

**FACULTY ENGINEERING**

**FACULTY OF ENGINEERING AND TECHNOLOGY**

**DEPARTMENT OF COMPUTER ENGINEERING, SE**

**CEF440: Internet and Mobile Programming**



**Presented by:**

| BOBGALA HARRISON DIMA | FE21A151 |
| --- | --- |
| KAMCHE YANN ARNAUD | **FE21A208** |
| MESSI II INNOCENT ROSTAND | **FE21A236** |
| NYANDO ONONGWENE | **FE21A290** |
| NKWI CYRIL AKINIMBOM | **FE21A281** |



**Table of Content**

[**1. Requirement Analysis and Conceptual Design 2**](#_299iefckpwku)

[**2. Logical Design and Schema Creation 2**](#_493shep21mdc)

[**3. Physical Design and Indexing 2**](#_3rdcrjn)

[**4. Constraints and Validation Rules 2**](#_6v359glsx0uv)

[**5. Implementation and Testing 2**](#_cx4ntnt3p1gt)

# **Requirement Analysis and Conceptual Design**

### Overview

The increasing frequency and severity of natural and man-made disasters underscore the need for an effective disaster management system. A Disaster Management Mobile Application aims to streamline the process of reporting incidents, coordinating responses, and disseminating critical information to the public and emergency responders.

### Requirement Analysis

Requirement analysis is a crucial step in the system development lifecycle. It involves gathering and analyzing requirements from stakeholders to understand what data needs to be stored, how it will be used, and what the business rules are. This process ensures that the final system meets the needs of its users and operates efficiently.

### Responsibilities:

1. **Gather Requirements from Stakeholders**:
   * Conduct interviews, surveys, and workshops with stakeholders, including emergency responders, government agencies, and potential users of the application.
   * Collect existing documentation and reports related to disaster management.
   * Review similar disaster management systems for reference.
2. **Identify Key Entities**:
   * Determine the main entities involved in the disaster management process, such as users, incidents, resources, locations, and services.
3. **Define Entities, Attributes, and Relationships**:
   * Establish the necessary entities, their attributes, and the relationships between them to model the disaster management process accurately.
4. **Create an Entity-Relationship Diagram (ERD)**:
   * Develop a high-level conceptual model that outlines the main entities and their relationships. This step helps visualize the database structure without worrying about the specifics of the database management system (DBMS).

### Conceptual Design

Conceptual design involves creating a high-level model that defines the structure of the database without considering the technical details of the DBMS. This step is crucial for ensuring that the database can efficiently support the application's requirements.

#### **Main Entities and Attributes:**

* **User**:
  + userId: Unique identifier
  + name: Name of the user
  + contactInfo: Contact information
* **Incident**:
  + incidentId: Unique identifier
  + location: Location of the incident
  + type: Type of incident
  + description: Description of the incident
  + timestamp: Time of occurrence
* **Volunteer**:
  + volunteerId: Unique identifier
  + skills: Skills or expertise
  + availability: Availability status
* **EmergencyService**:
  + serviceId: Unique identifier
  + type: Type of service
  + contactInfo: Contact information
* **GovernmentAgency**:
  + agencyId: Unique identifier
  + name: Name of the agency
  + directives: Directives issued
* **WeatherDataProvider**:
  + providerId: Unique identifier
  + weatherData: Real-time weather data
* **AlertSystem**:
  + alertId: Unique identifier
  + message: Alert message
  + timestamp: Time of alert

#### **Relationships:**

* **User** reports **Incident**
* **Incident** involves **Volunteer**
* **Incident** requires response from **EmergencyService**
* **GovernmentAgency** manages **Incident**
* **WeatherDataProvider** supplies data for **Incident**
* **AlertSystem** notifies **User** about **Incident**

### Importance for Disaster Management System

A well-designed database is fundamental to the effectiveness of a disaster management system. It ensures that:

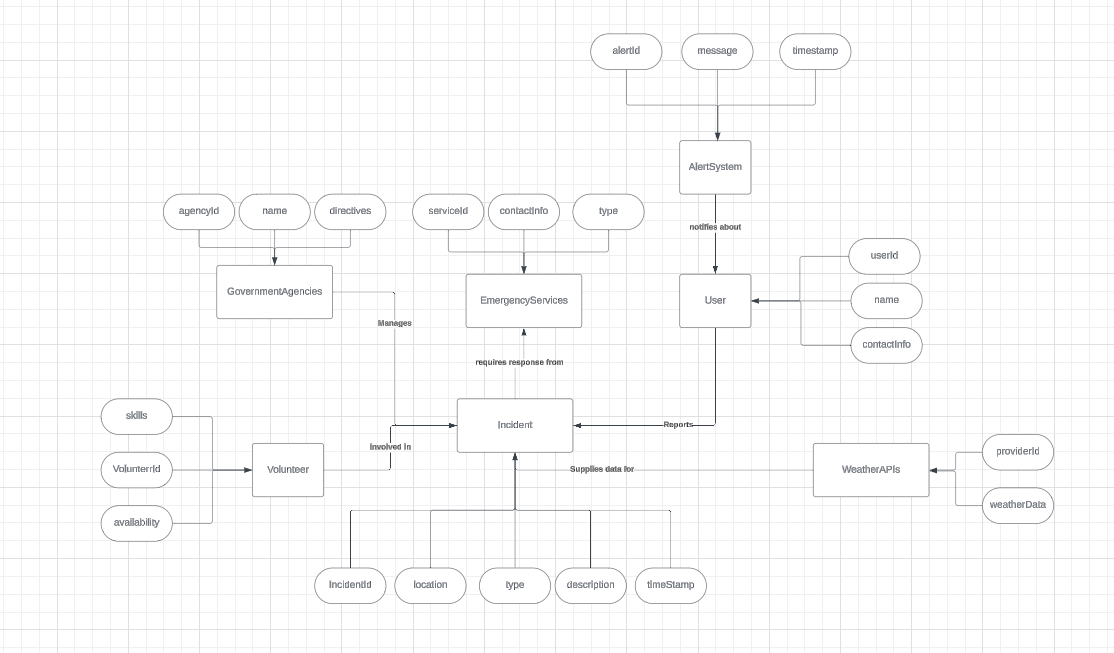
* **Data Integrity and Consistency**: Accurate and consistent data is maintained across the system, which is critical during emergencies.
* **Efficient Data Retrieval and Updates**: Fast access to information is crucial for timely responses to disasters.
* **Scalability**: The system can handle a growing amount of data as more incidents are reported and more users join the platform.
* **Improved Decision Making**: Reliable data supports better decision-making by emergency responders and government agencies.

### ER Diagram

An Entity-Relationship Diagram (ERD) is a graphical representation of the entities within a system and the relationships between those entities. It is a crucial part of database design that helps in visualizing the structure of the database.

#### **Components of an ERD:**

* **Entities**: These are objects or concepts about which data is stored. In a disaster management system, examples include User, Incident, Volunteer, EmergencyService, GovernmentAgency, WeatherDataProvider, and AlertSystem.
* **Attributes**: These are the properties or details of an entity. For instance, a User entity might have attributes such as userId, name, and contactInfo.
* **Relationships**: These define how entities interact with each other. For example, a User reports an Incident, and an Incident involves a Volunteer.



#### **Relationships:**

User reports Incident

Each user can report multiple incidents.

Each incident is reported by one user.

Incident involves Volunteer

Each incident can involve multiple volunteers.

Each volunteer can be involved in multiple incidents.

Incident requires response from EmergencyService

Each incident can require responses from multiple emergency services.

Each emergency service can respond to multiple incidents.

GovernmentAgency manages Incident

Each incident can be managed by multiple government agencies.

Each government agency can manage multiple incidents.

WeatherDataProvider supplies data for Incident

Each incident can receive data from multiple weather data providers.

Each weather data provider can supply data for multiple incidents.

AlertSystem notifies User about Incident

Each incident can generate multiple alerts.

Each alert is sent to multiple users.

Visual Representation

Here is a textual representation of the Entity-Relationship Diagram (ERD):

[User] ---reports---> [Incident]

[Incident] ---involves---> [Volunteer]

[Incident] ---requires response from---> [EmergencyService]

[GovernmentAgency] ---manages---> [Incident]

[WeatherDataProvider] ---supplies data for---> [Incident]

[AlertSystem] ---notifies about---> [User]

### Importance for a Disaster Management System

#### **1. Efficient Incident Reporting and Response:**

* A well-designed database ensures that incidents are reported, tracked, and responded to efficiently. The relationships between User, Incident, Volunteer, and EmergencyService are crucial for managing response efforts.

#### **2. Data Accuracy and Real-time Updates:**

* Accurate and up-to-date information is critical during disasters. An ERD helps in designing a database that maintains accurate records of incidents, resource availability, and communication logs.

#### **3. Integration with External Systems:**

* Disaster management systems often need to integrate with external systems like weather data providers and government databases. An ERD helps in designing the database to accommodate these integrations seamlessly.

#### **4. Scalability and Future Enhancements:**

* As the disaster management system evolves, the database will need to scale and possibly include new entities and relationships. An ERD provides a clear roadmap for such enhancements.

# **Logical Design and Schema Creation**

Logical database design is focused on turning basic ideas about what data should be stored into a more detailed plan. This plan is shown to illustrate how to organize the data, how different pieces of data are connected, and the rules for the data. The design does not depend on any specific software used to manage a database. Entity-Relationship Diagrams (ERDs) are often used to show how everything fits together.

Transforming the conceptual design (ERD) into a logical schema is done by refining the ERD into a more implementation-ready format that includes tables, columns, data types, and relationships appropriate for a relational database. The mapping of entities and relationships from the conceptual ERD into a logical schema is performed as follows:

**Logical Schema Design:**

- Each entity is transformed into a table, and each attribute into columns with specified data types.

- Relationships are implemented through foreign keys.

1. **User Table:**

- userId (INT, Primary Key)

- name (VARCHAR)

- contactInfo (VARCHAR)

2. **Incident Table:**

- incidentId (INT, Primary Key)

- location (VARCHAR)

- type (VARCHAR)

- description (TEXT)

- timestamp (DATETIME)

3. **Volunteer Table:**

- volunteerId (INT, Primary Key)

- skills (VARCHAR)

- availability (BOOLEAN)

4. **Emergency Service Table:**

- serviceId (INT, Primary Key)

- type (VARCHAR)

- contactInfo (VARCHAR)

5. **Emergency Contact Table:**

- contactId (INT, Primary Key)

- name (VARCHAR)

6. **Weather Data Provider Table:**

- providerId (INT, Primary Key)

- weatherData (TEXT)

7. **Government Agency Table:**

- agencyId (INT, Primary Key)

- name (VARCHAR)

- directives (TEXT)

8. **Alert System Table:**

- alertId (INT, Primary Key)

- message (TEXT)

- timestamp (DATETIME)

**Relationships with Foreign Keys include:**

- IncidentReports (to track incidents reported by users)

- VolunteerIncidentAssistance (to track volunteers assisting with incidents)

- EmergencyServiceResponse (to track emergency services responding to incidents)

- EmergencyContactNotification (to track notifications sent to emergency contacts for incidents)

- WeatherDataIncidents (to link weather data to specific incidents)

- GovernmentOversight (to track government oversight on incidents)

**Normalization** is done to ensure there is no unnecessary duplicate information and that the data is accurate. This includes:

1. First Normal Form (1NF): Each column in a table has only one value per row.

2. Second Normal Form (2NF): All information in a table that isn't the key depends completely on the key.

3. Third Normal Form (3NF): Non-key information doesn't depend on other non-key information.

4. Boyce-Codd Normal Form (BCNF): Every key actually qualifies as a key.

**Denormalization** is employed to make a database work faster by simplifying how data is retrieved:

- Redundant Data is added directly to tables where it’s often needed.

- Tables are merged if they are closely related and often accessed together.

- Precomputed Joins are set up to save time, especially for reports and analytics.

**Considerations for Optimization and Scalability include:**

- **Indexing**: Quick references are created for parts looked up often.

- **Partitioning**: Large tables are split into sections based on common attributes.

- **Caching**: Information used frequently but seldom changed is kept readily accessible.

- **Concurrency and Locking**: Strategies are devised to allow many users to update or delete data simultaneously without issues.

# **Physical Design and Indexing**

Physical design is the adjustment of the physical attributes of the model based on the logical model in order to optimize the database performance and improve the efficiency of service operation and application efficiency. The physical design should be adjusted in conjunction with the physical attributes of the target database product, with the ultimate goal of generating a deployable DDL for the target database.

# **Constraints and Validation Rules**

It is crucial to establish robust constraints and validation rules to ensure data integrity, consistency, and reliability.

**User Class**

**Constraints:**

- Primary Key: `userId` serves as the primary key to uniquely identify each user.

- Not Null: `userId`, `name`, and `contactInfo` are mandatory fields, ensuring all users have essential information stored.

**Validation:**

- Data Type: `userId` is an integer, `name` and `contactInfo` are strings, ensuring correct data types are enforced.

- Format: Validation rules can be applied to `contactInfo` to ensure it conforms to a valid format (e.g., email address or phone number).

**Volunteer Class**

**Constraints:**

- Primary Key: `volunteerId` uniquely identifies each volunteer.

- Not Null: `volunteerId`, `skill`, and `availability` are mandatory fields for each volunteer record.

**Validation:**

- Data Type: `volunteerId` is an integer, `skill` is a string, and `availability` is boolean, ensuring data consistency and accuracy.

**WeatherDataProvider Class**

**Constraints:**

- Primary Key: `providerId` uniquely identifies each weather data provider.

- Not Null: `providerId` and `weatherData` are mandatory fields.

**Validation:**

- Data Type: `providerId` is an integer, `weatherData` is a string, ensuring correct data types for weather information.

**WeatherData Class**

**Constraints:**

- Foreign Key: `providerId` establishes a relationship with `WeatherDataProvider.providerId`.

**Validation:**

- Referential Integrity: Ensure that `providerId` in `WeatherData` references an existing `providerId` in `WeatherDataProvider`.

**Incident (Disaster) Class**

**Constraints:**

- Primary Key: `IncidentId` uniquely identifies each disaster incident.

- Not Null: `IncidentId`, `location`, `type`, `description`, and `timestamp` are mandatory fields.

**Validation:**

- Data Type: `IncidentId` is an integer, `location`, `type`, and `description` are strings, `timestamp` is datetime.

- Business Rules: Validate that `timestamp` is a valid datetime format and other business-specific rules for incident reporting.

**EmergencyService Class**

**Constraints:**

- Primary Key: `serviceId` uniquely identifies each emergency service.

- Not Null: `serviceId`, `type`, and `contactInfo` are mandatory fields.

**Validation:**

- Data Type: `serviceId` is an integer, `type` and `contactInfo` are strings, ensuring correct data types and formats for emergency service information.

**EmergencyContact Class**

**Constraints:**

- Primary Key: `ContactId` uniquely identifies each emergency contact.

- Not Null: `ContactId`, `name`, and `contactInfo` are mandatory fields.

**2. Validation:**

- Data Type: `ContactId` is an integer, `name` and `contactInfo` are strings, enforcing correct data types and formats for emergency contact information.

**Message Class**

**Constraints:**

- Primary Key: `message` uniquely identifies each generated message.

- Not Null: `message` and `timestamp` are mandatory fields.

**Validation:**

- Data Type: `message` is an integer, `timestamp` is datetime, ensuring accurate storage of message generation details.

**GovernmentAgency Class**

**Constraints:**

- Primary Key: `agency` uniquely identifies each government agency.

- Not Null: `agency`, `name`, and `directives` are mandatory fields.

**Validation:**

- Data Type: `agency` is an integer, `name` and `directives` are strings, ensuring correct data types and formats for government agency information.

Overall System Constraints and Validation Considerations

* **Referential Integrity:** Ensure foreign keys correctly reference primary keys in related tables (e.g., `providerId` in `WeatherData` referencing `providerId` in `WeatherDataProvider`).
* **Business Rules Enforcement:** Implement specific business rules related to disaster management (e.g., validation of incident types, emergency service availability).
* **Data Consistency:** Use constraints like primary keys and unique constraints to prevent duplicate or inconsistent data entries.
* **Error Handling:** Implement validation checks to handle potential data entry errors and ensure data quality.
* **Performance Considerations:** Validate data types and constraints to optimize database performance and query efficiency.

# **Implementation and Testing**

In order to implement our database design, we are going to be using the PostgreSQL database management system. We choose this DBMS due to the following reason:

1. **ACID Compliance**:

* PostgreSQL is fully ACID-compliant (Atomicity, Consistency, Isolation, Durability), which ensures reliable transactions. This is crucial for disaster management systems where data integrity and consistency are paramount.

1. **Advanced Data Types and GIS Support**:

* PostgreSQL supports advanced data types and extensions like PostGIS, which adds support for geographic objects. This is highly beneficial for disaster management systems that need to manage and analyze spatial data (e.g., maps, geographic locations).

1. **Scalability**:

* PostgreSQL can handle large volumes of data and high-concurrency workloads. This is important for disaster management systems that need to scale as the volume of data increases during a disaster event.

1. **Robust Performance**:

* PostgreSQL provides excellent performance for both read and write operations. It includes features like indexing, query optimization, and caching, which help maintain high performance even under heavy load.

1. **Extensibility**:

* PostgreSQL’s extensible architecture allows you to add custom functions, data types, and extensions. This flexibility can be tailored to meet the specific needs of a disaster management system.

1. **Security**:

* PostgreSQL offers advanced security features, including robust authentication methods, data encryption, and access control mechanisms. These features ensure that sensitive data related to disaster management is protected.

1. **Reliability and Backup Solutions**:

* PostgreSQL includes powerful backup and restore capabilities. Tools like pg\_dump, continuous archiving, and point-in-time recovery are essential for ensuring data availability and disaster recovery.

1. **Open Source**:

* Being open source, PostgreSQL is cost-effective and benefits from a large community of developers and users. This community support can be invaluable for troubleshooting and improving the database system.

1. **Comprehensive Documentation**:

* PostgreSQL has extensive documentation and a wealth of online resources, tutorials, and community support, which can help developers and administrators effectively use and manage the database.

1. **Support for Complex Queries**:

* PostgreSQL can handle complex queries and provides advanced SQL functionalities. This is useful for analyzing data and generating reports in a disaster management context.