Certainly! Below are detailed explanations and two code examples for each of the topics: Data Masking and Encryption, Database Sharding, and Database Replication.

### 1. Data Masking and Encryption:

Data Masking:
Data masking involves replacing, encrypting, or scrambling original sensitive information with fake or pseudonymous data to protect sensitive information.

Example 1 - Using a simple masking function in Python:

```python
import random
import string

def mask_data(original_data):
    masked_data = ''.join(random.choices(string.ascii_letters + string.digits, k=len(original_data)))
    return masked_data

# Example usage:
sensitive_info = "123-45-6789"
masked_info = mask_data(sensitive_info)
print(f"Original Data: {sensitive_info}")
print(f"Masked Data: {masked_info}")
```

Data Encryption:
Data encryption involves transforming data into a different format that can only be read or processed with the correct decryption key.

Example 2 - Encrypting data using the cryptography library in Python:

```python
from cryptography.fernet import Fernet

def encrypt_data(original_data, key):
    cipher_suite = Fernet(key)
    encrypted_data = cipher_suite.encrypt(original_data.encode())
    return encrypted_data

def decrypt_data(encrypted_data, key):
    cipher_suite = Fernet(key)
    decrypted_data = cipher_suite.decrypt(encrypted_data).decode()
    return decrypted_data
```

```
# Example usage:
encryption_key = Fernet.generate_key()
sensitive_info = "123-45-6789"
encrypted_info = encrypt_data(sensitive_info, encryption_key)
decrypted_info = decrypt_data(encrypted_info, encryption_key)

print(f"Original Data: {sensitive_info}")
print(f"Encrypted Data: {encrypted_info}")
print(f"Decrypted Data: {decrypted_info}")
```

### 2. Database Sharding:

Horizontal Sharding:
Horizontal sharding involves dividing the data into smaller, independent pieces called shards based on specific criteria.

Example 3 - Simple horizontal sharding in Python using a list of dictionaries:

```python
def horizontal_sharding(data, shard_key):
    shards = {}
    for entry in data:
        shard_value = entry[shard_key]
        if shard_value not in shards:
            shards[shard_value] = []
        shards[shard_value].append(entry)
    return shards

# Example usage:
data_to_shard = [
    {"id": 1, "name": "John", "age": 25},
    {"id": 2, "name": "Jane", "age": 30},
    {"id": 3, "name": "Bob", "age": 22},
    {"id": 4, "name": "Alice", "age": 28}
]

sharded_data = horizontal_sharding(data_to_shard, "age")
print(f"Sharded Data: {sharded_data}")
```

Vertical Sharding:

Vertical sharding involves dividing data by columns, separating different attributes into different shards.

Example 4 - Simple vertical sharding in Python using nested dictionaries:

```python
def vertical_sharding(data):
    shards = {key: [] for key in data[0]}
    for entry in data:
        for key, value in entry.items():
            shards[key].append({key: value})
    return shards

# Example usage:
data_to_shard = [
    {"id": 1, "name": "John", "age": 25},
    {"id": 2, "name": "Jane", "age": 30},
    {"id": 3, "name": "Bob", "age": 22},
    {"id": 4, "name": "Alice", "age": 28}
]

sharded_data = vertical_sharding(data_to_shard)
print(f"Sharded Data: {sharded_data}")
```

### 3. Database Replication:

Master-Slave Replication:
Master-Slave replication involves copying data from one database (master) to another (slave) to ensure redundancy and fault tolerance.

Example 5 - Using SQLite for simple master-slave replication in Python:

```python
import sqlite3

# Master database
master_conn = sqlite3.connect('master.db')
master_cursor = master_conn.cursor()

# Slave database
slave_conn = sqlite3.connect('slave.db')
slave_cursor = slave_conn.cursor()
```

```
# Create a table in the master database
master_cursor.execute('CREATE TABLE users (id INTEGER PRIMARY KEY, name TEXT)')

# Insert data into the master database
master_cursor.execute('INSERT INTO users (name) VALUES (?)', ('John',))
master_conn.commit()

# Replicate data to the slave database
master_cursor.execute('SELECT * FROM users')
data_to_replicate = master_cursor.fetchall()
slave_cursor.executemany('INSERT INTO users (name) VALUES (?)', data_to_replicate)
slave_conn.commit()

# Close connections
master_conn.close()
slave_conn.close()
```

Multi-Master Replication:
Multi-Master replication involves multiple databases that can accept write operations, providing higher availability and fault tolerance.

Example 6 - Using Redis for simple multi-master replication in Python:

```python
import redis

# Create connections to multiple Redis servers
master1 = redis.StrictRedis(host='localhost', port=6379, decode_responses=True)
master2 = redis.StrictRedis(host='localhost', port=6380, decode_responses=True)

# Set data on the first master
master1.set('key', 'value')

# Replicate data to the second master
value_to_replicate = master1.get('key')
master2.set('key', value_to_replicate)

# Retrieve data from the second master
replicated_value = master2.get('key')
print(f"Replicated Value: {replicated_value}")
```

Note: For practical use cases, consider using dedicated database systems that support replication, such as MySQL, PostgreSQL, or MongoDB, depending on your requirements. The provided examples are simplified for educational purposes.

Certainly! Let's explore more examples for each topic:

### 1. Data Masking and Encryption:

Data Masking (Advanced):
In this example, we'll use a more sophisticated approach to mask sensitive data by applying a hash function.

```python
import hashlib

def mask_data_advanced(original_data):
    hashed_data = hashlib.sha256(original_data.encode()).hexdigest()
    return hashed_data

# Example usage:
sensitive_info = "123-45-6789"
masked_info_advanced = mask_data_advanced(sensitive_info)
print(f"Original Data: {sensitive_info}")
print(f"Masked Data (Advanced): {masked_info_advanced}")
```

Data Encryption (Advanced):
In this example, we'll use the `cryptography` library to implement AES encryption, which is more secure than the previous example.

```python
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend

def encrypt_data_advanced(original_data, key):
    cipher = Cipher(algorithms.AES(key), modes.CFB(), backend=default_backend())
    encryptor = cipher.encryptor()
    encrypted_data = encryptor.update(original_data.encode()) + encryptor.finalize()
    return encrypted_data

def decrypt_data_advanced(encrypted_data, key):
    cipher = Cipher(algorithms.AES(key), modes.CFB(), backend=default_backend())
    decryptor = cipher.decryptor()
    decrypted_data = decryptor.update(encrypted_data) + decryptor.finalize()
    return decrypted_data.decode()

# Example usage:
encryption_key_advanced = b'sixteenbyteskey'
sensitive_info_advanced = "123-45-6789"
encrypted_info_advanced = encrypt_data_advanced(sensitive_info_advanced,
encryption_key_advanced)
decrypted_info_advanced = decrypt_data_advanced(encrypted_info_advanced,
encryption_key_advanced)

print(f"Original Data: {sensitive_info_advanced}")
print(f"Encrypted Data (Advanced): {encrypted_info_advanced}")
print(f"Decrypted Data (Advanced): {decrypted_info_advanced}")
```

### 2. Database Sharding:

Composite Sharding:
In this example, we'll implement composite sharding, where data is sharded based on multiple keys.

```python
def composite_sharding(data, shard_keys):
    shards = {}
    for entry in data:
        shard_values = tuple(entry[key] for key in shard_keys)
        if shard_values not in shards:
            shards[shard_values] = []
        shards[shard_values].append(entry)
    return shards

# Example usage:
data_to_shard_composite = [
    {"id": 1, "name": "John", "age": 25, "country": "USA"},
    {"id": 2, "name": "Jane", "age": 30, "country": "Canada"},
    {"id": 3, "name": "Bob", "age": 22, "country": "USA"},
    {"id": 4, "name": "Alice", "age": 28, "country": "Canada"}
]

sharded_data_composite = composite_sharding(data_to_shard_composite, ["age", "country"])
print(f"Composite Sharded Data: {sharded_data_composite}")
```

Range Sharding:

In this example, we'll implement range sharding, where data is sharded based on a specified range of values.

```python
def range_sharding(data, shard_key, range_size):
    shards = {}
    for entry in data:
        shard_value = entry[shard_key] // range_size
        if shard_value not in shards:
            shards[shard_value] = []
        shards[shard_value].append(entry)
    return shards

# Example usage:
data_to_shard_range = [
    {"id": 1, "name": "John", "age": 25},
    {"id": 2, "name": "Jane", "age": 30},
    {"id": 3, "name": "Bob", "age": 22},
    {"id": 4, "name": "Alice", "age": 28}
]

range_size = 10
sharded_data_range = range_sharding(data_to_shard_range, "age", range_size)
print(f"Range Sharded Data: {sharded_data_range}")
```

### 3. Database Replication:

Asynchronous Master-Slave Replication:
In this example, we'll implement asynchronous master-slave replication using the `asyncio` library.

```python
import asyncio

async def replicate_data_async(master_data, slave_connection):
    # Simulating asynchronous replication
    await asyncio.sleep(1)
    await slave_connection.execute('INSERT INTO users (name) VALUES (?)', master_data)

# Example usage:
# Assume master_data is fetched from the master database
master_data_to_replicate = ("John",)
```

```
# Assume slave_connection is a connection to the slave database
slave_connection = None  # Replace None with your actual slave database connection

# Run the replication asynchronously
loop = asyncio.get_event_loop()
loop.run_until_complete(replicate_data_async(master_data_to_replicate, slave_connection))
```

Galera Cluster for Multi-Master Replication:
In this example, we'll use the Galera Cluster for MySQL to demonstrate multi-master replication.

```python
# This example assumes you have a Galera Cluster set up with multiple nodes
# Connect to the MySQL Galera Cluster using a MySQL connector library

import mysql.connector

# Define the connection configuration for one of the Galera nodes
galera_config = {
    'user': 'your_username',
    'password': 'your_password',
    'host': 'galera-node1',
    'database': 'your_database',
    'autocommit': True,
    'pool_size': 5,
}

# Connect to the Galera node
galera_conn = mysql.connector.connect(**galera_config)
galera_cursor = galera_conn.cursor()

# Perform write operations on the Galera node
galera_cursor.execute('INSERT INTO users (name) VALUES (%s)', ('Jane',))

# Close the connection
galera_conn.close()
```

Note: Make sure to replace placeholders such as `'your_username'`, `'your_password'`, `'galera-node1'`, `'your_database'`, etc., with your actual database credentials and connection details.