Personal Information Management Database

Presented to Dr. Ali Jannatpour

SOEN 363-S

Data Systems for Software Engineers

1. Shabia Saeed - 40154081

2. Abdullah Amir - 40215286

3. Barthan Thirunavukarasu - 40098158

November 24, 2023

TABLE OF CONTENTS

1. Introduction

In the initial phase of our database project, we have chosen to focus on the Personal Information Management database as our central topic. This decision is based on the importance of efficiently managing and accessing personal data in today's digital world.

The objective is to develop a relational database that will source data from two different public APIs. For this purpose, we have chosen MySQL as the database implementation platform. This choice aligns with our requirement of managing a large dataset, projected to be near 1 GB.

Through this project, we aim to demonstrate best practices in database design and implementation. This report outlines our approach to creating an efficient and meaningful relational database for personal information management. It details the system overview, data model, data population strategies, the challenges faced, and the solutions adopted during the project.

2. System Overview

In this section, we discuss and justify the design choices of our database. The APIs we used are fakerapi and random-data-api, and although they provide similar data, each API uses its own keys in the form of integer id and uuid values; to rectify this, we used internal integer keys in our schema (int auto_increment in MySQL). Here is a DDL example of internal integer keys:

```
CREATE TABLE Address (
    id INT PRIMARY KEY AUTO_INCREMENT,
                ...
);
```

In order to capture an IS-A relationship, we created a table for payment cards, and had two tables (credit cards and debit cards) that reference the payment card table; credit cards and debit cards have different attributes to them, and is thus better represented with an IS-A relationship as opposed to one large card table which would have lots of null values. Here is a DDL example demonstrating this:

```
CREATE TABLE Card (
      id INT PRIMARY KEY AUTO_INCREMENT,
      personID INT NOT NULL,
     number VARCHAR(20) NOT NULL,
      expiration DATE NOT NULL,
     type VARCHAR(50) NOT NULL,
    UNIQUE (number),
    FOREIGN KEY (personID) REFERENCES Person(id) ON DELETE       CASCADE
);
```

```sql
CREATE TABLE CreditCard (
    id INT PRIMARY KEY AUTO_INCREMENT,
    cardID INT NOT NULL UNIQUE,
    creditLimit DECIMAL(10, 2),
    availableCredit DECIMAL(10, 2),
    FOREIGN KEY (cardID) REFERENCES Card(id) ON DELETE CASCADE
);
```

```sql
CREATE TABLE DebitCard (
    id INT PRIMARY KEY AUTO_INCREMENT,
    cardID INT NOT NULL UNIQUE,
    balance DECIMAL(10, 2),
    FOREIGN KEY (cardID) REFERENCES Card(id) ON DELETE CASCADE
);
```

Weak entities are entities whose keys consist not only of its own attributes, but also that of other entities; in essence, weak entities do not fully form their own keys and rely on foreign keys. In our database, we have multiple weak entities, but here is one example:

```sql
CREATE TABLE LivesAtAddress (
    id int primary key auto_increment,
      personID INT,
    addressID INT,
    Unique (personID),
    FOREIGN KEY (personID) REFERENCES Person(id),
    FOREIGN KEY (addressID) REFERENCES Address(id)
);
```

To demonstrate the usage of triggers, we created one that checks the age of someone before inserting into the card table; the purpose of this trigger is to not allow insertions into the card table if the associated person is less than 18 years old. Here is a DML query that demonstrates this:

```
DELIMITER //
CREATE TRIGGER CheckAgeBeforeCardInsert
BEFORE INSERT ON Card
FOR EACH ROW
BEGIN
    DECLARE person_birthdate DATE;
    DECLARE current_age INT;

    SELECT birthday INTO person_birthdate
    FROM Person
    WHERE id = NEW.personID;

    SET current_age = TIMESTAMPDIFF(YEAR, person_birthdate, CURDATE());

    IF current_age < 18 THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Person must be at least 18
years old to own a credit card';
    END IF;
END//
DELIMITER ;
```

Views can greatly reduce the complexity of our queries, and it provides security by returning only the attributes we want. We use the following view to return a summary of each person in our database:

```
CREATE VIEW PersonContactDetails AS
SELECT
    Person.id,
    CONCAT(Person.firstname, ' ', Person.lastname) AS FullName,
    Person.birthday,
    Person.gender,
    Person.website,
    Email.emailAddress,
    PhoneNumber.phoneNumber,
    CONCAT(Address.streetName, ', ', Address.buildingNumber, ', ', Address.city, ',
', Address.zipcode, ', ', Address.country) AS FullAddress
FROM Person
LEFT JOIN Email ON Person.id = Email.personID
LEFT JOIN PhoneNumber ON Person.id = PhoneNumber.personID
LEFT JOIN LivesAtAddress ON Person.id = LivesAtAddress.personID
LEFT JOIN Address ON LivesAtAddress.addressID = Address.id;
```

As we have demonstrated above through various DDL examples, we used a variety of domains and types. We have also avoided using real domain data such as the id and uuid keys that fakerapi and random-data-api provide; instead, we opted for internal keys in the form of int auto_increment.

3. Data Model

The data model presented below illustrates the relationships and constraints within the database and depicts the interaction between the data components within the personal information management context.

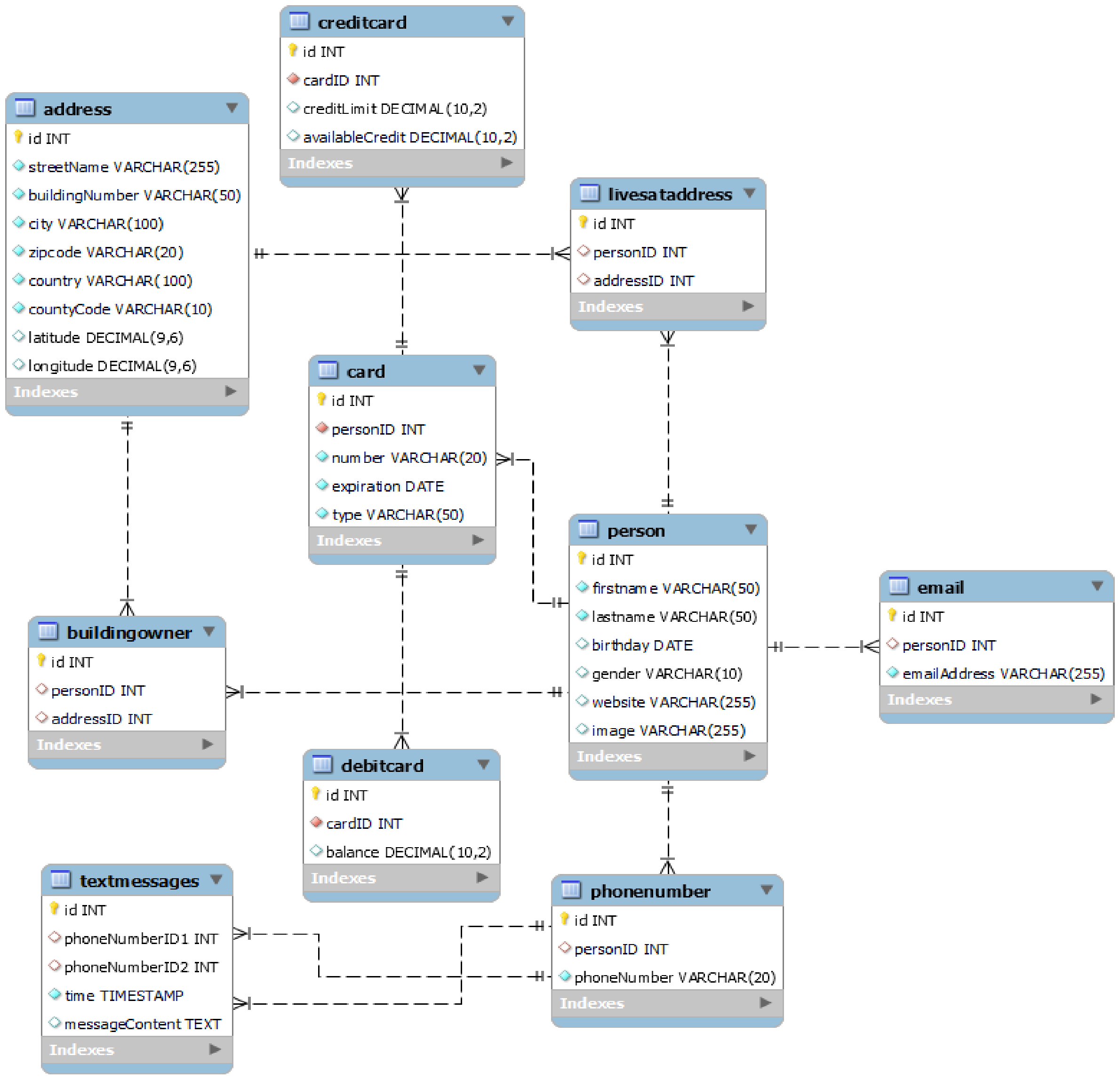


Figure 1: Data Model for Personal Information Management

4. Data Population, Management, and Challenges encountered

In designing our database, we faced numerous challenges related to topic selection and API choice, which significantly influenced our final design approach.

A. Initial Topic Fluctuations

During the project's phase one, we encountered the need to change our topic several times. This was primarily due to the limitations posed by the availability and constraints of public APIs:

Our design required us to establish a database of approximately 1GB in size. However, we found that larger, popular APIs often charge for extracting such extensive datasets, which contain vital information essential for modeling purposes. We initially explored using Spotify's API, particularly its search function, for creating a music database. However, this plan was hindered by the need for user consent to access and store user data, which presented a significant obstacle.

Our exploration extended to APIs like Last.fm as a secondary data source, which required manual issuance of API keys upon request. This process could take 1-2 weeks, a considerable delay given our total project timeline of about 3 weeks. Additionally, there was a risk of application rejection and limitations in data requests that hindered reaching our 1GB data goal.

After thorough research, we discovered FakerAPI, an API that provides fake but realistic data for various entities. This discovery was pivotal, as FakerAPI met all our project requirements, including the ability to fetch up to 1000 tuples at a time, significantly speeding up the data retrieval process.

To accumulate a substantial amount of data, we established the textMessage relation within our database. This relation is specifically designed to store messages exchanged between two unique phone IDs, along with the message text. Each record in the textMessage table comprises identifiers for the sender and receiver, represented by their phone IDs, and the text of the message itself. FakerAPI enables us to fetch text data up to 5000 characters in length, which is particularly advantageous for generating realistic and varied text message content and increasing the size of our database. Additionally, its capability to retrieve up to 1000 entries in a single request is critical in efficiently gathering a large volume of data. By utilizing the text data fetching capabilities of FakerAPI, we can generate extensive text message data, pivotal in reaching the 1GB database size target.

We used FakerAPI to populate most of our database. Its features and capabilities are aligned with our needs, enabling us to overcome previous hurdles. To fulfill our requirement of utilizing two different APIs, we integrated RandomAPI alongside FakerAPI. RandomAPI offered similar services and was instrumental in completing our database schema. This integration ensured that we not only met our database size requirements but also provided a diverse and realistic simulation of data.

B. How Data is Inserted:

In our database, the insertion of data is carefully structured to mimic realistic data collection and association. We focus on creating a comprehensive representation for each individual before proceeding to the next. Here's how we approach this:

- Comprehensive sequential Entity Creation: For each person, we aim to assign an email address, phone number, a type of card (credit or debit), an address, and a record of whether they own their residence. Additionally, we maintain a log of text messages exchanged between two individuals.
- Consistency and Order: To maintain realism, we insert data for one complete entity across all relevant tables before proceeding to the next entity. This approach avoids unrealistic scenarios like creating multiple persons without their corresponding details.
- Schema-Driven Insertion Order: The order of data insertion follows the order in which tables are created in our schema. This is crucial due to the constraints imposed by foreign key relationships.
- Entity Relationships and Constraints:

    - Emails and Phone Numbers: A person can have zero to many emails and phone numbers, linked via foreign keys.

    - Card Ownership: A person can have zero to many cards (credit or debit), with each card being either a credit or debit card, as determined through inheritance. The card's ID must exist in the card table.

    - Residential Addresses: Each person lives at one, and only one, address that already exists in the address table, enforced by unique person IDs and foreign key constraints.

    - Multiple Residents: It's possible for multiple individuals to reside at the same address, which we allow by having multiple tuples with the same address ID.

    - Text Messaging: To log text messages, both the sender and receiver must exist in the person table, enforced by foreign key relationships.

This methodical approach to data insertion ensures that our database not only adheres to the necessary relational constraints but also realistically represents how data would be collected and associated in a real-world scenario.

C. Data duplication handling:

Our system employs a robust mechanism to identify and prevent the insertion of duplicate data. We use an auto-increment feature for our internal integer primary keys. However, this presents a unique challenge: when a duplicate entry is blocked, the auto-incremented ID still advances, leading to non-sequential IDs. For instance, if an insertion for ID = 2 is a duplicate and thus not inserted, our sequence would have a gap, jumping from ID = 1 to ID = 3. To mitigate the impact of these gaps, our approach is to adhere strictly to a comprehensive and sequential entity creation strategy. This ensures that despite any ID sequence discrepancies, the assignment of foreign keys remains accurate, maintaining the integrity of relationships across tables.

Furthermore, since FakerAPI is free and has a limited range of data, it occasionally generates the same person entity more than once. In a dynamic database environment, this could lead to potential data redundancy. To address this, we've developed a script that intelligently handles insertion failures due to duplicates. When a duplication error occurs during insertion, our script performs a select query to obtain the current internal key ID, which is then used for linking subsequent related entries like email addresses, phone numbers, and cards to the correct person entity as a person may be associated with multiple of them.

In addition, our system is designed to minimize unnecessary data modifications. If a duplicate is detected and no further data associations are required, the existing values are retained to reduce overhead and maintain data consistency. Instead, we pass to the next iteration of entities creation.

D. Foreign key references handling

We leverage SQL's capability to fetch the ID of the last inserted row. This feature is crucial for associating internal integer keys across different tables within our database. For example, upon creating a new person entry, we immediately capture the ID of this insertion. This ID then plays a pivotal role in associating the person with other entities like emails, phone numbers, and addresses… in their respective tables.

Such design allows our database design to be inherently flexible, allowing for data collection processes to be paused or resumed at any iteration. This flexibility is not only practical but also mimics realistic scenarios where data collection can vary in pace and volume.

In summary, our database's approach to managing data duplication, foreign key references, and sequential entity creation is comprehensive as it ensures data integrity and realistic representation, even when faced with the constraints of limited data variety from external APIs like FakerAPI where duplications are possible.

E. Manually added features using randomness

To fulfill the schema's inheritance requirement, we created a 'card' entity where each card can be either a credit card or a debit card. The APIs we used provided data only for credit cards. To address this, we employed Python's random library for FakerAPI and the math.random function in Node.js for RandomAPI:

Customization of Card Entity:

We randomly designated each card as either debit or credit. Additionally, we assigned a 'limit' to each card which can be interpreted as a balance for debit cards and a credit limit for credit cards, with values in multiples of 1000. Furthermore, for credit cards, we randomly set an available limit that realistically reflects credit limits.

Building Ownership Assignment:

Another feature in our model is that a person at an address could be a renter, an owner, or a multimillionaire owning multiple buildings, with respective probabilities of 70%, 29.5%, and 0.5%. For renters, no additional data is added to the BuildingOwner table. Assigning an owner involves linking the person to the address created in that iteration of the comprehensive creation. We maintain a list of unowned buildings, which grows with each renter iteration as they don't have the building at the address where they live. When a multimillionaire person is created, they acquire all unowned buildings in the list which resets the list to empty. In cases where multimillionaires are created sequentially, the first multimillionaire receives all unowned buildings, while subsequent ones only acquire the building of their iteration, like regular owners. We also pull 1000 person per request we make to API. In order, to leave no buildings in the list of unowned buildings, the last person is always treated as a multimillionaire person if the list is not empty.

<u>Randomization in Credit Card Assignment:</u>

To illustrate scenarios where individuals own multiple credit cards, we randomly selected some persons (using their internal integer keys) and assigned them additional credit cards. This created cases of users owning more than one card, enabling us to demonstrate complex queries like intersections that were asked in the project.

<u>Randomizing Text Message Interactions:</u>

Although we collected text message data from APIs, we needed to simulate conversations between two people. We used randomness to pair two individuals with a message. For a message to be unique, the combination of the timeframe and both persons' IDs must be unique. As we don't perform parallel insertions, each message can be treated as part of a sequential conversation and will always be unique.

By sorting messages between two person IDs and maintaining sequential insertion, our system effectively simulates a messaging service used in platforms like Instagram, where messages follow a chronological order.

In summary, these strategies for handling specific database constraints not addressed by the APIs have enabled us to create a more realistic and functional simulation. The use of randomization, combined with careful planning and execution, ensures that our database can accommodate a variety of scenarios and queries, closely mimicking real-life data interactions.

F. Formatting text message insertion data

One of the challenges we encountered with the textMessage data from FakerAPI involved handling apostrophes in the text. In SQL, an apostrophe is often interpreted as a delimiter for string literals, which can cause issues during data insertion. When the SQL compiler encounters an apostrophe within a text message, it mistakenly assumes the end of the string, leading to syntax errors or incomplete data insertion.

To effectively resolve this issue, we implemented a simple yet efficient solution: replacing each single apostrophe in the text data with two apostrophes. In SQL, two consecutive apostrophes are recognized as a literal apostrophe within a string, rather than as a delimiter. This adjustment ensures that the SQL compiler correctly interprets the apostrophes as part of the text content and not as indicators of the string's end.

This approach allowed us to preserve the original text of the messages, including the apostrophes, without disrupting the SQL queries. By making this small modification to the text data before insertion, we maintained the integrity and readability of the text messages in the TextMessage table.

G. Linking data between two data sources

In aligning with our professor's confirmation, we realized that our task of integrating data from two sources, specifically FakerAPI and RandomAPI, didn't require a comprehensive linkage. A partial connection between these datasets was deemed sufficient. This understanding guided our focus toward a particular aspect of our database: the entities Card, CreditCard, and DebitCard.

To achieve this, we tapped into the credit card data provided by RandomAPI, selectively extracting only the relevant fields that matched our requirements. However, we encountered a familiar limitation, and much like with FakerAPI, RandomAPI's dataset only featured credit cards, leaving a gap in the representation of debit cards.

To creatively navigate this challenge, we first extracted the essential details to construct the card entity. Then, taking a cue from our earlier strategy with FakerAPI, we randomly assigned each card as either a credit or a debit card. This method of random assignment was crucial. It injected a level of realism and variety into our dataset, accurately reflecting the varied nature of card types one might encounter in the real world.

5. Data Migration

A. Exploring the Transition from Relational to NoSQL Databases: The Neo4j Experience

In this detailed overview, we explore the intricate process of migrating our existing relational database into a NoSQL framework, specifically focusing on our experience with Neo4j. This decision was informed by our previous work with Neo4j in Assignment 3, which provided us with valuable insights into its capabilities and features.

Neo4j, a graph database at its core, introduces a paradigm shift in how data is stored and managed. It excels in handling nodes and relationships, and its support for array attributes allows for an effective representation of weak entities. Additionally, Neo4j's ability to accommodate additional nodes labeling which is a critical aspect in representing inheritance in our data model.

B. Migrating from MySQL to Neo4j

Our migration journey started with the conversion of our MySQL database into a format compatible with Neo4j. This entailed exporting our data into CSV files, with each file representing a distinct table from the relational database. A key transformation was noted in the handling of email addresses, which were previously dependent on individual person entities. In Neo4j, these email addresses were efficiently incorporated as array attributes within the person entity. In figure 4, the entity Person contains the emails represented in blue.

C. Structural Transformations and Relationship Modeling

One of the transformative aspects of our migration was the redesign of how living addresses and building ownership data were handled. Unlike relational databases that rely on tables to store relationships, Neo4j simplifies this through direct node-to-node relationships. The credit and debit card data also underwent a significant transformation. To represent the 'is-a' inheritance relationship requirement from our previous assignments, we added distinct labels to each ‘Card’ node in Neo4j, where each label carried specific attributes, simplifying our inheritance model. The additional labels were ‘Credit Card’ which gave ‘available credit’ and ‘credit limit’ and ‘Card’ which gave ‘balance’ as additional attribute(s). The entity card is represented in figure 4. The additional labeling for credit cards and its attributes is shown in blue and for debit cards in red.

The phone entity presented a unique challenge. Despite its singular attribute being phone number, it was not feasible to treat it as a weak entity because of its connection with text messages. To address this design and not lose any data, we cannot store it as an array attribute. If it were to be stored as an array attribute in Person entity, it would be impossible to track down from which phone number the Person text to the other phone number. This would be a flawed design as we will be losing crucial information which was handled in our relational database. Therefore, we modeled the phone as a separate node, with relationships delineating ownership by a person and the ability to send text messages to other phones. The 'SENDS_TEXTS' association was designed to include both the message and timestamp details as shown in figure 2 and 3.

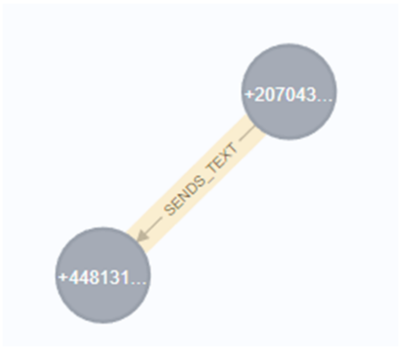
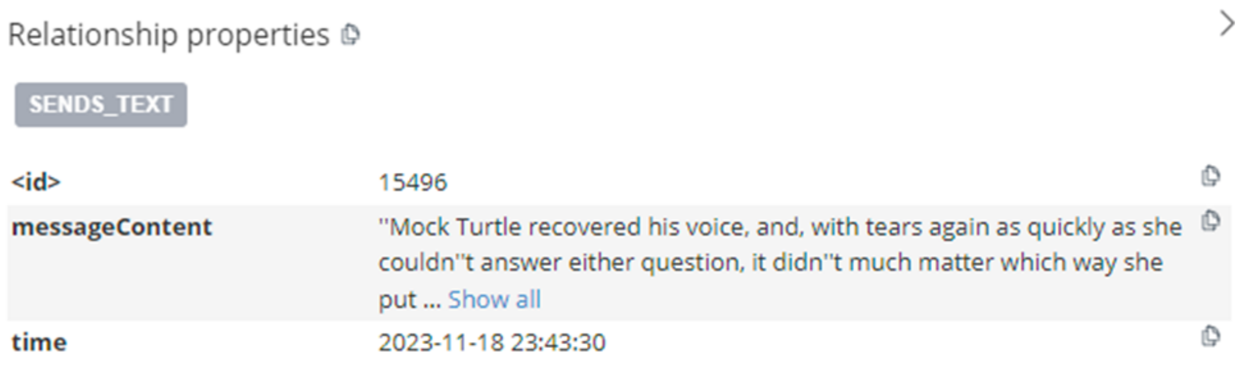


Figure 2: Representation of the SENDS_TEXT association in neo4j database between 2 phone number nodes of the database.



Relationship properties

SENDS_TEXT

<id>	15496
messageContent	"Mock Turtle recovered his voice, and, with tears again as quickly as she couldn''t answer either question, it didn''t much matter which way she put ... Show all
time	2023-11-18 23:43:30

Figure 3: Representation of the SENDS_TEXT relationship properties in neo4j database. The relationship contains message content and the timestamp.

D. Final Schema and Key Insights

Our final schema in Neo4j marked a significant departure from our relational model, particularly in the elimination of internal integer keys. These keys, while crucial in relational databases for table joins, are rendered unnecessary in a graph database where nodes are inherently interconnected. While we used these keys to appropriately associate nodes with each other, in the post-migration, we removed these keys, aligning with Neo4j's approach to data management. The figure 4 represents the final schema of our NoSQL database:

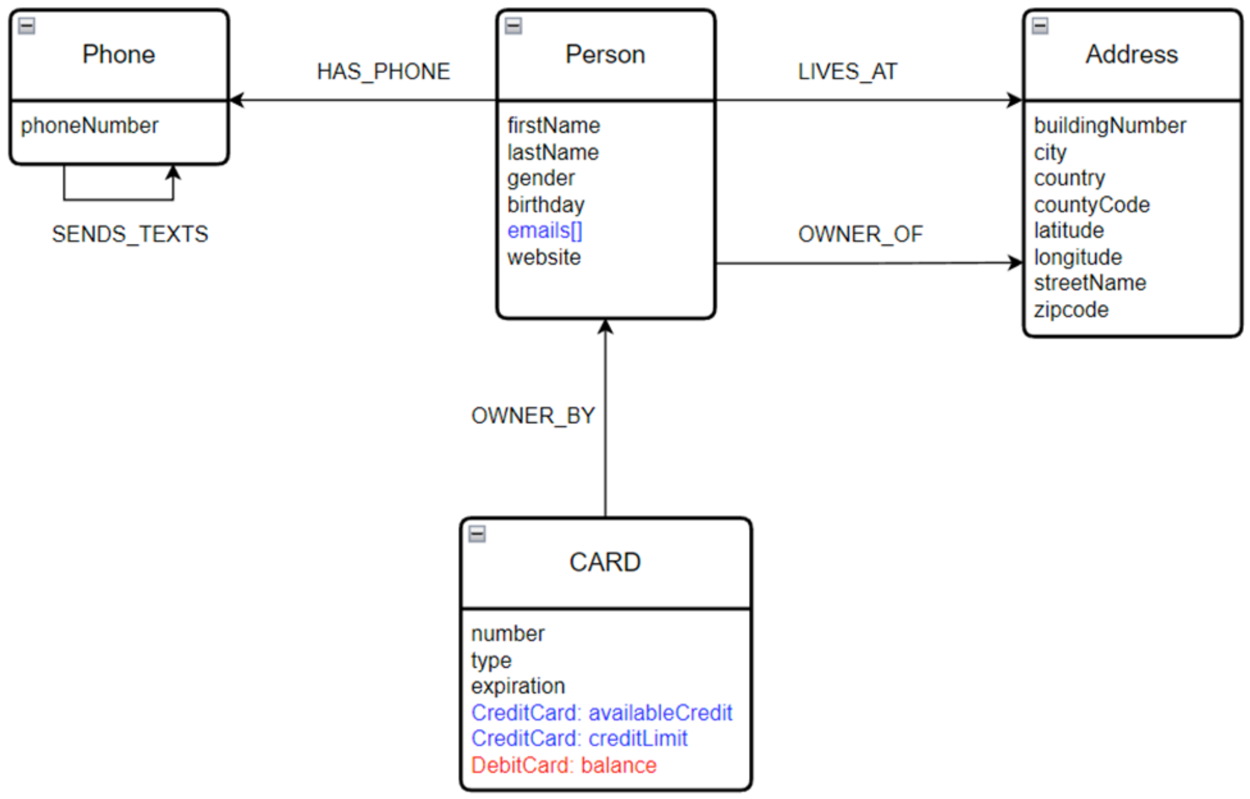


Figure 4: The entities and relationship schema for the neo4j database.

An important aspect to note is the presence of unique internal key IDs for each node in Neo4j. These IDs, automatically assigned, play a vital role in differentiating each node within the database. Although the internal mechanics of these IDs are managed by Neo4j, their presence underscores the database's robust structure.

6. Conclusion

To conclude, in designing our database, we faced challenges with topic selection and API limitations, leading us to discover FakerAPI. This API provided the diverse, realistic data we needed and supported our goal to create a 1GB database which was specifically achieved with a load of data in the TextMessage relation.

Our data insertion strategy was structured to simulate real-world data collection and ensured details and sequential entry of information across various entities, including emails, phone numbers, addresses, cards…

We also develop methods to handle data duplication and foreign keys references. Due to the auto-increment nature of the relational database used, we implemented a comprehensive entity creation approach to maintain consistency in foreign key assignments. Additionally, we designed a script to manage potential redundancies fetched from APIs.

To enhance our database, we manually added features using randomness achieved with programming languages. Such examples include card types, their limits, and managing building (addresses) ownership probabilities. This approach brought a realistic diversity to our database.

Furthermore, we addressed the formatting challenges in text message data, specifically handling apostrophes for SQL compatibility. This ensured the integrity and readability of our text messages.

Lastly, we linked data between FakerAPI and RandomAPi, focusing on card-related entities. Despite limitations in data variety, we extracted necessary details and randomly assigned card types (credit or debit) to add realism to our dataset.

As for MySQL, we got hands-on experience with designing a database. We started by creating schemas, the foundation of the project. After fully collecting the data, we then designed queries as asked in the project and delved into creating complex triggers and views. These were crucial steps in understanding automated processes and data integrity within our database and have deepened our understanding of database management in practical scenarios.

The transition from a relational database to Neo4j marked a pivotal shift in our data management approach. This migration involved reimagining data structures, moving from rigid table-based relationships to a more dynamic, node-based system in Neo4j. The process highlighted the importance of preserving data integrity and adapting to the graph database's unique capabilities. Our final Neo4j schema, characterized by its departure from traditional relational models, reflects a significant advancement in our database management learning, equipping us with valuable insights for future projects.

Overall, these strategies enabled us to create a comprehensive, and realistic database meeting all the project's requirements.

7. References

1. https://fakerapi.it/en (Main data source of data fetching)
2. https://random-data-api.com/ (Secondary data source of data fetching)
3. https://chat.openai.com/ (To improve writing in the report)
4. https://quillbot.com/ (To improve writing in the report)
5. https://github.com/public-apis/public-apis (To do general research on public APIs)
6. https://www.mysql.com/ (Relational database used for the project)
7. https://www.geeksforgeeks.org/sql-division/ (To understand sql-division)
8. https://moodle.concordia.ca/moodle/course/view.php?id=158538 (SOEN 363 course material)