DATA ENGINEERING RESEARCH PROJECT REPORT

RELATIONAL DATABASE FOR A PROFESSIONAL SOCIAL NETWORK

María Díaz Alba

Contents

1. Objective 2
2. Data selection 2
2.1. Dataset 2
2.2. Entities and relationships 3
3. Database Architecture 3
3.1. Database Overview: Primary Keys, Foreign Keys, and Constraints 3
3.2. Conceptual Data Model – Entity Relationship Diagram 5
3.3. Logical schema 6
4. Database Implementation 7
4.1. Initial DDL Queries 7
4.2. Initial DML Queries 8
4.3. Normalization 12
5. Views 12
6. Data Manipulation and querying 14
6.1. DML queries – CRUD Operations 14
6.2. Complex queries 15
7. Challenges faced and solutions applied 18
8. Conclusion 18

1. Objective

The objective of this project is to build a relational database that represents a simplified version of a professional social network, inspired by LinkedIn. The project begins with a manually constructed denormalized dataset containing user, post, and reaction data, and then applies the normalization process up to the Third Normal Form (3NF). The result is a well-structured, normalized relational database with appropriate keys and constraints.

2. Data selection

2.1. Dataset

The initial dataset consists of a single table called *linkedin*, which combines user profiles, post content, and post reactions into one structure. This table contains redundant information and multiple implied relationships between entities. It was designed as shown in Figures 1, 2 and 3.

```
CREATE TABLE linkedin (
    userId VARCHAR(30),
    fullName VARCHAR(100),
    email VARCHAR(100),
    location VARCHAR(100),
    industry VARCHAR(100),
    joinDate DATE,
    headline VARCHAR(150),
    postId INT,
    postDate DATE,
    content VARCHAR(500),
    visibility VARCHAR(20),
    reactionUserId VARCHAR(30),
    reactionType VARCHAR(20),
    reactionDate DATE
);
```

Figure 1. Table definition for the initial denormalized dataset linkedin

```
INSERT INTO linkedin VALUES
('ana.gomez',     'Ana Gómez',     'ana@abc.com',       'Madrid',     'Tech',       '2022-05-01', 'Data Scientist at ABC',
('ana.gomez',     'Ana Gómez',     'ana@abc.com',       'Madrid',     'Tech',       '2022-05-01', 'Data Scientist at ABC',
('ana.gomez',     'Ana Gómez',     'ana@abc.com',       'Madrid',     'Tech',       '2022-05-01', 'Data Scientist at ABC',
('carlos.ruiz',   'Carlos Ruiz',   'carlos@xyz.com',    'Barcelona',  'Tech',       '2021-08-10', 'Software Engineer at XYZ',
('carlos.ruiz',   'Carlos Ruiz',   'carlos@xyz.com',    'Barcelona',  'Tech',       '2021-08-10', 'Software Engineer at XYZ',
('marta.lopez',   'Marta López',   'marta.hr@mail.com', 'Miami',      'HR',         '2023-01-12', 'HR Manager',
('marta.lopez',   'Marta López',   'marta.hr@mail.com', 'Miami',      'HR',         '2023-01-12', 'HR Manager',
('laura.sanchez', 'Laura Sánchez', 'laura.iot@mail.com','Miami',      'Research',   '2023-07-21', 'IoT Researcher',
('elena.munoz',   'Elena Muñoz',   'elena.biz@mail.com','New York',   'Business',   '2022-03-17', 'Business Analyst',
('tomas.navarro', 'Tomás Navarro', 'tomas@sec.com',     'Madrid',     'Security',   '2020-09-25', 'Cybersecurity Specialist',
```

Figure 2. INSERT statements used to populate the linkedin table with user, post, and reaction data

Results | Messages

	userId	fullName	email	location	industry	joinDate	headline	postId	postDate	content	visibility	reactionUserId	reactionType	reactionDate
1	ana.gomez	Ana Gómez	ana@abc.com	Madrid	Tech	2022-05-01	Data Scientist at ABC	1	2024-01-01	Excited to start my role!	public	carlos.ruiz	like	2024-01-02
2	ana.gomez	Ana Gómez	ana@abc.com	Madrid	Tech	2022-05-01	Data Scientist at ABC	1	2024-01-01	Excited to start my role!	public	marta.lopez	celebrate	2024-01-02
3	ana.gomez	Ana Gómez	ana@abc.com	Madrid	Tech	2022-05-01	Data Scientist at ABC	2	2024-01-03	Sharing my latest article.	connections	elena.munoz	support	2024-01-04
4	carlos.ruiz	Carlos Ruiz	carlos@xyz.com	Barcelona	Tech	2021-08-10	Software Engineer at XYZ	3	2023-12-12	New backend project released.	public	ana.gomez	curious	2023-12-13
5	carlos.ruiz	Carlos Ruiz	carlos@xyz.com	Barcelona	Tech	2021-08-10	Software Engineer at XYZ	3	2023-12-12	New backend project released.	public	laura.sanchez	insightful	2023-12-14
6	marta.lopez	Marta López	marta.hr@mail.com	Miami	HR	2023-01-12	HR Manager	4	2024-01-08	We're hiring! Message me!	public	javier.torres	like	2024-01-09
7	marta.lopez	Marta López	marta.hr@mail.com	Miami	HR	2023-01-12	HR Manager	5	2024-02-01	Work culture matters.	connections	elena.munoz	celebrate	2024-02-02
8	laura.sanc...	Laura Sán...	laura.iot@mail.com	Miami	Rese...	2023-07-21	IoT Researcher	6	2023-11-05	IoT tips and tricks.	public	tomas.navarro	support	2023-11-06
9	elena.mu...	Elena Muñ...	elena.biz@mail.c...	New York	Busin...	2022-03-17	Business Analyst	7	2024-03-10	Data is the new oil.	public	paula.moreno	like	2024-03-11
10	tomas.nav...	Tomás Na...	tomas@sec.com	Madrid	Secur...	2020-09-25	Cybersecurity Specialist	8	2024-04-01	Cyber awareness week!	public	andres.perez	curious	2024-04-02

Figure 3. Resulting content of the linkedin table

Table 1 shows an overview of all data of this table, including a brief description of each attribute and the data types.

Field	Description	DataType
userId	ID of the user who created the post	VARCHAR(30)
fullName	Full name of the user	VARCHAR(100)
email	Email address of the user	VARCHAR(100)
location	User's city or region	VARCHAR(100)
industry	Industry or professional sector of the user	VARCHAR(100)
joinDate	Date the user joined the platform	DATE
headline	User's role or professional summary	VARCHAR(150)
postId	Unique ID of the post	INT
postDate	Date the post was created	DATE
content	Content or message of the post	VARCHAR(500)
visibility	Post visibility (public, connections, private)	VARCHAR(20)
reactionUserId	User who reacted to the post	VARCHAR(30)
reactionType	Type of reaction (like, celebrate, support, etc.)	VARCHAR(20)
reactionDate	Date the reaction was made	DATE

Table 1. Overview of linkedin table data

2.2. Entities and relationships

From the denormalized data, the following entities and relationships between them were identified:

Entities	
Users	Represents all users in the network.
Posts	Contains all published posts
Reactions	Stores user interactions with posts
Connections *(added later)*	Stores connection relationships between users

Table 2. Entities description

Relationships		
Users → Posts	A user can create multiple posts	One-to-many relationship
Users → Reactions	A user can react to multiple posts	One-to-many relationship
Posts → Reactions	A post can have multiple reaction	One-to-many relationship
Users → Users	Users can connect with other users	One-to-many relationship

Table 3. Relationships description

3. Database Architecture

This section presents the structural design of the database system, transitioning from conceptual design to logical implementation.

3.1. Database Overview: Primary Keys, Foreign Keys, and Constraints

This section provides an overview of the structural elements that define the integrity and logic of the database. It outlines the primary keys that uniquely identify each entity, the foreign keys that establish relationships between tables, and the key constraints that ensure consistency, validity, and adherence to the rules of the system.

3.1.1. Users Table

Purpose: Stores personal and professional information about each user on the platform.

Primary Key:

- userId (VARCHAR) — Unique username-style identifier (e.g., ana.gomez)

Key Attributes:

- *fullName, email, location, industry, joinDate, headline*
- email is UNIQUE (enforced via a filtered index to allow multiple NULL values)
- fullName and joinDate are marked as NOT NULL

Relationships:

- A user can create multiple posts (Posts)
- A user can react to multiple posts (Reactions), but each reaction belongs to a single user.
- A user can connect with other users (Connections)

3.1.2. Posts Table

Purpose: Represents content posted by users.

Primary Key:

- postId (INT)

Foreign Key:

- userId → References Users(userId) → ON DELETE CASCADE constraint: If a user is deleted, their posts are also deleted. The constraint is named FK_Posts_Users.

Key Attributes:

- *postDate, content, visibility*
- visibility is restricted with a CHECK constraint that ensures only valid visibility values are accepted (e.g., prevents typos like 'Publi').

Relationships:

- Each post belongs to a single **user**
- Each post can have multiple **reactions**

3.1.3. Reactions Table

Purpose: Stores reactions made by users to posts.

Primary Key:

- reactionId (INT)

Foreign Keys:

- postId → References Posts(postId) → ON DELETE CASCADE
- userId → References Users(userId) → ON DELETE NO ACTION to prevent conflict with cascading deletes from both Posts and Users.
- UNIQUE (postId, userId) ensures a user can only react once per post avoiding duplicate reactions. This constraint is named Unique_User_Post_React.

Key Attributes:

- *reactionDate, reactionType*
- reactionType restricted via CHECK constraint that ensures only valid reaction type values are accepted (e.g., 'like', 'celebrate', 'support').

Relationships:

- Each reaction is made by a **user**
- Each reaction belongs to a specific **post**

3.1.4. Connections Table – Additional Table

While the original linkedin table contained information about users, their posts, and reactions, it did not include any data regarding professional connections between users. For the purpose of enriching the relational model and better simulating a real-world social network platform, the Connections table was added as a logical extension to the initial dataset.

Purpose: Models connection (friendship) relationships between users.

Primary Key:

- connectionId (INT)

Foreign Keys:

- userId1 → References Users(userId) → ON DELETE NO ACTION
- userId2 → References Users(userId) → ON DELETE NO ACTION
    - These foreign keys do not cascade on delete, to mimic real-world behavior: if a user is deleted, their connections are not automatically removed.
- UNIQUE(userId1, userId2) prevents duplicate connections
- CHECK(userId1 < userId2) ensures that each connection is stored in a standardized direction (e.g., ana.gomez → carlos.ruiz, but not the reverse) and enforces consistent alphabetical order of user pairs.

Key Attributes:

- *connectionDate*

Relationships:

- Each row represents a bidirectional connection between two users

3.2. Conceptual Data Model – Entity Relationship Diagram

The Entity-Relationship Diagram (ERD) visually represents the core entities identified in the dataset, along with their attributes and interrelationships. It serves as a high-level abstraction of the database structure, helping to define how data is organized and how different components interact within the system. It is represented in Figure 4.

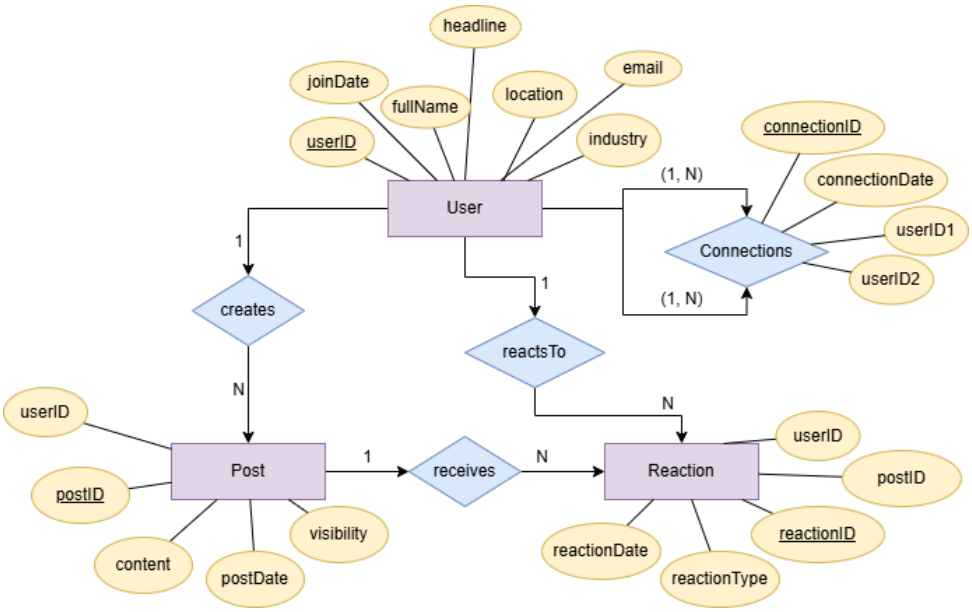

Figure 4. ER Diagram of linkedin database

3.3. Logical schema

The logical schema outlines the detailed structure of the database tables, including all column names, data types, and relationships. It represents the practical implementation of the conceptual model, providing the blueprint for creating the database using Data Definition Language (DDL) statements in SQL Server.

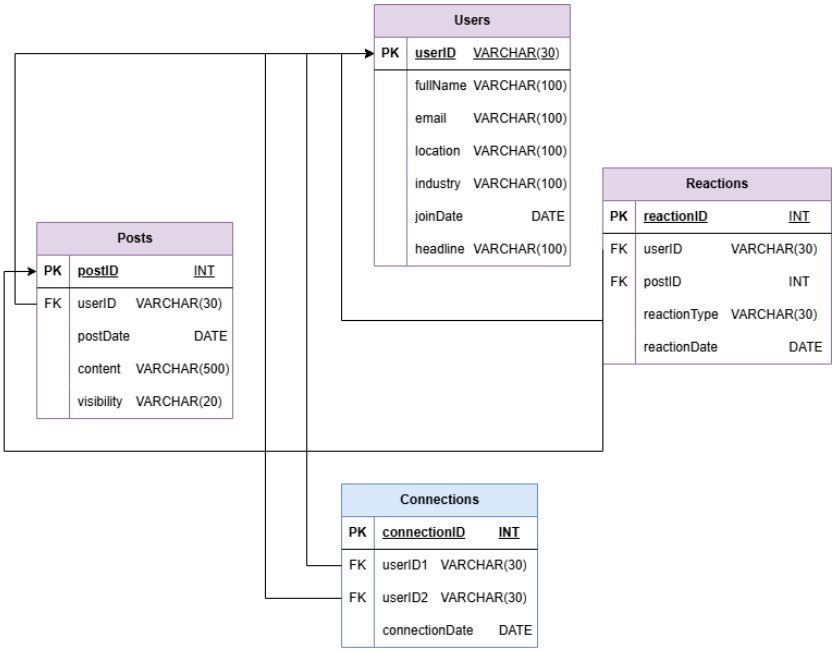

Figure 5. Logical schema of linkedin database

4. Database Implementation

4.1. Initial DDL Queries

This section includes the initial set of DDL queries used to build the normalized database. DDL statements such as CREATE, ALTER, and DROP are used to define the structure of each table, enforce relationships through foreign keys, apply constraints, and create indexes for optimization.

4.1.1. Create Users table

Figure 6 shows the initial query to create the Users table. An additional adjustment was made to the column email when defining it to allow NULL values, but to preserve the uniqueness of email addresses where present, we created a **filtered unique index**.

```
CREATE TABLE Users (
    userId VARCHAR(30) PRIMARY KEY,
    fullName VARCHAR(100) NOT NULL,
    email VARCHAR(100),
    location VARCHAR(100),
    industry VARCHAR(100),
    joinDate DATE NOT NULL,
    headline VARCHAR(150)
);

CREATE UNIQUE INDEX idx_unique_email_not_null
ON Users (email)
WHERE email IS NOT NULL;
```

Figure 6. Table definition for the Users data

4.1.2. Create Posts table

Figure 7 shows the SQL query used to create the Posts table.

```
CREATE TABLE Posts (
    postId INT PRIMARY KEY,
    userId VARCHAR(30) NOT NULL,
    postDate DATE NOT NULL,
    content VARCHAR(500) NOT NULL,
    visibility VARCHAR(20) NOT NULL CHECK (visibility IN ('public', 'connections', 'private')),
    CONSTRAINT FK_Posts_Users FOREIGN KEY (userId) REFERENCES Users(userId) ON DELETE CASCADE
);
```

Figure 7. Table definition for the Posts data

4.1.3. Create Reactions Table

Figure 8 shows the SQL query used to create the Reactions table.

```
CREATE TABLE Reactions (
    reactionId INT PRIMARY KEY,
    postId INT NOT NULL,
    userId VARCHAR(30) NOT NULL,
    reactionType VARCHAR(20) NOT NULL CHECK (reactionType IN ('like', 'celebrate', 'support', 'insightful', 'curious')),
    reactionDate DATE NOT NULL,
    CONSTRAINT FK_Reactions_Posts FOREIGN KEY (postId) REFERENCES Posts(postId) ON DELETE CASCADE,
    CONSTRAINT FK_Reactions_Users FOREIGN KEY (userId) REFERENCES Users(userId) ON DELETE NO ACTION,
    CONSTRAINT Unique_User_Post_React UNIQUE (postId, userId)
);
```

Figure 8. Table definition for the Reactions data

4.1.4. Create Connections table

Figure 9 shows the SQL query used to create the Connections table.

```sql
CREATE TABLE Connections (
    connectionId INT PRIMARY KEY,
    userId1 VARCHAR(30) NOT NULL,
    userId2 VARCHAR(30) NOT NULL,
    connectionDate DATE NOT NULL,
    CONSTRAINT FK_Conn_User1 FOREIGN KEY (userId1) REFERENCES Users(userId) ON DELETE NO ACTION,
    CONSTRAINT FK_Conn_User2 FOREIGN KEY (userId2) REFERENCES Users(userId) ON DELETE NO ACTION,
    CONSTRAINT Unique_Conn UNIQUE (userId1, userId2),
    CHECK (userId1 < userId2)
);
```

Figure 9. Table definition for the Connections data

4.1.5. Additional DDL queries

Besides the CREATE statements, we also used other DDL commands during development.

Figure 10 shows the use of ALTER TABLE to add NOT NULL constraints to ensure data completeness in the Posts table.

Figure 11 shows DROP TABLE IF EXISTS, which allowed us to safely delete and recreate tables during testing, avoiding errors when re-running scripts.

These commands helped refine the schema and keep the database consistent throughout the process.

```sql
ALTER TABLE Posts
ALTER COLUMN userId VARCHAR(30) NOT NULL;

ALTER TABLE Posts
ALTER COLUMN postDate DATE NOT NULL;

ALTER TABLE Posts
ALTER COLUMN content VARCHAR(500) NOT NULL;

ALTER TABLE Posts
ALTER COLUMN visibility VARCHAR(20) NOT NULL;
```

Figure 10. ALTER statements used to modify tables

```sql
DROP TABLE IF EXISTS Reactions;
DROP TABLE IF EXISTS Posts;
DROP TABLE IF EXISTS Users;
```

Figure 11. DROP statements used to remove tables

4.2. Initial DML Queries

Queries such as INSERT, UPDATE and DELETE are used to populate the tables with data derived from the denormalized source (linkedin).

4.2.1. Populate Users table

During the design process, we realized that some users only appear in the dataset as reaction authors (i.e., as reactionUserId) but never as post creators. Therefore, if we only inserted users based on the post authors (userId), those who only reacted to posts would be missing from the Users table.

To ensure all users were properly inserted, we restructured the Users table population using two separate INSERT statements:

1. First insert: for users who posted content, they have complete data in the linkedin table. See Figure 12.

2. Second insert: for users who only reacted to posts. Details like fullName were constructed from the reactionUserId (which follows the format name.surname). See Figure 13.

Figure 14 displays the final state of the Users table, now with both post authors and reaction users included.

```
INSERT INTO Users (userId, fullName, email, location, industry, joinDate, headline)
SELECT DISTINCT
    userId, fullName, email, location, industry, joinDate, headline
FROM linkedin;
```

Figure 12. INSERT statements used to populate the Users table

```
INSERT INTO Users (userId, fullName, email, location, industry, joinDate, headline)
SELECT DISTINCT
    reactionUserId AS userId,
    CONCAT(
        UPPER(LEFT(reactionUserId, 1)),
        LOWER(SUBSTRING(reactionUserId, 2, CHARINDEX('.', reactionUserId) - 2)),
        ' ',
        UPPER(SUBSTRING(reactionUserId, CHARINDEX('.', reactionUserId) + 1, 1)),
        LOWER(SUBSTRING(reactionUserId, CHARINDEX('.', reactionUserId) + 2, LEN(reactionUserId)))
    ) AS fullName,
    NULL AS email,
    NULL AS location,
    NULL AS industry,
    GETDATE() AS joinDate,
    NULL AS headline
FROM linkedin
WHERE reactionUserId IS NOT NULL
    AND reactionUserId NOT IN (SELECT userId FROM Users);
```

Figure 13. INSERT statements used to populate the Users table

Results | Messages

	userId	fullName	email	location	industry	joinDate	headline
1	ana.gomez	Ana Gómez	ana@abc.com	Madrid	Tech	2022-05-01	Data Scientist at ABC
2	andres.perez	Andres Perez	NULL	NULL	NULL	2025-04-15	NULL
3	carlos.ruiz	Carlos Ruiz	carlos@xyz.com	Barcelona	Tech	2021-08-10	Software Engineer at XYZ
4	elena.munoz	Elena Muñoz	elena.biz@ma...	New York	Busin...	2022-03-17	Business Analyst
5	javier.torres	Javier Torres	NULL	NULL	NULL	2025-04-15	NULL
6	laura.sanch...	Laura Sánc...	laura.iot@mail...	Miami	Rese...	2023-07-21	IoT Researcher
7	marta.lopez	Marta López	marta.hr@mail...	Miami	HR	2023-01-12	HR Manager
8	paula.more...	Paula More...	NULL	NULL	NULL	2025-04-15	NULL
9	tomas.nava...	Tomás Nav...	tomas@sec.c...	Madrid	Secur...	2020-09-25	Cybersecurity Specialist

Figure 14. Resulting content of the Users table

4.2.2. Populate Posts table

Figure 15 shows the SQL query used to populate the Posts table, while Figure 16 displays the final state of the table with all records successfully inserted.

```
INSERT INTO Posts (postId, userId, postDate, content, visibility)
SELECT DISTINCT
    postId, userId, postDate, content, visibility
FROM linkedin;
```

Figure 15.INSERT statements used to populate the Posts table

Results | Messages

	postId	userId	postDate	content	visibility
1	1	ana.gomez	2024-01-01	Excited to start my role!	public
2	2	ana.gomez	2024-01-03	Sharing my latest arti...	conn...
3	3	carlos.ruiz	2023-12-12	New backend project ...	public
4	4	marta.lop...	2024-01-08	We're hiring! Messag...	public
5	5	marta.lop...	2024-02-01	Work culture matters.	conn...
6	6	laura.san...	2023-11-05	IoT tips and tricks.	public
7	7	elena.mu...	2024-03-10	Data is the new oil.	public
8	8	tomas.na...	2024-04-01	Cyber awareness we...	public

Figure 16. Resulting content of the Posts table

4.2.3. Populate Reactions Table

Figure 17 shows the SQL query used to populate the Reactions table. Note that a WITH clause combined with ROW_NUMBER() is used to generate a unique reactionId for each record. This approach was necessary because the original denormalized table (linkedin) did not include a primary key for individual reactions. Since the Reactions table requires a unique identifier (reactionId as the primary key), we used ROW_NUMBER() to assign a sequential value to each reaction.

Figure 18 displays the final state of the table with all records successfully inserted.

```
CREATE TABLE Reactions (
    reactionId INT PRIMARY KEY,
    postId INT NOT NULL,
    userId VARCHAR(30) NOT NULL,
    reactionType VARCHAR(20) NOT NULL CHECK (reactionType IN ('like', 'celebrate', 'support', 'insightful', 'curious')),
    reactionDate DATE NOT NULL,
    CONSTRAINT FK_Reactions_Posts FOREIGN KEY (postId) REFERENCES Posts(postId) ON DELETE CASCADE,
    CONSTRAINT FK_Reactions_Users FOREIGN KEY (userId) REFERENCES Users(userId) ON DELETE NO ACTION,
    CONSTRAINT Unique_User_Post_React UNIQUE (postId, userId)
);

WITH ReactionsData AS (
    SELECT
        ROW_NUMBER() OVER (ORDER BY postId, reactionUserId) AS reactionId,
        postId,
        reactionUserId AS userId,
        reactionType,
        reactionDate
    FROM linkedin
    WHERE reactionUserId IS NOT NULL
)
INSERT INTO Reactions (reactionId, postId, userId, reactionType, reactionDate)
SELECT reactionId, postId, userId, reactionType, reactionDate
FROM ReactionsData;
```

Figure 17. INSERT statements used to populate the Reactions table

Results | Messages

	reactionId	postId	userId	reactionType	reactionDate
1	1	1	carlos.ruiz	like	2024-01-02
2	2	1	marta.lo...	celebrate	2024-01-02
3	3	2	elena.m...	support	2024-01-04
4	4	3	ana.go...	curious	2023-12-13
5	5	3	laura.sa...	insightful	2023-12-14
6	6	4	javier.tor...	like	2024-01-09
7	7	5	elena.m...	celebrate	2024-02-02
8	8	6	tomas.n...	support	2023-11-06
9	9	7	paula.m...	like	2024-03-11
10	10	8	andres....	curious	2024-04-02

Figure 18. Resulting content of the Reactions table

4.2.4. Populate Connections table

Figure 19 shows the SQL query used to populate the Connections table. This query ensures that rows are only inserted if both userId1 and userId2 exist in Users. It also maintains alphabetical order in data (userId1 < userId2) as required by CHECK.

Figure 20 displays the final state of the table with all records successfully inserted.

```
INSERT INTO Connections (connectionId, userId1, userId2, connectionDate)
SELECT *
FROM (
    VALUES
        (1,  'ana.gomez',     'carlos.ruiz',   '2023-01-15'),
        (2,  'ana.gomez',     'marta.lopez',   '2023-01-20'),
        (3,  'carlos.ruiz',   'javier.torres', '2023-02-11'),
        (4,  'laura.sanchez', 'marta.lopez',   '2023-03-01'),
        (5,  'diego.ramos',   'javier.torres', '2022-09-19'),
        (6,  'diego.ramos',   'laura.sanchez', '2023-05-02'),
        (7,  'elena.munoz',   'laura.sanchez', '2023-05-03'),
        (8,  'elena.munoz',   'tomas.navarro', '2023-06-10'),
        (9,  'paula.moreno',  'tomas.navarro', '2023-07-18'),
        (10, 'andres.perez',  'paula.moreno',  '2024-01-05')
) AS connections(connectionId, userId1, userId2, connectionDate)
WHERE userId1 IN (SELECT userId FROM Users)
  AND userId2 IN (SELECT userId FROM Users);
```

Figure 19. INSERT statements used to populate the Connections table

Results | Messages

	connectionId	userId1	userId2	connectionDate
1	1	ana.gomez	carlos.ruiz	2023-01-15
2	2	ana.gomez	marta.lopez	2023-01-20
3	3	carlos.ruiz	javier.torres	2023-02-11
4	4	laura.sanchez	marta.lopez	2023-03-01
5	7	elena.munoz	laura.sanchez	2023-05-03
6	8	elena.munoz	tomas.navarro	2023-06-10
7	9	paula.moreno	tomas.navarro	2023-07-18
8	10	andres.perez	paula.moreno	2024-01-05

Figure 20. Resulting content of the Connections table

4.2.5. Additional DML Queries

In addition to INSERT statements used to populate the tables, we also applied UPDATE and DELETE queries during the development process to maintain data accuracy and integrity, as shown in Figure 21 and 22.

```
UPDATE Reactions
SET reactionType = 'curious'
WHERE reactionType = 'insightful';
```

Figure 21. UPDATE statement used to update Reactions table

In this case, as we have created the data ourselves, we did not necessarily have to use the delete query at this stage, but an example is shown in Figure 22 by inserting an invented row to simulate the need to delete a row of data from a table.

```sql
INSERT INTO Reactions (reactionId, postId, userId, reactionType, reactionDate)
VALUES (999, 1, 'elena.munoz', 'like', '2025-04-15');

DELETE FROM Reactions
WHERE reactionId = 999;
```

Figure 22. DELETE statement used to remove data from Reactions table

4.3. Normalization

The database was carefully structured through the normalization process to eliminate redundancy, ensure data consistency, and optimize relational design. The process was applied to a manually created denormalized table named linkedin, which contained combined information about users, posts, and reactions.

- ✓ **First Normal Form (1NF)**
  - All attributes in the linkedin table contain atomic (indivisible) values.
  - There are no repeating groups or arrays.
  - Each row is uniquely identifiable and consistent in structure.
- ✓ **Second Normal Form (2NF)**
  - The table was decomposed to eliminate partial dependencies.
  - Attributes that did not depend on the entire composite key were moved to separate tables.
  - For example, user details (e.g., fullName, email, industry) were moved to the Users table, which is keyed by userId, not by combinations like postId + reactionUserId.
- ✓ Third Normal Form (3NF)
  - All transitive dependencies were removed.
  - Attributes now depend only on the primary key of each table. For example, headline depends directly on userId in the Users table, and reactionType depends only on reactionId in the Reactions table.
  - No attribute depends on a non-key column.

After normalization, each separate table now contains:

- A primary key
- Only attributes that are fully dependent on that key
- Foreign keys to relate entities without duplicating information

5. Views

Creating SQL Views is a powerful way to simplify complex queries, encapsulate business logic, and provide tailored representations of the data. We created several meaningful and reusable views, as shown in Figures 23, 24 and 25, which display the SQL definitions of three selected views.

```sql
-- View: All Posts with Author Information
CREATE VIEW View_PostsWithAuthor AS
SELECT
    p.postId,
    p.postDate,
    p.content,
    p.visibility,
    u.userId AS authorId,
    u.fullName AS authorName,
    u.location,
    u.headline
FROM Posts p
JOIN Users u ON p.userId = u.userId;
```

Figure 23. View joining post data with author details

```sql
-- View: Connections with Full Names
CREATE VIEW View_ConnectionsWithNames AS
SELECT
    c.connectionId,
    c.connectionDate,
    u1.fullName AS user1,
    u2.fullName AS user2
FROM Connections c
JOIN Users u1 ON c.userId1 = u1.userId
JOIN Users u2 ON c.userId2 = u2.userId;
```

Figure 24. View displaying users' connections & fullnames

```sql
-- View: Top Posts by Number of Reactions
CREATE VIEW View_TopPostsByReactions AS
SELECT
    p.postId,
    p.content,
    COUNT(r.reactionId) AS totalReactions
FROM Posts p
LEFT JOIN Reactions r ON p.postId = r.postId
GROUP BY p.postId, p.content;
```

Figure 25. View that lists posts along with the total number of reactions they have received

Once the views are created, we can interact with them just like with regular tables. These examples illustrate how views can enhance readability, reduce redundancy, and simplify querying in larger databases.

```sql
SELECT * FROM View_TopPostsByReactions;
```

100 %

Results | Messages

	postId	content	totalReactions
1	1	Excited to start my role!	3
2	2	Sharing my latest article.	1
3	3	New backend project released.	2
4	4	We're hiring! Message me!	1
5	5	Work culture matters.	1
6	6	IoT tips and tricks.	1
7	7	Data is the new oil.	1
8	8	Cyber awareness week!	1

Figure 26. Simple SELECT query retrieving all records from a view.

```sql
-- Posts with more than 2 reactions
SELECT *
FROM View_TopPostsByReactions
WHERE totalReactions > 2;
```

100 %

Results | Messages

	postId	content	totalReactions
1	1	Excited to start my role!	3

Figure 27. Demonstration of how to apply filters using a WHERE clause

```sql
-- Author and location of the post with more reactions
SELECT
    v.postId,
    v.content,
    v.totalReactions,
    u.fullName AS authorName,
    u.location
FROM View_TopPostsByReactions v
JOIN Posts p ON v.postId = p.postId
JOIN Users u ON p.userId = u.userId
WHERE v.totalReactions > 2
ORDER BY v.totalReactions DESC;
```

100 % | Results | Messages

	postId	content	totalReactions	authorName	location
1	1	Excited to start my role!	3	Ana Gómez	Madrid

Figure 28. How a view can be used in combination with other tables in a JOIN query

6. Data Manipulation and querying

This section demonstrates the practical application of SQL to manipulate and explore the data stored in the relational database. It is divided into two parts: basic DML operations and more complex queries that combine multiple tables and extract meaningful insights.

6.1. DML queries – CRUD Operations

6.1.1. Create – Add a new user to the platform

```sql
INSERT INTO Users (userId, fullName, email, location, industry, joinDate, headline)
VALUES ('roberto.mendez', 'Roberto Méndez', 'rmendez@mail.com', 'Barcelona', 'Product Management', '2023-12-01', 'Product Owner');
```

8	paula.more...	Paula More...	NULL	NULL	NULL	2025-04-15	NULL
9	roberto.me...	Roberto Mé...	rmendez@mail.com	Barcelona	Produc...	2023-12-01	Product Owner
10	tomas.nava...	Tomás Nav...	tomas@sec.com	Madrid	Security	2020-09-25	Cybersecurity Specialist

Figure 29. SQL query for a Create operation and its results

6.1.2. Read – Retrieve users who joined in 2023

```sql
SELECT userId, fullName, joinDate
FROM Users
WHERE YEAR(joinDate) = 2023;
```

100 % | Results | Messages

	userId	fullName	joinDate
1	laura.sanchez	Laura Sánchez	2023-07-21
2	marta.lopez	Marta López	2023-01-12
3	roberto.mendez	Roberto Méndez	2023-12-01

Figure 30. SQL query for a Read operation and its results

6.1.3. Update – Change visibility of all public posts published on 2024.

```sql
UPDATE Posts
SET visibility = 'connections'
WHERE YEAR(postDate) = 2024;


SELECT visibility, postDate
FROM Posts;
```

100 % — Results — Messages

	visibility	postDate
1	connections	2024-01-01
2	connections	2024-01-03
3	public	2023-12-12
4	connections	2024-01-08
5	connections	2024-02-01
6	public	2023-11-05
7	connections	2024-03-10
8	connections	2024-04-01

Figure 31. SQL query for an Update operation and its results

6.1.4. Delete – Remove users with "owner" in their headline

Before applying DELETE statement, we can retrieve data that will be affected to ensure it is correct

```sql
SELECT userID, headline
FROM Users
WHERE LOWER(headline) LIKE '%owner%';
```

100 % — Results — Messages

	userID	headline
1	roberto.mendez	Product Owner

Figure 32. SQL query for retrieve data

```sql
DELETE FROM Users
WHERE LOWER(headline) LIKE '%owner%';

SELECT userID, headline
FROM Users;
```

100 % — Results — Messages

	userID	headline
1	ana.gomez	Data Scientist at ABC
2	andres.perez	NULL
3	carlos.ruiz	Software Engineer at XYZ
4	elena.munoz	Business Analyst
5	javier.torres	NULL
6	laura.sanchez	IoT Researcher
7	marta.lopez	HR Manager
8	paula.moreno	NULL
9	tomas.navarro	Cybersecurity Specialist

Figure 33. SQL query for a Delete operation and its results

6.2. Complex queries

We can use complex queries for analysis of data:

- **Count how many posts each user has made**

```sql
SELECT u.fullName, COUNT(p.postId) AS totalPosts
FROM Users u
LEFT JOIN Posts p ON u.userId = p.userId
GROUP BY u.fullName
ORDER BY totalPosts DESC;
```

Results — Messages

	fullName	totalPosts
1	Ana Gómez	2
2	Marta López	2
3	Laura Sánchez	1
4	Tomás Navarro	1
5	Carlos Ruiz	1
6	Elena Muñoz	1
7	Javier Torres	0
8	Andres Perez	0
9	Paula Moreno	0

Figure 34. SQL query for task 1

- **Which reaction type is most common**

```sql
SELECT reactionType, COUNT(*) AS total
FROM Reactions
GROUP BY reactionType
ORDER BY total DESC;
```

100 % — Results — Messages

	reactionType	total
1	like	4
2	curious	3
3	celebrate	2
4	support	2

Figure 35. SQL query for task 2

- **Average of reactions per post**

```sql
SELECT AVG(reactionCount) AS avgReactionsPerPost
FROM (
    SELECT postId, COUNT(*) AS reactionCount
    FROM Reactions
    GROUP BY postId
) AS PostReactions;
```

100 %

Results | Messages

	avgReactionsPerPost
1	1

Figure 36. SQL query for task 3

- **Who are the most active users reacting**

```sql
SELECT u.userId, u.fullName, COUNT(r.reactionId) AS totalReactionsMade
FROM Users u
JOIN Reactions r ON u.userId = r.userId
GROUP BY u.userId, u.fullName
ORDER BY totalReactionsMade DESC;
```

100 %

Results | Messages

	userId	fullName	totalReactionsMade
1	elena.munoz	Elena Muñoz	3
2	javier.torres	Javier Torres	1
3	laura.sanchez	Laura Sánchez	1
4	marta.lopez	Marta López	1
5	paula.moreno	Paula Moreno	1
6	tomas.navarro	Tomás Navarro	1
7	ana.gomez	Ana Gómez	1
8	andres.perez	Andres Perez	1
9	carlos.ruiz	Carlos Ruiz	1

Figure 37. SQL query for task 4

- **How many connections each user has**

```sql
SELECT u.userId, u.fullName, COUNT(*) AS totalConnections
FROM Users u
LEFT JOIN (
    SELECT userId1 AS userId FROM Connections
    UNION ALL
    SELECT userId2 AS userId FROM Connections
) AS all_connections ON u.userId = all_connections.userId
GROUP BY u.userId, u.fullName
ORDER BY totalConnections DESC;
```

100 %

Results | Messages

	userId	fullName	totalConnections
1	ana.gomez	Ana Gómez	2
2	carlos.ruiz	Carlos Ruiz	2
3	elena.munoz	Elena Muñoz	2
4	laura.sanchez	Laura Sánchez	2
5	marta.lopez	Marta López	2
6	paula.moreno	Paula Moreno	2
7	tomas.navarro	Tomás Navarro	2
8	javier.torres	Javier Torres	1
9	andres.perez	Andres Perez	1

Figure 38. SQL query for task 5

We can apply specific filters and conditions

- **Posts with public visibility and 2 or more reactions**

```sql
SELECT
    p.postId,
    p.content,
    COUNT(r.reactionId) AS totalReactions
FROM Posts p
JOIN Reactions r ON p.postId = r.postId
WHERE p.visibility = 'public'
GROUP BY p.postId, p.content
HAVING COUNT(r.reactionId) >= 2;
```

100 %

Results | Messages

	postId	content	totalReactions
1	3	New backend project released.	2

Figure 39. SQL query for task 6

- **Users who have never posted anything**

```sql
SELECT u.userId, u.fullName
FROM Users u
LEFT JOIN Posts p ON u.userId = p.userId
WHERE p.postId IS NULL;
```

100 %

Results | Messages

	userId	fullName
1	andres.perez	Andres Perez
2	javier.torres	Javier Torres
3	paula.moreno	Paula Moreno

Figure 40. SQL query for task 7

- **Users who have only reacted but have never posted**

```sql
SELECT DISTINCT u.userId, u.fullName
FROM Users u
JOIN Reactions r ON u.userId = r.userId
WHERE u.userId NOT IN (SELECT userId FROM Posts)
```

100 %

Results | Messages

	userId	fullName
1	andres.perez	Andres Perez
2	javier.torres	Javier Torres
3	paula.moreno	Paula Moreno

Figure 41. SQL query for task 8

- **Find users with more than 2 total connections, ordered by older users.**

```sql
SELECT
    u.userId,
    u.fullName,
    u.joinDate,
    COUNT(c.connectionId) AS totalConnections
FROM Users u
JOIN (
    SELECT userId1 AS userId FROM Connections
    UNION ALL
    SELECT userId2 FROM Connections
) AS allConnections ON u.userId = allConnections.userId
JOIN Connections c ON c.userId1 = u.userId OR c.userId2 = u.userId
GROUP BY u.userId, u.fullName, u.joinDate
HAVING COUNT(c.connectionId) > 2
ORDER BY u.joinDate ASC;
```

Results | Messages

	userId	fullName	joinDate	totalConnections
1	tomas.navarro	Tomás Navarro	2020-09-25	4
2	carlos.ruiz	Carlos Ruiz	2021-08-10	4
3	elena.munoz	Elena Muñoz	2022-03-17	4
4	ana.gomez	Ana Gómez	2022-05-01	4
5	marta.lopez	Marta López	2023-01-12	4
6	laura.sanchez	Laura Sánchez	2023-07-21	4
7	paula.moreno	Paula Moreno	2025-04-15	4

Figure 42. SQL query for task 9

We can use complex queries to retrieve data and find different relationships between data, format data

- **List all users with the number of posts they've written, the number of reactions they have received, and how many connections they have.**

```sql
SELECT
    u.userId,
    u.fullName,
    COUNT(DISTINCT p.postId) AS totalPosts,
    COUNT(DISTINCT r.reactionId) AS totalReactionsReceived,
    COUNT(DISTINCT c.connectionId) AS totalConnections
FROM Users u
LEFT JOIN Posts p ON u.userId = p.userId
LEFT JOIN Reactions r ON p.postId = r.postId
LEFT JOIN Connections c ON u.userId = c.userId1 OR u.userId = c.userId2
GROUP BY u.userId, u.fullName
ORDER BY totalReactionsReceived DESC;
```

Results | Messages

	userId	fullName	totalPosts	totalReactionsReceived	totalConnections
1	ana.gomez	Ana Gómez	2	4	2
2	carlos.ruiz	Carlos Ruiz	1	2	2
3	marta.lopez	Marta López	2	2	2
4	laura.sanchez	Laura Sánchez	1	1	2
5	tomas.navarro	Tomás Navarro	1	1	2
6	elena.munoz	Elena Muñoz	1	1	2
7	javier.torres	Javier Torres	0	0	1
8	andres.perez	Andres Perez	0	0	1
9	paula.moreno	Paula Moreno	0	0	2

Figure 43. SQL query for task 10

- **Show user information with formatted name, email domain, join year, weekday, and time since joined**

```sql
SELECT
    userId,
    UPPER(fullName) AS upperName,
    SUBSTRING(email, CHARINDEX('@', email) + 1, LEN(email)) AS emailDomain,
    YEAR(joinDate) AS joinYear,
    DATENAME(MONTH, joinDate) AS joinMonth,
    DATENAME(WEEKDAY, joinDate) AS joinWeekday,
    DATEDIFF(YEAR, joinDate, GETDATE()) AS yearsOnPlatform
FROM Users
WHERE email IS NOT NULL
ORDER BY joinYear;
```

Results | Messages

	userId	upperName	emailDomain	joinYear	joinMonth	joinWeekday	yearsOnPlatform
1	tomas.navarro	TOMÁS NAVARRO	sec.com	2020	September	Friday	5
2	carlos.ruiz	CARLOS RUIZ	xyz.com	2021	August	Tuesday	4
3	elena.munoz	ELENA MUÑOZ	mail.com	2022	March	Thursday	3
4	ana.gomez	ANA GÓMEZ	abc.com	2022	May	Sunday	3
5	laura.sanchez	LAURA SÁNCHEZ	mail.com	2023	July	Friday	2
6	marta.lopez	MARTA LÓPEZ	mail.com	2023	January	Thursday	2

Figure 44. SQL query for task 11

7. Challenges faced and solutions applied

Throughout the development of this project, several challenges emerged that required careful problem-solving. One major challenge was designing a normalized schema from scratch, starting from a denormalized table that combined user, post, and reaction data. To solve this, I applied normalization techniques up to 3NF, creating separate tables with proper primary and foreign keys to preserve data integrity.

Another challenge involved enforcing constraints such as UNIQUE, CHECK, and referential integrity through foreign keys. These often led to errors during data insertion (e.g., foreign key violations or conflicts due to multiple cascade paths). I resolved these by adjusting the order of data population and rethinking the ON DELETE behavior depending on the relationship logic.

Additionally, retrieving meaningful insights through complex SQL queries — involving JOINs, GROUP BY, subqueries, and built-in functions — required several iterations to ensure performance and accuracy. I addressed this by progressively testing smaller queries and building them up with confidence. The use of LEFT JOIN and aggregate functions like COUNT, AVG, and DATEDIFF allowed me to derive useful analytics from the dataset.

8. Conclusion

This report presents the design, implementation, and querying of a relational database that simulates a simplified version of a professional social network. It reflects the application of key database concepts such as normalization, data integrity through constraints, and the use of SQL for both data manipulation and analysis.

Although the database contains a limited amount of data, it successfully demonstrates the structure and logic of a relational model. The schema design and implemented queries provide a clear example of how relationships between entities can be defined and leveraged to extract relevant insights, even in small-scale environments.