Database Management System

Unit-5

Dharmendra Kumar(Associate Professor)
Department of Computer Science and Engineering
United College of Engineering and Research, Prayagraj

CONCURRENCY CONTROL

When several transactions execute concurrently in the database, however, the isolation property may no longer be preserved. To ensure that it is, the system must control the interaction among the concurrent transactions. The mechanism used to control the interaction of transactions is called concurrency control scheme.
There are number of concurrency control schemes.

1. Lock based protocol
2. Time stamp based protocol
3. Validation based protocol
4. Multiple granularity protocol
5. Multi-version protocol

1 Lock based protocol

A lock is a mechanism to control concurrent access to a data item.
Data item can be locked in two modes.
Shared mode(S): If a transaction T_i has obtained a shared-mode lock on item Q, then T_i can read, but cannot write, Q.
Exclusive mode(S): If a transaction T_i has obtained an exclusive-mode lock on item Q, then T_i can both read and write Q.

Note: The transaction makes the lock request to the concurrency control manager. Transaction can process only after lock request is granted.

Compatibility function
Given a set of lock modes, we can define a compatibility function on them as follows.
Let A and B represent arbitrary lock modes. Suppose that a transaction T_i requests a lock of mode A on item Q on which transaction T_j ($T_i \neq T_j$) currently holds a lock of mode B. If transaction T_i can be granted a lock on Q immediately, in spite of the presence of the mode B lock, then we say mode A is compatible with mode B. Such a function can be represented conveniently by a matrix. An element comp(A, B) of the matrix has the value **true** if and only if mode A is compatible with mode B.

Note:

- A transaction requests a shared lock on data item Q by executing the lock-S(Q) instruction.

- A transaction requests an exclusive lock on data item Q by executing the lock-X(Q) instruction.
- To unlock the data item Q, we use unlock(Q) instruction.

Note:
To access a data item, transaction T_i must first lock that item. If the data item is already locked by another transaction in an incompatible mode, the concurrency control manager will not grant the lock until all incompatible locks held by other transactions have been released. Thus, T_i is made to wait until all incompatible locks held by other transactions have been released.

Example: Consider the following two transactions T_1 and T_2 with locking modes.

Consider the following schedule-1 of these transactions.
Suppose that the values of accounts A and B are $100and200$, respectively. If these two transactions are executed serially, either in the order T_1, T_2 or the order T_2, T_1, then transaction T_2 will display the value \$300. If, however, these transactions are executed concurrently, then schedule 1 is possible. In this case, transaction T_2 displays \$250, which is incorrect. The reason for this mistake is that the transaction T_1 unlocked data item B too early, as a result of which T_2 saw an inconsistent state. **Example:** Consider the following two transactions T_3 and T_4 with locking modes.

Consider the following schedule-1 of these transactions.

Consider the partial schedule-2 for T_3 and T_4. Since T_3 is holding an exclusive-mode lock on B and T_4 is requesting a shared-mode lock on B, T_4 is waiting for T_3 to unlock B. Similarly, since T_4 is holding a shared-mode lock on A and T_3 is requesting an exclusive-mode lock on A, T_3 is waiting for T_4 to unlock A. Thus, we have arrived at a state where neither of these transactions can ever proceed with its normal execution. This situation is called deadlock.

Note: When deadlock occurs, the system must roll back one of the two transactions.

Locking protocol This is the set of rules indicating when a transaction may lock and unlock each of the data items.
Note: A schedule S is legal under a given locking protocol if S is a possible schedule for a set of transactions that follow the rules of the locking protocol.
Note: A locking protocol ensures conflict serializability if and only if all legal schedules are conflict serializable.

Starvation

Suppose a transaction T_2 has a shared-mode lock on a data item, and another transaction T_1 requests an exclusive-mode lock on the data item. Clearly, T_1 has to wait for T_2 to release the shared-mode lock. Meanwhile, a transaction T_3 may request a shared-mode lock on the same data item. The lock request is compatible with the lock granted to T_2, so T_3 may be granted the shared-mode lock. At this point T_2 may release the lock, but still T_1 has to wait for T_3 to finish. But again, there may be a new transaction T_4 that requests a shared-mode lock on the same data item, and is granted the lock before

Compatibility Matrix

	S	X
S	true	false
X	false	false

Transaction T_1 and T_2

T_1: lock-X(B);
read(B);
$B := B - 50$;
write(B);
unlock(B);
lock-X(A);
read(A);
$A := A + 50$;
write(A);
unlock(A).

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

Schedule-1

T_1	T_2	concurrency-control manager
lock-X(B)		
		grant-X(B, T_1)
read(B)		
$B := B - 50$		
write(B)		
unlock(B)		
	lock-S(A)	
		grant-S(A, T_2)
	read(A)	
	unlock(A)	
	lock-S(B)	
		grant-S(B, T_2)
	read(B)	
	unlock(B)	
	display($A + B$)	
lock-X(A)		
		grant-X(A, T_2)
read(A)		
$A := A + 50$		
write(A)		
unlock(A)		

Transaction T_3 and T_4

T_3: lock-X(B);
read(B);
$B := B - 50$;
write(B);
lock-X(A);
read(A);
$A := A + 50$;
write(A);
unlock(B);
unlock(A).

T_4: lock-S(A);
read(A);
lock-S(B);
read(B);
display($A + B$);
unlock(A);
unlock(B).

T_3 releases it. In fact, it is possible that there is a sequence of transactions that each requests a shared-mode lock on the data item, and each transaction releases the lock a short while after it is granted, but T_1 never gets the exclusive-mode lock on the data item. The transaction T_1 may never make progress, and is said to be starved. This situation is said to be **starvation**.

1.1 Two-phase locking protocol

This protocol requires that each transaction issue lock and unlock requests in two phases:
1. Growing phase: A transaction may obtain locks, but may not release any lock.
2. Shrinking phase: A transaction may release locks, but may not obtain any new locks.

Initially, a transaction is in the growing phase. The transaction acquires locks as needed. Once the transaction releases a lock, it enters the shrinking phase, and it can issue no more lock requests.
Example: Transactions T_3 and T_4 are locked in two phase. While, transactions T_1 and T_2 are not locked in two phase.

Note: Two-phase locking protocol ensures conflict serializability. The serializability order of transactions will be based on lock point in the transactions.

Lock point: Lock point of a transaction is a point in the schedule where the transaction has obtained its final lock (the end of its growing phase).
Note: Two-phase locking does not ensure freedom from deadlock.

Observe that transactions T_3 and T_4 are in two phase, but, in schedule 2, they are deadlocked.

Note: In addition to being serializable, schedules should be cascadeless. Cascading rollback may occur under two-phase locking.
Example: Consider the partial schedule in the following figure:-
Each transaction observes the two-phase locking protocol, but the failure of T_5 after the read(A) step of T_7 leads to cascading rollback of T_6 and T_7.
Note: Cascading rollbacks can be avoided by a modification of two-phase locking called the strict two-phase locking protocol.

Strict two-phase locking protocol
This protocol requires not only that locking be two phase, but also that all exclusive-mode locks taken by a transaction be held until that transaction commits.
Rigorous two-phase locking protocol
Another variant of two-phase locking is the rigorous two-phase locking protocol, which requires that all locks be held until the transaction commits.
Note: With rigorous two-phase locking, transactions can be serialized in the order in which they commit.

Lock Conversion
Upgrade: We denote conversion from shared to exclusive modes by upgrade.

Downgrade: We denote conversion from exclusive to shared by downgrade.

Note: Lock conversion cannot be allowed arbitrarily. Rather, upgrading can take place in only the growing phase, whereas downgrading can take place in only the shrinking phase.
Note: Strict two-phase locking and rigorous two-phase locking (with lock conversions) are used extensively in commercial database systems.
Note: A simple but widely used scheme automatically generates the appropriate lock and unlock instructions for a transaction, on the basis of read and write requests from the transaction:

- When a transaction T_i issues a read(Q) operation, the system issues a lock- S(Q) instruction followed by the read(Q) instruction.
- WhenT_i issues a write(Q) operation, the system checks to see whether T_i already holds a shared lock on Q. If it does, then the system issues an upgrade(Q) instruction, followed by the write(Q) instruction. Otherwise, the system issues a lock-X(Q) instruction, followed by the write(Q) instruction.
- All locks obtained by a transaction are unlocked after that transaction commits or aborts.

1.2 Graph-Based Protocols

For this type of protocol, we need some prior knowledge of database. To acquire such prior knowledge, we impose a partial ordering $\rightarrow$ on the set D = $\{d_1, d_2, ..., d_n\}$ of all data items. If $d_i \rightarrow d_j$, then any transaction accessing both d_i and d_j must access d_i before accessing d_j .

The partial ordering implies that the set D may now be viewed as a directed acyclic graph, called a database graph. Here, we will consider graph with rooted tree. Therefore, we will study tree protocol.

In the **tree protocol,** the only lock instruction allowed is lock-X. Each transaction T_i can lock a data item at most once, and must observe the following rules:

1. The first lock by T_i may be on any data item.
2. Subsequently, a data item Q can be locked by T_i only if the parent of Q is currently locked by T_i.
3. Data items may be unlocked at any time
4. A data item that has been locked and unlocked by T_i cannot subsequently be relocked by T_i.

All schedules that are legal under the tree protocol are conflict serializable.

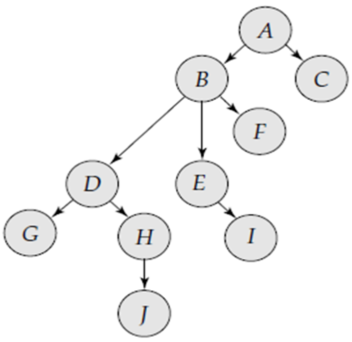
Example: Consider the database graph of the following figure:-
The following four transactions follow the tree protocol on this graph. We show only the lock and unlock instructions:

Schedule-2

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)	

Partial schedule under
two-phase locking protocol

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)



T_{10}: lock-X(B); lock-X(E); lock-X(D); unlock(B); unlock(E); lock-X(G); unlock(D); unlock(G).
T_{11}: lock-X(D); lock-X(H); unlock(D); unlock(H).
T_{12}: lock-X(B); lock-X(E); unlock(E); unlock(B).
T_{13}: lock-X(D); lock-X(H); unlock(D); unlock(H).

One possible schedule in which these four transactions participated appears in the following figure:-
Observe that the schedule in this figure is conflict serializable. It can be shown not only that the tree protocol ensures conflict serializability, but also that this protocol ensures freedom from deadlock.
The tree protocol in this figure does not ensure recoverability and cascadelessness.

Advantage:

1. The tree-locking protocol has an advantage over the two-phase locking protocol in that, unlike two-phase locking, it is deadlock-free, so no rollbacks are required.

2. The tree-locking protocol has another advantage over the two-phase locking protocol in that unlocking may occur earlier. Earlier unlocking may lead to shorter waiting times, and to an increase in concurrency.

1.3 Timestamp-Based Protocols

Timestamps With each transaction T_i in the system, we associate a unique fixed timestamp, denoted by TS(T_i). This timestamp is assigned by the database system before the transaction Ti starts execution. If a transaction T_i has been assigned timestamp TS(T_i), and a new transaction T_j enters the system, then TS(T_i) ¡ TS(T_j). There are two simple methods for implementing this scheme:

1. Use the value of the system clock as the timestamp; that is, a transaction's timestamp is equal to the value of the clock when the transaction enters the system.

2. Use a logical counter that is incremented after a new timestamp has been assigned; that is, a transaction's timestamp is equal to the value of the counter when the transaction enters the system.

The timestamps of the transactions determine the serializability order. Thus, if TS(T_i) ¡ TS(T_i), then the system must ensure that the produced schedule is equivalent to a serial schedule in which transaction T_i appears before transaction T_j.

To implement this scheme, we associate with each data item Q two timestamp values:

- **W-timestamp(Q)** denotes the largest timestamp of any transaction that executed write(Q) successfully.

- **R-timestamp(Q)** denotes the largest timestamp of any transaction that executed read(Q) successfully.

These timestamps are updated whenever a new read(Q) or write(Q) instruction is executed.

1.3.1 Timestamp-Ordering Protocol

The timestamp-ordering protocol ensures that any conflicting read and write operations are executed in timestamp order. This protocol operates as follows:

1. Suppose that transaction T_i issues read(Q).
   (a) If TS(T_i) $<$ W-timestamp(Q), then the read operation is rejected, and T_i is rolled back.
   (b) If TS(T_i) $\geq$ W-timestamp(Q), then the read operation is executed, and R-timestamp(Q) is set to the maximum of R-timestamp(Q) and TS(T_i).
2. Suppose that transaction T_i issues write(Q).
   (a) If TS(T_i) $<$ R-timestamp(Q), then the system rejects the write operation and rolls T_i back.
   (b) If TS(Ti) $<$ W-timestamp(Q), then the system rejects this write operation and rolls T_i back.
   (c) Otherwise, the system executes the write operation and sets W-timestamp(Q) to TS(T_i).

If a transaction T_i is rolled back by the concurrency-control scheme as result of issuance of either a read or write operation, the system assigns it a new timestamp and restarts it.

Example: Consider transactions T_{14} and T_{15}. Transaction T_{14} displays the contents of accounts A and B:

T_{14}: read(B);
read(A);
display(A + B).

Transaction T_{15} transfers \$50 from account A to account B, and then displays the contents of both:

T_{15}: read(B);
B := B - 50;
write(B);
read(A);
A := A + 50;
write(A);
display(A + B).

Following schedule is possible under timestamp ordering protocol.
Note:

1. The timestamp-ordering protocol ensures conflict serializability.
2. This protocol also ensures freedom from deadlock.
3. There is a possibility of starvation.

T_{10}	T_{11}	T_{12}	T_{13}
lock-x(B)			
	lock-x(D)		
	lock-x(H)		
	unlock(D)		
lock-x(E)			
lock-x(D)			
unlock(B)			
unlock(E)			
		lock-x(B)	
		lock-x(E)	
	unlock(H)		
lock-x(G)			
unlock(D)			
			lock-x(D)
			lock-x(H)
			unlock(D)
			unlock(H)
		unlock(E)	
		unlock(B)	
unlock (G)			

T_{14}	T_{15}
read(B)	
	read(B)
	$B := B - 50$
	write(B)
read(A)	
	read(A)
display($A + B$)	
	$A := A + 50$
	write(A)
	display($A + B$)

4. This protocol can generate schedules that are not recoverable.

1.3.2 Thomas' Write Rule

The modification to the timestamp-ordering protocol, called Thomas' write rule, is this: Suppose that transaction Ti issues write(Q).

1. If TS(T_i) ¡ W-timestamp(Q), then the read operation is rejected, and T_i is rolled back.
2. If TS(T_i) ¡ W-timestamp(Q), then T_i is attempting to write an obsolete value of Q. Hence, this write operation can be ignored.
3. Otherwise, the system executes the write operation and sets W-timestamp(Q) to TS(T_i).

Example: Consider following schedule:-
Clearly, this schedule is not conflict serializable and, thus, is not possible under any of two-phase locking, the tree protocol, or the timestamp-ordering protocol. Under Thomas' write rule, the write(Q) operation of T_{16} would be ignored. The result is a schedule that is view equivalent to the serial schedule ¡ T_{16}, T_{17}¿.

2 Multiple Granularity

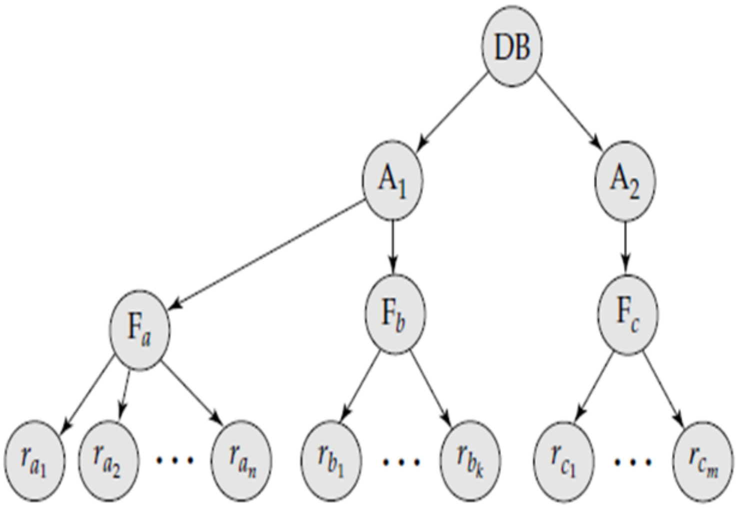
Consider the following granularity hierarchy. This tree consists of four levels of nodes. The highest level represents the entire database. Below it are nodes of type area; the database consists of exactly these areas. Each area in turn has nodes of type file as its children. Each area contains exactly those files that are its child nodes. No file is in more than one area. Finally, each file has nodes of type record. As before, the file consists of exactly those records that are its child nodes, and no record can be present in more than one file.

This protocol uses the following compatibility matrix to lock the data items. There is an intention mode associated with shared mode, and there is one with exclusive mode. If a node is locked in intention-shared (IS) mode, explicit locking is being done at a lower level of the tree, but with only shared-mode locks. Similarly, if a node is locked in intention-exclusive (IX) mode, then explicit locking is being done at a lower level, with exclusive-mode or shared-mode locks. Finally, if a node is locked in shared and intention-exclusive (SIX) mode, the sub-tree rooted by that node is locked explicitly in shared mode, and that explicit locking is being done at a lower level with exclusive-mode locks.

The multiple-granularity locking protocol, which ensures serializability, is this:
Each transaction T_i can lock a node Q by following these rules:

1. It must observe the lock-compatibility function shown in above matrix.
2. It must lock the root of the tree first, and can lock it in any mode.
3. It can lock a node Q in S or IS mode only if it currently has the parent of Q locked in either IX or IS mode.

T_{16}	T_{17}
read(Q)	
	write(Q)
write(Q)	



Lock Compatibility Matrix

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

4. It can lock a node Q in X, SIX, or IX mode only if it currently has the parent of Q locked in either IX or SIX mode.

5. It can lock a node only if it has not previously unlocked any node (that is, T_i is two phase).

6. It can unlock a node Q only if it currently has none of the children of Q locked.

Clearly, the multiple-granularity protocol requires that locks be acquired in top-down (root-to-leaf) order, whereas locks must be released in bottom-up (leaf-to-root) order.

Example:
Consider the tree shown in the above figure and these transactions:

- Suppose that transaction T_{18} reads record r_{a_2} in file F_a. Then, T_{18} needs to lock the database, area A_1, and F_a in IS mode (and in that order), and finally to lock r_{a_2} in S mode.
- Suppose that transaction T_{19} modifies record r_{a_9} in file F_a. Then, T_{19} needs to lock the database, area A_1, and file F_a in IX mode, and finally to lock r_{a_2} in X mode.
- Suppose that transaction T_{20} reads all the records in file F_a. Then, T_{20} needs to lock the database and area A_1 (in that order) in IS mode, and finally to lock F_a in S mode.
- Suppose that transaction T_{21} reads the entire database. It can do so after locking the database in S mode.

Clearly, transactions T_{18}, T_{20}, and T_{21} can access the database concurrently. Transaction T_{19} can execute concurrently with T_{18}, but not with either T_{20} or T_{21}.

This protocol enhances concurrency and reduces lock overhead. It is particularly useful in applications that include a mix of

- Short transactions that access only a few data items
- Long transactions that produce reports from an entire file or set of files

Note: Deadlock is possible in this protocol.

3 Multiversion Schemes

In multiversion concurrency control schemes, each write(Q) operation creates a new version of Q. When a transaction issues a read(Q) operation, the concurrencycontrol manager selects one of the versions of Q to be read. The concurrency-control scheme must ensure that the version to be read is selected in a manner that ensures serializability.

3.1 Multiversion Timestamp Ordering

With each data item Q, a sequence of versions $< Q_1, Q_2, ..., Q_m >$ is associated. Each version Q_k contains three data fields:

- **Content** is the value of version Q_k.
- **W-timestamp(Q_k)** is the timestamp of the transaction that created version Q_k.
- **R-timestamp(Q_k)** is the largest timestamp of any transaction that successfully read version Q_k.

A transaction T_i creates a new version Q_k of data item Q by issuing a write(Q) operation. The content field of the version holds the value written by T_i. The system initializes the W-timestamp and R-timestamp to TS(T_i). It updates the R-timestamp value of Q_k whenever a transaction T_j reads the content of Q_k, and R-timestamp(Q_k) ¡ TS(T_j).

The multiversion timestamp-ordering scheme operates as follows. Suppose that transaction T_i issues a read(Q) or write(Q) operation. Let Q_k denote the version of Q whose write timestamp is the largest write timestamp less than or equal to TS(T_i).

1. If transaction T_i issues a read(Q), then the value returned is the content of version Q_k.
2. If transaction T_i issues write(Q), and if TS(T_i)¡R-timestamp(Q_k), then the system rolls back transaction T_i. On the other hand, if TS(T_i) = W-timestamp(Q_k), the system overwrites the contents of Q_k; otherwise it creates a new version of Q.

Versions that are no longer needed are removed according to the following rule. Suppose that there are two versions, Q_k and Q_j , of a data item, and that both versions have a W-timestamp less than the timestamp of the oldest transaction in the system. Then, the older of the two versions Q_k and Q_j will not be used again, and can be deleted.

Note:

1. The multiversion timestamp-ordering scheme ensures serializability.
2. The multiversion timestamp-ordering scheme does not ensure recoverability and cascadelessness.

4 Deadlock Handling

A system is in a deadlock state if there exists a set of transactions such that every transaction in the set is waiting for another transaction in the set. More precisely, there exists a set of waiting transactions $\{T_0, T_1, ..., T_n\}$ such that T_0 is waiting for a data item that T_1 holds, and T_1 is waiting for a data item that T_2 holds, and . . ., and T_{n-1} is waiting for a data item that T_n holds, and T_n is waiting for a data item that T_0 holds. None of the transactions can make progress in such a situation.

There are two principal methods for dealing with the deadlock problem. We can use a deadlock prevention protocol to ensure that the system will never enter a deadlock state. Alternatively, we can allow the system to enter a deadlock state, and then try to recover by using a deadlock detection and deadlock recovery scheme.

Note: Prevention is commonly used if the probability that the system would enter a deadlock state is relatively high; otherwise, detection and recovery are more efficient.

4.1 Deadlock Prevention

Two different deadlock prevention schemes using timestamps have been proposed:

1. **wait–die:** This scheme is a non-preemptive technique. When transaction T_i requests a data item currently held by T_j , T_i is allowed to wait only if it has a timestamp smaller than that of T_j (that is, T_i is older than T_j). Otherwise, T_i is rolled back (dies).
   For example, suppose that transactions T_1, T_2, and T_3 have timestamps 5, 10, and 15, respectively. If T_1 requests a data itemheld by T_2, then T_1 will wait. If T_3 requests a data item held by T_2, then T_3 will be rolled back.

2. **wound–wait:** This scheme is a preemptive technique. It is a counterpart to the wait–die scheme. When transaction T_i requests a data item currently held by T_j , T_i is allowed to wait only if it has a timestamp larger than that of T_j (that is, T_i is younger than T_j). Otherwise, T_j is rolled back (T_j is wounded by T_i).
   Returning to our example, with transactions T_1, T_2, and T_3, if T_1 requests a data item held by T_2, then the data item will be preempted from T_2, and T_2 will be rolled back. If T_3 requests a data item held by T_2, then T_3 will wait.

4.2 Deadlock Detection and Recovery

If a system does not employ some protocol that ensures deadlock freedom, then a detection and recovery scheme must be used. An algorithm that examines the state of the system is invoked periodically to determine whether a deadlock has occurred. If one has, then the system must attempt to recover from the deadlock.

4.2.1 Deadlock Detection

To identify the deadlock is present in the system, we use a directed graph called **wait-for-graph**.
In this graph, vertices are corresponding to transactions. When transaction T_i requests a data item currently being held by transaction T_j , then the edge $T_i \rightarrow T_j$ is inserted in the wait-for graph. This edge is removed only when transaction $T_j$is no longer holding a data item needed by transaction T_i.

A deadlock exists in the system if and only if the wait-for graph contains a cycle. Each transaction involved in the cycle is said to be deadlocked. To detect deadlocks, the system needs to maintain the wait-for graph, and periodically to invoke an algorithm that searches for a cycle in the graph.

Example: Consider the wait-for graph show in the following figure,
which depicts the following situation:

- Transaction T_{25} is waiting for transactions T_{26} and T_{27}.
- Transaction T_{27} is waiting for transaction T_{26}.
- Transaction T_{26} is waiting for transaction T_{28}.

Since the graph has no cycle, the system is not in a deadlock state.
Suppose now that transaction T_{28} is requesting an item held by T_{27}. The edge $T_{28} \rightarrow T_{27}$is added to the wait-for graph, resulting in the new system state in following figure.
This time, the graph contains the cycle
$T_{26} \rightarrow T_{28} \rightarrow T_{27} \rightarrow T_{26}$.
implying that transactions T_{26}, T_{27}, and T_{28} are all deadlocked.

4.2.2 Recovery from Deadlock

When a detection algorithm determines that a deadlock exists, the system must recover from the deadlock. The most common solution is to roll back one or more transactions to break the deadlock. Three actions need to be taken:

1. **Selection of a victim:** Given a set of deadlocked transactions, we must determine which transaction (or transactions) to roll back to break the deadlock.We should roll back those transactions that will incur the minimum cost. Unfortunately, the term minimum cost is not a precise one. Many factors may determine the cost of a rollback, including
   (a) How long the transaction has computed, and how much longer the transaction will compute before it completes its designated task.
   (b) How many data items the transaction has used.
   (c) How many more data items the transaction needs for it to complete.
   (d) How many transactions will be involved in the rollback.
2. **Rollback:** Once we have decided that a particular transaction must be rolled back, we must determine how far this transaction should be rolled back.
   The simplest solution is a total rollback:
3. **Starvation:** In a system where the selection of victims is based primarily on cost factors, it may happen that the same transaction is always picked as a victim. As a result, this transaction never completes its designated task, thus there is starvation. We must ensure that transaction can be picked as a victim only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor.

4.3 The Phantom Phenomenon

Consider transaction T_{29} that executes the following SQL query on the bank database:

```
select sum(balance)
from account
where branch-name = 'Perryridge'
```

Transaction T_{29} requires access to all tuples of the account relation pertaining to the Perryridge branch.
Let T_{30} be a transaction that executes the following SQL insertion:

```
insert into account
values (A-201, 'Perryridge', 900)
```

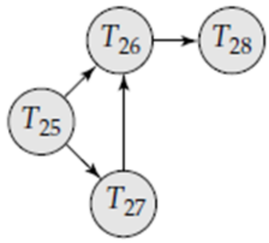
Let S be a schedule involving T_{29} and T_{30} . We expect there to be potential for a conflict for the following reason:

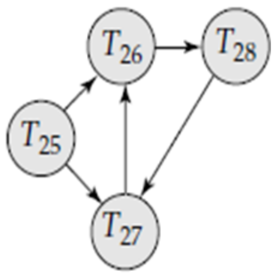
- If T_{29} uses the tuple newly inserted by T_{30} in computing sum(balance), then T_{29} read a value written by T_{30}. Thus, in a serial schedule equivalent to S, T_{30} must come before T_{29}.
- If T_{29} does not use the tuple newly inserted by T_{30} in computing sum(balance), then in a serial schedule equivalent to S, T_{29} must come before T_{30}.

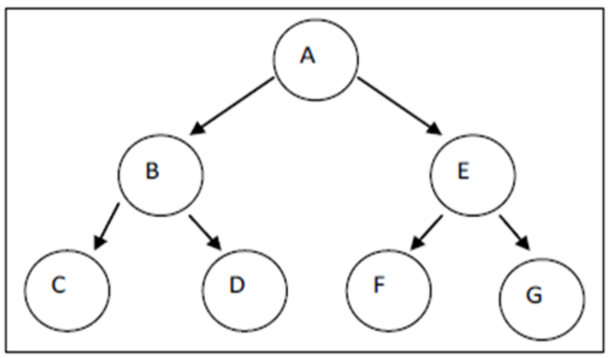
The second of these two cases is curious. T_{29} and T_{29} do not access any tuple in common, yet they conflict with each other! In effect, T_{29} and T_{29} conflict on a phantom tuple. If concurrency control is performed at the tuple granularity, this conflict would go undetected. This problem is called the **phantom phenomenon**.
To prevent the phantom phenomenon, we allow T_{29} to prevent other transactions from creating new tuples in the account relation with branch-name = "Perryridge."

5 AKTU Examination Questions

1. Define Concurrency Control.
2. Explain the phantom phenomena. Discuss a Time Stamp Protocol that avoids the phantom phenomena.
3. Discuss about deadlock prevention schemes.
4. Explain Concurrency Control. Why it is needed in database system?
5. What is deadlock? What are necessary conditions for it? How it can be detected and recovered?
6. Explain two phase locking protocol with suitable example.
7. Write the salient features of graph based locking protocol with suitable example.
8. What do you mean by multiple granularity? How the concurrency is maintained in this case. Write the concurrent transactions for the following graph.
   - T_1 wants to access Item C in read mode
   - T_2 wants to access item D in Exclusive mode
   - T_3 wants to read all the children of item B
   - T_4 wants to access all items in read mode
9. Define Exclusive Lock.


T_{26}
T_{28}
T_{25}
T_{27}


T_{26}
T_{28}
T_{25}
T_{27}


A
B
E
C
D
F
G

10. What is Two phase Locking (2PL)? Describe with the help of example.
11. What are multi version schemes of concurrency control? Describe with the help of an example. Discuss the various Time stamping protocols for concurrency control also.
12. Define timestamp.
13. Discuss about the deadlock prevention schemes.
14. Explain the following protocols for concurrency control.
    i) Lock based protocols
    ii) Time Stamp based protocols
15. What are the pitfalls of lock-based protocol?
16. Describe major problems associated with concurrent processing with examples. What is the role of locks in avoiding these Problems.
17. Explain the phantom phenomenon. Devise a time stamp based protocol that avoids the phantom phenomenon.
18. What do you mean by multiple granularties? How it is implemented in transaction system?