UNIT IV

OBJECT ORIENTED AND EXCEPTION HANDLING

Classes and Objects - creating a class - class methods - class inheritance. Exceptions Handling-Build in Exceptions- Files, File operations, reading a file content, writing a file, change position, controlling file I/O, Manipulating file paths.

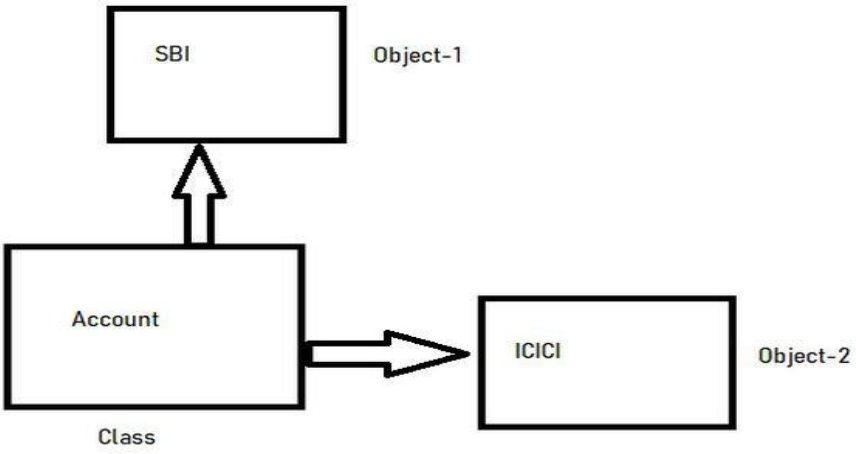
In object-oriented programming (OOP), classes and objects are fundamental concepts. Let's break them down:

Classes:

- A **class** is like a blueprint or a template for creating objects. It defines a set of attributes and methods that the created objects (instances) will have.
- **Attributes** are the characteristics of the class, also known as properties or fields. They define the state or data that the objects will hold.
- **Methods** are the functions defined inside a class that describe the behaviors of the objects created from the class.

Objects:

- An **object** is an instance of a class. When a class is defined, no memory is allocated until an object of that class is created.
- Each object can have its own set of attribute values, and the methods of the class can be used to perform actions using these attributes.



Creating a class:

Creating a class in Python is straightforward. Here's a step-by-step guide to help you understand how to create a class, define attributes, and methods, and then create objects (instances) of that class.

Define a Class:

You start by using the class keyword followed by the class name. The class name should be descriptive and follow the PascalCase convention (e.g., MyClass).

Example:

Let's create a class called Person

```
class Person:
    pass
```

Here, Person is the name of the class, and pass is a placeholder that indicates the class is currently empty.

Attributes:

Attributes are variables that hold data related to the class. We typically define them within the __init__ method, which is a special method called when a new object is instantiated.

```
class Person:
    def __init__(self, name, age, gender):
        self.name = name  # Attribute
        self.age = age  # Attribute
        self.gender = gender  # Attribute
    def greet(self):  # Method
        print(f"Hello, my name is {self.name}.")
    def display_info(self):  # Method
        print(f"Name: {self.name}, Age: {self.age}, Gender: {self.gender}")
```

Class methods:

In Python, class methods are methods that are bound to the class rather than its instance. They can modify the class state that applies across all instances of the class, rather than individual instance state.

Here's a quick overview of how to define and use class methods:

Defining a Class Method:

To define a class method, use the @classmethod decorator and ensure the method takes cls as its first parameter. This cls parameter refers to the class itself, not an instance of the class.

Here's a basic example:

```
class MyClass:
    class_variable = 0
    def __init__(self, value):
        self.instance_variable = value
    @classmethod

    def class_method(cls):
        print(f"Class method called. Class variable value: {cls.class_variable}")
    @classmethod

    def increment_class_variable(cls):
        cls.class_variable += 1

# Using the class method
MyClass.class_method()  # Output: Class method called. Class variable value: 0
MyClass.increment_class_variable()
MyClass.class_method()  # Output: Class method called. Class variable value: 1
```

Example with Class and Instance Interaction:

```
class Counter:
    count = 0

    def __init__(self):
        Counter.count += 1
    @classmethod

    def get_count(cls):
        return cls.count
    @classmethod

    def reset_count(cls):
        cls.count = 0
# Creating instances
c1 = Counter()
c2 = Counter()

print(Counter.get_count())  # Output: 2
# Resetting count
Counter.reset_count()

print(Counter.get_count())  # Output: 0
```

class inheritance:

Inheritance in DBMS allows us to create a relationship between tables, similar to how classes inherit properties and methods in object-oriented programming. It enables us to establish a hierarchical structure among tables, where child tables inherit attributes and behaviors from their parent table.

When a class (the subclass) inherits from another class (the superclass), it gains access to all the attributes and methods of the superclass.

Here's a simple example:

```
class Animal:
    def __init__(self, name):
        self.name = name
    def speak(self):
        raise NotImplementedError("Subclass must implement this method")
class Dog(Animal):
    def speak(self):
        return "Woof!"
class Cat(Animal):
    def speak(self):
        return "Meow!"
# Creating instances
dog = Dog("Buddy")
cat = Cat("Whiskers")
print(dog.name)   # Output: Buddy
print(dog.speak())  # Output: Woof!
print(cat.name)   # Output: Whiskers
print(cat.speak())  # Output: Meow!
```

. Exceptions Handling-Build in Exceptions

In Python, exceptions are used to handle errors and other exceptional conditions that may arise during the execution of a program. Python provides a rich set of built-in exceptions to handle common error scenarios. Here's a guide to handling exceptions and an overview of the built-in exceptions in Python:

Basic Exception Handling

To handle exceptions, you use try, except, else, and finally blocks.

- **try Block**: The code that might raise an exception is placed inside this block.
- **except Block**: This block handles the exception if it occurs.
- **else Block**: This block, if present, executes if no exceptions are raised in the try block.
- **finally Block**: This block always executes, regardless of whether an exception occurred or not.

Example:

```
try:
    # Code that might raise an exception
    result = 10 / 0
except ZeroDivisionError:
    # Code that executes if an exception occurs
    print("You can't divide by zero!")
else:
    # Code that executes if no exception occurs
    print("Division was successful.")
finally:
    # Code that always executes
    print("This block always executes.")
```

File operations

File operations in the context of a Database Management System (DBMS) involve various tasks related to reading, writing, managing, and maintaining the files that store the database's data and metadata. Here's an overview of common file operations and concepts related to DBMS:

1. File Creation

- **Purpose**: Establish new files for storing data, logs, indexes, etc.
- **Operation**: Involves initializing file structures and allocating space on disk.
- **Example**: When creating a new database or table, the DBMS generates new data files or index files to store the table data

```
CREATE DATABASE my_database;

CREATE TABLE my_table (id INT, name VARCHAR(50));
```

2. File Opening

- **Purpose**: Access files for reading or writing operations.
- **Operation**: Opens the file and prepares it for interaction.
- **Example**: When a query is executed, the DBMS opens relevant data files to retrieve or update information.

```
# In a file-based DBMS like SQLite

import sqlite3

conn = sqlite3.connect('my_database.db')
```

3. File Reading

- **Purpose**: Retrieve data from files.
- **Operation**: Involves reading records or data blocks from the file into memory.
- **Example**: Reading records from a table file in a relational database

SELECT * FROM my_table;

4. File Writing

- **Purpose**: Modify or insert data into files.
- **Operation**: Involves writing data or updates to the file.
- **Example**: Inserting a new record into a table, which involves writing to the data file

INSERT INTO my_table (id, name) VALUES (1, 'Alice');

5. File Updating

- **Purpose**: Modify existing data within files.
- **Operation**: Updates records in the file, often involving locating the record, applying changes, and rewriting the data.
- **Example**: Updating a record's value in a table file.

UPDATE my_table SET name = 'Bob' WHERE id = 1;

6. File Deletion

- **Purpose**: Remove data or entire files.
- **Operation**: Involves deleting records from files or removing entire files from the filesystem.
- **Example**: Deleting a record from a table or dropping a table

DELETE FROM my_table WHERE id = 1;

DROP TABLE my_table;

change position:

When talking about changing the position of files or data within a Database Management System (DBMS), the context can vary. It could involve moving or reorganizing files on disk, reordering data within files, or modifying the

layout of data structures. Here's a breakdown of what "changing position" could mean in different contexts:

1. Changing File Location on Disk

- **Purpose**: Move database files to different locations on disk for reasons like performance optimization, reorganizing storage, or following administrative policies.
- **Operation**: This typically involves stopping the DBMS, moving the files, and updating the DBMS configuration to point to the new file locations.

Example (General Steps for Relocating Database Files):

1. **Stop the DBMS**: Ensure no active connections are using the files.
2. **Move Files**: Physically move the files to the new location.
3. **Update Configuration**: Modify the DBMS configuration files or settings to reflect the new file locations.
4. **Restart the DBMS**: Start the DBMS and verify that it's using the new file locations correctly.

ALTER DATABASE my_database

MODIFY FILE (NAME = 'my_database_data', FILENAME = '/new/location/my_database_data.mdf')

- **Changing Position of Files**: Moving files on disk or reorganizing their physical layout for performance or administrative reasons.
- **Reorganizing Data**: Optimizing or defragmenting data within files to improve access and performance.
- **Changing Data Order**: Modifying how data is sorted or indexed within tables.
- **Renaming and Moving Data**: Managing file names and transferring data between tables or databases.

controlling file I/O:

Controlling file input/output (I/O) is crucial for efficient file management and performance in database management systems (DBMS) and general programming. Here's a comprehensive guide to controlling file I/O, focusing on key concepts and techniques:

1. File Opening and Closing

- **Purpose**: To access files for reading, writing, or both. Properly closing files after operations is essential to free system resources and ensure data integrity.
- **Operation**: Use appropriate functions to open and close files

Opening a file for reading

file = open('example.txt', 'r')

Reading from the file

content = file.read()

Closing the file

file.close()

2. File Modes

- **Purpose**: Determine how the file is accessed (read, write, append, etc.).
- **Operation**: Specify the mode when opening a file.

Common Modes (Python):

- 'r': Read (default mode, file must exist)
- 'w': Write (creates a new file or truncates an existing file)
- 'a': Append (writes to the end of the file)

- 'b': Binary mode (e.g., 'rb', 'wb' for binary files)

```
# Opening a file for writing (creates or truncates)
with open('example.txt', 'w') as file:
    file.write('Hello, world!')
```

File Operations: Includes opening, closing, reading, writing, and managing file access modes.

Buffering: Improves performance by temporarily storing data.

File Positioning: Controls where in the file you are reading or writing.

File Locking: Manages concurrent access to prevent conflicts.

Error Handling: Ensures graceful management of errors during file operations.

Compression: Reduces file size for efficiency.

File System Operations: Handles file and directory management tasks.

Asynchronous I/O: Provides non-blocking file operations for improved performance.

Manipulating file paths:

Manipulating file paths is a common task in programming, especially when working with file systems. It involves constructing, modifying, and managing paths to files and directories. This is essential for tasks like reading and writing files, organizing data, and ensuring cross-platform compatibility. Here's an overview of how to manipulate file paths effectively using various programming languages and tools.

1. Basic Concepts

- **File Path**: A string that specifies the location of a file or directory in the filesystem. It can be absolute (full path from the root) or relative (path from the current directory).
- **Directory Separator**: The character used to separate directories in a path. It's / on Unix-like systems (Linux, macOS) and \ on Windows.

2. Python Path Manipulation

Python provides several libraries for handling file paths:

Using os.path

The os.path module offers functions to manipulate file paths. Here's how to use it:

- **Join Paths**: Combine directory names and filenames.

```
import os

path = os.path.join('folder', 'subfolder', 'file.txt')
```

Split Paths: Separate the directory and filenames,

```
directory, filename = os.path.split('/folder/subfolder/file.txt')
```

Using pathlib

The pathlib module (introduced in Python 3.4) provides an object-oriented approach to path manipulation:

Create Path Objects: Instantiate Path objects

```
from pathlib import Path

p = Path('folder') / 'subfolder' / 'file.txt'
```

Access Parts of the Path: Retrieve different components of the path

```
p.name       # 'file.txt'
p.stem       # 'file'
p.suffix     # '.txt'
p.parent     # 'folder/subfolder'
```

Check Path Properties: Determine if the path is a file or directory.

```
p.exists()    # Check if the path exists
p.is_file()   # Check if it's a file
p.is_dir()    # Check if it's a directory
```

UNIT V

Transactions: Transaction concept – Transaction State – Implementation of Atomicity and Durability – Concurrent Executions – Serializability – Testing for Serializability. Concurrency Control: Lock-Based Protocols – TimestampBased Protocols. Recovery System: Failure Classification – Storage Structure – Recovery and Atomicity – Log-Based Recovery – Shadow Paging.

Transaction concept:

In Database Management Systems (DBMS), a transaction is a sequence of operations performed as a single logical unit of work. The transaction concept is fundamental to ensuring the consistency, reliability, and integrity of a database. Here's a detailed look at the concept of transactions in DBMS:

Key Concepts of Transactions

1. **Transaction Definition**
    - A transaction is a set of database operations that are executed as a single unit. These operations could include reading or writing data.
    - The transaction is intended to be atomic, meaning either all of its operations are completed successfully, or none are applied (rolled back).
2. **ACID Properties** Transactions in a DBMS adhere to the ACID properties to ensure reliable processing:
    - **Atomicity**:
        - A transaction is atomic; it either completes entirely or does not execute at all. If any part of the transaction fails, the entire transaction is rolled back to maintain the database's consistency.
        - **Example**: If a bank transfer transaction deducts money from one account but fails to deposit it into another,

atomicity ensures that the money is neither lost nor created. Both operations are rolled back if one fails.

- **Consistency**:
  - A transaction transforms the database from one consistent state to another. The database must satisfy all integrity constraints before and after the transaction.
  - **Example**: If a transaction updates account balances, it must maintain the overall consistency of the account balances, ensuring that no funds are arbitrarily created or lost.
- **Isolation**:
  - Transactions are isolated from one another. Intermediate results of a transaction should not be visible to other transactions until the transaction is complete.
  - **Example**: If two transactions are updating the same account balance, isolation ensures that each transaction operates as if it were the only transaction running at that time.
- **Durability**:
  - Once a transaction is committed, its effects are permanent, even in the event of a system failure.
  - **Example**: After a successful bank transfer transaction is committed, the changes to account balances will persist despite any subsequent system crashes.

Transaction State:

In a Database Management System (DBMS), a transaction progresses through various states during its lifecycle. Understanding these states is crucial for managing transactions effectively and ensuring database consistency. Here's an overview of the transaction states:

1. Active

- **Description**: The transaction is in progress and has not yet been completed, committed, or aborted.

- **Characteristics**:
    - The transaction is executing its operations, which could include reading or writing data.
    - The transaction can still be committed or rolled back.

2. Partially Committed

- **Description**: The transaction has executed its final operation but has not yet reached the commit stage.
- **Characteristics**:
    - All changes made by the transaction are tentative and may still be undone.
    - The database is in an intermediate state, reflecting the results of the transaction's operations but not yet finalized.

3. Committed

- **Description**: The transaction has successfully completed all operations and the changes have been permanently applied to the database.
- **Characteristics**:
    - All changes made by the transaction are now visible to other transactions.
    - The transaction cannot be rolled back; its effects are durable and persistent.

4. Failed

- **Description**: The transaction has encountered an error or issue that prevents it from completing successfully.
- **Characteristics**:
    - The transaction may be aborted or rolled back to undo any changes made.
    - The failure might be due to system errors, application errors, or violations of database constraints.

5. Aborted

- **Description**: The transaction has been rolled back, and all changes made during the transaction have been undone.
- **Characteristics**:
    - The database state is restored to what it was before the transaction began.
    - An aborted transaction does not affect the database state; it effectively cancels the transaction's operations.

2. Partially Committed

- **Description**: The transaction has executed its final operation but has not yet reached the commit stage.
- **Characteristics**:
    - All changes made by the transaction are tentative and may still be undone.

3. Committed

- **Description**: The transaction has successfully completed all operations and the changes have been permanently applied to the database.
- **Characteristics**:
    - All changes made by the transaction are now visible to other transactions.
    - The transaction cannot be rolled back; its effects are durable and persistent.

4. Failed

- **Description**: The transaction has encountered an error or issue that prevents it from completing successfully.
- **Characteristics**:
    - The transaction may be aborted or rolled back to undo any changes made.
    - The failure might be due to system errors, application errors, or violations of database constraints.

5. Aborted

- **Description**: The transaction has been rolled back, and all changes made during the transaction have been undone.
- **Characteristics**:
    - The database state is restored to what it was before the transaction began.
    - An aborted transaction does not affect the database state; it effectively cancels the transaction's operations.
    -

Implementing Atomicity and Durability

Implementing **Atomicity** and **Durability** in a Database Management System (DBMS) ensures that transactions are reliable, consistent, and resistant to system failures. Here's how these two properties are implemented:

1. Atomicity

Atomicity ensures that a transaction is an all-or-nothing operation. Either all of its operations are completed successfully, or none are applied if an error occurs. This is typically achieved through the following mechanisms:

1.1. Transaction Log (Write-Ahead Logging)

- **Concept**: Before any changes are applied to the database, a transaction log records all operations. This log ensures that if a transaction fails, the system can roll back to a consistent state.
- **Implementation**:
    - **Before Writing**: The DBMS writes a log entry describing the operation (e.g., a new record insertion) before making any changes to the actual database.
    - **Commit Record**: Once the transaction is successfully completed, a commit record is written to the log, indicating that the transaction can be permanently applied.

    - **Rollback**: If the system crashes or the transaction fails, the DBMS uses the log to undo any

- **Atomicity** ensures that a transaction is fully completed or fully rolled back, using mechanisms like transaction logs and the Two-Phase Commit protocol.
- **Durability** guarantees that committed transactions persist even after system failures, utilizing techniques such as Write-Ahead Logging, Checkpoints, and redundant storage.

Concurrent executions:

Concurrent executions in a Database Management System (DBMS) refer to the simultaneous execution of multiple transactions or queries. Handling concurrency effectively is crucial for maintaining the integrity and performance of the database. Here's an overview of how concurrency is managed, including concepts, challenges, and techniques:

1. Concepts of Concurrency

1.1. Concurrent Transactions

- **Definition**: Multiple transactions are executed at the same time, possibly interacting with the same data.
- **Objective**: Maximize system utilization and throughput by overlapping transactions.

1.2. Isolation Levels

- **Definition**: Isolation levels define how and when the changes made by one transaction become visible to other concurrent transactions.
- **Types**:
    - **Read Uncommitted**: Transactions can read uncommitted changes from other transactions. This level is the least restrictive but can lead to dirty reads.
    - **Read Committed**: Transactions can only read committed changes from other transactions. It avoids dirty reads but might still face non-repeatable reads.

- **Repeatable Read**: Ensures that if a transaction reads a record, it will see the same data if it reads again. This prevents non-repeatable reads but might still face phantom reads.
- **Serializable**: The strictest level, where transactions are executed as if they were run sequentially. It ensures consistency but can reduce concurrency.

2. Challenges in Concurrency

2.1. Lost Updates

- **Description**: Occurs when two transactions update the same data concurrently, and one update overwrites the other's changes.
- **Example**: Two transactions both modify an account balance, but only one update is saved.

2.2. Dirty Reads

- **Description**: Occurs when a transaction reads data modified by another transaction that has not yet been committed. If the other transaction is rolled back, the read data becomes invalid.
- **Example**: A transaction reads an account balance that is modified by another transaction that is later rolled back.

2.3. Non-Repeatable Reads

- **Description**: Occurs when a transaction reads the same data multiple times, and the data changes in between reads due to other transactions.
- **Example**: A transaction reads a customer's address, but another transaction updates the address before the first transaction completes.

2.4. Phantom Reads

- **Description**: Occurs when a transaction reads a set of rows that match a certain condition, but another transaction inserts or deletes rows that match the condition before the first transaction completes.
- **Example**: A transaction reads all orders for a particular date, but another transaction inserts new orders for that date.

3. Concurrency Control Techniques

3.1. Locking Mechanisms

Locks: Mechanisms to control access to data by multiple transactions. Common types include:

- **Shared Locks**: Allow multiple transactions to read but not modify data.
    - **Example**: Multiple transactions can read the same account balance, but none can update it.
- **Exclusive Locks**: Allow a single transaction to read and modify data, blocking other transactions from accessing the data.
    - **Example**: A transaction that updates an account balance acquires an exclusive lock, preventing others from reading or modifying the balance until the transaction is complete.
- **Two-Phase Locking (2PL)**: A protocol where a transaction first acquires all necessary locks (growing phase) and then releases them (shrinking phase). This prevents deadlocks and ensures serializability.
    - **Example**: A transaction locks the rows it needs and does not release any locks until it is ready to commit.

3.2. Timestamp Ordering

- **Concept**: Transactions are ordered based on timestamps, and their operations are applied according to this order.
- **Implementation**:
    - **Timestamp-based Protocol**: Each transaction gets a unique timestamp. The system uses these timestamps to decide the order of operations.
    - **Conflict Resolution**: If a transaction attempts to perform an operation that conflicts with an earlier transaction, it is rolled back and restarted with a new timestamp.

Example:

plaintext

Copy code

Transaction T1 (timestamp t1)

Transaction T2 (timestamp t2)

If T1 and T2 conflict:

Apply operations based on their timestamps

3.3. Optimistic Concurrency Control

- **Concept**: Transactions execute without locks but validate their changes before committing. If validation fails (e.g., due to conflicts with other transactions), the transaction is rolled back and retried.
- **Implementation**:
    - **Read Phase**: The transaction reads data and performs its operations.
    - **Validation Phase**: Before committing, the system checks for conflicts with other transactions.
    - **Write Phase**: If validation is successful, changes are applied; otherwise, the transaction is rolled back.

Example:

plaintext

Copy code

Transaction reads data

Transaction performs operations

Validation checks for conflicts

If no conflicts, commit changes

Else, rollback and retry

3.4. Multiversion Concurrency Control (MVCC)

- **Concept**: Each transaction works with a snapshot of the database. Multiple versions of data are maintained to handle concurrent read and write operations.

- **Implementation**:
    - **Snapshot Isolation**: Each transaction sees a consistent view of the data as it was at the start of the transaction.
    - **Versioning**: The database maintains different versions of data items to handle concurrent updates.

Example:

plaintext

Copy code

Transaction T1 reads data version 1

Transaction T2 writes data, creating version 2

Transaction T1 continues with version 1, ensuring consistency

4. Deadlock Management

Deadlock: A situation where two or more transactions are waiting for each other to release locks, leading to a standstill.

- **Deadlock Prevention**: Design protocols to avoid situations that could lead to deadlocks. For example, always acquire locks in a predefined order.
- **Deadlock Detection**: Periodically check for deadlocks and resolve them by aborting one or more transactions to break the cycle.
- **Deadlock Recovery**: After detecting a deadlock, rollback one of the transactions to resolve the deadlock and allow the remaining transactions to proceed.

Example:

plaintext

Copy code

Transaction T1 and T2 are in deadlock:

- Detect deadlock using a wait-for graph

- Abort one transaction to resolve the deadlock

Serializability:

Serializability is a fundamental concept in database management systems (DBMS) related to the correctness of concurrent transactions. It ensures that the outcome of executing multiple transactions concurrently is the same as if they were executed serially (one after the other) without overlapping.

Concept of Serializability

Serializability guarantees that the execution of concurrent transactions preserves the consistency of the database. In other words, even though transactions are executed simultaneously, their effects are as if they were executed in some sequential order.

Types of Serializability

1. **Conflict Serializability**
    - **Definition**: A schedule (sequence of operations) is conflict-serializable if it can be transformed into a serial schedule by swapping non-conflicting operations.
    - **Conflicting Operations**: Two operations are considered to conflict if they meet the following criteria:
        - They belong to different transactions.
        - They access the same data item.
        - At least one of them is a write operation.
    - **Example**:

plaintext

Copy code

Transaction T1: Read(A), Write(A)

Transaction T2: Read(A), Write(A)

2. In this case, the operations of T1 and T2 conflict because they both read and write the same data item (A). To check for conflict serializability, you can construct a serialization graph or precedence graph and ensure it is acyclic.

3. **View Serializability**
    - **Definition**: A schedule is view-serializable if it is equivalent to a serial schedule in terms of the data values read by transactions.
    - **Criteria**:
        - **Initial Reads**: Each transaction reads the initial values of data items in the same order as in a serial schedule.
        - **Reads of Writes**: Each transaction reads the values written by other transactions in the same order as in a serial schedule.
        - **Final Writes**: The final write of each data item in the schedule matches the final write in the corresponding serial schedule.
    - **Example**:

plaintext

Copy code

Transaction T1: Write(A), Read(B), Write(B)

Transaction T2: Read(A), Write(A), Read(B)

4. Here, you would check if the result of T1 and T2 can be achieved by some serial ordering of T1 and T2, considering the above criteria.

Checking Serializability

1. Serialization Graph (Precedence Graph)

- **Definition**: A directed graph used to check conflict serializability.
- **Nodes**: Represent transactions.
- **Edges**: Represent dependencies between transactions. An edge from T1 to T2 indicates that T1 must precede T2 due to conflicting operations.

Steps:

- Create a node for each transaction.

- Add a directed edge from Ti to Tj if there is a conflict between Ti and Tj's operations such that Ti's operation must occur before Tj's operation.
- Check if the graph is acyclic. If it is acyclic, the schedule is conflict-serializable; otherwise, it is not.

Example:

plaintext

Copy code

T1 -> T2 (conflict due to write-read or write-write)

2. Serializability Tests

- **Conflict-Serializability Test**: Use the serialization graph to determine conflict serializability.
- **View-Serializability Test**: Compare the schedule with possible serial schedules to verify if it is view-serializable.

Techniques for Ensuring Serializability

1. Two-Phase Locking (2PL)

- **Concept**: A concurrency control protocol that ensures conflict-serializability by acquiring all required locks before releasing any.
- **Phases**:
  - **Growing Phase**: A transaction acquires all necessary locks.
  - **Shrinking Phase**: After the transaction starts releasing locks, it cannot acquire any new locks.

Example:

plaintext

Copy code

Transaction T1:

Acquire lock on A

Read A

Write A

Release lock on A

2. Serializable Schedules

- **Definition**: Schedules that can be transformed into a serial schedule through simple reordering of non-conflicting operations.
- **Implementation**: The DBMS ensures that schedules are serializable by using protocols that guarantee serializability, such as 2PL or using advanced concurrency control techniques.

3. Strict 2PL

- **Concept**: A stricter version of Two-Phase Locking where transactions are not allowed to release any locks until they have completed their execution. This guarantees serializability.

Example:

plaintext

Copy code

Transaction T1:

Acquire locks on all data items needed

Perform operations

Release all locks

Concurrency control:

Concurrency control in database management systems (DBMS) ensures that multiple transactions can execute concurrently without violating database consistency. Two primary methods for concurrency control are **Lock-Based Protocols** and **Timestamp-Based Protocols**. Each method has its own approach to managing concurrent access to data.

1. Lock-Based Protocols

Lock-based protocols use locks to manage concurrent access to data. They ensure that transactions are isolated from each other and that

conflicts are handled appropriately. Here's an overview of key lock-based protocols:

1.1. Basic Locking Mechanisms

- **Shared Lock (S-Lock)**: Allows a transaction to read a data item but not modify it. Multiple transactions can hold a shared lock on the same data item simultaneously.
    - **Example**: Transactions T1 and T2 both read data item X with a shared lock.
- **Exclusive Lock (X-Lock)**: Allows a transaction to read and write a data item. An exclusive lock is held by only one transaction at a time, and no other locks (shared or exclusive) can be granted on that data item.
    - **Example**: Transaction T1 updates data item X and holds an exclusive lock, preventing other transactions from accessing X until T1 releases the lock.

1.2. Two-Phase Locking (2PL)

- **Concept**: A protocol that ensures conflict-serializability by dividing the transaction execution into two phases: the growing phase and the shrinking phase.
    - **Growing Phase**: The transaction acquires all the locks it needs.
    - **Shrinking Phase**: After releasing a lock, the transaction cannot acquire any new locks.
- **Strict Two-Phase Locking**: A stricter variant where a transaction is not allowed to release any locks until it has completed its execution. This ensures serializability by preventing any reordering of operations that could lead to inconsistencies.

Example:

plaintext

Copy code

Transaction T1:

Growing Phase:

Acquire lock on A

Acquire lock on B

Perform operations (e.g., read/write A and B)

Shrinking Phase:

Release lock on B

Release lock on A

1.3. Deadlock Handling

- **Deadlock Prevention**: Techniques to avoid deadlock situations, such as requiring transactions to acquire locks in a predefined order.
- **Deadlock Detection**: Methods to detect deadlocks using wait-for graphs and periodically checking for cycles.
- **Deadlock Recovery**: Resolving deadlocks by aborting one or more transactions involved in the deadlock cycle and rolling back their changes.

Example:

plaintext

Copy code

Deadlock Detection:

- Construct wait-for graph

- Detect cycles in the graph

- Abort one transaction to break the cycle

2. Timestamp-Based Protocols

Timestamp-based protocols use timestamps to manage the order of transaction operations and maintain serializability. Each transaction is assigned a unique timestamp when it starts, which is used to resolve conflicts.

2.1. Basic Timestamp Ordering

- **Concept**: Transactions are ordered based on their timestamps. Operations are executed according to this order, ensuring that the schedule is equivalent to some serial schedule.
- **Rules**:
    - **Read Rule**: A transaction T with timestamp TS(T) can read a data item X if X's last write operation was done by a transaction with a timestamp less than TS(T).
    - **Write Rule**: A transaction T with timestamp TS(T) can write to a data item X if no other transaction with a timestamp greater than TS(T) has already read or written X.

Example:

plaintext

Copy code

Transaction T1 (TS = 1) writes X

Transaction T2 (TS = 2) reads X (allowed because T1's timestamp < T2's timestamp)

2.2. Thomas's Write Rule

- **Concept**: An extension of basic timestamp ordering that allows some transactions to write to a data item even if it would violate basic timestamp rules, provided the data item's value will not be needed by transactions with later timestamps.
- **Rule**: If a transaction T with timestamp TS(T) tries to write a data item X, it can do so if no later transaction will need the old value of X.

Example:

plaintext

Copy code

Transaction T1 (TS = 1) writes X

Transaction T2 (TS = 2) can overwrite X if it does not affect future transactions

2.3. Validation-Based Protocols

- **Concept**: Transactions execute without locks but are validated before committing to ensure serializability. Transactions are divided into three phases: read, validate, and write.
- **Phases**:
    - **Read Phase**: Transactions execute their read and write operations on a local copy of the data.
    - **Validation Phase**: Before committing, the transaction is validated against a serializable schedule.
    - **Write Phase**: If validation is successful, changes are applied to the database.

Example:

plaintext

Copy code

Transaction T1:

Read data

Perform operations

Validate against serializable schedule

Commit if validation is successful

Comparison of Lock-Based and Timestamp-Based Protocols

- **Lock-Based Protocols**:
    - **Pros**: Simple to implement and understand. Provides strict guarantees on conflict-serializability.
    - **Cons**: Can lead to deadlocks and requires complex deadlock handling mechanisms. Performance may degrade due to locking overhead.
- **Timestamp-Based Protocols**:
    - **Pros**: Avoids deadlocks and is easier to manage in distributed systems. Provides guarantees on serializability based on timestamps.

- **Cons**: May lead to higher abort rates (especially with basic timestamp ordering) and can be more complex to implement compared to simple locking protocols.

Recovery System: Failure Classification:

In database management systems (DBMS), the recovery system is essential for maintaining data integrity and consistency in the face of various types of failures. Failure classification helps in understanding the nature of problems that can occur and determining the appropriate recovery strategies. Here's a detailed overview of the different types of failures:

1. Failure Classification

1.1. Transaction Failures

- **Definition**: Failures that occur due to issues within individual transactions.
- **Causes**:
  - **Logical Errors**: Errors in the transaction logic or code that cause it to fail.
  - **Constraint Violations**: Transactions that attempt to violate database constraints (e.g., unique constraints, foreign key constraints).
  - **Deadlocks**: Situations where two or more transactions are waiting for each other to release locks, resulting in a standstill.
- **Recovery Approach**: Rollback the failed transaction to its initial state to ensure that its partial changes do not affect the database.
  - **Example**: If a transaction fails while transferring funds between accounts, the transaction is rolled back to ensure no partial transfers are made.

1.2. System Failures

- **Definition**: Failures that affect the entire database system or its underlying resources.
- **Causes**:

    - **Operating System Crashes**: Failures in the operating system that cause the DBMS to stop functioning.
    - **Hardware Failures**: Issues such as disk crashes, memory corruption, or processor malfunctions.
    - **Power Failures**: Unexpected power outages leading to system shutdowns.
- **Recovery Approach**: Use transaction logs to restore the database to a consistent state. Apply committed changes and roll back any uncommitted transactions.
    - **Example**: After a system crash, the database recovery process involves replaying the log entries to restore the database to the last consistent state.

1.3. Media Failures

- **Definition**: Failures that impact the storage media where database files are kept.
- **Causes**:
    - **Disk Drive Failures**: Physical problems with the storage disk leading to data corruption or loss.
    - **File Corruption**: Corruption of database files due to hardware issues or software bugs.
- **Recovery Approach**: Restore the database from backups and apply transaction logs to recover changes made since the last backup.
    - **Example**: If a disk storing the database becomes corrupted, the database is restored from the most recent backup, and any changes recorded in the logs are reapplied to bring it up to date.

1.4. Human Errors

- **Definition**: Failures caused by mistakes or accidental actions performed by users or administrators.
- **Causes**:

    - **Accidental Deletion**: Unintended deletion of data by users or administrators.
    - **Incorrect Modifications**: Incorrect updates or changes made to the database.
- **Recovery Approach**: Restore from backups or use point-in-time recovery techniques to revert to a state before the error occurred.
    - **Example**: If a user accidentally deletes important records, the database can be restored from a backup taken just before the deletion occurred.

1.5. Application Failures

- **Definition**: Failures related to issues within the application interacting with the database.
- **Causes**:
    - **Bugs in Application Code**: Application bugs that lead to incorrect data operations or queries.
    - **Application Configuration Issues**: Misconfigurations that affect how the application interacts with the database.
- **Recovery Approach**: Correct application bugs or configuration issues and use the recovery system to handle any inconsistencies or data corruption resulting from application failures.
    - **Example**: If an application bug causes incorrect data to be written to the database, correcting the bug and using logs to fix the inconsistencies would be necessary.

Recovery and Atomicity:

Recovery and atomicity are fundamental concepts in database management systems (DBMS) that ensure data integrity and consistency in the face of various types of failures. Here's a detailed overview of these concepts and how they are interrelated:

1. Atomicity

Atomicity is one of the core properties of transactions in a DBMS, encapsulated in the ACID properties (Atomicity, Consistency, Isolation, Durability). Atomicity ensures that a transaction is treated as a single, indivisible unit of work. This means that either all operations within the transaction are successfully completed, or none are applied. If a transaction cannot be completed successfully, it should be rolled back to its initial state.

Key Aspects of Atomicity

- All-or-Nothing Principle: A transaction is atomic in the sense that it either completes entirely or does not happen at all. This ensures that partial changes made by a transaction do not affect the database if the transaction fails.
- Rollback: If an error occurs during a transaction, the system will roll back the transaction, undoing all operations to restore the database to its state before the transaction began.
- Commit: If a transaction completes successfully, a commit operation is performed to make all changes permanent.

Example

Consider a transaction that transfers money from Account A to Account B. The transaction consists of two operations:

1. Deduct the amount from Account A.
2. Add the amount to Account B.

If the system crashes after deducting from Account A but before adding to Account B, atomicity ensures that the transaction is rolled back, and Account A is restored to its original state, thus maintaining consistency.

2. Recovery

Recovery in a DBMS involves restoring the database to a consistent state after a failure. This process is crucial for handling various types of failures, such as system crashes, media failures, or transaction failures. Recovery mechanisms use transaction logs and backup files to achieve this.

Key Concepts in Recovery

- Transaction Logs: Logs record all changes made to the database, including the start, commit, and rollback of transactions. They are essential for recovering the database to a consistent state.
    - Write-Ahead Logging (WAL): Ensures that changes are written to the log before they are applied to the database. This guarantees that the log can be used to redo or undo changes as needed.
- Checkpointing: A checkpoint is a saved state of the database at a specific point in time. It helps to reduce the recovery time by providing a consistent starting point.
- Undo and Redo Operations:
    - Undo: Reverts changes made by uncommitted transactions during recovery.
    - Redo: Reapplies changes made by committed transactions to ensure that they are reflected in the recovered database.

Recovery Process

Identify the Point of Failure: Determine the point at which the failure occurred using the logs and checkpoint information.

Shadow Paging:

Shadow Paging is a technique used in database management systems (DBMS) to provide a reliable mechanism for recovery and to ensure the atomicity and durability of transactions. It is a form of recovery technique that avoids some of the complexities associated with traditional logging methods.

How Shadow Paging Works

Shadow paging maintains two page tables:

1. **Current Page Table**: This table reflects the most recent state of the database, mapping logical pages to their current physical locations.
2. **Shadow Page Table**: This is a snapshot of the page table from a point in time before a transaction started, representing the state of the database before the transaction.

Process Overview

1. **Transaction Start**:
    - When a transaction begins, a copy of the **current page table** is made and saved as the **shadow page table**.
2. **During the Transaction**:
    - Changes made by the transaction are applied to new or modified pages.
    - The **current page table** is updated to reflect these changes. However, the **shadow page table** remains unchanged, preserving the database's state before the transaction.
3. **Transaction Commit**:
    - Upon successful commit, the **shadow page table** is discarded, and the **current page table** becomes the official page table.
    - The changes made by the transaction are thus made permanent.
4. **Transaction Abort or Failure**:

- If the transaction fails or is aborted, the **current page table** is discarded.
- The system reverts to using the **shadow page table** to restore the database to its pre-transaction state.

Advantages of Shadow Paging

- **Simplicity**: The shadow paging technique is conceptually simpler than logging-based recovery because it primarily involves managing and swapping page tables.
- **Consistency**: It inherently maintains database consistency by either keeping the database in its pre-transaction state or committing the transaction changes.
- **No Need for Logs**: Unlike traditional recovery methods that require extensive logging, shadow paging does not require logging of each change.

Disadvantages of Shadow Paging

- **Space Overhead**: Maintaining shadow page tables requires additional storage space, which can be significant for large databases.
- **Performance Impact**: Copying the page table and managing shadow tables can lead to increased overhead, impacting performance.
- **Complexity with Concurrent Transactions**: Managing shadow tables in a multi-user environment with concurrent transactions can be complex and might require additional mechanisms to handle concurrency.