# TASK 1

Sophie Fidan
21068639

UFCFU3-15-3
Advanced

# Databases
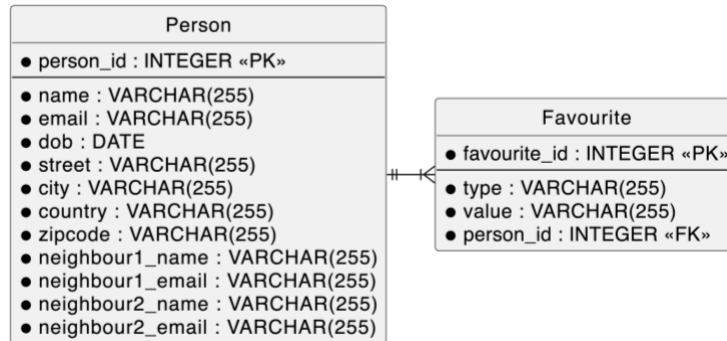
# TABLE OF CONTENTS

# FIRST NORMAL FORM (1NF)

In the First Normal Form, each column should contain an atomic value, and each record should be unique. Additionally, each row should have a unique identifier (primary key).



## PERSON

The *Person* table is created with a primary key of *person_id*. This *person_id* is automatically incremented for each new entry, providing a unique identifier for each row.

## FAVOURITE

The *Favourite* table consists of the user's favourite information. This table is created with attributes of *favoruite_id* as the primary key, *type* and *value*. The type is the type of the favourite, which in this case, is either drink, activity or book. The value is the value of this type, which is the *favourite_book*, *favoruite_drink*, and *favourite_activity*.

There are many-to-one relationships with the Person table, meaning multiple favourites can belong to one person. This relationship is represented with *person_id* as a foreign key to the *Person* table in the diagram.

**Design Rationale**

The Favourite attributes () could have been placed in a *Favourite* table with separate columns for each type of favourite that can be null. However, creating a *Favourite* table with *type* and *value* columns reduces the database design cost and enhances the scalability of the database design for the following reasons:

- This type-value structure uniforms the data model. Therefore, the system can accommodate new types of favourites without altering the table schema, providing more dynamic expansion. For instance, adding a new favourite type called favourite destination only requires a new row in the table.

1

- This design simplifies queries to retrieve favourite information. Instead of querying multiple columns for different favourite types, the system can query a single table and filter by type.

If there is any additional information about the book, activity, or drink (such as the author of the book, or the drink's flavour), then the creation of separate tables for each type would be necessary. However, the current use case did not provide such information, making the type-value structure more suitable.

# SECOND NORMAL FORM (2NF)

In the Second Normal Form, all partial dependencies should be removed so that every non-key attribute should depend on the primary key. For this purpose, separate tables are created as *Person*, *Address*, , and *NeighboursPair*. *Favourite* Table from 1NF stayed as it is since it already satisfies 2NF.
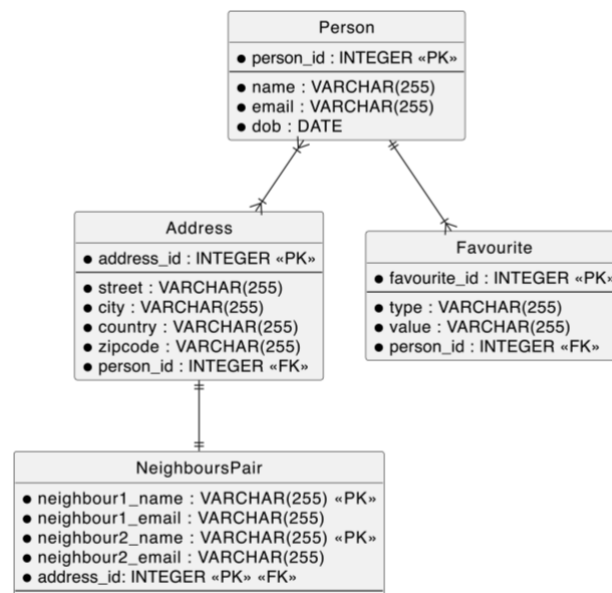


## PERSON

The *Person* table only contains information about the person such as *name*, *email* and *dob*(date of birth) with a primary key of *person_id*.

## ADDRESS

The address attributes (*street*, *city*, *country* and *zipcode*) were separated into a new table called *Address* with a primary key of *address_id*.

The *Address* table has many-to-many relationships to the *Person* table with a foreign key of *person_id*, meaning multiple addresses can belong to one person, and two different people

2

can share the same address. Even though the use case did not include any example of two people sharing the same address, this design enhances the flexibility of the database.

## NEIGHBOURSPAIR

The *NeighboursPair* table contains the neighbour information for each address, including their names and emails as *neighbour1_name*, *neighbour2_name*, *neighbour1_email*, and *neighbour2_email*.

This table has a one-to-one relationship with the *Address* table, which means every address can exactly have one neighbour pair (two neighbours) associated with it. The relationship is indicated with *address_id* as a primary key of the *NeighboursPair* table as well as a foreign key to the *Address* table.

## FAVOURITE

By fulfilling the 1NF, the *Favourite* table already satisfies the requirements of 2NF.

# THIRD NORMAL FORM (3NF)

In the Third Normal Form, all the attributes must be functionally dependent on the primary key, and there should be no transitive dependency, which means non-key attributes should not be dependent on other non-key attributes. Therefore, the associated entities that have many-to-many relationships should be further normalised.

## PERSON

By fulfilling the 2NF, the *Person* table already satisfies the requirements of 3NF. Below is the SQL statement for creating the table:

```
CREATE TABLE
    IF NOT EXISTS Person (
        person_id INT AUTO_INCREMENT PRIMARY KEY,
        name VARCHAR(255) NOT NULL,
        email VARCHAR(255) NOT NULL,
        dob DATE NOT NULL
    );
```

The insertion of the data is achieved using Python. Below is the function that performs this operation:

```
def insert_person(row):
    cursor.execute("""
        INSERT INTO Person (name, email, dob)
        VALUES (%s, %s, %s)
```

3

```
    """, (row['Name'], row['Email'], row['DOB']))
    return cursor.lastrowid
```
The table contains 20 entries from the use case:

| * person_id<br>int | * name<br>varchar(255) | * email<br>varchar(255) | * dob<br>date |
|---|---|---|---|
| 1 | Person 1 | person1@email.com | 1995-03-15 |
| 2 | Person 2 | person2@email.com | 1993-06-22 |
| 3 | Person 3 | person3@email.com | 1991-09-10 |
| 4 | Person 4 | person4@email.com | 1998-12-05 |
| 5 | Person 5 | person5@email.com | 1983-11-30 |
| 6 | Person 6 | person6@email.com | 1989-07-18 |
| 7 | Person 7 | person7@email.com | 1996-04-25 |
| 8 | Person 8 | person8@email.com | 1990-01-09 |
| 9 | Person 9 | person9@email.com | 1993-08-17 |
| 10 | Person 10 | person10@email.com | 1997-10-22 |
| 11 | Person 11 | person11@email.com | 1992-05-13 |
| 12 | Person 12 | person12@email.com | 1986-02-27 |
| 13 | Person 13 | person13@email.com | 1991-11-25 |
| 14 | Person 14 | person14@email.com | 1987-02-01 |
| 15 | Person 15 | person15@email.com | 1984-08-12 |
| 16 | Person 16 | person16@email.com | 1990-03-09 |
| 17 | Person 17 | person17@email.com | 1995-11-17 |
| 18 | Person 18 | person18@email.com | 1994-06-20 |
| 19 | Person 19 | person19@email.com | 1992-12-11 |
| 20 | Person 20 | person20@email.com | 1988-09-25 |

## PERSONADDRESS

The *PersonAddress* serves as a pivotal junction table that resolves the many-to-many relationship by establishing a many-to-one relationship with the *Person* table and a many-to-one relationship with the *Address* table. This means that multiple entries in the *PersonAddress* table can reference the same person and multiple entries in the *PersonAddress* can reference the same address.

**Composite Primary Key:** The *PersonAddress* table uses a composite primary key, which is a combination of *person_id* and *address_id*. This composite key uniquely identifies each entry in the *PersonAddress* table, ensuring that each combination of person and address is unique. This design eliminates redundancy by preventing duplicate entries for the same person-address pair.

In this refined schema, each person can be associated with multiple addresses, and each address can be linked to multiple persons. This fulfils the requirements of 3NF, ensuring

4

there are no transitive dependencies, thereby maintaining the data integrity and reduce data redundancy.

Below is the SQL statement for creating the table:

```sql
CREATE TABLE IF NOT EXISTS PersonAddress (
    person_id INTEGER NOT NULL,
    address_id INTEGER NOT NULL,
    FOREIGN KEY (person_id) REFERENCES Person(person_id),
    FOREIGN KEY (address_id) REFERENCES Address(address_id),
    PRIMARY KEY (person_id, address_id)
);
```

The insertion of the data is achieved using Python. Below is the function that performs this operation:

```python
def insert_person_address(person_id, address_id):
    cursor.execute("""
        INSERT INTO PersonAddress (person_id, address_id)
        VALUES (%s, %s)
    """, (person_id, address_id))
```

The table contains 20 entries from the use case:

| * person_id int | * address_id int |
| --- | --- |
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
| 4 | 4 |
| 5 | 5 |
| 6 | 6 |
| 7 | 7 |
| 8 | 8 |
| 9 | 9 |
| 10 | 10 |
| 11 | 11 |
| 12 | 12 |
| 13 | 13 |
| 14 | 14 |
| 15 | 15 |
| 16 | 16 |
| 17 | 17 |
| 18 | 18 |
| 19 | 19 |
| 20 | 20 |

5

# ADDRESS

Since the *PersonAddress* table is used to manage the many-to-many relationship between *Person* and *Address*, the foreign key *person_id* in the *Address* table is redundant and can be removed. This is because the *PersonAddress* table already captures the relationship between persons and addresses. Below is the SQL statement for creating the table:

```sql
CREATE TABLE
  IF NOT EXISTS Address (
    address_id INT AUTO_INCREMENT PRIMARY KEY,
    street VARCHAR(255) NOT NULL,
    city VARCHAR(255) NOT NULL,
    country VARCHAR(255) NOT NULL,
    zipcode VARCHAR(20) NOT NULL
  );
```

The insertion of the data is achieved using Python. Below is the function that performs this operation:

```python
def insert_address(row):
  cursor.execute("""
    INSERT INTO Address (street, city, country, zipcode)
    VALUES (%s, %s, %s, %s)
  """, (row['Street'], row['City'], row['Country'], row['Zip Code']))
  return cursor.lastrowid
```

The table contains 20 entries from the use case:

| * address_id<br>int | * street<br>varchar(255) | * city<br>varchar(255) | * country<br>varchar(255) | * zipcode<br>varchar(20) |
| --- | --- | --- | --- | --- |
| 1 | 12 Maple St | London | England | E1 6AN |
| 2 | 45 Oak Ave | Manchester | England | M1 2WD |
| 3 | 89 Pine Rd | Birmingham | England | B1 1AB |
| 4 | 23 Birch St | Edinburgh | Scotland | EH1 1YZ |
| 5 | 67 Cedar Ln | Bristol | England | BS1 3XE |
| 6 | 56 Elm St | Liverpool | England | L1 1AA |
| 7 | 12 Maple St | Glasgow | Scotland | G1 2TF |
| 8 | 89 Oak Dr | Leeds | England | LS1 3AB |
| 9 | 123 Pine Rd | Newcastle | England | NE1 2AB |
| 10 | 15 Elm St | Cardiff | Wales | CF10 3AF |
| 11 | 78 Oak Ln | Sheffield | England | S1 4GT |
| 12 | 56 Birch Rd | Nottingham | England | NG1 2PB |
| 13 | 10 Holy St | Cardiff | Wales | CF10 2NF |
| 14 | 34 Willow Rd | Edinburgh | Scotland | EH1 1AB |
| 15 | 78 Cedar Ave | Cambridge | England | CB1 2SE |
| 16 | 45 Maple Rd | Oxford | England | OX2 6TP |
| 17 | 23 Birch Ave | Southampton | England | SO14 3HL |
| 18 | 12 Elm Blvd | Leicester | England | LE1 3PL |
| 19 | 56 Oak Rd | Norwich | England | NR1 4BE |
| 20 | 89 Pine Ave | Cardiff | Wales | CF10 3BC |

6

# NEIGHBOURSPAIR

By fulfilling the 2NF, the *NeighboursPair* table already satisfies the requirements of 3NF.
Below is the SQL statement for creating the table:

```sql
CREATE TABLE
  IF NOT EXISTS NeighboursPair (
    neighbour1_name VARCHAR(255) NOT NULL,
    neighbour1_email VARCHAR(255) NOT NULL,
    neighbour2_name VARCHAR(255) NOT NULL,
    neighbour2_email VARCHAR(255) NOT NULL,
    address_id INTEGER NOT NULL,
    FOREIGN KEY (address_id) REFERENCES Address (address_id),
    PRIMARY KEY (neighbour1_name, neighbour2_name, address_id)
  );
```

The insertion of the data is achieved using Python. Below is the function that performs this operation:

```python
def insert_neighbours_pair(row, address_id):
  cursor.execute("""
    INSERT INTO NeighboursPair (neighbour1_name, neighbour1_email, neighbour2_name, neighbour2_email, address_id)
    VALUES (%s, %s, %s, %s, %s)
  """, (row['Neighbour 1 Name'], row['Neighbour 1 Email'], row['Neighbour 2 Name'], row['Neighbour 2 Email'], address_id))
```

The table contains 20 entries from the use case:

| * neighbour1_name varchar(255) | * neighbour1_email varchar(255) | * neighbour2_name varchar(255) | * neighbour2_email varchar(255) | * address_id int |
|---|---|---|---|---|
| Neighbour A | neighbourA@email.com | Neighbour B | neighbourB@email.com | 1 |
| Neighbour AA | neighbourAA@email.com | Neighbour AB | neighbourAB@email.com | 14 |
| Neighbour AC | neighbourAC@email.com | Neighbour AD | neighbourAD@email.com | 15 |
| Neighbour AE | neighbourAE@email.com | Neighbour AF | neighbourAF@email.com | 16 |
| Neighbour AG | neighbourAG@email.com | Neighbour AH | neighbourAH@email.com | 17 |
| Neighbour AI | neighbourAI@email.com | Neighbour AJ | neighbourAJ@email.com | 18 |
| Neighbour AK | neighbourAK@email.com | Neighbour AL | neighbourAL@email.com | 19 |
| Neighbour AM | neighbourAM@email.com | Neighbour AN | neighbourAN@email.com | 20 |
| Neighbour C | neighbourC@email.com | Neighbour D | neighbourD@email.com | 2 |
| Neighbour E | neighbourE@email.com | Neighbour F | neighbourF@email.com | 3 |
| Neighbour G | neighbourG@email.com | Neighbour H | neighbourH@email.com | 4 |
| Neighbour I | neighbourI@email.com | Neighbour J | neighbourJ@email.com | 5 |
| Neighbour K | neighbourK@email.com | Neighbour L | neighbourL@email.com | 6 |
| Neighbour M | neighbourM@email.com | Neighbour N | neighbourN@email.com | 7 |
| Neighbour O | neighbourO@email.com | Neighbour P | neighbourP@email.com | 8 |
| Neighbour Q | neighbourQ@email.com | Neighbour R | neighbourR@email.com | 9 |
| Neighbour S | neighbourS@email.com | Neighbour T | neighbourT@email.com | 10 |
| Neighbour U | neighbourU@email.com | Neighbour V | neighbourV@email.com | 11 |
| Neighbour W | neighbourW@email.com | Neighbour X | neighbourX@email.com | 12 |
| Neighbour Y | neighbourY@email.com | Neighbour Z | neighbourZ@email.com | 13 |

## FAVOURITE

By fulfilling the 1NF and 2NF, the *Favourite* table already satisfies the requirements of 3NF. Below is the SQL statement for creating the table:

```sql
CREATE TABLE
    IF NOT EXISTS Favourite (
        favourite_id INT AUTO_INCREMENT PRIMARY KEY,
        type VARCHAR(255) NOT NULL,
        value VARCHAR(255) NOT NULL,
        person_id INT,
        FOREIGN KEY (person_id) REFERENCES Person (person_id)
    );
```

The insertion of the data is achieved using Python. Below is the function that performs this operation:

```python
def insert_favourite(row, person_id):
    cursor.execute("""
        INSERT INTO Favourite (type, value, person_id)
        VALUES ('Book', %s, %s)
    """, (row['Favourite Book'], person_id))
    cursor.execute("""
        INSERT INTO Favourite (type, value, person_id)
        VALUES ('Drink', %s, %s)
    """, (row['Favourite Drink'], person_id))
    cursor.execute("""
        INSERT INTO Favourite (type, value, person_id)
        VALUES ('Activity', %s, %s)
    """, (row['Favourite Activity'], person_id))
```

The table contains 60 entries from the use case. Below is an example of some records:

| favourite_id int | type varchar(255) | value varchar(255) | person_id int |
|---|---|---|---|
| 1 | Book | A New Beginning | 1 |
| 2 | Drink | Lemonade | 1 |
| 3 | Activity | Outdoor Running | 1 |
| 4 | Book | The Road to Success | 2 |
| 5 | Drink | Coffee | 2 |
| 6 | Activity | Hiking | 2 |
| 7 | Book | Endless Possibilities | 3 |
| 8 | Drink | Smoothie | 3 |
| 9 | Activity | Swimming | 3 |
| 10 | Book | Journey of Life | 4 |
| 11 | Drink | Iced Tea | 4 |
| 12 | Activity | Traveling | 4 |
| 13 | Book | The Adventure Continues | 5 |
| 14 | Drink | Green Tea | 5 |
| 15 | Activity | Gardening | 5 |
| 16 | Book | Finding Inner Peace | 6 |
| 17 | Drink | Coconut Water | 6 |
| 18 | Activity | Reading | 6 |
| 19 | Book | Exploring New Horizons | 7 |
| 20 | Drink | Fruit Juice | 7 |
| 21 | Activity | Cycling | 7 |

# ENTITY RELATIONSHIP DIAGRAM

The Entity Relationship diagram is created using PlantUML as below:
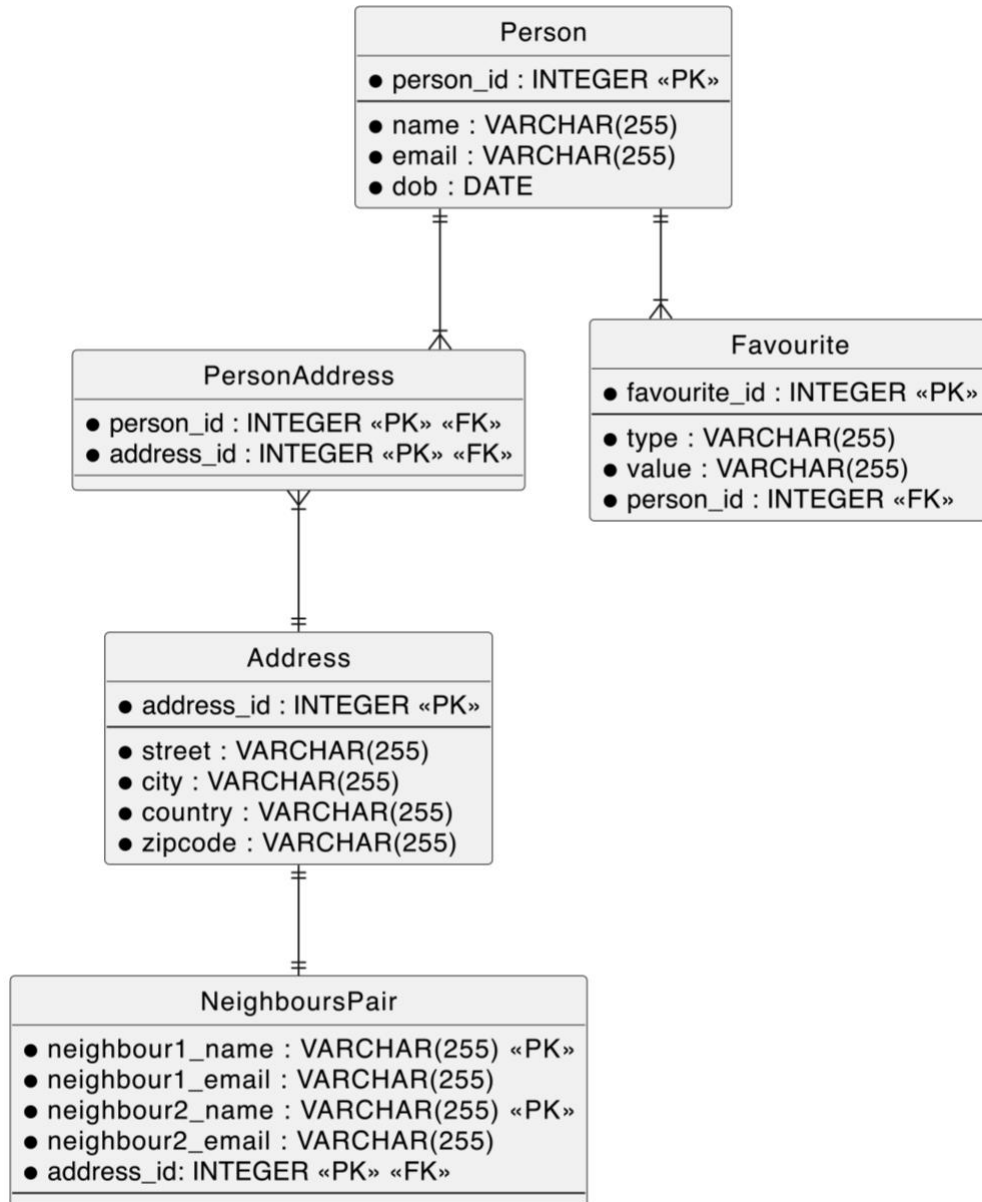


An additional visibility modifier ● to identify mandatory attributes that cannot be null.

# SQL QUERIES

To make SQL queries more efficient, unique indexes are created:

```
CREATE UNIQUE INDEX person_id_index ON Person (person_id);
CREATE UNIQUE INDEX address_id_index ON Address (address_id);
CREATE UNIQUE INDEX favourite_id_index ON Favourite (favourite_id);
CREATE UNIQUE INDEX person_address_index ON PersonAddress (person_id, address_id);
CREATE UNIQUE INDEX neighbours_pair_index ON NeighboursPair (neighbour1_name, neighbour2_name, address_id);
```

## QUERY 1: DISPLAY ALL PERSONS' NAME AND THEIR AGES IN YEARS

```
SELECT name,
    TIMESTAMPDIFF(YEAR, dob, CURDATE()) AS age
FROM Person;
```

| • name varchar(255) | age bigint |
|---|---|
| Person 1 | 29 |
| Person 2 | 31 |
| Person 3 | 33 |
| Person 4 | 26 |
| Person 5 | 41 |
| Person 6 | 35 |
| Person 7 | 28 |
| Person 8 | 35 |
| Person 9 | 31 |
| Person 10 | 27 |
| Person 11 | 32 |
| Person 12 | 38 |
| Person 13 | 33 |
| Person 14 | 38 |
| Person 15 | 40 |
| Person 16 | 34 |
| Person 17 | 29 |
| Person 18 | 30 |
| Person 19 | 32 |
| Person 20 | 36 |

The benchmark of Query 1 before and after adding a unique index can be found in Appendix A.

10

## QUERY 2: GROUP PERSONS BY THEIR FAVOURITE DRINK AND RETURN AVERAGE AGE OF EACH GROUP

```sql
SELECT f.value AS favourite_drink,
    ROUND(AVG(TIMESTAMPDIFF(YEAR, p.dob, CURDATE())), 2) AS average_age
FROM Person p
JOIN Favourite f ON p.person_id = f.person_id
WHERE f.type = 'Drink'
GROUP BY f.value;
```

| favourite_drink varchar | average_age decimal |
|---|---|
| Lemonade | 33.50 |
| Coffee | 31.00 |
| Smoothie | 33.00 |
| Iced Tea | 26.00 |
| Green Tea | 35.00 |
| Coconut Water | 32.50 |
| Fruit Juice | 31.00 |
| Water | 35.50 |
| Hot Chocolate | 31.00 |
| Fruit Smoothie | 27.00 |
| Sparkling Water | 32.00 |
| Herbal Tea | 35.50 |
| Iced Coffee | 40.00 |

The benchmark of Query 2 before and after adding a unique index can be found in Appendix B.

## QUERY 3: DISPLAY THE AVERAGE AGE OF PEOPLE WHO LIKE HIKING

```sql
SELECT f.value as activity, ROUND(AVG(TIMESTAMPDIFF(YEAR, dob, CURDATE())), 2) AS average_age
FROM Person p
JOIN Favourite f ON p.person_id = f.person_id
WHERE f.type = 'Activity' AND f.value = 'Hiking';
```

11

| activity varchar | average_age decimal |
|---|---|
| Hiking | 33.00 |

The benchmark of Query 3 before and after adding a unique index can be found in Appendix C.

## QUERY 4: DISPLAY THE TOTAL NUMBER OF PEOPLE FROM EACH CITY AND SORT IT IN ASCENDING ORDER BY TOTAL NUMBER OF PEOPLE

```
SELECT a.city, COUNT(pa.person_id) AS total_number_of_people
FROM Address a
JOIN PersonAddress pa ON a.address_id = pa.address_id
GROUP BY a.city
ORDER BY total_number_of_people ASC;
```

| city varchar | total_number_of_people bigint |
|---|---|
| Newcastle | 1 |
| Norwich | 1 |
| Leicester | 1 |
| Southampton | 1 |
| Oxford | 1 |
| Cambridge | 1 |
| Nottingham | 1 |
| Sheffield | 1 |
| London | 1 |
| Leeds | 1 |
| Glasgow | 1 |
| Liverpool | 1 |
| Bristol | 1 |
| Birmingham | 1 |
| Manchester | 1 |
| Edinburgh | 2 |
| Cardiff | 3 |

The benchmark of Query 4 before and after adding a unique index can be found in Appendix D.

# QUERY 5: DISPLAY NAME OF PERSON(S) WHOSE NEIGHBOUR IS NEIGHBOUR C

```sql
SELECT p.name AS person_name, np.neighbour1_name AS neighbour_name
FROM Person p
JOIN PersonAddress pa ON p.person_id = pa.person_id
JOIN NeighboursPair np ON pa.address_id = np.address_id
WHERE np.neighbour1_name = 'Neighbour C' OR np.neighbour2_name = 'Neighbour C';
```

| person_name varchar | neighbour_name varchar |
|---|---|
| Person 2 | Neighbour C |

The benchmark of Query 5 before and after adding a unique index can be found in Appendix E.

# APPENDICES

## A) BENCHMARK OF QUERY 1

Display all persons' name and their ages in years

```
SELECT name,
       TIMESTAMPDIFF(YEAR, dob, CURDATE()) AS age
FROM Person;   2ms
```

*Figure 1: The benchmark of Query 1 without indexing*

```
SELECT name,
       TIMESTAMPDIFF(YEAR, dob, CURDATE()) AS age
FROM Person;   1ms
```

*Figure 2: The benchmark of Query 1 after creating unique indexes*

## B) BENCHMARK OF QUERY 2

Group Persons by their favourite drink and return average age of each group

```
SELECT f.value AS favourite_drink,
       ROUND(AVG(TIMESTAMPDIFF(YEAR, p.dob, CURDATE())), 2) AS average_age
FROM Person p
JOIN Favourite f ON p.person_id = f.person_id
WHERE f.type = 'Drink'
GROUP BY f.value;   2ms
```

*Figure 3: The benchmark of Query 2 without indexing*

```
SELECT f.value AS favourite_drink,
       ROUND(AVG(TIMESTAMPDIFF(YEAR, p.dob, CURDATE())), 2) AS average_age
FROM Person p
JOIN Favourite f ON p.person_id = f.person_id
WHERE f.type = 'Drink'
GROUP BY f.value;   1ms
```

*Figure 4: The benchmark of Query 2 after creating unique indexes*

## C) BENCHMARK OF QUERY 3

Display average age of people who likes Hiking

```
SELECT f.value as activity, ROUND(AVG(TIMESTAMPDIFF(YEAR, dob, CURDATE())), 2) AS average_age
FROM Person p
JOIN Favourite f ON p.person_id = f.person_id
WHERE f.type = 'Activity' AND f.value = 'Hiking';   2ms
```

*Figure 5: The benchmark of Query 3 without indexing*

```
SELECT f.value as activity, ROUND(AVG(TIMESTAMPDIFF(YEAR, dob, CURDATE())), 2) AS average_age
FROM Person p
JOIN Favourite f ON p.person_id = f.person_id
WHERE f.type = 'Activity' AND f.value = 'Hiking';   1ms
```

*Figure 6: The benchmark of Query 3 after creating unique indexes*

## D)  BENCHMARK OF QUERY 4

Display the total number of people from each City and sort it in ascending order by total number of people

```
SELECT a.city, COUNT(pa.person_id) AS total_number_of_people
FROM Address a
JOIN PersonAddress pa ON a.address_id = pa.address_id
GROUP BY a.city
ORDER BY total_number_of_people ASC;   3ms
```

*Figure 7: The benchmark of Query 4 without indexing*

```
SELECT a.city, COUNT(pa.person_id) AS total_number_of_people
FROM Address a
JOIN PersonAddress pa ON a.address_id = pa.address_id
GROUP BY a.city
ORDER BY total_number_of_people ASC;   1ms
```

*Figure 8: The benchmark of Query 4 after creating unique indexes*

## E)  BENCHMARK OF QUERY 5

Display name of person(s) whose neighbour is neighbour C

```
SELECT p.name AS person_name, np.neighbour1_name AS neighbour_name
FROM Person p
JOIN PersonAddress pa ON p.person_id = pa.person_id
JOIN NeighboursPair np ON pa.address_id = np.address_id
WHERE np.neighbour1_name = 'Neighbour C' OR np.neighbour2_name = 'Neighbour C';   4ms
```

*Figure 9: The benchmark of Query 5 without indexing*

```
SELECT p.name AS person_name, np.neighbour1_name AS neighbour_name
FROM Person p
JOIN PersonAddress pa ON p.person_id = pa.person_id
JOIN NeighboursPair np ON pa.address_id = np.address_id
WHERE np.neighbour1_name = 'Neighbour C' OR np.neighbour2_name = 'Neighbour C';   1ms
```

*Figure 10: The benchmark of Query 5 after creating unique indexes*