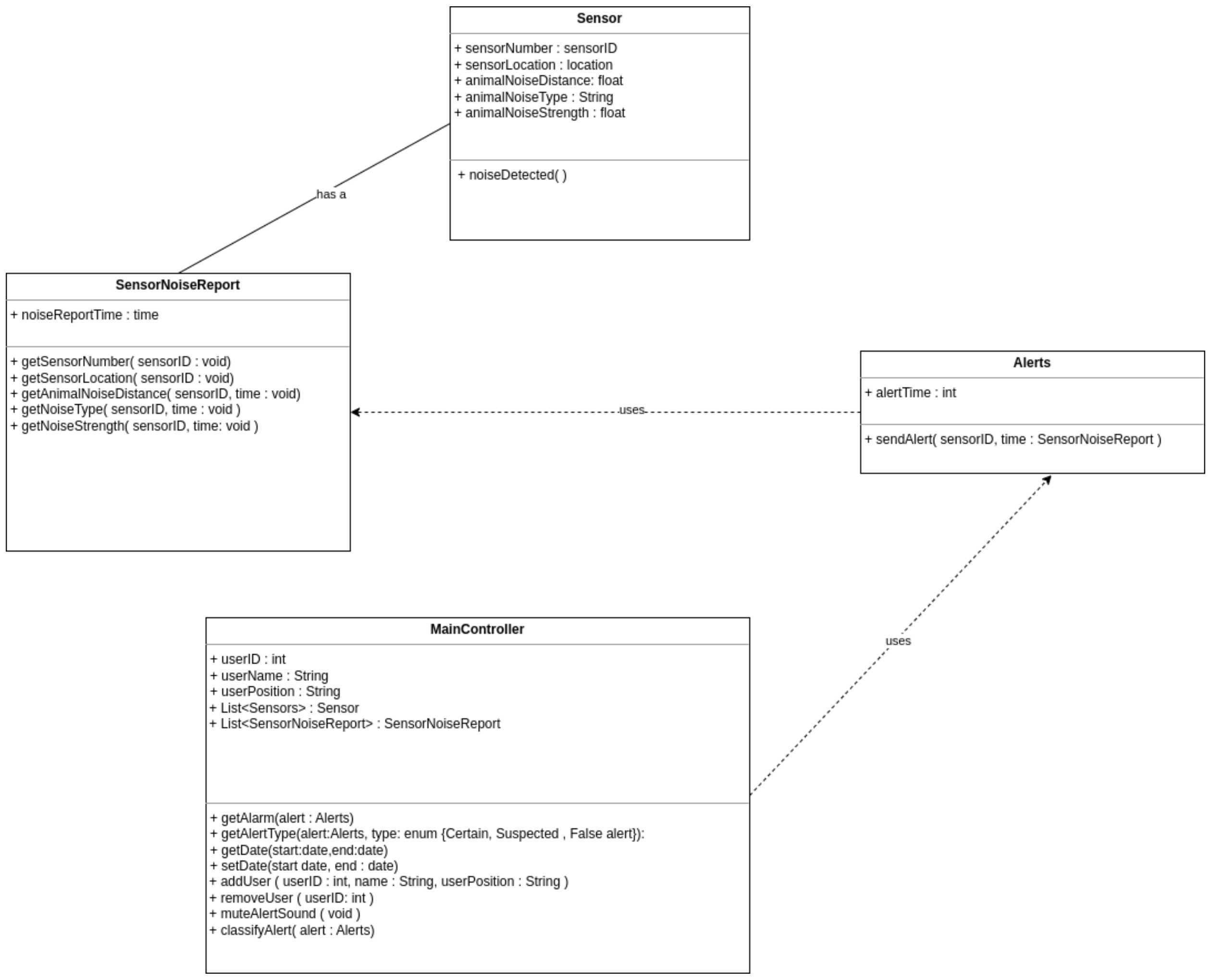
Assignment 4 Software Design 2.0

Project Title: Mountain Lion Detection System

Team Members: Jai Sharma, Leo Findley and Todd Hutchison

System Description: The Mountain Lion Detection System is designed to detect Various types of animal noises and issue alert messages to the controlling computers of authorities to further analyze these alerts and also notify the rangers in the national park. This system uses the animal detection system developed by the company Animals-R-HERE and has the ability to detect noises within a zone by the strategic placement of noise detection sensors within that area. After detection, it sends the alert message to a main controlling computer on the basis of the strength of the voice and classifies these into different alert messages. This is a highly advanced system and can provide detailed information in the form of alert messages which include the type of noise, strength, and also the precise location to within 3 miles.

UML CLASS Diagram


Sensor
+ sensorNumber : sensorID
+ sensorLocation : location
+ animalNoiseDistance: float
+ animalNoiseType : String
+ animalNoiseStrength : float
+ noiseDetected( )
has a
SensorNoiseReport
+ noiseReportTime : time
+ getSensorNumber(sensorID : void)
+ getSensorLocation(sensorID : void)
+ getAnimalNoiseDistance(sensorID, time : void)
+ getNoiseType(sensorID, time : void)
+ getNoiseStrength(sensorID, time: void)
uses
Alerts
+ alertTime : int
+ sendAlert(sensorID, time : SensorNoiseReport)
uses
MainController
+ userID : int
+ userName : String
+ userPosition : String
+ List<Sensors> : Sensor
+ List<SensorNoiseReport> : SensorNoiseReport
+ getAlarm(alert : Alerts)
+ getAlertType(alert:Alerts, type: enum {Certain, Suspected , False alert}):
+ getDate(start:date,end:date)
+ setDate(start date, end : date)
+ addUser (userID : int, name : String, userPosition : String)
+ removeUser (userID: int)
+ muteAlertSound (void)
+ classifyAlert(alert : Alerts)

Description of Classes

1. Class: Sensor

a.)Attributes:

sensorNumber : sensorID

sensorLocation : location

animalNoiseDistance: float

animalNoiseType : String

animalNoiseStrength : float

b.)Methods:

noiseDetected(void)

How it works: This class represents the sensors installed at various locations in the park and has attributes representing the IDs of sensors and their locations along with the type of noise they detect. The noiseDetected parameter detects the intensity of the noise and issues an alert message according to the intensity of the noise.

2. Class: Alerts

a.) Attributes:

alertTime : int

b.) Methods:

sendAlert(sensorID, time : SensorNoiseReport)

How it works: the Alerts class represents the alerts generated when the sensors detect sound. The attributes it has help us transmit each sensor's specific ID (sensorID), location (sensorLocation), time the alert was sent (alertTime), type of sound detected (animalNoiseType), intensity of the noise (animalNoiseStrength), and distance of noise with respect to the location of the sensor (animalNoiseDistance).

3. Class: SensorNoiseReport

a.) Attributes:

noiseReportTime : String

b.) Methods:

getSensorNumber(sensorID: void)

getSensorLocation(sensorID : void)

getAnimalNoiseDistance(sensorID, time : void)

getNoiseType(sensorID, time : void)

getNoiseStrength(sensorID, time : void)

How it works: the SensorNoiseReport class generates the data used in the Alerts sent to the Main Controller class. When the sensor detects a noise it generates a SensorNoiseReport with a time stamp. The Alerts class uses this sensor data to create an alert to the Main Controller class which uses an audible alarm to alert the ranger. The Main Controller class will also use the data from the alert to store varying amounts of information based on the time the alert was created.

4. Class: MainController

a.) Attributes:

userID : int

userName : String

userPosition : String

List<Sensors> : Sensor

List<SensorNoiseReport> : SensorNoiseReport

b.) Methods:

getAlarm(alert : Alerts)

getAlertType(alert:Alerts, type: enum {Certain, Suspected , False alert})

getDate(start:date,end:date)

setDate(start date, end : date)

addUser (userID : int, name : String, userPosition : String)

removeUser (userID: int)

muteAlertSound (void)

classifyAlert(alert : Alerts)

How it Works: This class is kind of like the brain behind our system and is responsible for issuing alerts and managing the alert system in general. It has methods to sound alarms and also helps classify them into certain, suspected and false alarms categories.

Updates to the UML Diagram:

Added methods:

getAlarm(alert : Alerts)

getAlertType(alert:Alerts, type: enum {Certain, Suspected , False alert})

getDate(start:date,end:date)

setDate(start date, end : date)

addUser (userID : int, name : String, userPosition : String)

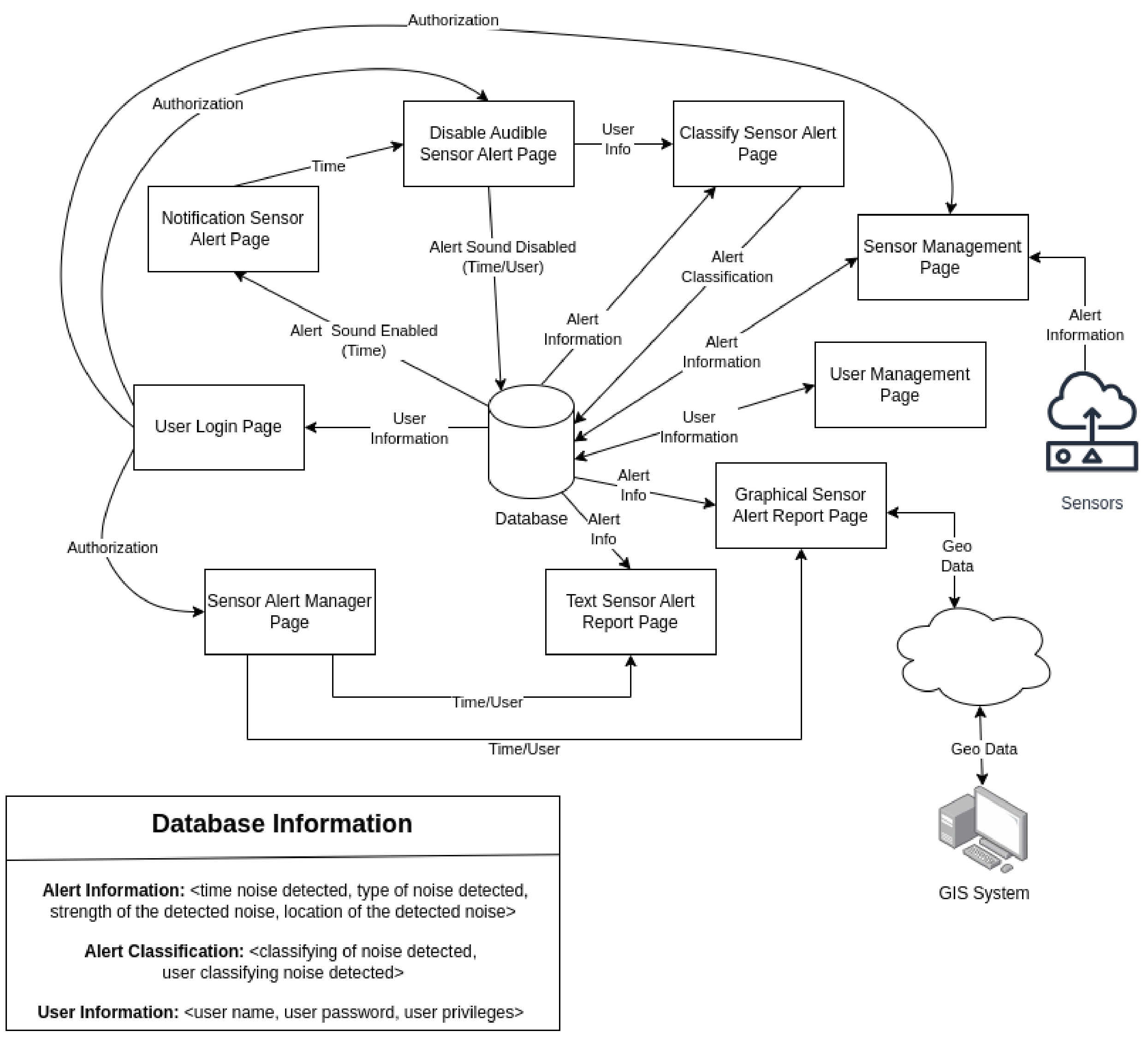
removeUser (userID: int)

muteAlertSound (void)

classifyAlert(alert : Alerts)

Architectural Diagram (Updated):

Mountain Lion Detection System - System Architecture



Database Information

Alert Information: <time noise detected, type of noise detected, strength of the detected noise, location of the detected noise>

Alert Classification: <classifying of noise detected, user classifying noise detected>

User Information: <user name, user password, user privileges>

For our updated Architecture diagram, we decided to go more in-depth and show more of the processes of the system in regards to the database. We also decided to make a more detailed diagram than just the main high-level components of the system which includes more pages of the user interface. We also added a GIS system to help with the graphical reports of the animal noises. The functionality of the system and system architecture remains relatively unchanged.

Major components of the Mountain Lion Detection System:

1. Sensors: The sensors are key to detecting the lions as they are responsible for recording noises and sending them to authorities for classification.
2. Alerting System: The sensors relay the message to our alerting system which processes the noise and classifies them in order to know whether the mountain lions are present or not.
3. Database: Stores alerts and alert notifications. The duration for alerts to be saved within the storage is 30 days and the alert summary older than 30 days but within one year. It also stores user information for the park rangers.
4. User interface: The system also provides an easy-to-use user interface for the rangers to interact with the system in an easier and more efficient manner.
5. GIS System: The GIS system provides the geographic data for the park and allows the system to map all of the alerts.

Interactions of the major components of the Mountain Lion Detection System

1. Sensors: They are responsible for sending noise to the sensor management, sending information to the sensor management page.
2. Sensor Management System: Sensor management is responsible for sending to to alert processing system.
3. Alert Processing: This receives alerts from sensors and process them to classify them as serious alert or false alerts which it then sends to the database storage along with the user interface.

4. User interface and Database: They are responsible for communicating with each other as the user interface requires reports and classification from database storage.
5. GIS System and Database: The GIS system allows the database to map all of the alerts onto a geographical graph and display that information for the rangers to see.

How the Architectural diagram works:

1. Firstly we would install sensors in different parts of the national park with unique IDs these sensors would detect animal noises and their strength which would then be sent to the sensor management.
2. The sensor management processes the noise and sends it back to the alert Processing system.
3. Alert processing further sends these alerts to the database and these alerts are then shown on the user interface as alarms.
4. The Rangers can then use the simple-to-use user interface to their advantage to classify the alert as certain, suspected, or false alert. The Control computer sounds an alarm when an alert is received and rangers can manually turn them off using the user interface. The user interface also helps with generating reports which can be saved up to one year on the database. The GIS system helps in generating reports that pertain to the geographical information of the noises.

Development Plan and Timeline:

Using the Waterfall Model

Requirement Analysis Phase (2 weeks)

- Gather detailed requirements from stakeholders, including park rangers and system users.
- Define the scope and constraints for the system.
- Create a comprehensive requirement specification document.

System Design Phase (4 weeks)

- Design the system architecture and define the technical specifications.
- Identify the technology stack and tools required for development.
- Create a detailed system design document and review it with stakeholders.

Implementation Phase (16 weeks)

- Develop the Sensor Management module for handling sensor data.
- Implement the Alert Processing module for analyzing and storing alerts.
- Create the Database Storage module for efficient data storage and retrieval.
- Develop the User Interface module for ranger interaction and reporting.

Testing Phase (6 weeks)

- Conduct unit testing for individual modules to ensure their functionality.
- Perform testing to verify that the system meets the system design specifications
- Perform integration testing to validate the interactions between different system components.

Deployment Phase (2 weeks)

- Prepare the system for deployment in the park environment.
- Configure the necessary hardware and software components for the system.
- Conduct a final round of testing to ensure a smooth deployment process.

Training and Documentation (2 weeks)

- Provide training sessions for park rangers and system administrators on how to use and maintain the system.
- Create comprehensive documentation including user manuals, technical guides, and troubleshooting documents.

Maintenance and Support (Ongoing)

- Establish a dedicated support team to address any issues or concerns raised by users.
- Regularly update the system based on user feedback and emerging requirements.
- Perform routine maintenance tasks to ensure the system's smooth operation.

Verification Test Plan

For the Project Mountain Lion Detection System, we will be focusing on 2 test sets each covering all three granularities which are Unit testing, Integration testing, and System testing.

Test Set 1: Sensor And Alert Processing

1. **Test case 1: Valid Noise Detection (Unit Testing)**

   A.)Purpose: This test case aims to check whether the method “noiseDetected” in the Sensor class is correctly able to detect and process the noise coming to it.

B.)Test Vector: We would be generating a valid noise detection scenario where the sensor is made to listen to the recorded sound of a mountain lion.

C.)Coverage: This test case covers the general use of the sensor class which is to listen for mountain lion noise as well as other noises in the area

D.)Failure Scenarios: Noise could possibly be outside the audible frequency for the audio devices we possess, the noise type could be incorrect or the sensors could malfunction.

2. **Test case 2: Data Transmission and Alert Generation(Integration)**

   A.)Purpose: This test case aims to check whether The sensor and alert classes are interacting properly.

   B.)Test Vector: To test this we would be simulating the conditions where the Sensor class sends noise data to the Alerts class and actively an alert is generated.

   C.) Coverage: This test case would validate the integration between the two classes(sensor and alerts) and check for alert generation.

D.)Failure Scenarios: The possible failure scenarios could be individual problems with data transmission, data accuracy, and alert generation. The failure of any of them could lead to the failure of this test case.

3. **<u>Test Case 3: Real-time alert Management and User Interface Interaction (System)</u>**

A.)Purpose: This test case aims to test the real-time alert processing and user interface interaction.

B.)Test Vector: To test this we would have to simulate real-time alerts generated by the sensors which would then be processed by the system and in the end would interact with the user interface.

C.)Coverage: This test case checks whether the entire system is working including the real-time alert processing and user interaction.

D.)Failure Scenarios: We should check for problems with real-time processing, alert handling, and user interface to avoid any scenarios.

Test Set 2: Database and Reporting

1. **<u>Test Case 1: Data Storage and Retrieval(Unit)</u>**

A.)Purpose: The aim of this test case is to verify the credibility of storing this data and its retrieval.

B.)Test Vector: To Test this we would have to simulate the environment to store and retrieve alert data from our database for the Mountain Lion Detection System.

C.)Coverage: This test would cover the storage prospects of our database.

D.)Failure Scenario: We should try checking for performance issues, data processing issues and system crashes when the System is made to work with high load.

2. **<u>Test Case 2: User Interface Generation of Reports (Integration)</u>**

A.) Purpose: The aim of this test case is to check whether the user interface is accurately able to generate and present reports.

B.)Test Vector: To test this test case we can try simulating the report generation through the user interface and its interaction with stored data.

C.) Coverage: This test case would aim to verify the interaction between the database storage module and the user interface.

D.) Failure Scenarios: We should try checking for issues with report generation

3. Test Case 3: Security and Unauthorized Access (System)

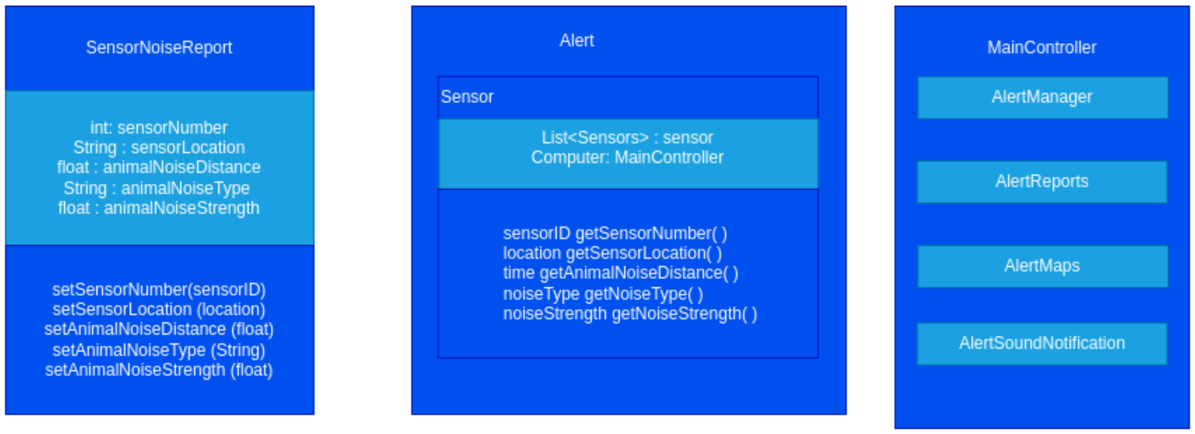
A.) Purpose: The aim of this test case is to ensure the security of the system and keep unauthorized users from accessing the system.

B.)Test Vector: We would engage the testing by hiring pen-testers and try simulating various security attacks and attempts to access the system.

C.)Coverage: This test case focuses on the system's security and access control in general.

D.) Failure Scenarios: Check for unauthorized access to the system, potential breaches, and security vulnerabilities which could be bad for the system in the long run.

Testing Levels Diagram


SensorNoiseReport
int: sensorNumber
String : sensorLocation
float : animalNoiseDistance
String : animalNoiseType
float : animalNoiseStrength
setSensorNumber(sensorID)
setSensorLocation (location)
setAnimalNoiseDistance (float)
setAnimalNoiseType (String)
setAnimalNoiseStrength (float)
Alert
Sensor
List<Sensors> : sensor
Computer: MainController
sensorID getSensorNumber( )
location getSensorLocation( )
time getAnimalNoiseDistance( )
noiseType getNoiseType( )
noiseStrength getNoiseStrength( )
MainController
AlertManager
AlertReports
AlertMaps
AlertSoundNotification

Unit Testing

SensorNoiseReport.setSensorNumber(sensorID)
SensorNoiseReport.setSensorLocation (location)
SensorNoiseReport.setAnimalNoiseDistance (float)
SensorNoiseReport.setAnimalNoiseType (String)
SensorNoiseReport.setAnimalNoiseStrength (float)

Functional/Integration Testing

Alert.sendAlert(Sensor s)

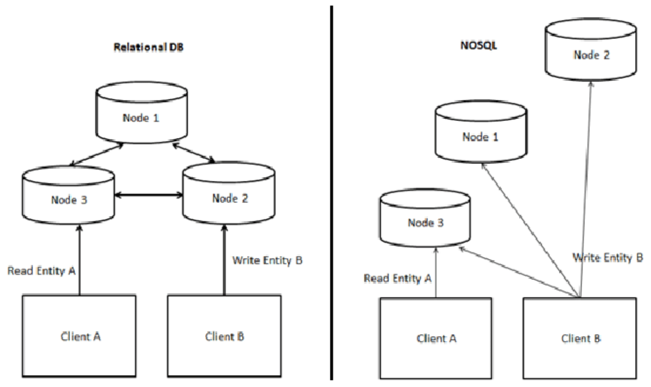
System/Acceptance Testing

Login
Logout
Add a sensor
Remove a sensor
Set a sensor location
Disable Alert Sound
Classify alert message
Create report for a specific sensor
Create a map report for all sensors

Data Management Strategy for Mountain Lion Detection System

2.1 Choice Of Database Strategy:

We plan on using a relational database(RDBMS) for storing our data in a sequential and structured order. We also plan on using SQL for our database due to it being easy to learn and reproduce in other systems as well as due to its proven reliability, and compliance with ACID(Atomicity, Consistency, Isolation, Durability), and we personally believe that it will be suitable for making our database well defined as compared to NoSql. We have also attached a diagram for why Sql fulfills specific needs.



Source: https://www.researchgate.net/figure/Relational-DB-vs-NoSQL_fig4_336746925

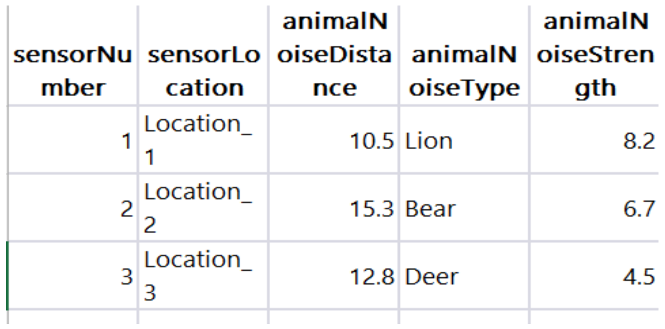
2.2 Number Of Databases

We have currently chosen a single database to store all of our data for two reasons which are budget constraints and simplicity. This decision was mainly driven by the moderate scale of our application, making it simple, and efficient, and saving us money on maintaining multiple databases. The money saved can be used elsewhere in the project such as fixing broken sensors which will make the project more monetarily feasible. In the future we might plan on adding a database just for redundancy purposes as if data is lost we might still have backup data.

2.3 Logical Partition of Data

The Logical partitioning of data in the Mountain Lion Detection System involves organining all the data into distinct tables within a Relational Database Management System(RDBMS). Each of the tables has a specific purpose and we were able to make 4 such tables which are further discussed below.

Table 1: Sensors



sensorNumber	sensorLocation	animalNoiseDistance	animalNoiseType	animalNoiseStrength
1	Location_1	10.5	Lion	8.2
2	Location_2	15.3	Bear	6.7
3	Location_3	12.8	Deer	4.5

Attributes:(sensorNumber,sensorLocation,animalNoiseDistance,animalNoiseType,animalNoiseStrength)

a.) sensorNumber: It will be our primary key in the case and has the function of being the unique identifier for each sensor.

b.)sensorlocation: It represents the actual geographical location of the sensor.

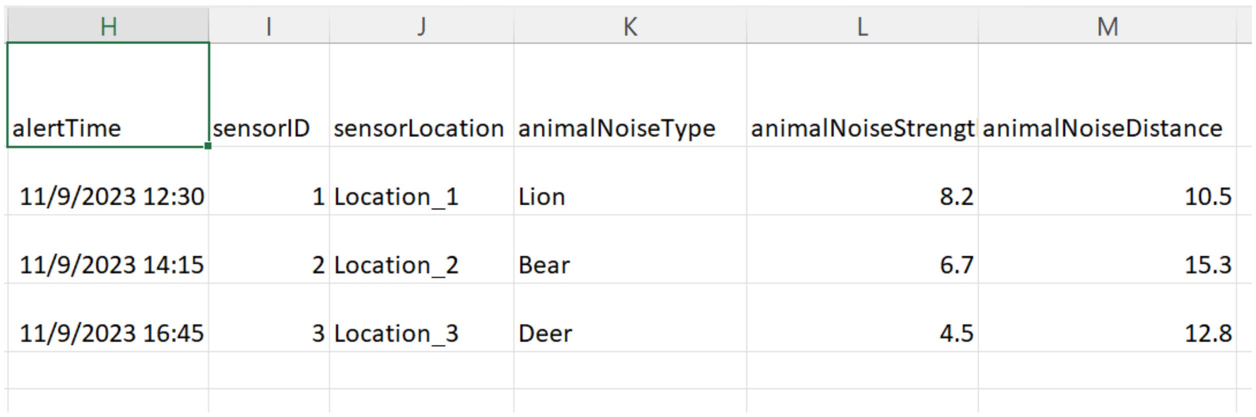
c.)animalNoiseDistance: This tells us the distance between the detected animal noise and the sensor.

d.)animalNoiseType: This Describes the type of animal noise whose noise is detected by the sensor

e.) animalNoiseStrength: Represents the intensity of the animal noise detected

The explanation forTable 1: This table will store data about each sensor deployed in the national park and the sensorNumber, in this case, serves as the primary key to ensure uniqueness and other attributes store critical information about the sensor such as its geographical location and its ability to detect the noises of wild animals.

Table 2: Alerts



H	I	J	K	L	M
alertTime	sensorID	sensorLocation	animalNoiseType	animalNoiseStrengt	animalNoiseDistance
11/9/2023 12:30	1	Location_1	Lion	8.2	10.5
11/9/2023 14:15	2	Location_2	Bear	6.7	15.3
11/9/2023 16:45	3	Location_3	Deer	4.5	12.8

Attributes:(alertTime,sensorId,sensorocation.animalNoiseType,animalNoiseDistance)

alertTime: This is the timestamp which tells us when the alert was generated

sensorID: This is the foreign key that would be used to link to the sensorNumber key in the sensors table.

sensorLocation: This tells us the geographical location of the sensor triggering the alarm.

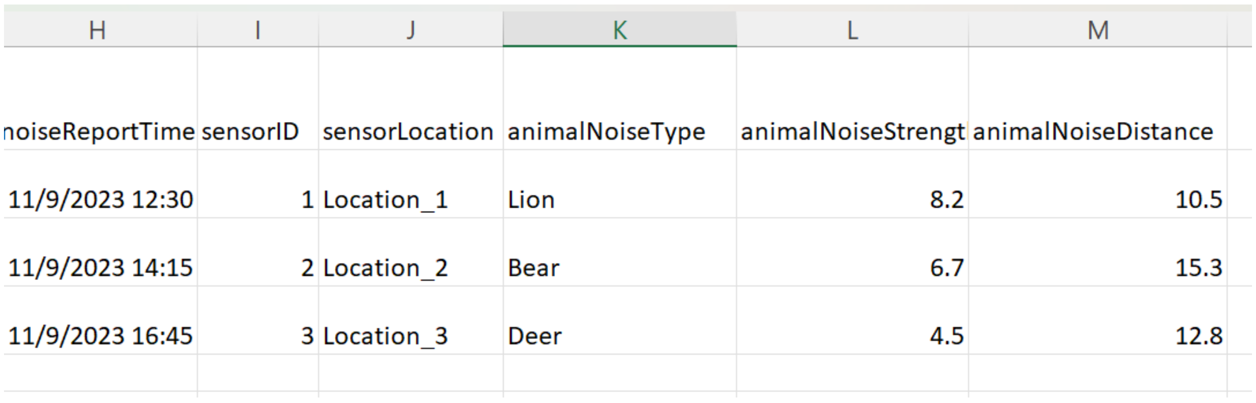
animalNoiseType: This tries to identify the animal that could have made the noise.

animalNoiseStrength: this tries to tell us the intensity of the noise detected.

animalNoiseDistance: This tells us the distance between the animal noise and the sensor.

The Explanation for Table 2: This table actively stores information about the alerts generated by the system and the ‘sensorID’ serves as a foreign key that is used to link tables in SQL.This links each alert generated to the specific sensor that detected the noise. The other attributes are also critical as they capture crucial details such as the time, location, type and also the distance of the detected animal noise.

Table 3: SensorNoiseReport



H	I	J	K	L	M
noiseReportTime	sensorID	sensorLocation	animalNoiseType	animalNoiseStrengt	animalNoiseDistance
11/9/2023 12:30	1	Location_1	Lion	8.2	10.5
11/9/2023 14:15	2	Location_2	Bear	6.7	15.3
11/9/2023 16:45	3	Location_3	Deer	4.5	12.8

Attributes:(alertTime,sensorId,sensorocation.animalNoiseType,animalNoiseDistance)

alertTime: This is the timestamp which tells us when the alert was generated.

sensorID: This is the foreign key that would be used to link to the sensorNumber key in the sensor table.

sensorLocation: This tells us the geographical location of the sensor triggering the alarm.

animalNoiseType: This tries to identify the animal which could have made the noise.

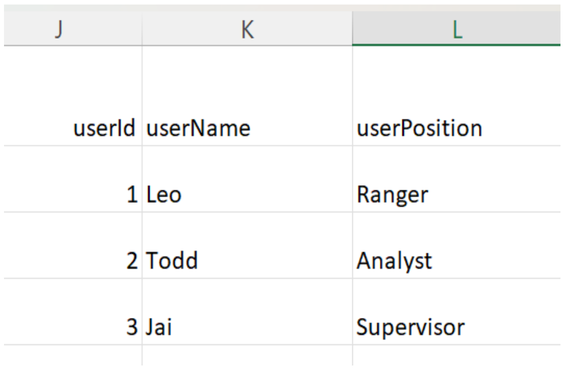
animalNoiseStrength: this tries to tell us the intensity of the noise detected.

animalNoiseDistance: This tells us the distance between the animal noise and the sensor.

Explanation of Table 3:

This table is responsible for storing the noise reports generated by the sensors. Just like the Alerts table this utilizes the ‘sensorId’ as a foreign key to get linked to the Sensors table and has all the other attributes used for similar reasons to the Alerts table.

Table 4: SuperController



J	K	L
userId	userName	userPosition
1	Leo	Ranger
2	Todd	Analyst
3	Jai	Supervisor

Attributes:(userId,userName,userPosition)

userId: this will be used as the unique identifier for each user and thus will be used as the primary key

userName : this will help us store the names of the users.

userPosition: This will help us tell the position of the users and their role in the system so that there is hierarchical access given in the park.

Explanation of Table 4:

This table is responsible for maintaining the user information within the system including their unique work IDs, their names and their positions.

The Overall logical partition we use for each of the 4 tables helps in efficiently organizing and retrieving data. We have effectively used primary and foreign keys to establish relationships between tables and have thus ensured data integrity and cohesion. We hope that this structured approach will definitely save our organization time and money by simplifying data management and supporting effective system functionality.

2.4Alternative Considerations

A.)NoSql database: We could have used the NoSql database for the flexibility and scalability it offers but went with the Sql approach due to the nature of our data being structured and our dire need for the ACID(Atomicity, Consistency, Isolation, Durability) properties Sql offers.

B.) The need for more Databases: We could have gone for more databases for separation of concerns and redundancy but have taken the approach for a single database as it simplifies queries and maintenance for our organization while being cheaper.

2.5 TradeOff

Our choice for a single SQL database is a calculated compromise by us between simplicity and efficiency. This database is perfect for our current needs as we only need to manage a moderate amount of data currently and this system facilitates easy data management and retrieval, potential challenges might arise as the system grows in the future. It would be difficult for the database to shadow the growth of the system with our single SQL database but given the current scope of the project, we find this tradeoff to be acceptable.