**3.5 ORDER PRESERVING ENCRYPTION (OPE) FOR WEIGHT ATTRIBUTE (EXTRA POINTS)**

**DESIGN**:

* The **Weight** attribute is encrypted using Order Preserving Encryption (OPE), which allows range queries (e.g., finding all records with weight between 50 and 80).

**EXPLANATION**:

* OPE preserves the order of encrypted values, enabling efficient queries on encrypted data.
* The system implements OPE for the weight attribute, allowing range-based queries while keeping the data encrypted.

**3.6 EXTRA SECURITY FEATURE REPORT**

* **Order Preserving Encryption Explanation**: OPE allows encrypted data to maintain its original order. This means that range queries can be executed on encrypted data without needing to decrypt it, which is essential for maintaining privacy while allowing meaningful queries.

In this implementation, we use a combination of AES and RSA encryption methods to secure sensitive data. The Weight attribute is protected using the AES encryption scheme, specifically the AES-256 algorithm in CBC mode. Here’s a breakdown of the approach:

1. **AES Encryption for Weight Attribute**:
   * **AES Key Generation**: The AES key is either generated or loaded from a file (aes\_key.txt). If the key doesn't exist, it is created using os.urandom(32), which generates a random 256-bit key for AES-256 encryption.
   * **AES Block Size**: The block size for AES is set to 128 bits (16 bytes), which is the standard for AES.
   * **Encryption Process**:
     + A random Initialization Vector (IV) is generated using os.urandom(BLOCK\_SIZE) to ensure that the encryption is non-deterministic and enhances security.
     + The value to be encrypted (in this case, the weight attribute) is first converted into bytes and padded to a multiple of the AES block size using PKCS7 padding. This ensures that the length of the data is compatible with the block size required by AES.
     + The data is then encrypted using the AES algorithm in CBC (Cipher Block Chaining) mode, which provides enhanced security by chaining the blocks of data together.
     + The IV is concatenated with the encrypted data and returned, ensuring that the decryption process can be correctly performed.
2. **RSA Encryption for SSN**:
   * The SSN (Social Security Number) of individuals in the healthcare table is encrypted using RSA encryption, which is an asymmetric encryption algorithm.
   * The RSA public key is used to encrypt the SSN with the OAEP (Optimal Asymmetric Encryption Padding) scheme, which provides enhanced security against certain types of attacks.
   * The encrypted SSN is stored in the encrypted\_ssn column of the healthcare table.

**APPLICATION TO WEIGHT ATTRIBUTE:**

For the Weight attribute, the value is securely encrypted before being stored in the database. Here's how it is applied:

* The value of the weight field is passed to the encrypt\_value\_aes function, where it is encrypted using AES-256 in CBC mode. This ensures that the weight data is stored securely and cannot be accessed without the proper decryption key. Only authorized users or systems with the correct AES key can decrypt and view the original weight data.

This method ensures that the sensitive weight attribute, along with other data like SSN, is protected while still allowing for secure and efficient storage in the database.

**SOURCE CODE:**

# AES block size (128 bits)

BLOCK\_SIZE = 16

# Check if the AES key already exists; if not, generate a new one

if not os.path.exists('aes\_key.txt'):

    AES\_KEY = os.urandom(32)  # AES Key (256 bits for AES-256)

    # Save the AES Key to a text file

    with open('aes\_key.txt', 'wb') as aes\_key\_file:

        aes\_key\_file.write(AES\_KEY)

    print("AES key generated and saved to aes\_key.txt.")

else:

    # If the key file exists, load the existing key

    with open('aes\_key.txt', 'rb') as aes\_key\_file:

        AES\_KEY = aes\_key\_file.read()

    print("AES key loaded from aes\_key.txt.")

# RSA Key Generation (Secure RSA Key pair)

private\_key = rsa.generate\_private\_key(

    public\_exponent=65537,

    key\_size=2048,

    backend=default\_backend()

)

public\_key = private\_key.public\_key()

# Save the RSA private key and public key to text files

with open('private\_key.pem', 'wb') as private\_key\_file:

    private\_key\_file.write(private\_key.private\_bytes(

        encoding=serialization.Encoding.PEM,

        format=serialization.PrivateFormat.TraditionalOpenSSL,

        encryption\_algorithm=serialization.NoEncryption()

    ))

with open('public\_key.pem', 'wb') as public\_key\_file:

    public\_key\_file.write(public\_key.public\_bytes(

        encoding=serialization.Encoding.PEM,

        format=serialization.PublicFormat.SubjectPublicKeyInfo

    ))

# Encryption function using AES (CBC mode)

def encrypt\_value\_aes(value):

    # Generate a random IV

    iv = os.urandom(BLOCK\_SIZE)

    # Create AES cipher object

    cipher = Cipher(algorithms.AES(AES\_KEY), modes.CBC(iv), backend=default\_backend())

    encryptor = cipher.encryptor()

    # Ensure the value is converted to bytes and then padded

    value\_bytes = str(value).encode('utf-8')

    # Pad the value to be a multiple of BLOCK\_SIZE

    padder = padding.PKCS7(BLOCK\_SIZE \* 8).padder()

    padded\_value = padder.update(value\_bytes) + padder.finalize()

    # Encrypt the value

    encrypted\_value = encryptor.update(padded\_value) + encryptor.finalize()

    # Return the IV concatenated with the encrypted value

    return iv + encrypted\_value

# Encryption function using RSA

def encrypt\_value\_rsa(value):

    value\_bytes = str(value).encode('utf-8')

    encrypted\_value = public\_key.encrypt(

        value\_bytes,

        asymmetric\_padding.OAEP(

            mgf=asymmetric\_padding.MGF1(algorithm=hashes.SHA256()),

            algorithm=hashes.SHA256(),

            label=None

        )  )

    return encrypted\_value

**NOTE: The above code is included in the main source code(database\_setup.py)**