REPUBLIC OF CAMEROON

PEACE - WORK – FATHERLAND

MINISTRY OF HIGHER EDUCATION

REPUBLIQUE DU CAMEROUN

PAIX – TRAVAIL – PATRIE

MINISTERE DE L'ENSEIGNEMENT SUPERIEUR

UNIVERSITY OF BUEA

FACULTY OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER ENGINEERING

CEF440

INTERNET PROGRAMMING AND MOBILE PROGRAMMING

BY

GROUP 25

NAME	MATRICULE
FOFIE FOPA ELISABETH	FE21A189
MBISHU FABRICE YENVEN	FE21A232
EDI EDISON FORNANG	FE21A178
TAKOH CLOVERT NFUA	FE21A311
KIMBI CYRIL BONGNYU NFOR	FE21A216

COURSE INSTRUCTOR: DR NKEMINI Valery **JUNE 2024**

TABLE OF CONTENT

DATABASE DESIGN AND IMPLEMENTATION 3

1. INTRODUCTION 3

2. RELATIONAL SCHEMA 3

2.1 CUSTOMUSER 3

2.2 ALERTCHOICES 3

2.3 LOCATION 3

2.4 ALERT 4

2.5 DISASTER FEEDBACK 4

3. NORMALIZATION 4

3.1 FIRST NORMAL FORM (1NF) 4

3.2 SECOND NORMAL FORM (2NF) 5

3.3 THIRD NORMAL FORM (3NF) 5

4. ENTITY-RELATIONSHIP DIAGRAM (ERD) 5

5. RELATIONSHIPS IN THE DATABASE 6

1. **CUSTOMUSER AND ALERT** 6
2. **ALERTCHOICES AND ALERT** 6
3. **LOCATION AND ALERT** 6
4. **ALERT AND DISASTERFEEDBACK** 7

6. SQL SCHEMA DEFINITIONS 7

7. CACHING NEEDS IN THE DATABASE WITH REDIS 9

1. **FREQUENTLY ACCESSED TABLES** 9
2. **USER SESSION DATA** 9
3. **RECENT ALERTS** 9
4. **FEEDBACK FOR RECENT ALERTS** 9
5. **STATISTICS AND AGGREGATED DATA** 9
6. **IN-MEMORY CACHING WITH REDIS** 10
7. **TIME-BASED CACHING** 11
8. **QUERY CACHING** 11
9. **APPLICATION-LEVEL CACHING** 11

DATABASE DESIGN AND IMPLEMENTATION

1. INTRODUCTION

This document outlines the design of a database system for an emergency management system. It includes a detailed relational schema, normalization process, entity-relationship diagram (ERD), and SQL schema definitions. The system includes functionalities for user management, emergency reporting, safety information, real-time incident reporting, resource management, alerts and notifications, and more.

2. RELATIONAL SCHEMA

This section provides the relational schema for the classes and attributes described in the system.

2.1 CUSTOMUSER

- **userId** (Primary Key): INT - Unique identifier for the user.
- **username**: VARCHAR(150) - Username for the user.
- **password**: VARCHAR(128) - Password for user authentication.
- **userType**: VARCHAR(50) - Type of user (e.g., Normal User, Emergency Agent, etc.).
- **phone number**: VARCHAR(15) - Phone number of the user.
- **otp**: VARCHAR(100) - One-time password for the user.

2.2 ALERTCHOICES

- **alertChoiceId** (Primary Key): INT - Unique identifier for the alert choice.
- **emergency _name**: VARCHAR(25) - Name of the emergency type.

2.3 LOCATION

- **locationId** (Primary Key): INT - Unique identifier for the location.
- **longitude**: FLOAT - Longitude of the location.

- **latitude**: FLOAT - Latitude of the location.

2.4 ALERT

- **alertId** (Primary Key): INT - Unique identifier for the alert.
- **userId** (Foreign Key): INT - Identifier for the user raising the alert.
- **alertChoiceId** (Foreign Key): INT - Identifier for the type of alert.
- **date time of _alert**: DATETIME - Timestamp of the alert.
- **locationId** (Foreign Key): INT - Identifier for the location of the alert.
- **description**: TEXT - Description of the alert.
- **first aid response**: TEXT - First aid response details.

2.5 DISASTER FEEDBACK

- **feedbackId** (Primary Key): INT - Unique identifier for the feedback.
- **alertId** (Foreign Key): INT - Identifier for the related alert.
- **description**: TEXT - Description of the feedback.
- **date time of _feedback**: DATETIME - Timestamp of the feedback.

3. NORMALIZATION

The database schema is normalized to reduce redundancy and improve data integrity. The tables are designed to be in the third normal form (3NF).

3.1 FIRST NORMAL FORM (1NF)

All tables have a primary key and each field contains only atomic values.

3.2 SECOND NORMAL FORM (2NF)

All non-key attributes are fully functionally dependent on the primary key. Composite keys are split into separate tables.

3.3 THIRD NORMAL FORM (3NF)

All attributes are functionally dependent only on the primary key. Transitive dependencies are eliminated.

4. ENTITY-RELATIONSHIP DIAGRAM (ERD)

The ERD provides a visual representation of the database structure and relationships between the entities.

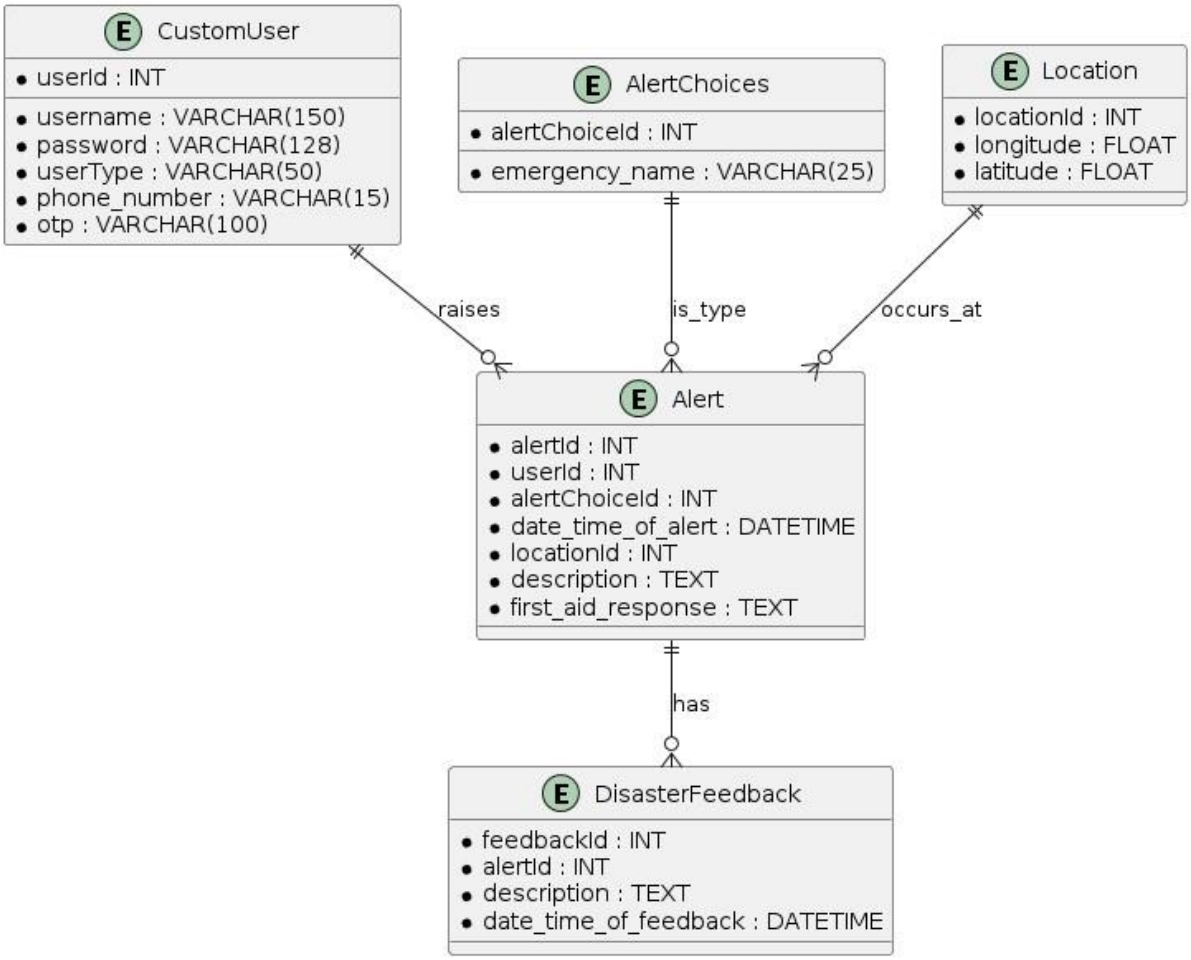

Figure 1: ER Diagram

5. RELATIONSHIPS IN THE DATABASE

The relationships between the entities in our database are fundamental to ensuring that the system accurately captures the interactions between users, alerts, locations, and feedback. Below, we detail each relationship and its significance.

1. CUSTOMUSER AND ALERT

Relationship: One-to-Many

Type: CustomUser ||--o{ Alert : raises

Description: This relationship indicates that each user can raise multiple alerts, but each alert is associated with only one user. This is crucial for identifying the source of each alert and ensuring accountability. When a CustomUser raises an alert, their unique userId is recorded in the Alert table. This connection allows the system to track which user reported each incident, facilitating follow-up actions and communication. For instance, if an alert needs to be verified or additional information is required, the system can easily reference the CustomUser who raised the alert.

2. ALERTCHOICES AND ALERT

Relationship: One-to-Many

Type: AlertChoices ||--o{ Alert : is_type

Description: Each type of emergency that can be reported (e.g., fire, flood) is recorded in the AlertChoices table. An Alert is linked to one of these choices through the alertChoiceId. This relationship ensures that every alert is categorized correctly, enabling efficient response and data analysis. By having a predefined set of alert types, the system standardizes the kinds of emergencies that can be reported, making it easier to organize response efforts and analyze trends in alert data. For example, the system can quickly generate reports on the frequency of different types of emergencies, helping to allocate resources where they are most needed.

3. LOCATION AND ALERT

Relationship: One-to-Many

Type: Location ||--o{ Alert : occurs_at

Description: Alerts are linked to specific geographical locations, which are stored in the Location table. Each Alert is associated with one Location through the locationId, indicating where

the incident occurred. This relationship is vital for spatial analysis and response planning. By associating alerts with specific locations, the system can map incidents, identify hotspots, and deploy resources effectively. For instance, if multiple alerts originate from the same location, it may indicate a recurring issue that requires further investigation or a more robust response strategy.

4. ALERT AND DISASTERFEEDBACK

Relationship: One-to-Many

Type: Alert ||--o{ DisasterFeedback : has

Description: This relationship links each alert to any feedback received regarding the incident. The DisasterFeedback table stores feedback entries that reference an alertId from the Alert table. This one-to-many relationship allows each alert to have multiple feedback entries, capturing various responses and observations from different users or responders. The feedback collected can be invaluable for assessing the effectiveness of the response and understanding the impact of the incident. For example, feedback can provide insights into the adequacy of the first aid response or highlight areas for improvement in emergency handling. This continuous loop of raising alerts and receiving feedback helps in refining emergency response protocols and improving overall disaster management.

Importance of Relationships

The defined relationships ensure that our database is well-structured and capable of capturing complex interactions within the system. They enable efficient data retrieval, meaningful analysis, and effective management of alerts and responses. By maintaining clear links between users, alerts, alert types, locations, and feedback, the database supports robust reporting and decision-making processes, ultimately enhancing the system's capability to manage emergencies and disasters effectively.

6. SQL SCHEMA DEFINITIONS

This section provides the SQL statements to create the database tables.

Listing 1: SQL Schema Definition

```
CREATE TABLE CustomUser ( userId INT PRIMARY
    KEY AUTO_INCREMENT, username
    VARCHAR(150) NOT NULL UNIQUE, password
    VARCHAR(128) NOT NULL, userType
    VARCHAR(50) NOT NULL, phone_number
    VARCHAR(15) NULL, otp VARCHAR(100)
    NULL UNIQUE
);

CREATE TABLE AlertChoices ( alertChoiceId INT PRIMARY
    KEY AUTO_INCREMENT, emergency_name
    VARCHAR(25) DEFAULT NULL
);

CREATE TABLE Location ( locationId INT PRIMARY KEY
    AUTO_INCREMENT, longitude FLOAT DEFAULT
    NULL, latitude FLOAT DEFAULT NULL
);

CREATE TABLE Alert ( alertId INT PRIMARY KEY
    AUTO_INCREMENT, userId INT, alertChoiceId INT DEFAULT
    NULL, date_time_of_alert DATETIME DEFAULT
    CURRENT_TIMESTAMP, locationId INT DEFAULT NULL,
    description TEXT,
    first_aid_response TEXT DEFAULT NULL,
    FOREIGN KEY (userId) REFERENCES CustomUser(userId),
    FOREIGN KEY (alertChoiceId) REFERENCES AlertChoices(alertChoiceId),
    FOREIGN KEY (locationId) REFERENCES Location(locationId)
);

CREATE TABLE DisasterFeedback ( feedbackId INT
    PRIMARY KEY AUTO_INCREMENT, alertId INT,
    description TEXT,
    date_time_of_feedback DATETIME DEFAULT
CURRENT_TIMESTAMP, FOREIGN KEY (alertId) REFERENCES
Alert(alertId) );
```

7. CACHING NEEDS IN THE DATABASE WITH REDIS

Caching can significantly improve the performance of our database system by reducing the time it takes to retrieve frequently accessed data. We utilize Redis for caching in various parts of the database. Below, we outline which parts of the database are cached and the specific strategies used.

1. FREQUENTLY ACCESSED TABLES

AlertChoices: This table contains predefined types of alerts. Since these types do not change frequently and are likely to be accessed often, we cache this table in Redis. This reduces the number of database reads and speeds up the process of categorizing alerts.

Location: Geographical locations are accessed frequently when querying or displaying alerts. We cache location data in Redis to improve performance, particularly since the location data set is relatively static.

2. USER SESSION DATA

CustomUser Information: When a user logs in, their profile information (e.g., username, userType, phone number) is cached in Redis. This reduces the need for repeated database queries and improves performance for user-specific operations during a session. Session data is typically cached with an expiration time to ensure data consistency.

3. RECENT ALERTS

Alerts: Recently raised alerts are cached in Redis, as they are frequently accessed by users and responders. We use a time-based caching strategy, storing alerts from the past hour or day to ensure quick access while maintaining data relevance.

4. FEEDBACK FOR RECENT ALERTS

DisasterFeedback: Feedback on recent alerts is also cached in Redis. This allows responders to quickly access and review feedback without imposing additional load on the database. Feedback entries are cached with a similar time-based strategy as alerts.

5. STATISTICS AND AGGREGATED DATA

Summary Reports for CRM: Our CRM system requires frequent access to statistics and aggregated data, such as the number of alerts by type or location, and average response times. We

cache these precomputed results in Redis to optimize performance. Cached data is periodically updated to reflect the most current statistics, ensuring the CRM system has access to up-to-date information.

Frequently Queried Aggregations: Common queries, such as the total number of alerts per day or the most common alert types, are precomputed and cached in Redis. This saves significant processing time and ensures rapid response times for these queries.

Implementation Details with Redis

We utilize Redis for caching the aforementioned parts of our database. Below are the implementation details for each caching strategy:

6. IN-MEMORY CACHING WITH REDIS

Setup: Redis is configured as an in-memory data store, optimized for fast read operations.

Caching Strategy:

- **AlertChoices and Location**: Data is cached indefinitely since changes are rare. Manual cache invalidation is performed when updates occur.
- **User Session Data**: Cached with a time-to-live (TTL) value to expire session data after a predefined period, ensuring data consistency and security.
- **Recent Alerts and DisasterFeedback**: Cached with a TTL to keep recent data accessible while discarding stale data after the defined period.
- **Summary Reports and Aggregations**: Cached data is updated periodically, using background jobs to refresh the cache at regular intervals.

Data Consistency: Cache invalidation and refresh mechanisms are implemented to ensure data consistency. For example, when new alerts are raised or feedback is provided, the corresponding cache entries are invalidated, and new data is fetched from the database.

7. TIME-BASED CACHING

For data that changes over time, such as recent alerts and feedback, we implement time-based invalidation strategies to ensure the cache is refreshed periodically. This ensures that users and responders have access to the most up-to-date information while maintaining optimal performance.

8. QUERY CACHING

We enable query caching on the database level for complex queries that are frequently executed but do not change often. By caching the results of these queries, we reduce the load on the database and improve response times for these operations.

9. APPLICATION-LEVEL CACHING

We implement caching at the application level for user session data and commonly accessed static data. Using libraries that integrate with the application framework, we manage cache lifecycle and invalidation efficiently. This approach ensures that the application remains responsive and scalable.