A cartoon of a coat of arms

AI-generated content may be incorrect.

TASK 1

Sophie Fidan 21068639

UFCFU3-15-3 Advanced Databases

|  |
| --- |
| FIRST NORMAL FORM (1nf) |

A screenshot of a computer screen

AI-generated content may be incorrect.A white box with black text

AI-generated content may be incorrect.In 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). For this purpose, a separate row was created for each neighbour of the person. The *id* was created to ensure each row in the table is uniquely identifiable, even if other columns contain duplicate values. This *id* was automatically incremented for each new entry, providing a unique identifier for each row.

CREATE TABLE `1NF` (

`id` int NOT NULL AUTO\_INCREMENT,

`name` varchar(255) NOT NULL,

`email` varchar(255) NOT NULL,

`dob` date NOT NULL,

`street` varchar(255) NOT NULL,

`city` varchar(255) NOT NULL,

`country` varchar(255) NOT NULL,

`zipcode` varchar(20) NOT NULL,

`favourite\_book` varchar(255) NOT NULL,

`favourite\_drink` varchar(255) NOT NULL,

`favourite\_activity` varchar(255) NOT NULL,

`neighbour\_name` varchar(255) NOT NULL,

`neighbour\_email` varchar(255) NOT NULL,

PRIMARY KEY (`id`)

)

|  |
| --- |
| 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 were created as *Person*, *Address*, *Favourite*, *Neighbour* and *Person-Neighbour*.

The Person table only contains information about the person:

CREATE TABLE `Person` (

`id` int NOT NULL AUTO\_INCREMENT,

`name` varchar(255) NOT NULL,

`email` varchar(255) NOT NULL,

`dob` date NOT NULL,

PRIMARY KEY (`id`)

)

A screenshot of a phone

AI-generated content may be incorrect.

The address attributes were separated into a new table called Address. The *Person* table could have included the address information and still be in 2NF because the address attributes (*street*, city, *country* and *zipcode*) depend on the primary key of the *Person* table. However, separating the address information enhances the flexibility and scalability of the database design:

1. Even though it was not part of the user, this design avoids repeating the same address information for multiple people who might share the same address in the future and, therefore, reduces data redundancy.
2. This design allows for easier management and expansion as the system grows. This way, if any address information needs to be updated, the changes need to be made only once in the *Address* table rather than updating multiple records of the Person table when multiple users share the same address.
3. This separation also provides a cleaner and more modular database design.

CREATE TABLE `Address` (

`id` int NOT NULL AUTO\_INCREMENT,

`person\_id` int DEFAULT NULL,

`street` varchar(255) NOT NULL,

`city` varchar(255) NOT NULL,

`country` varchar(255) NOT NULL,

`zipcode` varchar(20) NOT NULL,

PRIMARY KEY (`id`),

KEY `person\_id` (`person\_id`),

CONSTRAINT `Address\_ibfk\_1` FOREIGN KEY (`person\_id`) REFERENCES `Person` (`id`)

)

A screenshot of a computer

AI-generated content may be incorrect.

The *Favourite* table only consists of the user’s favourite information. This table was created with attributes as *id* for primary key, *type*, value, and *person\_id* as a foreign key to the Person table. The Favourite attributes (favourite\_books, *favoruite\_drink*, *favourite\_activity*) could have been placed in a *Favourite* table with separate columns for each type of favourite and still be in 2NF. However, creating a *Favourite* table with *type* and value columns enhances the flexibility and scalability of the database design:

1. By using a type-value structure, uniforms the data model. This way, the system can easily 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. This avoids the need to add a new column called *favourite\_destination* to the table and update existing records with null values for this attribute.
2. 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.

CREATE TABLE `Favourite` (

`id` int NOT NULL AUTO\_INCREMENT,

`type` varchar(255) NOT NULL,

`value` varchar(255) NOT NULL,

`person\_id` int DEFAULT NULL,

PRIMARY KEY (`id`),

KEY `person\_id` (`person\_id`),

CONSTRAINT `Favourite\_ibfk\_1` FOREIGN KEY (`person\_id`) REFERENCES `Person` (`id`)

)

A screenshot of a black screen

AI-generated content may be incorrect.

The Neighbour table contains the neighbour information:

CREATE TABLE `Neighbour` (

`id` int NOT NULL AUTO\_INCREMENT,

`name` varchar(255) NOT NULL,

`email` varchar(255) NOT NULL,

PRIMARY KEY (`id`)

)

A screenshot of a black screen

AI-generated content may be incorrect.

The *PersonNeighbour* table is a junction table that establishes a many-to-many relationship between the *Person* and *Neighbour* entities to eliminate partial dependency. Each row in the *PersonNeighbour* table represents a unique relationship between a person and their neighbour. The *Neighbour* table could have included person\_id as a foreign key to the *Person* table, but this would imply that each neighbour is associated with only one person, or their information would need to be repeated for each person, leading to data duplication. This does not accurately represent the real-world scenario where a neighbour can also be another person’s neighbour even though the use case did not include any example. This design minimises data redundancy and maintains data integrity properly.

CREATE TABLE `PersonNeighbour`

`id` int NOT NULL AUTO\_INCREMENT,

`person\_id` int DEFAULT NULL,

`neighbour\_id` int DEFAULT NULL,

PRIMARY KEY (`id`),

KEY `person\_id` (`person\_id`),

KEY `neighbour\_id` (`neighbour\_id`),

CONSTRAINT `PersonNeighbour\_ibfk\_1` FOREIGN KEY (`person\_id`) REFERENCES `Person` (`id`),

CONSTRAINT `PersonNeighbour\_ibfk\_2` FOREIGN KEY (`neighbour\_id`) REFERENCES `Neighbour` (`id`)

)

A screenshot of a black wall

AI-generated content may be incorrect.

|  |
| --- |
| THIRD NORMAL FORM (3nf) |

In 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. By fulfilling the 2NF for each table, the design already satisfies the requirements of 3NF.

* The *Person* table has a one-to-many relationship with the *Favourite* table, represented by the foreign key *person\_id* in the Favourite table, which means the one person can have 0 or more favourites.
* The *Person* table has a one-to-one relationship with the *Address* table, represented by the foreign key *id* in the *Address* table, which means the one person can have only one address.
* The *Person* table has a many-to-many relationship with the *Neighbour* table, represented by *PersonNeighbour* junction table, which means one person can have 0 or more neighbours.

|  |
| --- |
| ENTITY RELATIONSHIP DIAGRAM |

The Entity Relationship diagram is created using PlantUML as below:

A diagram of a person

AI-generated content may be incorrect.

|  |
| --- |
| sql queries |

-- Display person's name and their age in years:

SELECT name,

TIMESTAMPDIFF(YEAR, dob, CURDATE()) AS age

FROM Person;

A screenshot of a black screen

AI-generated content may be incorrect.

-- 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.id = f.person\_id

WHERE f.type = 'Drink'

GROUP BY f.value;

A screenshot of a menu

AI-generated content may be incorrect.

-- 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.id = f.person\_id

WHERE f.type = 'Activity' AND f.value = 'Hiking';

A screenshot of a computer

AI-generated content may be incorrect.

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

SELECT a.city, COUNT(p.id) AS total\_number\_of\_people

FROM Person p

JOIN Address a ON p.id = a.person\_id

GROUP BY a.city

ORDER BY total\_number\_of\_people ASC;

A screenshot of a black screen

AI-generated content may be incorrect.

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

SELECT p.name as person\_name, n.name as neighbour\_name

FROM Person p

JOIN PersonNeighbour pn ON p.id = pn.person\_id

JOIN Neighbour n ON pn.neighbour\_id = n.id

WHERE n.name = 'Neighbor C';

A screenshot of a phone

AI-generated content may be incorrect.