**Module-6**

**Basic Information:**

Two-Phase Locking (2PL): 2PL is a widely used concurrency control protocol in database management systems (DBMS). It ensures serializability by enforcing a two-phase process: a growing phase where transactions acquire locks on data items before accessing them, and a shrinking phase where locks are released only after completing the transaction. This protocol allows transactions to obtain exclusive locks (write locks) or shared locks (read locks) on data items, preventing conflicts and maintaining consistency. However, it can lead to potential deadlocks and reduced concurrency under heavy contention for locks.

Timestamp-Based Concurrency Control (TBCC): TBCC is an alternative concurrency control algorithm used in DBMS to ensure serializability. It assigns unique timestamps to transactions based on their start times or arrival order. Transactions are allowed to access data items if their timestamps precede the timestamps of previous conflicting transactions. This approach promotes high concurrency by allowing transactions to proceed concurrently as long as they don't conflict in their read and write operations. TBCC typically operates at the granularity of individual data items, making it suitable for fine-grained concurrency control. However, it may suffer from increased abort rates in scenarios with frequent conflicts.

These two concurrency control mechanisms play essential roles in ensuring data consistency and transaction isolation in database systems, each with its own set of characteristics and trade-offs. The choice between them depends on factors such as the nature of the application, concurrency requirements, and system performance considerations.

**Algorithms Used:**

**2 phase lock concurrency control:**

class Transaction:

initialize with id

Method acquire\_lock(resource):

If phase is Growing:

If resource not in locks:

Add resource to locks

Print message: Transaction id acquired lock on resource

Else:

Print message: Transaction id already holds lock on resource

Else:

Print message: Transaction id cannot acquire lock during Shrinking phase

Method release\_lock(resource):

If resource in locks:

Remove resource from locks

Print message: Transaction id released lock on resource

Else:

Print message: Transaction id does not hold lock on resource

Method end\_transaction():

Set phase to Shrinking

Print message: Transaction id entered Shrinking phase

Method read\_write():

If phase is Growing:

Print message: Transaction id performing read/write operations

Else:

Print message: Transaction id cannot perform read/write operations during Shrinking phase

class ResourceManager:

Initialize locked\_resources for ResourceManager

Method acquire\_lock(resource):

If resource not in locked\_resources:

Add resource to locked\_resources

Print message: Resource locked

Return True

Else:

Print message: Resource already locked

Return False

Method release\_lock(resource):

If resource in locked\_resources:

Remove resource from locked\_resources

Print message: Resource released

Return True

Else:

Print message: Resource not locked

Return False

**Timestamp based concurrency control:**

Initialize the TimestampConcurrencyControl

Create several Transaction objects with unique IDs and timestamps

For each Transaction object:

Start the transaction using the TimestampConcurrency

Check for conflicts with other transactions:

If there are no conflicts:

Start the transaction using the TimestampConcurrency

End the transaction using the TimestampConcurrency

Check for conflicts with other transactions:

If there are no conflicts:

Start the transaction using TimestampConcurrency

End all transactions

**Comparison:**

1. **Principle:**
   * **2PL:** The 2PL protocol ensures serializability by ensuring that transactions acquire locks on data items before accessing them and release these locks only after completing the transaction. It has two phases: the growing phase, during which locks are acquired, and the shrinking phase, during which locks are released.
   * **TBCC:** TBCC uses timestamps assigned to transactions to determine their relative order. It ensures serializability by allowing a transaction to access a data item if its timestamp is earlier than the timestamp of the last transaction that modified that data item.
2. **Granularity:**
   * **2PL:** It supports both coarse-grained and fine-grained locking. Coarse-grained locking involves locking entire tables or large portions of data, while fine-grained locking involves locking individual data items.
   * **TBCC:** TBCC typically operates at the level of individual data items, making it suitable for fine-grained concurrency control.
3. **Concurrency:**
   * **2PL:** It can lead to more contention for locks, especially in scenarios where transactions hold locks for long periods of time, which can reduce concurrency and potentially lead to deadlocks.
   * **TBCC:** TBCC generally allows for higher concurrency since transactions can access different data items concurrently as long as they don't conflict in their read and write operations.
4. **Deadlock Handling:**
   * **2PL:** Deadlocks can occur in 2PL due to the locking mechanism. Most database systems implement deadlock detection and resolution mechanisms to handle deadlocks, such as timeouts or deadlock detection algorithms.
   * **TBCC:** TBCC does not inherently lead to deadlocks since it does not involve locking. However, deadlock detection and resolution mechanisms may still be necessary for other reasons in a database system.
5. **Ease of Implementation:**
   * **2PL:** Implementation of 2PL is generally straightforward, as it involves acquiring and releasing locks based on a predefined protocol. However, ensuring proper lock granularity and deadlock handling can require careful design.
   * **TBCC:** Implementation of TBCC can be more complex, especially in distributed systems where timestamps need to be synchronized across multiple nodes. Additionally, ensuring that transactions adhere to their assigned timestamps can require careful coordination.
6. **Performance:**
   * **2PL:** Performance of 2PL can degrade under heavy contention for locks, especially in scenarios with long-running transactions. However, it can perform well in scenarios with moderate concurrency and appropriately chosen lock granularity.
   * **TBCC:** TBCC can offer better performance in scenarios with high concurrency and short transactions, as it avoids the overhead of lock management. However, it may suffer from increased abort rates if transactions frequently conflict.

**Top of Form**

**2 phase locking:**

class Transaction:

    def \_\_init\_\_(self, id):

        self.id = id

        self.locks = set()

        self.phase = "Growing"

    def acquire\_lock(self, resource):

        if self.phase == "Growing":

            if resource not in self.locks:

                self.locks.add(resource)

                print(f"Transaction {self.id} acquired lock on resource {resource}.")

            else:

                print(f"Transaction {self.id} already holds lock on resource {resource}.")

        else:

            print(f"Transaction {self.id} cannot acquire lock during Shrinking phase.")

    def release\_lock(self, resource):

        if resource in self.locks:

            self.locks.remove(resource)

            print(f"Transaction {self.id} released lock on resource {resource}.")

        else:

            print(f"Transaction {self.id} does not hold lock on resource {resource}.")

    def end\_transaction(self):

        self.phase = "Shrinking"

        print(f"Transaction {self.id} entered Shrinking phase.")

    def read\_write(self):

        if self.phase == "Growing":

            print(f"Transaction {self.id} performing read/write operations.")

        else:

            print(f"Transaction {self.id} cannot perform read/write operations during Shrinking phase.")

class ResourceManager:

    def \_\_init\_\_(self):

        self.locked\_resources = set()

    def acquire\_lock(self, resource):

        if resource not in self.locked\_resources:

            self.locked\_resources.add(resource)

            print(f"Resource {resource} locked.")

            return True

        else:

            print(f"Resource {resource} already locked.")

            return False

    def release\_lock(self, resource):

        if resource in self.locked\_resources:

            self.locked\_resources.remove(resource)

            print(f"Resource {resource} released.")

            return True

        else:

            print(f"Resource {resource} not locked.")

            return False

if \_\_name\_\_ == "\_\_main\_\_":

    rm = ResourceManager()

    t1 = Transaction(1)

    t2 = Transaction(2)

    t1.acquire\_lock('A')

    t2.acquire\_lock('B')

    rm.acquire\_lock('A')

    rm.acquire\_lock('B')

    rm.acquire\_lock('C')

    rm.release\_lock('A')

    t1.release\_lock('A')

    t1.end\_transaction()

    t1.read\_write()

    t2.acquire\_lock('A')

    t2.release\_lock('B')

    t2.end\_transaction()

**Output:**

**A screenshot of a computer program

Description automatically generated**

**Timestamp based concurrency control:**

**Code:**

class Transaction:

    def \_\_init\_\_(self, id, timestamp):

        self.id = id

        self.timestamp = timestamp

    def get\_id(self):

        return self.id

    def get\_timestamp(self):

        return self.timestamp

class TimestampConcurrencyControl:

    def \_\_init\_\_(self):

        self.active\_transactions = []

    def start\_transaction(self, transaction):

        self.active\_transactions.append(transaction)

        print(f"Transaction {transaction.get\_id()} started at timestamp {transaction.get\_timestamp()}.")

    def end\_transaction(self, transaction):

        self.active\_transactions.remove(transaction)

        print(f"Transaction {transaction.get\_id()} ended at timestamp {transaction.get\_timestamp()}.")

    def check\_conflict(self, transaction):

        for active\_transaction in self.active\_transactions:

            if active\_transaction.get\_timestamp() > transaction.get\_timestamp():

                print(f"Conflict detected between Transaction {transaction.get\_id()} and Transaction {active\_transaction.get\_id()}.")

                return True

        return False

if \_\_name\_\_ == "\_\_main\_\_":

    concurrency\_control = TimestampConcurrencyControl()

    t1 = Transaction(1, 1)

    t2 = Transaction(2, 2)

    t3 = Transaction(3, 3)

    t4 = Transaction(4, 4)

    t5 = Transaction(5, 5)

    concurrency\_control.start\_transaction(t1)

    concurrency\_control.start\_transaction(t2)

    concurrency\_control.start\_transaction(t3)

    if not concurrency\_control.check\_conflict(t4):

        concurrency\_control.start\_transaction(t4)

    if not concurrency\_control.check\_conflict(t5):

        concurrency\_control.start\_transaction(t5)

    concurrency\_control.end\_transaction(t1)

    concurrency\_control.end\_transaction(t2)

    if not concurrency\_control.check\_conflict(t5):

        concurrency\_control.start\_transaction(t5)

    concurrency\_control.end\_transaction(t3)

    concurrency\_control.end\_transaction(t4)

    concurrency\_control.end\_transaction(t5)

**Output:**

A screen shot of a computer screen

Description automatically generated