LECTURE SESSION SIX

CONCURRENCY CONTROL TECHNIQUES

Learning Objectives

6.1 Introduction

Welcome back to class, last week we discussed about scheduling and serializability of concurrent transactions. In this session, we shall discuss various techniques used to ensure concurrency transactions do not conflict. I hope you are going to enjoy, welcome.

6.2 Learning Outcomes

At the end of this lecture, you should be able to:

1. Explain how database transactions are managed
2. Discuss concurrency control and the role it plays in maintaining the database's integrity
3. Apply locking methods to solve concurrency problems

6.3 Concurrency Control Techniques

Before we begin to discuss about the techniques of concurrency control, we shall first refresh ourselves on what concurrency control is all about. To start off our discussion I would like to pose to you the following trivial questions:

Questions:

1. Why do you think that we should have interleaving execution of transactions if it may lead to problems such as irrecoverable schedule, inconsistency and many more threats?
2. Why not just let it be Serial schedules and we may live peacefully, no complications at all?

Great!! I believed that you have responded that the objective of multi-user DBMS is to maximize the degree of concurrency or parallelism in the system in order to achieve high throughput and high resource utilization. On the other hand, serial schedules have low throughput and less resource utilization.

You can also recall that concurrency control is a mechanism or a method used to ensure that transactions are executed in a safe manner and follow the ACID rules. In order to achieve concurrency control, the DBMS must be able to ensure that only serializable, recoverable schedules are allowed, and that no actions of committed transactions are lost while undoing aborted transactions.

There are different concurrency control techniques, each of these techniques provide different advantages between the amount of concurrency they allow and the amount of overhead that they impose. We shall discuss the following techniques:

- Concurrency control with **Locking Methods**
- Concurrency Control with **Time Stamping Methods**
- Concurrency Control with **Optimistic Methods**

6.4 Locking Methods

Locking is a procedure used to control concurrent access to data. The locking protocol is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules. A transaction must acquire a lock prior to data access. The lock is released (Unlock) when the transaction is completed so that another transaction can lock the data item for its exclusive use. Locks are therefore used by a transaction to deny data access to other transactions and so prevent incorrect updates.

6.4.1 Types of Locks

The two main lock types are:

a. Binary Locks

Simple Lock-based protocol or Binary locking can be in one of two states i.e., the locked states (denoted by 1) and the unlocked state (denoted by 0). Locked objects are unavailable to other objects and this is managed by the DBMS. Unlocked objects however, are open to any order transaction. Its each transaction that locks its objects and the same unlocks it when complete. It is also possible to change DBMS default with LOCK TABLE and other SQL commands.

b. Shared/ Exclusive Locks

i. **Shared Lock (S-Lock),** also known as Read-only lock, is used during read operations since read operations cannot conflict. Since read operations

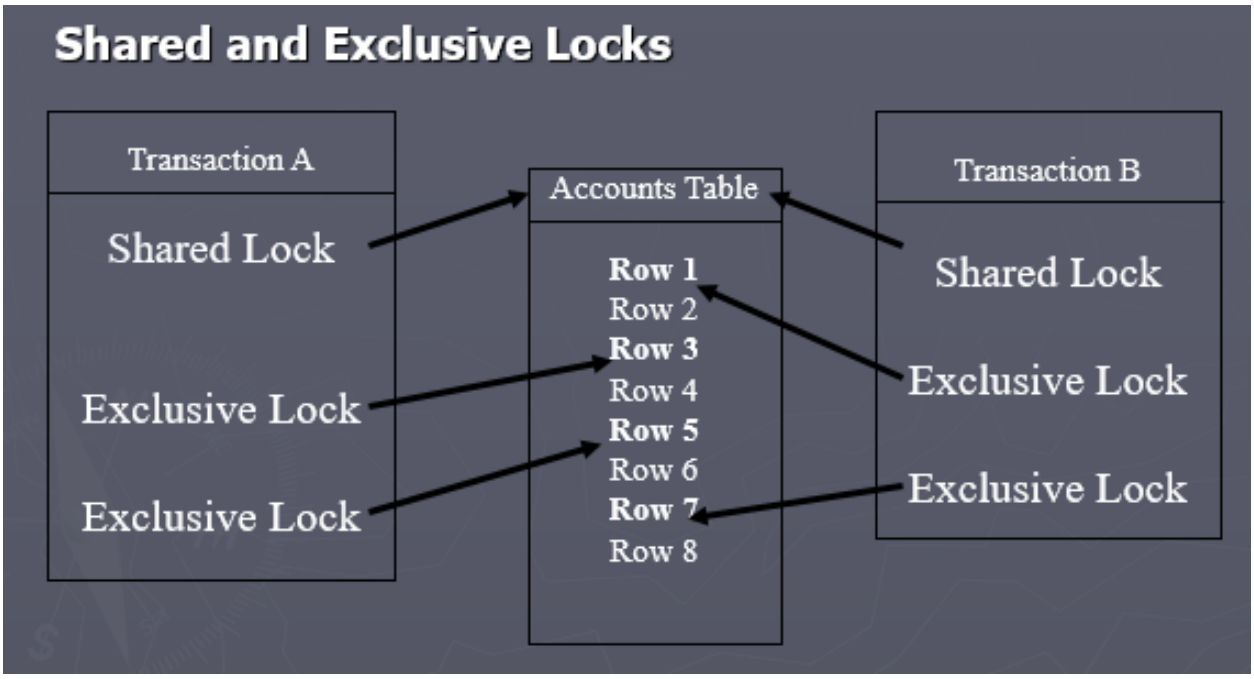
cannot conflict, it is permissible for more than one transaction to hold shared locks simultaneously on the same data item.

ii. **Exclusive Locks (X-Lock),** also Write lock allows data item that is locked with exclusive lock to be both read as well as written. As long as a transaction holds an exclusive lock no other transaction can read or update that data item. X-lock is requested using lock-X instruction.

Lock Manager

Transactions request for locks from the scheduler called lock manager. A Lock manager can be implemented as a separate process to which transactions send lock and unlock requests. The lock manager replies to a lock request by ending a lock grant messages (or a message asking the transaction to roll back, in case of a deadlock). The requesting transaction waits until its request is answered. The lock manager maintains a data structure called a lock table to record granted locks and pending requests

Example of a Lock Table:



The diagram below shows the Lock-compatibility Matrix:

	S	X
S	true	false
X	false	false

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions
- Any number of transactions can hold shared locks on an item. But if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.
- If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted.

The locking procedure is as follows:

1. Any transaction that needs to access a data item must first lock the item by requesting a shared lock for read only access or an exclusive lock for both read and write access.
2. If an item is not already locked by another transaction, the lock will be granted.
3. If the item is currently locked, the DBMS determines whether the request is compatible with the existing lock. If a shared lock is requested on an item that has already a shared lock on it, the request will be granted, otherwise the transaction must wait until the existing lock is released.
4. A transaction continues to hold a lock until it explicitly releases it either during execution or when it terminates (aborts or commits). It is only when the exclusive lock has been released that the effects of the write operation will be made visible to other transactions.

Example of a transaction performing locking:

T_2: lock-S*(A)*;
read *(A)*;
unlock*(A)*;
lock-S*(B)*;
read *(B)*;
unlock*(B)*;
display*(A+B)*

Note: Locking as above is not sufficient to guarantee serializability — if A and B get updated in-between the read of A and B, the displayed sum would be wrong.

Drawback of Simple Lock-based protocol (or Binary Locking)

Simple lock-based protocols have disadvantages in that:

i. They do not guarantee Serializability - schedules may follow the preceding rules but a non-serializable schedule may result.
ii. May lead to starvation and Deadlock.

Before we introduce 2-Phase Locking (2-PL) which will use the concept of Locks to avoid deadlock, we have said that by applying simple locking, we may not always produce serializable results and that it may lead to deadlock inconsistency. Let's discuss briefly about starvation and Deadlock.

Deadlock

A deadlock is an impasse that may result when two (or more) transactions are each waiting for locks to be released that are held by the other.

Consider the following example:

Example: Deadlock

T_3	T_4
lock-x (B)	
read (B)	
$B := B - 50$	
write (B)	
	lock-s (A)
	read (A)
	lock-s (B)
lock-x (A)	

In this partial schedule, T_3 holds an Exclusive lock over B, and T_4 holds a Shared lock over A. Consider line 7, T_4 requests for lock on B, while in line 8 T_3 requests lock on A. This as you may notice imposes a deadlock as none can proceed with their execution. neither T_3 nor T_4 can make progress — executing lock-S(B) causes T_4 to wait for T_3 to release its lock on B, while executing lock-X(A) causes T_3 to wait for T_4 to release its lock on A. Such a situation is called a deadlock. To handle a deadlock one of T_3 or T_4 must be rolled back/aborted and its locks released.

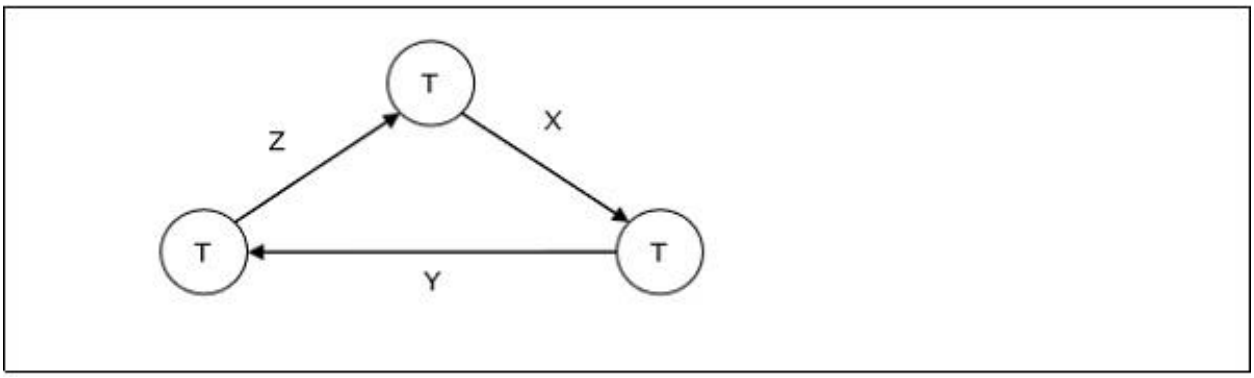
Take Note

Neither of the transactions in a deadlocked system can continue because each is waiting for a lock it cannot obtain until the other completes

Once a deadlock occurs, the applications involved cannot resolve the problem, instead the DBMS has to recognise that deadlock exists and break the deadlock in some way. The only way to break a deadlock is to abort one or more transactions.

A deadlock can be indicated by a cycle in the **wait-for-graph**. This is a directed graph in which the vertices denote transactions and the edges denote waits for data items.

Example: In the wait-for-graph below, transaction T_1 is waiting for data item X which is locked by T_3. T_3 is waiting for Y which is locked by T_2 and T_2 is waiting for Z which is locked by T_1. Hence, a waiting cycle is formed, and none of the transactions can proceed executing.



Activity

Examine the following transactions. Is it deadlocked or not? briefly explain your answer.

Time	**T_3**	**T_4**
t_1	begin_transaction	
t_2	Write_lock(bal_x)	begin_transaction
t_3	read(bal_x)	Write_lock(bal_y)
t_4	$bal_x = bal_x - 10$	read(bal_y)
t_5	write(bal_x)	$bal_y = bal_y + 100$
t_6	Write_lock(bal_y)	write(bal_y)
t_7	wait	Write_lock(bal_x)
t_8	wait	wait
t_9	wait	wait
t_{10}	wait	wait
t_{11}	**:**	**:**

Starvation

In addition to deadlocks, there is a possibility of starvation. Starvation occurs if the concurrency control manager is badly designed. For example: A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request are granted an S-lock on the same item. This leads to same transaction is repeatedly rolled back due to deadlocks. However, concurrency control manager can be designed to prevent starvation.

I hope you are now familiar with why we should study Concurrency Control Protocols. Moreover, you should be familiar with basics of Lock Based Protocols and problems with Simple Locking. Next, we'll discuss 2-PL and its categories, implementation along with the advantages and pitfalls of using them.

4.4.2 Two Phase (2PL) Locking Protocol

Now, having said that there are two types of Locks available Shared (S) and Exclusive (X)**.** If we implementing this lock system without any restrictions as we have done so far gives us the Simple Lock-based protocol (or Binary Locking).

To guarantee serializability, we must follow some additional protocol concerning the positioning of locking and unlocking operations in every transaction. This is where the concept of Two-Phase Locking(2-PL) comes into the picture, 2-PL ensures serializability. Now, let's dig deep! To ensure serializability, the 2- phase locking protocol defines how transaction acquire and relinquish locks. Although 2-phase locking guarantees serializability, it does not prevent deadlocks.

A transaction is said to follow the Two-Phase Locking protocol if Locking and Unlocking can be done in two phases:

(a) Growing phase in which, a transaction acquires all the required locks without unlocking/releasing any lock/data. Once all the locks have been acquired the transaction is in its locked point (lock point is the point where a transaction acquired its final lock).
(b) Shrinking phase in which a transaction releases existing locks and cannot obtain any new lock.

Rules governing the 2-Phase protocol are:

i. A transaction must acquire a lock on an item before operating on the item. The lock may be read or write, depending on the type of access needed.
ii. Once the transaction releases a lock, it can never acquire any new locks.
iii. Upgrading of locks can only take place in the growing phase.
iv. Downgrading can only take place during the shrinking phase.

Additional:

i. 2 transactions cannot have conflicting locks
ii. No unlock operation can proceed an unlock operation in the same transaction.
iii. No data is affected until all locks are obtained i.e., until the transaction is in the locked point.

An example of 2-phase locking with lock conversions (upgrade/downgrade):

First Phase(growing):

- can acquire a lock-S on item
- can acquire a lock-X on item
- can convert a lock-S to a lock-X (upgrade)

Second Phase(shrinking):

- can release a lock-S
- can release a lock-X
- can convert a lock-X to a lock-S (downgrade)

2-PL protocol assures serializability. But still relies on the programmer to insert the various locking instructions.

Example:2-PL

T_5	T_6	T_7
lock-X(A)		
read(A)		
lock-S(B)		
read(B)		
write(A)		
unlock(A)		
	lock-X(A)	
	read(A)	
	write(A)	
	unlock(A)	
		lock-S(A)
		read(A)

Further Examples:

a. Lost Update Problem

Time	**T 1**	**T 2**	**Bal$_x$**
t_1	-	begin_transaction	100
t_2	begin_transaction	read(bal_x)	100
t_3	read(bal_x)	$bal_x = bal_x + 100$	100
t_4	$bal_x = bal_x - 10$	write(bal_x)	200
t_5	write(bal_x)	Commmit	90
t_6	Commmit	-	90

Transaction T_1 updates the same record updated earlier by T_2 at time t_5 on the basis of values read at t_3.

Preventing the Lost Update Problem using 2PL

Time	**T 1**	**T 2**	**Bal$_x$**

t_1	-	begin_transaction	100
t_2	begin_transaction	Write_lock(bal_x)	100
t_3	Write_lock(bal_x)	read(bal_x)	100
t_4	Wait	bal_x= bal_x +100	100
t_5	Wait	write(bal_x)	200
t_6	Wait	Commmit/unlock(bal_x)	200
t_7	read(bal_x)		200
t_8	bal_x= bal_x -10		200
t_9	write(bal_x)		190
t_{10}	Commmit/unlock(bal_x)		190

b. Uncommitted Dependency Problem

Time	**T_3**	**T_4**	**Bal_x**
t_1	-	begin_transaction	100
t_2	-	read(bal_x)	100
t_3	-	bal_x= bal_x +100	100
t_4	begin_transaction	write(bal_x)	200
t_5	read(bal_x)	:	200
t_6	bal_x= bal_x -10	rollback	100
t_7	write(bal_x)	-	190
t_8	commmit	-	190

Preventing the Uncommitted Dependency Problem using 2PL

Time	**T_3**	**T_4**	**Bal_x**
t_1	-	begin_transaction	100
t_2	-	Write_lock(bal_x)	100
t_3	-	read(bal_x)	100

t_4	begin_transaction	$bal_x = bal_x + 100$	200
t_5	Write_lock(bal_x)	write(bal_x)	200
t_6	wait	Rollback/unlock(bal_x)	100
t_7	read(bal_x)		100
t_8	$bal_x = bal_x - 10$		100
t_9	write(bal_x)		90
t_{10}	commmit/unlock(bal_x)		90

c. Inconsistent Analysis Problem

Time	**Transaction A**	**Transaction B**	**Acc1**	**Acc2**	**Acc3**	**Sum**
t_1	Read Acc1	Read Acc1	100	50	25	0
t_2	Acc1 - 10	Sum + Acc1	100	50	25	100
t_3	Write Acc1	Read Acc2	90	50	25	100
t_4	Read Acc3	Sum + Acc2	90	50	25	150
t_5	Acc3 + 10	-	90	50	25	150
t_6	Write Acc3	-	90	50	35	150
t_6	Commit	Read Acc3	90	50	35	150
t_7	-	Sum + Acc3	90	50	35	185
t_8	-	Commit	90	50	35	185

* **Sum = 185 not 175!!**

Activity

Using 2PL, provide a solution to the inconsistent analysis problem discussed earlier.

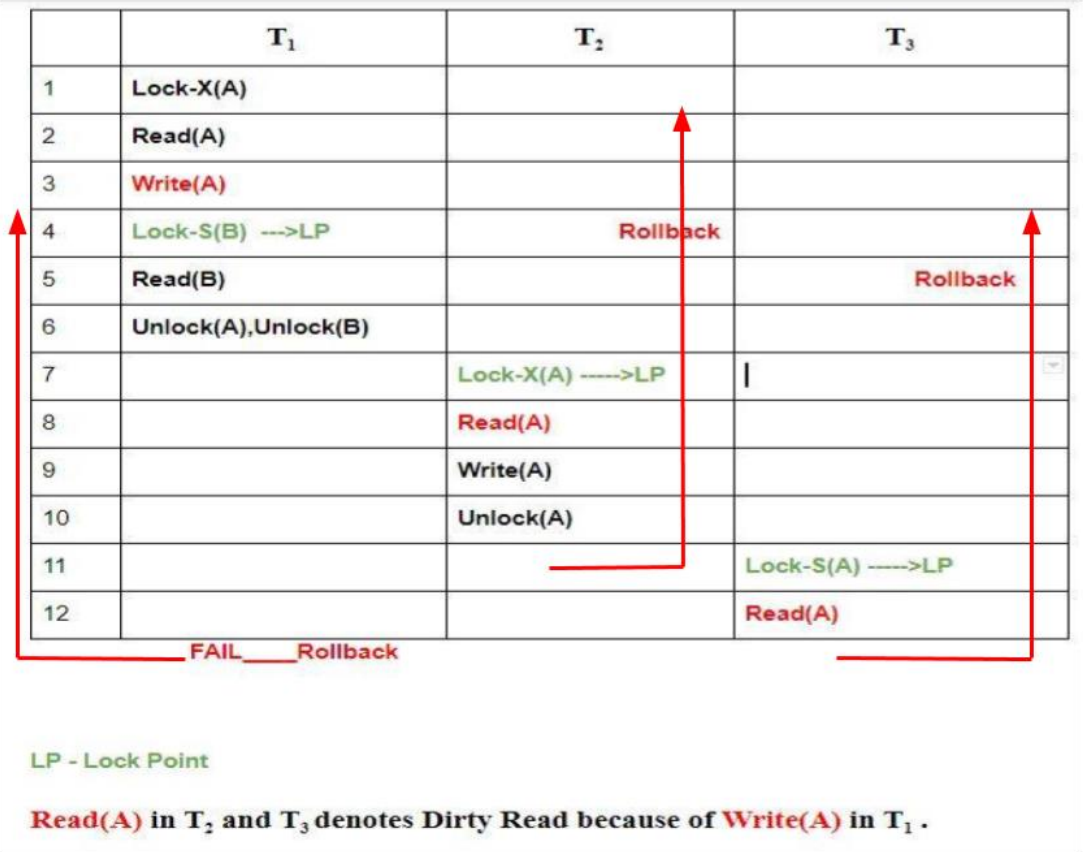
Drawbacks of Two-Phase Locking

Although we have said that 2-PL ensures serializability, but there are still some drawbacks of 2-PL. Let's glance at two of these drawbacks:

- Cascading Rollback is possible under 2-PL.
- Deadlocks and Starvation are possible.

i. Cascading Rollback

Let's examine the following Schedule:



Take a moment to analyze the schedule. Yes, you're correct, because of Dirty Read (uncommitted dependency problem) in T_2 and T_3 in lines 8 and 12 respectively, when T_1 failed we have to roll back others also. Hence, Cascading Rollbacks are possible in 2-PL. I have taken skeleton schedules as examples because it's easy to understand when it's kept simple. When explained with real-time transaction problems with many variables, it becomes very complex.

There are many protocols derived from 2-PL. To avoid Cascading Rollbacks, we use the following:

- **Strict two-phase locking**. Here a transaction must hold all its exclusive locks till it commits.
- **Rigorous two-phase locking**: This is even stricter, here all locks (shared and exclusive) are held till commit. In this protocol transactions can be serialized in the order in which they commit.
- **Graph-based protocol:** we fix an order of accessing data. If a transaction has to update Row2 and read Row1, it has to access these data in a predefined order.

ii. Deadlocks and Deadlock Handling

Two-phase locking does not ensure freedom from deadlocks as seen earlier. There are three classical approaches for deadlock handling, namely:

- Timeouts
- Deadlock prevention
- Deadlock detection and removal

Techniques to Control Deadlocks

i. Timeouts

A transaction that requests for a lock will wait for only a system-defined period of time. If the lock has not been granted within this period, the lock requests times out and the transaction is rolled back and restarted. DBMS assumes the transaction may be deadlocked, even if it may not be, and it aborts and automatically restarts the transaction. Though using timeouts is simple to implement; but starvation is possible. Another challenge with timeouts is that it's difficult to determine good value of the timeout interval.

ii. Deadlock Prevention

Using deadlock prevention technique, a transaction requesting a new lock is aborted if there is a possibility that a dead lock can occur. Before aborting, the DBMS looks ahead to determine if a transaction would cause deadlock, and never allows deadlock to occur. The convention is that when more than one transaction request for locking the same data item, only one of them is granted the lock. If the transaction is aborted, all the changes made by this transaction are rolled back and all locks obtained by the transaction are released. The transaction is then rescheduled for execution. Deadlock prevention works because it avoids the conditions that lead to deadlock.

The deadlock prevention/avoidance approach handles deadlocks before they occur by analyzing the transactions and the locks to determine whether or not waiting leads to a deadlock. The lock manager checks whether the lock is available. If it is available, the lock manager allocates the data item and the transaction acquires the lock. However, if the item is locked by some other transaction in incompatible mode, the lock manager runs an algorithm to test whether keeping the transaction in waiting state will cause a deadlock or not. Accordingly, the algorithm decides whether the transaction can wait or one of the transactions should be aborted.

There are two algorithms for this purpose that use transaction **timestamps** for the sake of deadlock avoidance/prevention alone:

- **wait-die scheme** (non-preemptive): using this algorithm, older transaction may wait for younger one to release data item. (Older means smaller timestamp). Younger transactions never wait for older ones; they are rolled back instead. A transaction may die several times before acquiring needed data item.

- **wound-wait** scheme (preemptive): older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it. Younger transactions may wait for older ones. Wound-wait algorithm may have fewer rollbacks than *wait-die* scheme.

Both in ***wait-die*** and in ***wound-wait*** schemes, a rolled back transactions are restarted with its original timestamp. Older transactions thus have precedence over newer ones, and starvation is hence avoided.

Example: Let us assume that there are two transactions, T1 and T2, where T1 tries to lock a data item which is already locked by T2. The algorithms are as follows:

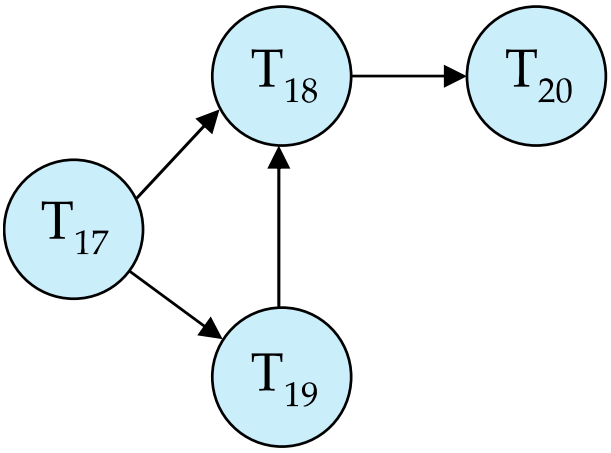
- **Wait-Die** − If T1 is older than T2, T1 is allowed to **wait**. Otherwise, if T1 is younger than T2, T1 is aborted(**die**) and later restarted.
- **Wound-Wait** − If T1 is older than T2, T2 is aborted(**wounded**) and later restarted. Otherwise, if T1 is younger than T2, T1 is allowed to **wait**.

iii. Deadlock Detection and Removal

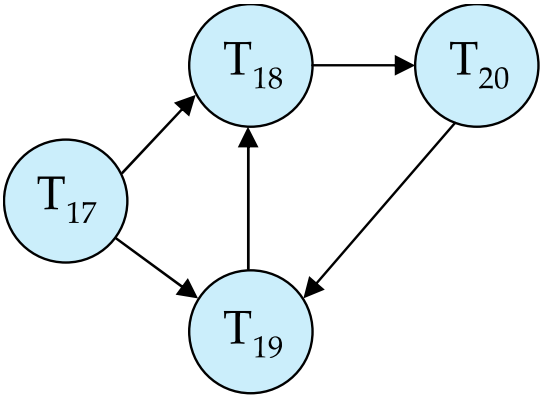
The deadlock detection and removal approach runs a deadlock detection algorithm periodically and removes deadlock in case there is one. This approach does not check for deadlock when a transaction places a request for a lock. When a transaction requests a lock, the lock manager checks whether it is available. If it is available, the transaction is allowed to lock the data item; otherwise, the transaction is allowed to wait. Since there are no precautions while granting lock requests, some of the transactions may be deadlocked.

To detect deadlocks, the lock manager periodically checks if the wait-for graph has cycles. If the system is deadlocked, the lock manager chooses a victim

transaction from each cycle. The victim is aborted and rolled back; and then restarted later.



Wait-for graph without a cycle



Wait-for graph with a cycle

Some of the methods used for victim selection are:

a. Choose the youngest transaction.
b. Choose the transaction with fewest data items.
c. Choose the transaction that has performed least number of updates.

d. Choose the transaction having least restart overhead.
e. Choose the transaction which is common to two or more cycles.

This approach is primarily suited for systems having transactions low and where fast response to lock requests is needed.

Take Note

The best deadlock control method depends on the database environment, if the probability is low, deadlock detection is recommended, if probability is high, deadlock prevention is recommended and if response time is not high on the system priority list deadlock avoidance might be employed.

6.5 Time Stamping Method

The time stamping approach, to schedule concurrent transactions assigns a global unique time stamp to each transaction. The time stamp value uses an explicit order in which transactions are submitted to the DBMS. The stamps must have 2 properties;

i. Uniqueness - which assures that no equal time stamp values can exist.
ii. Monotonicity - which assures that time stamp values always increase.

All database operations read and write within the same transaction must have the same time stamp. The DBMS executes conflicting operations in the time stamp order thereby ensuring serialisability of the transactions.

If 2 transactions conflict, one is often stopped, re-scheduled and assigned a new time stamp value. The main draw back of time stamping approach is that each value stored in the database requires 2 additional time- stamp fields, one for the last time the field was read and one for the last update. Time stamping thus increases the memory needs and the databases.

6.6 Optimistic Methods

The optimistic approach is based on the assumption that the majority of database operations do not conflict. The optimistic approach does not require locking or time stamping techniques; instead a transaction is executed without restrictions until it is committed. In this approach, each transaction moves through 2 or 3 phases; read, validation and write phase.

4.6.1 Optimistic Concurrency Control phases

a. Read Phase

The transaction reads the database, executes the needed computations and makes the updates to private copy of the database values. All update operations of the transaction are recorded in a temporary update file, which is not accessed by the remaining transactions.

b. Validation Phase

The transaction is validated to ensure that the changes made will not affect the integrity and consistency of the database. If a validation phase is negative, the transaction is restarted and the changes are discarded.

c. Write Phase

The changes are permanently applied (written) to the database.

Take Note

The optimistic approach is acceptable for mostly read or query database system that require very few update transactions.

6.7 Uses of Transaction Management

- The DBMS is used to schedule the access of data concurrently. It means that the user can access multiple data from the database without being interfered with each other. Transactions are used to manage concurrency.
- It is also used to satisfy ACID properties.
- It is used to solve Read/Write Conflict.
- It is used to implement Recoverability, Serializability, and Cascading.
- Transaction Management is also used for Concurrency Control Protocols and Locking of data.

6.8 Summary

In this session we discussed concurrency control techniques that included locking methods, time stamping methods and optimistic methods. Lastly, we saw the uses of transaction management. I hope you enjoyed the session.

6.9 Student Activity

i. Discuss the differences between Concurrency Control with Locking and Concurrency Control without Locking.
ii. Discuss the differences between Pessimistic concurrency control and Optimistic concurrency control.
iii. Describe the basic timestamp ordering protocol for concurrency control, and how does it differ from locking based protocols.

6.10 Reference Materials

Core Books

i. Coronel, C., & Morris, S. (2017). *Database Systems: Design, Implementation, & Management* (12th ed.). Boston, MA: Cengage Learning. ISBN: 1305627482.
ii. Elmasri, R., & Navathe, S. B. (2017). *Fundamentals of Database Systems* (7th ed.). Hoboken, NJ: Pearson Education Ltd. ISBN: 0133970779.

iii. Connolly, T. M., & Begg, C. E. (2015). *Database Systems: A Practical Approach to Design, Implementation, and Management* (6th ed.). Boston, MA: Pearson Education Ltd. ISBN: 0132943263.

Core Journals

i. Journal of Database Management. ISSN: 1063-8016.
ii. Database Management & Information Retrieval. ISSN: 1862-5347.
iii. International Journal of Information Technology and Database Systems. ISSN: 2231-1807.

Recommended Text Books

i. Hernandez, M. J. (2013). *Database Design for Mere Mortals: A Hands-On Guide to Relational Database Design* (3rd ed.). Harlow, UK: Addison-Wesley. ISBN: 0321884493.
ii. Rankins, R., Bertucci, P., Gallelli, C., & Silverstein, A. (2015). *Microsoft SQL Server 2014 Unleashed*. Indianapolis, IN: Sams Publishing. ISBN: 0672337290.
iii. Comeau, A. (2016). *MySQL Explained: Your Step By Step Guide to Database Design*. Bradenton, FL: OSTraining. ISBN: 151942437X.

Recommended Journals

i. International Journal of Intelligent Information and Database Systems. ISSN: 1751-5858.
ii. Database Systems Journal. ISSN: 2069-3230.
iii. Distributed and Parallel Databases. ISSN: 0926-8782.
iv. International Journal of Database Management Systems. ISSN: 0975 - 5985.
v. Journal of Database Management. ISSN:1063-8016.