer v,
          (
```

```sql
            SELECT
              r.id AS request_id
            FROM
              request r
            EXCEPT
            SELECT
              rs.request_id AS request_id
            FROM
              request_skill rs
          ) s
      ) rv,
      volunteer_range vr,
      request_location rl
    WHERE
      rv.request_id = rl.request_id
      AND vr.volunteer_id = rv.volunteer_id
      AND rl.city_id = vr.city_id
  ) rv
GROUP BY
  rv.request_id,
  rv.volunteer_id
ORDER BY
  rv.volunteer_id,
  m DESC,
  rv.request_id;
```

Interpretation & explanations:

This starts the same way that query A2 starts, by counting the number of matching skills, filtering so only requests with more than 2 matching skills are left. Then using UNION to

combine it with the requests that don't require any skills, which is created by eliminating all requests that require skills using EXCEPT. Then the resulting table gets selected again to remove duplicates, and account for volunteer_range and request_location. Finally, the rows are combined using SUM to remove redundant 0's leftover from the UNION and ordered appropriately.

Query results:

	volunteer_id	request_id	m
0	010469-981A	283	8.0
1	010469-981A	2	7.0
2	010469-981A	12	7.0
3	010469-981A	20	7.0
4	010469-981A	21	7.0

6. For each volunteer, list all the requests where the title matches their area of interest and are still available to register.

Sql code:

```sql
SELECT
  v.id AS volunteer_id,
  v.name AS volunteer_name,
  r.id AS request_id,
  r.title AS request_title
FROM
  volunteer v
  JOIN interest_assignment ia
    ON v.id = ia.volunteer_id
  JOIN request r
    ON replace(r.title, ' ', '') ILIKE '%%' || ia.interest_name || '%%'
WHERE
  r.register_by_date > CURRENT_DATE;
```

Interpretation & explanations:

We select the appropriate columns from volunteer and requests, where the interest_name is inside of the request title. We had to remove all spaces from the title and use case sensitive ILIKE because interest name have no spaces. Finally, we filter to get only requests which is valid, in this case, it is that register_by_date have not passed by.

Query results:

	volunteer_id	volunteer_name	request_id	request_title
0	011095-974M	Ulla Saari-Tiainen	24	food help needed
1	011095-974M	Ulla Saari-Tiainen	88	food help needed
2	011095-974M	Ulla Saari-Tiainen	220	food help needed
3	011095-974M	Ulla Saari-Tiainen	314	food help needed
4	011095-974M	Ulla Saari-Tiainen	3	guide and teach needed

** For part A question 1, we deleted needed for the request_title column, but for this question, we just leave them as it is (for example, food help needed), to show there are different ways presenting the query results and it is just individual preference. **

7. List the request ID and the volunteers who applied to them (name and email) but are not within the location range of the request. Order volunteers by readiness to travel.

Sql code:

```sql
SELECT
  va.request_id,
  v.name,
  v.email
FROM
  volunteer_application va
  JOIN volunteer v
    ON va.volunteer_id = v.id
  JOIN request_location rl
    ON va.request_id = rl.request_id
WHERE
  NOT EXISTS
  (
    SELECT
      1
    FROM
      volunteer_range vr
    WHERE
      vr.volunteer_id = v.id
      AND vr.city_id = rl.city_id
  )
ORDER BY
  travel_readiness DESC;
```

Interpretation & explanations:

We select the appropriate columns from volunteer applications which don't have the location in their volunteer range with the NOT EXIST command. The subquery checks if there is no entry in the volunteer_range table for the volunteer's ID and the city ID of the request location. If there is no such entry, it means the volunteer is not within the location range of the request.

Query results:

	request_id	name	email
0	20	Kalervo Rinne	kalervo.rinne@dbcourse.cs.aalto.fi
1	143	Kalervo Rinne	kalervo.rinne@dbcourse.cs.aalto.fi
2	189	Kalervo Rinne	kalervo.rinne@dbcourse.cs.aalto.fi
3	237	Kalervo Rinne	kalervo.rinne@dbcourse.cs.aalto.fi
4	146	Kalervo Rinne	kalervo.rinne@dbcourse.cs.aalto.fi

8. Order the skills overall (from all requests) in the most prioritized to least prioritized (average the importance value).

Sql code:

```sql
SELECT
  rs.skill_name,
  AVG(r.priority_value) AS average_value
FROM
  request_skill rs
  JOIN request r
    ON r.id = rs.request_id
GROUP BY
  rs.skill_name
ORDER BY
  average_value DESC;
```

Interpretation & explanations:

rs.skill_name selects the name of the skill. AVG(r.priority_value) AS average_value calculates the average priority value for each skill and assigns it the alias average_value. FROM: request_skill rs: it specifies the request_skill table with an alias rs. JOIN the request table with the request_skill table on the request_id. GROUP BY the results by the skill_name to calculate the average priority value for each skill. ORDER BY the result set by the average priority value in descending order.

Query results:

	skill_name	average_value
0	Organizational	2.622378
1	EventOrganization	2.596026
2	HealthCareOrFirstAid	2.573529
3	PublicPerformances	2.554688
4	DigitalCompetence	2.545455

[free choice]: Construct 4 queries of your choice and explain why these are important for the VMS.

9. *For each city, give the number of applications and the total amount of volunteers needed (request multiplied by the amount of volunteer evenly divided if they operate in multiple cities)*

Rationale: by understanding the approximate demand and activity in each city, the Red Cross can allocate resources (such as recruitment efforts or additional support) to cities with higher needs or lower volunteer engagement.

Sql code:

```sql
SELECT
  t1.city_name,
  volunteer_supply,
  volunteer_demand
FROM (
  SELECT
    c."name" AS city_name,
    COUNT(DISTINCT vr.volunteer_id) AS volunteer_supply
  FROM
    city c
    JOIN volunteer_range vr
      ON vr.city_id = c.id
  GROUP BY
    c.id
  ) t1
  JOIN (
    SELECT
      c."name" AS city_name,
      SUM(r.number_of_volunteers::FLOAT / lc.COUNT) AS volunteer_demand
```

```sql
    FROM
      city c
      JOIN request_location rl
        ON rl.city_id = c.id
      JOIN request r
        ON r.id = rl.request_id
      JOIN (
        SELECT
          rl.request_id,
          COUNT(rl.city_id)
        FROM
          request_location rl
        GROUP BY
          rl.request_id
      )
      lc
      ON lc.request_id = rl.request_id
    GROUP BY
      c.id
  ) t2
    ON t1.city_name = t2.city_name
ORDER BY
  volunteer_demand / volunteer_supply DESC;
```

Interpretation & explanations:

Table t1 is the total number of volunteers by city. Table t2 is the number of volunteers needed by each city with the assumption that the requests are distributed evenly if the requests are in multiple cities. This is achieved by first counting the number of different cities each request is in by table "lc", then taking the sum of request volunteer amount divided by this number (this will result in an approximate volunteer demand for each city, which might not be a whole number). Finally, we joined both and ordered them by

demand divided by supply to find which cities has the most demand compared to supply of volunteers.

In this query, we assumed that volunteers could volunteer for multiple requests, but each request can only take in the number of volunteers needed without repetition in multiple cities. This explains why the totle amount of volunteer supply is higher than the totle number of volunteers and why we did not divide the volunteer suply by how many cities they operate in like with the requests.

Query results:

	city_name	volunteer_supply	volunteer_demand
0	Rusko	63	1462.516667
1	Rautavaara	55	1230.283333
2	Tammela	56	962.450000
3	Myrskylä	58	884.850000
4	Hailuoto	61	908.866667

10. *List the city by the total priority value of Requests in that city.*

Rationale: by listing cities based on the total priority value, the Red Cross can identify which cities have the most critical needs to pay more attention to these cities and facilitate strategic planning, such as immediate intervention.

Sql code:

```sql
SELECT
  c."name",
  SUM(r.priority_value) AS total_priority_value
FROM
  city c
  JOIN request_location rl
    ON rl.city_id = c.id
  JOIN request r
    ON r.id = rl.request_id
GROUP BY
  c.id
ORDER BY
  total_priority_value DESC;
```

Interpretation & explanations:

We take the name and sum of all priority values for requests, grouped by cities. Note that this, unlike the previous query, counts the priority multiple times if the requests are in multiple cities. This is because the priority are not concrete numeric concepts like needed

volunteers in the previous queries, so an overview given by the total priority value even with double counting are valuable.

Query results:

	name	total_priority_value
0	Rusko	485
1	Rautavaara	390
2	Myrskylä	352
3	Hailuoto	329
4	Tammela	328

11. *For each request, show a list of applications only if it is valid and time modified is before the register by date.*

Rationale: ensures that only relevant and timely applications are taken into account; filtering out invalid or outdated applications so that the system can more effectively math volunteers.

Sql code:

```sql
SELECT
  r.id AS request_id,
  va.id AS application_id
FROM
  request r
  JOIN volunteer_application va
    ON va.request_id = r.id
WHERE
  va.is_valid AND va.modified <= r.register_by_date
ORDER BY
  request_id,
  application_id;
```

Interpretation & explanations:

We joined re quest and volunteer applications on the request id, then filter out the rows that are not valid or modified after the register_by_date, which is considered too late. The ordering in this query is not mandatory, but it is quite nice to have.

Query results:

	request_id	application_id
0	1	274
1	1	344
2	1	583
3	1	652
4	1	795

12. *For each city, list the number of volunteers who are interested in certain area of interests.*

Rationale: this helps in matching volunteers to requests that align with their interests so to increase volunteer satisfaction and return to engage in more volunteer work later; this can also inform the development of initiatives to narrow down to popular areas of interest to make themselves more sustainable.

Sql code:

```sql
SELECT
  c."name",
  ia.interest_name,
  COUNT(DISTINCT v.id)
FROM
  city c
  JOIN volunteer_range vr
    ON vr.city_id = c.id
  JOIN volunteer v
    ON v.id = vr.volunteer_id
  JOIN interest_assignment ia
    ON ia.volunteer_id = v.id
GROUP BY
  c.id,
  ia.interest_name;
```

Interpretation & explanations:

We join table city with volunteer by their ranges and interest. Then using a select with a COUNT command, we can get the number of volunteers who have a specific interest grouped by city.

Query results:

	name	interest_name	count
0	Hailuoto	CollectDonations	34
1	Hailuoto	FirstAid	30
2	Hailuoto	FoodHelp	29
3	Hailuoto	GuideAndTeach	31
4	Hailuoto	HelpInCrisis	31

B. ADVANCED

a) Views

1. Create a view that lists next to each beneficiary the average number of volunteers that applied, the average age that applied, and the average number of volunteers they need across all of their requests.

Sql code:

```sql
create or replace view beneficiary_spread as
with ageCalculation as (
       select v.id as volunteer_id,
              v.birthdate,
              date_part('year', AGE(v.birthdate)) as age
       from volunteer v
),
applicationStats as (
       select r.beneficiary_id,
              count(va.volunteer_id) as number_of_volunteers_that_applied,
              AVG(ac.age) as avg_age_applied
       from request r
       left join volunteer_application va on r.id = va.request_id
       left join ageCalculation ac on va.volunteer_id = ac.volunteer_id
       group by r.beneficiary_id
),
requestStats as (
       select beneficiary_id,
              AVG(number_of_volunteers) as avg_volunteers_needed
       from request
       group by beneficiary_id
)
select b.id as beneficiary_id,
       b.name as beneficiary,
       coalesce(app_stats.number_of_volunteers_that_applied, 0) as avg_num_volunteers_applied,
       coalesce(app_stats.avg_age_applied, 0) as avg_age_applied,
       coalesce(req_stats.avg_volunteers_needed, 0) as avg_volunteers_needed
from beneficiary b
left join applicationStats app_stats on b.id = app_stats.beneficiary_id
left join requestStats req_stats on b.id = req_stats.beneficiary_id;
```

Interpretation & explanations:

The query uses 3 table expressions to calculate and store the average number of volunteers that applied, the average age of those who applied, and the average number of volunteers needed respectively. The first table ageCalculation is used to calculate the age of each volunteer. The applicationStats table is used to calculate the number of volunteers that applied and the average age of each beneficiary. RequestStats is used to calculate the number of volunteers needed for each beneficiary. It then joins them with the beneficiary table based on beneficiary_id. It uses coalesce to account for cases when there are no matching records in the tables.

Query results:

	beneficiary_id	beneficiary	avg_num_volunteers_applied	avg_age_applied	avg_volunteers_needed
0	1	Hospital	253	42.881423	20.650000
1	2	Food Bank	327	45.792049	18.976744
2	3	Elderly Care	250	43.744000	17.852941
3	4	Youth Centre	323	44.823529	17.767442
4	5	Homeless Shelter	326	44.837423	17.395349

2. *List the number of volunteers interested in each skill, broken down by city, along with the average priority value of request for those skills in the respective cities.*

Rationale: the Red Cross can identify skill gaps in different cities by understanding the distribution of volunteers' skills and the corresponding demand.

Sql code:

```sql
create or replace view volunteer_skill_details as
select
  v.id as volunteer_id,
  v.name as volunteer_name,
  v.email,
  v.city_id,
  c.name as city_name,
  string_agg(s.name, ', ') as skills,
  count(s.name) as skill_count
from
  volunteer v
left join
  skill_assignment sa on v.id = sa.volunteer_id
left join
  skill s on sa.skill_name = s.name
left join
  city c on v.city_id = c.id
group by
  v.id, v.name, v.email, v.city_id, c.name
order by
  v.name;
```

Interpretation & explanations:

The query constructs a view that lists the ID, name, email, and city_id of each volunteer along with the city name, a list of skills that the volunteer possesses and count of the number of skills. It does this by joining the volunteer and city tables, matches the values and finally orders them by volunteer name.

The view provides a comprehensive overview of each volunteer's details, their skills, and the city they are associated with. It is much easier to obtain data than the skill assignment table.

Query results:

	volunteer_id	volunteer_name	email	city_id	city_name
0	250672-987C	Aimo Vuorela	aimo.vuorela@dbcourse.cs.aalto.fi	886	Ulvila
1	020572-947E	Alexander Ahonen	alexander.ahonen@dbcourse.cs.aalto.fi	886	Ulvila
2	200304A9898	Alexander Jääskeläinen	alexander.jaaskelainen@dbcourse.cs.aalto.fi	426	Liperi
3	230605A923R	Alexander Jokela-Saastamoinen	alexander.jokela-saastamoinen@dbcourse.cs.aalto.fi	687	Rautavaara
4	160263-970F	Anita Virtanen	anita.virtanen@dbcourse.cs.aalto.fi	72	Hailuoto

b) Trigger and Functions

1. Create a check constraint for the volunteer table with a function that validates a volunteer ID when a new volunteer is inserted.

Sql code:

```sql
create or replace function finnish_id_checker(volunteer_id TEXT)
returns boolean as $$
declare birth_date INT;
        character_in_the_middle CHAR;
        individual_string TEXT;
        Control_character CHAR;
        remainder INT;
        expected_control_char CHAR;
        coontrol_chars TEXT = '0123456789ABCDEFHJKLMNPRSTUVWXY';
begin
    if length(volunteer_id) != 11 then
       return false;
    end if;

birth_date = SUBSTRING(volunteer_id from 1 for 6);
character_in_the_middle = SUBSTRING(volunteer_id from 7 for 1);
individual_string = SUBSTRING(volunteer_id FROM 8 FOR 3);
Control_character = SUBSTRING(volunteer_id FROM 11 FOR 1);
if character_in_the_middle not in ('+', '-', 'A', 'B', 'C', 'D', 'E', 'F', 'X', 'Y', 'W', 'V', 'U') then
       return false;
end if;
remainder = (birth_date::int * 1000 + individual_string::int) % 31;
expected_control_char = SUBSTRING(coontrol_chars from remainder+1 for 1);
if
  Control_character != expected_control_char then
  return false;
end if;
return true;
end;
$$ language plpgsql;

--constraint
ALTER TABLE volunteer
ADD CONSTRAINT valid_finnish_id
CHECK (finnish_id_checker(id));
```

Interpretation & explanations:

The query creates a trigger that checks whether the volunteer_id of a volunteer complies with the requirements of a valid Finnish ID. It creates a function that checks the IDs and adds that as a constraint to the volunteer table.

The function checks the validity by first declaring variable to hold intermediate results such as remainder, expected_control_character, or to declare a specific range such as control_chars.

It first checks if the ID is the correct length, and then splits it into smaller parts to check the other constraints. The function checks if the middle character is within the approved range and that whether the remainder of the birthdate in a specific formula corresponds to the correct control character.

2. Create a trigger that updates the number of volunteers for a request whenever the minimum need for any of its skill requirements is changed. The total number of volunteers needed for each request is calculated as the sum of unskilled volunteers (those without any skill requirements) and the minimum need for each required skill.

Sql code:

```sql
create or replace view unskilled_volunteers_count as
select count(*) as count
from volunteer v
where not exists (
  select 1
  from skill_assignment sa
  where sa.volunteer_id = v.id
);

create or replace function update_number_of_volunteers()
returns trigger as $$
declare
  total_skilled_volunteers int;
  total_unskilled_volunteers int;
begin
  select coalesce(sum(min_need), 0)
  into total_skilled_volunteers
  from request_skill
  where request_id = new.request_id;

  select count
  into total_unskilled_volunteers
  from unskilled_volunteers_count;

  update request
  set number_of_volunteers = total_unskilled_volunteers + total_skilled_volunteers
  where id = new.request_id;

  return new;
end;
$$ language plpgsql;
```

```sql
drop trigger if exists trg_update_number_of_volunteers on request_skill;

create trigger trg_update_number_of_volunteers
after insert or update of min_need on request_skill
for each row
execute function update_number_of_volunteers();
```

Interpretation & explanations:

The query creates a trigger that updates the number of volunteers whenever the min_need field of request_skill is changed. A view is created to count the number of volunteers that

do not have any skills assigned to them. This is done by checking if a skill_assignment record exists for each volunteer. If the record does not exist then we consider the volunteer unskilled.

The total number of skilled volunteers is calculated as the sum of the min_need of all the skills in request_skill for a new request. The total number of unskilled volunteers is obtained from the unskilled_volunteers_count view. The function then updates the number_of_volunteers field in the request table with the sum of these two values.

The trigger trg_update_number_of_volunteers is created to execute the update_number_of_volunteers function after each insert or update operation on the min_need field of the request_skill table. This ensures that the number_of_volunteers field contains accurate data is properly updated whenever a change occurs.

c) Transactions

1. Create a transaction that will read valid applications for a request.

Sql code:

```sql
CREATE OR REPLACE PROCEDURE fillRequest(req request) AS $$
DECLARE
    going BOOLEAN = 1;
    is_accepted BOOLEAN = 1;
    volunteerNum integer = 0;
    req_skill RECORD;
    vol_app RECORD;
BEGIN
FOR req_skill IN
    SELECT *
    FROM request_skill AS rs
    WHERE rs.request_id = req.id
    ORDER BY rs.value DESC
LOOP
    FOR vol_app IN
        SELECT *
        FROM volunteer_application va
        JOIN skill_assignment sa ON sa.skill_name = req_skill.skill_name
        JOIN volunteer_range vr ON vr.volunteer_id = va.volunteer_id
        JOIN request_location rl ON rl.request_id = va.request_id
        WHERE req.id=va.request_id AND va.is_valid AND sa.volunteer_id = va.volunteer_id AND rl.city_id = vr.city_id
    LOOP
        IF EXISTS (SELECT 1 FROM skill_assignment as sa WHERE sa.volunteer_id = vol_app.volunteer_id
                   AND req_skill.skill_name = sa.skill_name
                   AND NOT EXISTS (SELECT 1 FROM volunteer_assignment WHERE volunteer_id = vol_app.volunteer_id)) THEN
            INSERT INTO volunteer_assignment (request_id, volunteer_id) VALUES (req.id, vol_app.volunteer_id);
            volunteerNum := volunteerNum + 1;
        END IF;
        IF volunteerNum >= req_skill.min_need THEN
            EXIT;
        END IF;
    END LOOP;
    IF req.register_by_date < CURRENT_DATE AND volunteerNum < req_skill.min_need THEN
        ROLLBACK;
    END IF;
END LOOP;
COMMIT;
END;
$$LANGUAGE plpgsql;
--Run the transaction
DO $$
DECLARE
    req request;
BEGIN
    FOR req IN
        SELECT *
        FROM request r
    LOOP
        CALL fillrequest(req);
    END LOOP;
    COMMIT;
END;
$$ language plpgsql;
```

Interpretation & explanations:

For the procedure, first we loop through all the skills needed in the request by order of priority. Then we loop through individual volunteer applications, making sure that they have the required skill, and the requests are in volunteer range. For each application, we check if they have already been assigned to a request, if not, assign the volunteer to that request by inserting the values into the volunteer_assignment table, and add a volunteerNum to keep track of the volunteers assigned. Then if the volunteerNum is larger than the required number, we exit the loop. Outside of the applications loop, if the register deadline has not passed and the request skill has not been filled, we rollback. If the deadline has passed or the request skill has been filled, however, then we commit to the changes. The reasoning is that after the deadline has been passed, even if the quota has not been filled, we still want as many volunteers as we can get.

The entire procedure is then called in a for loop for every request.

**There are some redundancies in the code, we checked that the application have the required skill both in the loop and in the checking stage, for example. And the procedure assumes that one volunteer can only be assigned to one request even if they applied to multiple. We tried to change it so that one volunteer can do multiple requests, but runs into problems and in the end, we ran out of time. The procedure also doesn't order the application or request when processing the applications, making the entire transaction inconsistent and dependent on the ordering of the database (which should never be done in reality). **

2. Create a transaction that will remove applications that were last modified over 1 year ago. If the number of applications left is less than the number of volunteers needed, then rollback.

Rationale: Old applications have the possibility of being outdated or inactive. And leaving them in the system long term can cause inefficiency or even wrongful assignments on an outdated application. So, it is important to remove old applications from the system if the requests are not missing applications.

Sql code:

```sql
CREATE OR REPLACE PROCEDURE remove_outdated_application(req request) AS $$
DECLARE
    going BOOLEAN = 1;
    is_accepted BOOLEAN = 1;
    volunteerNum integer = 0;
    vol_app RECORD;
BEGIN
    FOR vol_app IN
        SELECT *
        FROM volunteer_application va
        WHERE req.id = va.request_id
    LOOP
        IF req.register_by_date < CURRENT_DATE - INTERVAL '1 years' THEN
            DELETE FROM volunteer_application WHERE id = vol_app.id;
        ELSE
            volunteerNum := volunteerNum + 1;
        END IF;
    END LOOP;

    IF volunteerNum < req.number_of_volunteers THEN
        ROLLBACK;
    END IF;
COMMIT;
END;
$$LANGUAGE plpgsql;

--Run the transaction
DO $$
DECLARE
    req request;
BEGIN
    FOR req IN
        SELECT *
        FROM request r
    LOOP
        CALL remove_outdated_application(req);
    END LOOP;
    COMMIT;
END;
$$ language plpgsql;
```

Interpretation & explanations:

For the procedure, we loop through all applications, check if they are over 1 year old, and delete them if it is. If it is not, we add that to the count. After the loop, we check if the amount of application not deleted and for the request is larger than the amount of volunteer required. If it is not we roll back the deletion because the request might need those application since they don't have enough.

The final block just runs the procedure on every requests.

Currently, the transaction doesn't delete any applications because there are just not enough applications for most requests to meet the number of volunteers. However, we do believe that this function is useful and necessary if the database is used long term.

**The code currently makes no distinction between valid and invalid applications, perhaps we should only count valid applications for the volunteerNum. The code also doesn't consider the skills and volunteer range of the applications, which can make the transaction delete applications without leaving enough valid applications left. These problems, however, are overlooked because of the time constraints we are working with. **

d) Analysis

Question 1

Visualize the number of volunteers available by city (according to their volunteer range, note: a volunteer can be available in more than 1 city) compared to the number of volunteers that applied for a request in that city. What's the city with the most (top 2 volunteers and the least (bottom 2)? Make sure your visualization gives a good overview of the current situation and quickly shows the most information.

The code is found in Part_B_d)_1.py

The query retrieves the count of volunteers available and those who have applied, categorized by city. The visualize_volunteers_by_city function generates a bar chart using Plotly to compare available and applied volunteers for each city. Two groups of bars are produced: one for available volunteers and another for applied volunteers. The top 2 cities with the highest number of available volunteers and the bottom 2 cities with the lowest number of available volunteers are emphasized. However, it is notable from the graph that there are two cities, Liperi and Rautavaara, both showing a count of 55. In this instance, as per the given instruction regarding "2" bottom cities, only one of these two cities with 55 available volunteers has been highlighted.

Below is the output after running python code in Part_B_d)_1.py:

```
Top 2 cities with most available volunteers:
  city_name  available_volunteers
4     Rusko                    63
0  Hailuoto                    61

Bottom 2 cities with least available volunteers:
  city_name  available_volunteers
5    Säkylä                    49
1    Liperi                    55
```

Question 2

> Create your own scoring system to calculate the matching percentage from all the attributes of a volunteer that you find relevant: e.g. interest, travel readiness, volunteer range, number of skill matches, etc. Make a compelling case for your scoring scheme and suggest a top 5 candidates for each request according to this system. Does it match the candidates you have found in past questions?

The code is found in Part_B_d)_2.py

the case is demonstrated along with the scoring system explanations; not written separately. The case might not be "compelling" because we simply chose the case from request_id '1' and the first volunteer_id who chose request_id '1'.

Below are the listed attributes in the provided data excel/the database for designing the scoring system purpose:

City: name, id, geolocation

Volunteer: id, birthdate, city_id, name, email, address, travel_readiness

Volunteer Range: volunteer_id, city_id

Skill: name, description

Skill Assignment: volunteer_id, skill_name

Interest: name, description

Interest Assignment: volunteer_id, interest_name

Beneficiary: id, name, address, city_id

Request:
id, title, beneficiary_id, number_of_volunteers, priority_value, start_date, end_date, register_by_date

Request Skill: request_id, skill_name, min_need, value

Request Location: request_id, city_id

Volunteer Application: id, request_id, volunteer_id, modified, is_valid

First, in the volunteer_application sheet, we must exclude volunteers who have not submitted any applications and whose volunteer_application is no longer valid in order to calculate the legit score.

This scoring system aims to prioritize volunteers according to their appropriateness for particular requests. We take into account the following attributes (might not be all attributes that are relevant):

1.Number of skill matches (40%),

2.Interest match (20%),

3. The priority of the request (10%),

4. Travel readiness (20%),

5. Volunteer range (10%),

In total 100%

We calculate a total score by adding together weighted 5 different components. This approach to scoring resembles that of various other college course grading methods.

1. **Number of skill matches** is counted for 40% of the overall score. We calculate this score by using this equation:

(the weighted sum of the matching skills between each volunteer_id and each request / the total request skills for each request) x 40

Rationale: first, we need to filter the volunteer_application to exclude those where is_valid is False; then sort and list each 'request_id' correlated with the respective 'volunteer_id'; then **for example**, pick one volunteer_id for request_id '1', and list all skill_name of the 'request_id' 1 from request_skill -> list the volunteer_id correlated skill_name from skill_assignment to see how much these two main components match -> calculate the score for what is mentioned above: because there is only one match the score is 1 out of skills needed for 'request_id' 1, which is ¼.

Example: we picked this volunteer_id as the example: 160903A941P

We get:

Skills needed for request_id 1: EventHosting, CommunicationAndMarketing, PhotographyAndVideo, EventOrganization;

Skills of the selected volunteer: Rescue, PublicPerformances, CookingAndBaking, HealthCareOrFirstAid, Organizational, PhotographyAndVideo;

Match count for request_id 1: the volunteer has the skill "PhotographyAndVideo"

Because we get only one match -> 1/4, the score of number of skill matches for this volunteer is ¼ x 40 = 10

The python code produces the whole list with the match scores for each volunteer and request pair.

2. **Interest match** is counted for 20% of the overall score. We calculate this score by using this logic: because each 'id' in the 'request' sheet only has one 'title' which will or will not correspond with the 'volunteer_id''s 'interest_assginment'. The score should be either 0 or 20 -> match or not match.

Rationale: select the 'title' for each of the request_id from the 'request' sheet; choose a specific volunteer_id and list all interest_name that belongs to that volunteer_id from the interest_assignment sheet; compare 'title' and each of the 'interest_name' to see if they match; finally get the match score.

Example (continued):

Let's find this volunteer_id as the example: 160903A941P again from 'interest_assignment':

WorkWithElderly	160903A941P
FirstAid	160903A941P
GuideAndTeach	160903A941P
WorkWithYoung	160903A941P
PromoteWellbeing	160903A941P
WorkInMulticulturalEnvironment	160903A941P
CollectDonations	160903A941P
ImmigrantSupport	160903A941P
OrganiseActivities	160903A941P

id	title	beneficiary_id	number_of_volunteers	priority_value	start_date	end_date	register_by_date
1	work in team needed	3	14	1	2024-07-25 22:15:00	2024-07-28 18:00:00	2024-07-18 01:00:00

We notice that this volunteer is interested in many aspects from the 'interest' sheet. However, the 'title' for request 1 is 'work in team needed', so nothing matched between 'interest_assignment' and 'title' from this volunteer_id -> the score of 'interest match' for this volunteer is therefore (0/1) x 20 = 0.

**the issue we encountered with this interest match is that 'title' are written as work in team needed, work with young needed, but 'interest_name' are written as WorkInATeam, WorkWithYoung, due to time constrains we do not have too much time to rename and clean the data to let two similar names match exactly. Therefore, for this part, we have noticed that the matching score are all 0 even though for example 'WorkInATeam' should match 'work in team needed'. ** → this issue is then fixed by creating a mapping between the 'title' in the 'request' sheet and the 'interest_name' in the 'interest_assignment' sheet. This mapping is used to compare and determine if there is a match, setting the score to 20 if there is a match.

3. **The priority of the request** is counted for 10% of the overall score. It is directly proportional to the request's priority value (0 to 5). We calculate this score by using this equation: *(priority value / 5) x 10*

Rationale: find each of the priority_value of each id from 'request' sheet, then calculate *(priority value / 5) x 10*. The python code lists the score of priority of the request for each volunteer and request pair.

Example (continued):

id	title	beneficiary_id	number_of_volunteers	priority_value	start_date	end_date	register_by_date
1	work in team needed	3	14	1	2024-07-25 22:15:00	2024-07-28 18:00:00	2024-07-18 01:00:00

We need to find the priority_value of id '1' from 'request' sheet, which is '1', for the same volunteer_id 160903A941P, then calculate (1/5) x 10 which is 2.

4. **Travel readiness** is counted for 20% of the overall score. The volunteer gets full points if the volunteer's city ID matches either of the request locations.

Travel readiness defines in this case how long it will take for the volunteer to get ready to travel [3].→ the larger the number the lower the score.

We use the following formula: 20 – (travel_readiness / max_travel_readiness) x 20

We define the max_travel_readiness (minutes) = 1500 because all numbers under the 'travel_readiness' column in the 'volunteer' sheet are <= 1500.

'(travel_readiness / max_travel_readiness)' normalizes the volunteer's travel readiness time to a value between 0 and 1. If 'travel_readiness' is equal to 'max_travel_readiness', this term will be 1. If 'travel_readiness' is less than 'max_travel_tradiness', this term will be a fraction between 0 and 1.

Rationale: first, identify volunteer information: find each of the volunteer_id from the 'volunteer' sheet; retrieve the 'city_id' and 'travel_readiness'. Find the 'request_id' from the 'volunteer_application' sheet; retrieve the ' beneficiary_id' for the given 'request_id' from the 'request' sheet. Find the 'city_id' corresponding to the 'beneficiary_id' from the 'beneficiary' sheet. Compare the volunteer's 'city_id' with the beneficiary's 'city_id'. Finally, calculate the travel readiness score.

Example (continued): same volunteer_id: 160903A941P

id	title	beneficiary_id	number_of_volunteers	priority_value	start_date	end_date	register_by_date
1	work in team needed	3	14	1	2024-07-25 22:15:00	2024-07-28 18:00:00	2024-07-18 01:00:00

id	birthdate	city_id	name	email	address	travel_readiness
160903A941P	2003-09-16 00:00:00	072	Johannes Ja	johannes.ja	Fahlanderia	1056

→ this row shows both city_id and travel_readiness for this specific volunteer.

First, find the 'id' from 'volunteer' sheet, then check the 'city_id' of the same 'id' which is '072', then go to 'request' sheet to find this 'id' '1', then 'beneficiary_id' which is '3'. Then go to the 'beneficiary' sheet to find 'id' '3', find its corresponding 'city_id' which is '704', and '704' does not match '072'. We need more steps for calculating 'travel readiness' score we go to 'volunteer' sheet to find the volunteer 'id', then we see 'travel_readiness' column which is '1056' minutes, then use the formula presented above and we get the score for this volunteer is: 20 – (1056/1500) x 20 = 5.92 (this score is also shown in the generated Excel after running python code).

5. **Volunteer range** is counted for 10% of the overall score.

Rationale and **example** explanations (continued and combined):

In the 'geolocation' in the 'city' sheet, we need to calculate the "volunteer range" score, under the 'geolocation' column, the value is written as '13.4134995/45.792650'. 'geolocation' has its own corresponding 'id' then we need to find the 'id' from the 'volunteer' sheet, which means we locate the specific volunteer, for example, the same volunteer mentioned before: 160903A941P → this 'volunteer_id' 'city_id' is '072'. We need to go to 'id' '1' from the 'request' sheet to find the 'beneficiary_id', which is '3', then we go to 'beneficiary' sheet to check 'id' '3' 'city_id', which is '704' → noticed '072' does not match '704'. Therefore, we need to go a few more steps for this calculation: go back to the 'city' sheet to find 'id' '072' 'geolocation' which is '-79.132557/-40.995129' → find 'id' '704' 'geolocation' which is '13.4134995/45.792650'. The main idea is, the further between these 2 geolocations, the lower the score.

name	id	geolocation
Rusko	704	13.4134995/45.792650
Hailuoto	072	-79.132557/-40.995129
Liperi	426	47.244262/80.880444
Ulvila	886	18.6986795/-17.160223
Tammela	834	37.935520/12.226293
Rautavaara	687	66.6004235/-62.724453
Myrskylä	504	45.475847/-105.227997
Säkylä	783	-14.344729/-104.974555

In this specific example, one geolocation is '13.4134995/45.792650', the other geolocation is '-79.132557/-40.995129'. → this format means 'latitude/longtitude'. We need to split this string to get the latitude and longitude values in the python code. We decide to use the *Haversine* [2] formula to calculate the distance between two geolocations for this volunteer_range score, again, closer distances yield higher scores.

The maximum distance should be set to cover the farthest possible points on Earth, and the Earth's circumference is around 40,000 km → we define a maximum distance for scaling for this part of the scoring system = 20000 (halfway around the globe)

The Haversine formula to calculate the great-circle distance between two points on the Earth's surface is:

$$d = 2r \arcsin\left(\sqrt{\sin^2\left(\frac{\Delta\phi}{2}\right) + \cos(\phi_1)\cos(\phi_2)\sin^2\left(\frac{\Delta\lambda}{2}\right)}\right)$$

Where:

- ϕ_1 and ϕ_2 are the latitudes of the two points (in radians)
- λ_1 and λ_2 are the longitudes of the two points (in radians)
- $\Delta\phi = \phi_2 - \phi_1$
- $\Delta\lambda = \lambda_2 - \lambda_1$
- r is the Earth's radius (mean radius = 6,371 km)

(above are typed in MacTex)

In this example, the volunteer range score for this volunteer '160903A941P' is around: 4,298

The total score regarding volunteer_id '160903A941P' = skill_score + interest_score + priority_score + travel_score + range_score = 22,218

request_id	request_title	volunteer_id	unteer_city	eficiary_cit	vel_readin	travel_score	lunteer_sk	kills_neede	ill_matchin	_match_co	skill_score	nteer_inte	pped_inter	terest_scor	riority_valu	riority_scor	range_score	total_score
1	work in team needed	160903A941P	72	704	1056	5,92	['Rescue', 'P	['EventHost	['Photograp	1	10	['WorkWitl	WorkInATe	0	1	2	4,297668	22,21767

The total score of this volunteer for request '1' is not really high compared to other volunteer candidates who have applied for request '1'.

The complete generated scoring system Excel link can be found here:

https://aaltofi-my.sharepoint.com/:x:/g/personal/meishan_lin_aalto_fi/ERfMCQpwtuhIhpPzhwZXkMkBQGXOzwS6RSEB6iVDV19l5g?e=2mVNUm

Suggest the top 5 candidates for each request according to this system.

After modifying python code, the generated Excel for suggesting the top 5 candidates for each request is below:

https://aaltofi-my.sharepoint.com/:x:/g/personal/meishan_lin_aalto_fi/Efps4F4Wd71KlnPvq_phjIABD3CMkbNarpFJZoP1r6dl1w?e=wrORMg

Does it match the candidates you have found in past questions?

It is clear after comparing the output of question B part c1 to the output of this scoring that there is absolutely no match, any matching that occurred is purely incidental. This can be attributed to the randomness in our solution in the volunteer assignment transaction. In that transaction, the order in which the applications are processed (which largely determines which ones are picked) are not ordered in any way, making the algorithm dependent on the order of the database.

Question 3

For each month, what are the number of valid volunteer applications compared to the number of valid requests?

The code is found in Part_B_d)_3.py

Each month, the number of volunteer applications varies greatly, with peaks towards the end of the timeline (2024-2025) as indicated by the blue line in the first graph. In contrast, the number of requests remains relatively steady with minor fluctuations, shown by the orange line. However, there is also a spike trend between June and September 2024 when comparing requests itself over time.

What months have the most and least for each?

Most Valid Applications: September 2024: around 160 valid applications. Least Valid Applications: November 2020: around 7 valid applications. Most Requests: September 2024: around 22 requests. One of the instances with the fewest requests occurred in December 2019, with only one request. There are also additional months where the number of orange dots is really close to zero.

How about the difference between the requests and volunteers for each month?

The blue bars in the second chart show consistently more applications than requests, with the largest negative difference occurring around September 2024. Largest Negative Difference: September 2024: Difference of -129. Smallest Negative Difference: September 2022: Difference of -3.

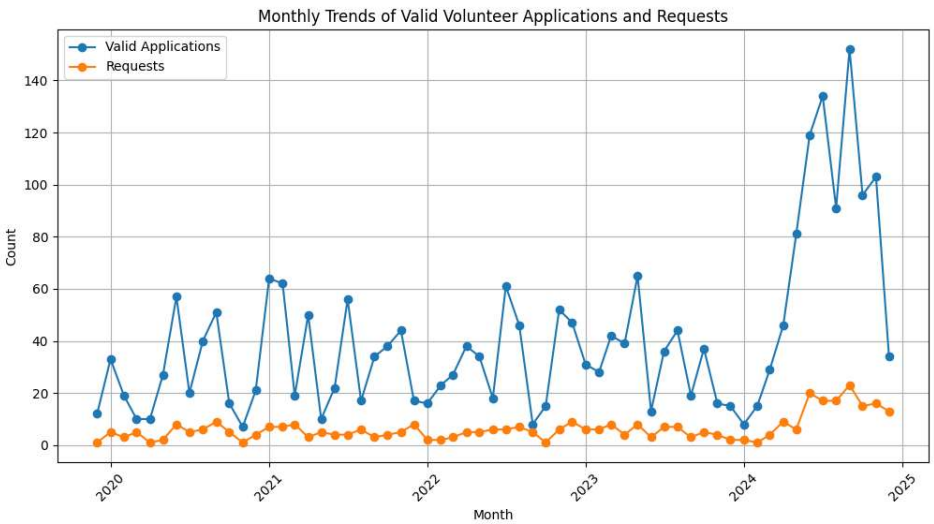
Is there a general/seasonal trend?

The number of valid volunteer applications generally increases over time, with significant peaks in 2024-2025. There does not appear to be a consistent seasonal trend for requests.

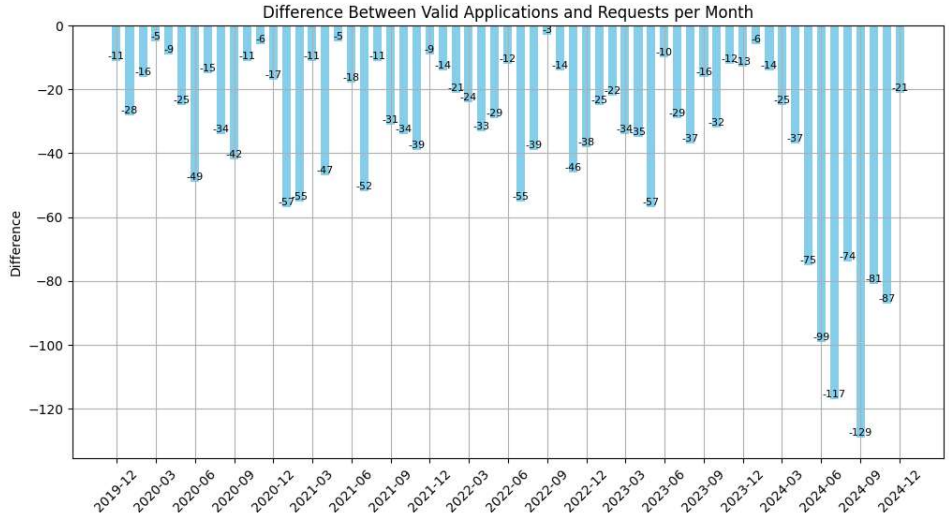
Is there any correlation between the time of the year and number of requests and volunteers?

Graph 3 displays a positive correlation, and the correlation matrix reveals a strong positive relationship with a coefficient of 0.858 between valid applications and requests. There is also an almost perfect negative correlation (-0.996) shown in the matrix between valid applications and the difference, suggesting that as the number of valid applications goes up, the difference (requests - valid applications) tends to decrease dramatically.

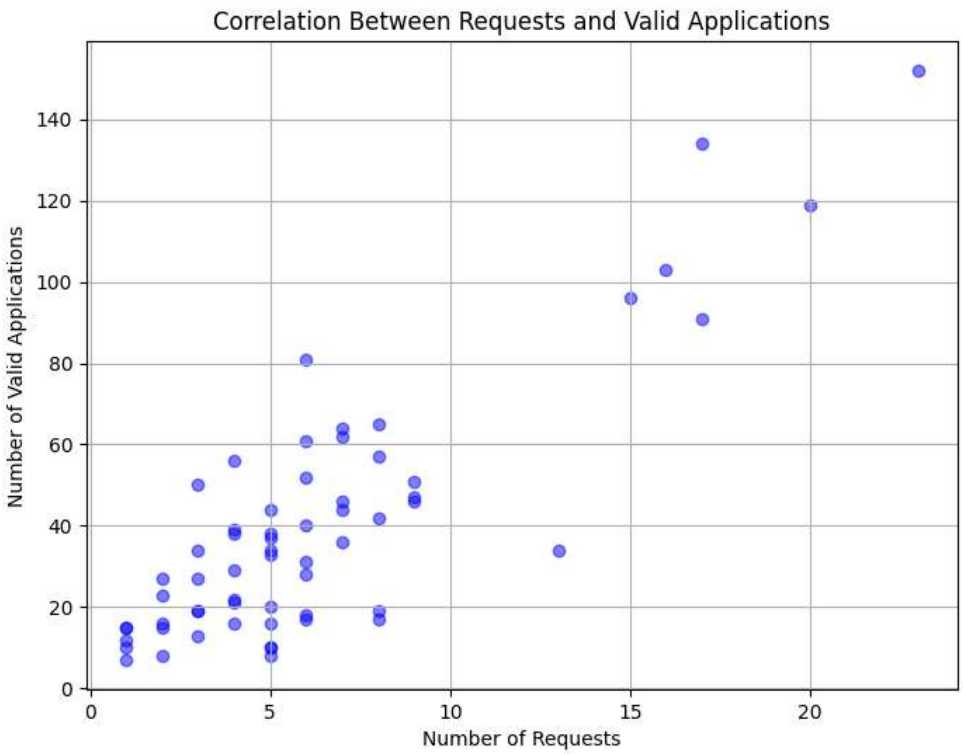
```
Correlation Matrix:
                    valid_applications  ...  difference
valid_applications            1.000000  ...   -0.996147
requests                      0.858110  ...   -0.809771
difference                   -0.996147  ...    1.000000
```



There is a noticeable increase in valid applications over time, with significant peaks around 2024-2025. Requests remain relatively stable with slight fluctuations.



Most months show a negative difference, indicating that the number of valid applications consistently exceeds the number of requests. The largest negative differences occur in the latter part of the timeline, around 2024-2025.



There is a positive correlation between the number of requests and valid applications, indicating that as requests increase, the number of valid applications also tends to increase.

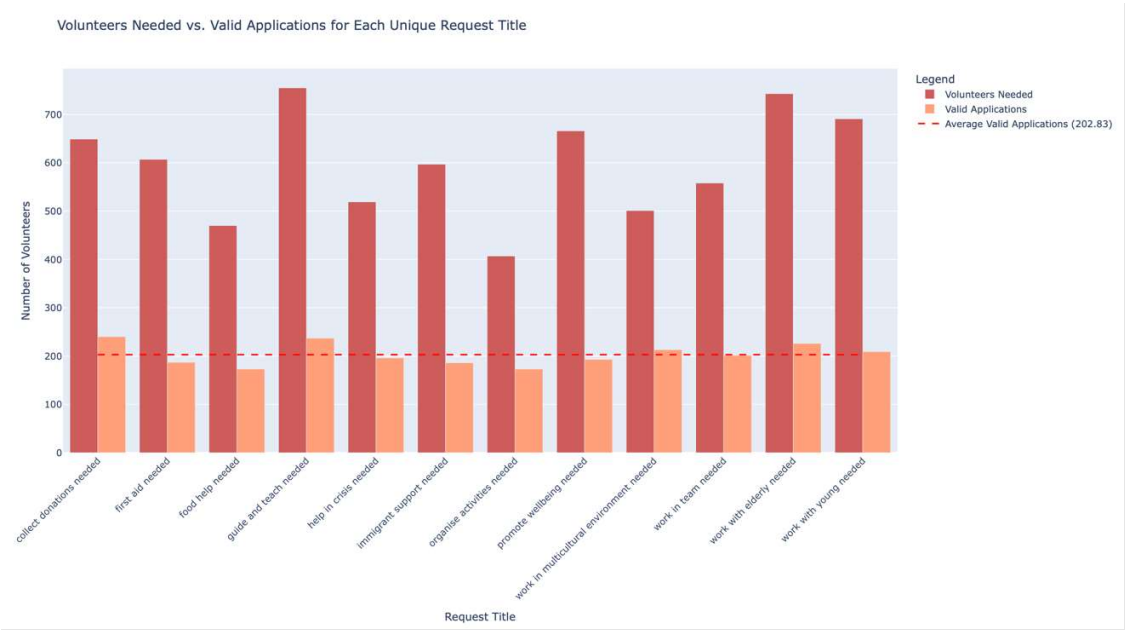
Question 4

> [free choice] Choose a question/topic/problem/aspect you would like to analyze and visualize.

The code is found in Part_B_d)_4.py

For this question, we created two visualizations: request fulfillment rate analysis and volunteer retention analysis shown below.

Part 1: Request Fulfillment Rate Analysis

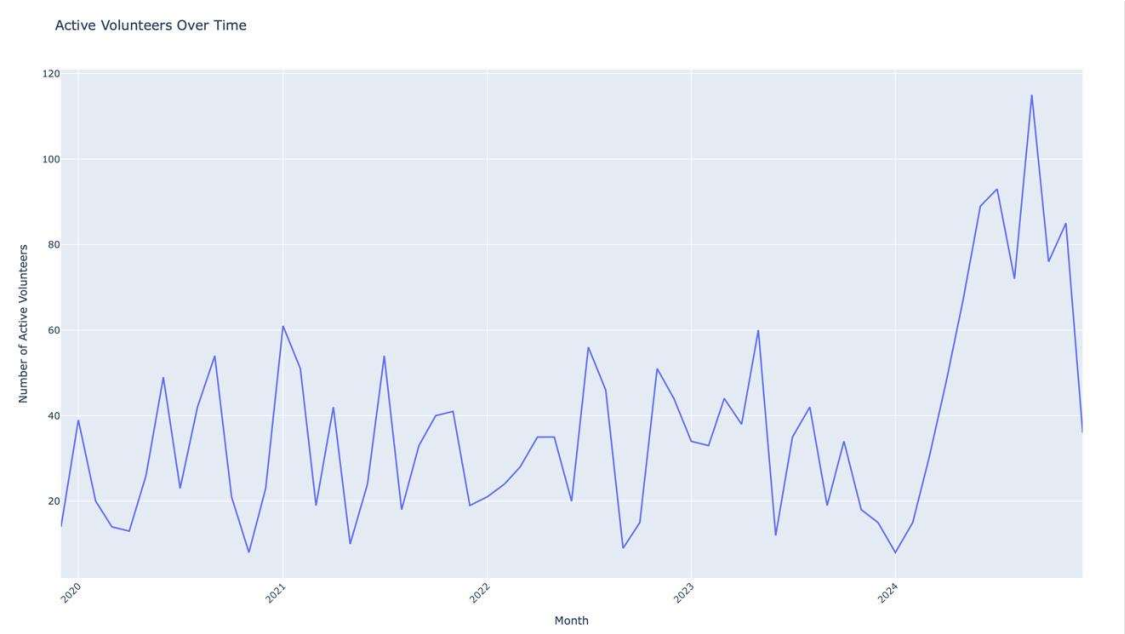


The bar graph displays the quantity of volunteers required (represented by red bars) as compared to the number of acceptable applications (depicted by orange bars) for each specific request title. An average line in red dashed format is placed horizontally, denoting the mean number of valid applications for all types of requests. This feature facilitates a swift determination of whether a particular request falls below or exceeds the standard application rate.

Part 1 rationale: The analysis of request fulfillment rates provides insight into the extent to which volunteer applications align with the need for volunteers across different types of

requests. Evaluating both the required number of volunteers for each request and the quantity of valid applications received enables Red Cross to assess whether there is adequate volunteer capacity to meet all requests or if shortages exist in particular areas.

Part 2: Volunteer Retention Analysis



Part 2 rationale: the analysis of volunteer retention involves monitoring the fluctuation in the number of ongoing volunteers to assist Red Cross in recognizing patterns related to volunteer retention. Instances with noticeable declines in active volunteers can be examined to identify potential causes, such as modifications in volunteer management procedures, external occurrences, or other factors influencing volunteer involvement. This chart may appear similar to the one titled "Monthly Trends of Valid Volunteer Applications and Requests," although this graph is intended to emphasize "retention" and has been generated using Plotly for dynamic visualization. In contrast, "Monthly Trends of Valid Volunteer Applications and Requests" was produced using Matplotlib, which provides a static representation.

6.3 PRESENTATION SLIDES LINK

https://aaltofi-my.sharepoint.com/:p:/g/personal/meishan_lin_aalto_fi/EcnrM7KxosdLu8xOBgtygAgBIokMPypdrdHCgBCJ4Rmolw?e=WtDNus

6.4 GROUP WORK TIMELINE AND REFLECTIONS

Reference

[1] N. Sawhney, A. Chahla, D. To, F. Afrooz, K. Modrzyński, M. Ha Le, and R. Innanen, *"Designing a Volunteer Matching System (VMS) with the Finnish Red Cross (FRC),"* CS-A1155: Databases for Data Science 2024, Aalto University, Computer Science Department, 2024, pp. 1-8.

[2] M. K. Paneri, "Haversine formula: Find distance between two points," Medium, https://medium.com/@manishkp220/haversine-formula-find-distance-between-two-points-2561d66c2d79.

[3] Slack (no date). Available at: https://aalto-databases-2024.slack.com/archives/C06Q9DQTE2Z/p1717866106858669?thread_ts=1717863085.772119