CS3492 DATABASE MANAGEMENT SYSTEMS

UNIT I RELATIONAL DATABASES **10**

Purpose of Database System – Views of data – Data Models – Database System Architecture – Introduction to relational databases – Relational Model – Keys – Relational Algebra – SQL fundamentals – Advanced SQL features – Embedded SQL– Dynamic SQL

UNIT II DATABASE DESIGN **8**

Entity-Relationship model – E-R Diagrams – Enhanced-ER Model – ER-to-Relational Mapping – Functional Dependencies – Non-loss Decomposition – First, Second, Third Normal Forms, Dependency Preservation – Boyce/Codd Normal Form – Multi-valued Dependencies and Fourth Normal Form – Join Dependencies and Fifth Normal Form

UNIT III TRANSACTIONS **9**

Transaction Concepts – ACID Properties – Schedules – Serializability – Concurrency Control – Need for Concurrency – Locking Protocols – Two Phase Locking – Deadlock – Transaction Recovery - Save Points – Isolation Levels – SQL Facilities for Concurrency and Recovery.

UNIT IV IMPLEMENTATION TECHNIQUES **9**

RAID – File Organization – Organization of Records in Files – Indexing and Hashing –Ordered Indices – B+ tree Index Files – B tree Index Files – Static Hashing – Dynamic Hashing – Query Processing Overview – Algorithms for SELECT and JOIN operations – Query optimization using Heuristics and Cost Estimation.

UNIT V ADVANCED TOPICS **9**

Distributed Databases: Architecture, Data Storage, Transaction Processing – Object-based Databases: Object Database Concepts, Object-Relational features, ODMG Object Model, ODL, OQL - XML Databases: XML Hierarchical Model, DTD, XML Schema, XQuery – Information Retrieval: IR Concepts, Retrieval Models, Queries in IR systems.

TOTAL: 45 PERIODS

TEXT BOOKS:

1. Abraham Silberschatz, Henry F. Korth, S. Sudharshan, —Database System Concepts||,Sixth Edition, Tata McGraw Hill, 2011
2. Ramez Elmasri, Shamkant B. Navathe, —Fundamentals of Database Systems||, Sixth Edition, Pearson, 2011.

REFERENCES:

1. C. J. Date, A.Kannan, S. Swamynathan, —An Introduction to Database Systems||, Eighth Edition, Pearson Education, 2006.
2. Raghu Ramakrishnan, —Database Management Systems||, Fourth Edition, McGraw-Hill College Publications, 2015.
3. G.K.Gupta,"Database Management Systems||, Tata McGraw Hill, 2011.

UNIT III TRANSACTIONS

Transaction Concepts – ACID Properties – Schedules – Serializability – Concurrency Control – Need for Concurrency – Locking Protocols – Two Phase Locking – Deadlock – Transaction Recovery - Save Points – Isolation Levels – SQL Facilities for Concurrency and Recovery.

Transaction Concept

A **transaction**is a *unit* of program execution that accesses and possibly updates various data items.

E.g. transaction to transfer $50 from account A to account B:

1. **read**(*A*)
2. *A* :=*A* – 50
3. **write**(*A*)
4. **read**(*B*)
5. *B* :=*B* + 50
6. **write**(*B)*

Two main issues to deal with:

- ✓ Failures of various kinds, such as hardware failures and system crashes
- ✓ Concurrent execution of multiple transactions

Example of Fund Transfer:-

Transaction to transfer $50 from account A to account B:

1. read(A)
2. A :=A – 50
3. write(A)
4. read(B)
5. B :=B + 50
6. write(B)

ACID Properties:-

A transaction is a unit of program execution that accesses and possibly updates various data items.

To preserve the integrity of data the database system must ensure

Atomicity:

Either all operations of the transaction are properly reflected in the database or none are.

Consistency: Execution of a transaction in isolation preserves the consistency of the database.

Isolation:

Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.

That is, for every pair of transactions Tiand Tj, it appears to Tithat either Tj, finished execution before Ti started, or Tj started execution after Ti finished.

Durability:

After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Atomicity

- Failure could be due to software or hardwarethe system should ensure that updates of a partially
- executed transaction are not reflected in the database

Atomicity requirement:

- if the transaction fails after step 3 and before step 6, money will be "lost" leading to an inconsistent database state

Durability requirement:

once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.

Transaction to transfer $50 from account A to account B:

1. **read**(*A*)
2. *A* :=*A* – 50
3. **write**(*A*)
4. **read**(*B*)
5. *B* :=*B* + 50
6. **write**(*B*)

Consistency requirement

In above example: the sum of A and B is unchanged by the execution of the transaction

In general, consistency requirements include

- Explicitly specified integrity constraints such as primary keys and foreign keys
- Implicit integrity constraints

- e.g. sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand

- A transaction must see a consistent database.
- During transaction execution the database may be temporarily inconsistent.
- When the transaction completes successfully the database must be consistent

Erroneous transaction logic can lead to inconsistency

Isolation requirement: if between steps 3 and 6, another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database (the sum A + *B* will be less than it should be).

	T1	**T2**
1.	**read(*A*)**	
2.	***A* :=*A* – 50**	
3.	**write(*A*)**	
		read(A), read(B), print(A+B)
4.	**read(*B*)**	
5.	***B* :=*B* + 50**	
6.	**write(*B***	

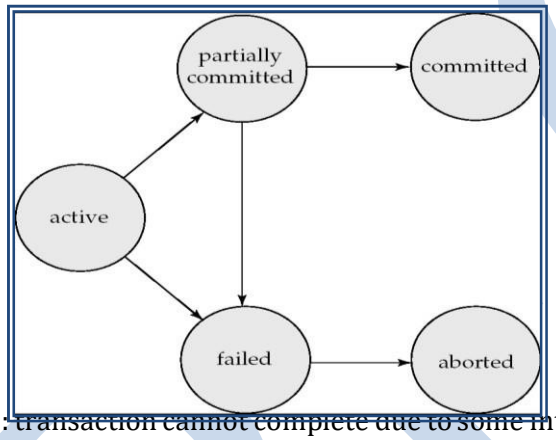
Isolation can be ensured trivially by running transactions serially that is, one after the other. However, executing multiple transactions concurrently has significant benefits.

Transaction State:

Active –the initial state; the transaction stays in this state while it is executing
Partially committed –after the final statement has been executed.
Failed -- after the discovery that normal execution can no longer proceed.
Aborted – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction.
Two options after it has been aborted:

- ✓ restart the transaction can be done only if no internal logical error kill the transaction
- ✓ Committed – after successful completion.

Transaction State



Recovery System

Failure Classification:
Transaction failure:

- **Logical errors**: transaction cannot complete due to some internal error condition
- **System errors**: the database system must terminate an active transaction due to an error
- condition (e.g., deadlock)

System crash: a power failure or other hardware or software failure causes the system to crash.
Fail-stop assumption: non-volatile storage contents are assumed to not be corrupted by system crash
Database systems have numerous integrity checks to prevent corruption of disk data
Disk failure:

- a head crash or similar disk failure destroys all or part of disk storage

Destruction is assumed to be detectable: disk drives use checksums to detect failures
Recovery Algorithms:
Recovery algorithms are techniques to ensure database consistency and transaction atomicity and durability despite failures
Recovery algorithms have two parts

1. Actions taken during normal transaction processing to ensure enough information exists to recover from failures
2. Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and
3. durability

Storage Structure:
Volatile storage:

- does not survive system crashes

Examples: main memory, cache memory
Nonvolatile storage:

- Survives system crashes

Examples: disk, tape, flash memory, non-volatile (battery backed up) RAM

Stable storage: Nonvolatile storage:

- non-volatile (battery backed up) RAM

Stable:

It is a mythical form of storage that survives all failures approximated by maintaining multiple copies on distinct nonvolatile media

Stable-Storage Implementation Maintain multiple copies of each block on separate disks, copies can be at remote sites to protect against disasters such as fire or flooding.

Failure during data transfer can still result in inconsistent copies:

Block transfer can result in

1. Successful completion
2. Partial failure: destination block has incorrect information
3. Total failure: destination block was never updated

- Protecting storage media from failure during data transfer (one solution):
- Execute output operation as follows (assuming two copies of each block):
- Write the information onto the first physical block.
- When the first write successfully completes, write the same information onto the second physical block.
- The output is completed only after the second write successfully completes.

Recovery and Atomicity:

1. Modifying the database without ensuring that the transaction will commit may leave the database
2. in an inconsistent state.
3. Consider transaction T_i that transfers \$50 from account A to account B; goal is either to perform all
4. database modifications made by $T_i$or none at all.
5. Several output operations may be required for T_i (to output A and B). A failure may occur after one
6. of these modifications have been made but before all of them are made.
7. To ensure atomicity despite failures, we first output information describing the modifications to stable
8. storage without modifying the database itself.

We study two approaches:

- ✓ log-based recovery, and
- ✓ shadow-paging

We assume (initially) that transactions run serially, that is, one after the other.

Log-Based Recovery:

A log is kept on stable storage.

The log is a sequence of log records, and maintains a record of update activities on the database.

- When transaction T_i starts, it registers itself by writing a $<T_i$ start>log record
- *Before* T_i executes write(X), a log record $<T_i, X, V_1, V_2>$ is written, where V_1 is the value of X before the write, and
- V_2 is the value to be written to X.
- Log record notes that Ti has performed a write on data item XjXjhad value V1before the write, and
- will have value V2after the write.
- When Ti finishes it last statement, the log record <Ticommit> is written.
- We assume for now that log records are written directly to stable storage (that is, they are not buffered)

Two approaches using logs

- ✓ Deferred database modification
- ✓ Immediate database modification

Serializability

Basic Assumption:–Each transaction preserves database consistency.

Thus serial execution of a set of transactions preserves database consistency.

A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule.

Different forms of schedule equivalence give rise to the notions of:

1. **Conflict serializability**
2. **View serializability**

Simplified view of transactions:

- We ignore operations other than **read** and **write** instructions
- We assume that transactions may perform arbitrary computations on data in local buffers in between
- reads and writes.
- Our simplified schedules consist of only **read** and **write** instructions.

Conflicting Instructions:

Instructions l_i and l_j of transactions T_i and T_j respectively, **conflict** if and only if there exists some item Q accessed by both l_i and l_j, and at least one of these instructions wrote Q.

1. l_i = **read**(Q), l_j = **read**(Q). l_i and $l_j$don't conflict.
2. l_i = **read**(Q), l_j = **write**(Q). They conflict.
3. l_i = **write**(Q), l_j = **read**(Q). They conflict
4. l_i = **write**(Q), l_j = **write**(Q). They conflict

Intuitively, a conflict between $l_i$and l_j forces a (logical) temporal order between them.

If l_i and l_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged

Conflict Serializability

- If a schedule S can be transformed into a schedule S' by a series of swaps of
- non-conflicting instructions, we say that S and S' are **conflict equivalent**.

- We say that a schedule S is **conflict serializable** if it is conflict equivalent to a serial schedule

Schedule 3 can be transformed into Schedule 6, a serial schedule where T2 follows T1, by series of swaps of non-conflicting instructions.
Therefore Schedule 3 is conflict serializable.
Example of a schedule that is not conflict serializable.

We are unable to swap instructions in the above schedule to obtain either the serial schedule $<T_3, T_4>$, or the serial schedule $<T_4, T_3>$.

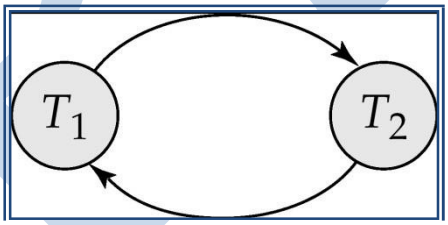
View Serializability

- A schedule S is **view serializable** if it is view equivalent to a serial schedule.
- Every conflict serializable schedule is also view serializable.
- Below is a schedule which is view-serializable but *not* conflict serializable.
- What serial schedule is above equivalent to?
  Every view serializable schedule that is not conflict serializable has **blind writes.**

Testing for Serializability

Consider some schedule of a set of transactions $T_1, T_2, ..., T_n$

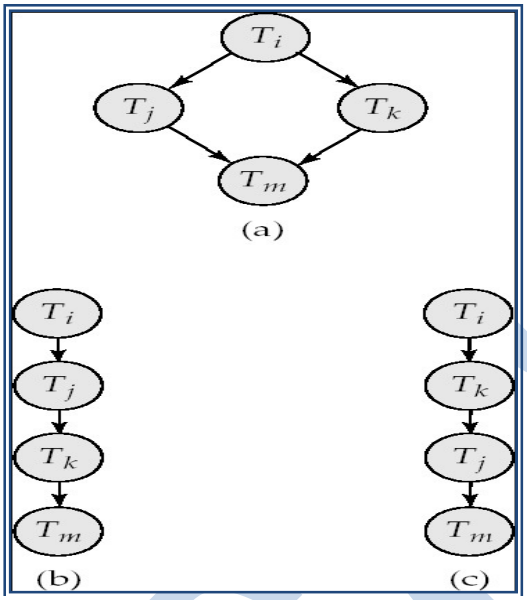
Precedence graph:- a direct graph where the vertices are the transactions (names).
We draw an arc from Ti to Tj if the two transaction conflict, and Ti accessed the data item on which the conflict arose earlier.
We may label the arc by the item that was accessed.
Example:-



Test for Conflict Serializability

A schedule is conflict serializable if and only if its precedence graph is acyclic.
Cycle-detection algorithms exist which takes order n2 time, where n is the number of vertices in the graph.
(Better algorithms take order n + e where e is the number of edges.). If precedence graph is acyclic, the serializability order can be obtained by a topological sorting of the graph.
This is a linear order consistent with the partial order of the graph.
For example, a serializability order for Schedule A would be T5→T1→T3→T2→T4

- Are there others?



Test for View Serializability

The precedence graph test for conflict serializability cannot be used directly to test for view serializability.

Extension to test for view serializability has cost exponential in the size of the precedence graph.

The problem of checking if a schedule is view serializable falls in the class of NP-complete problems.

Thus existence of an efficient algorithm is extremely unlikely.

However practical algorithms that just check some sufficient conditions for view serializability can still be used.

Lock-Based Protocols

A lock is a mechanism to control concurrent access to a data item

Data items can be locked in two modes :

1. **exclusive (X) mode:**
   Data item can be both read as well as written. X-lock is requested using lock-X instruction.
2. **shared (S) mode:**
   Data item can only be read. S-lock is requested using lock-S instruction.

Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.

Lock-Based Protocols

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.
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*)

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.
A locking protocol is a set of rules followed by all transactions while requesting and releasing locks.
Locking protocols restrict the set of possible schedules.

Pitfalls of Lock-Based Protocols:

Consider the partial schedule

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)	

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 and its locks released.
The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.
Starvation is also possible if 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 and are granted an S-lock on the same item.
The same transaction is repeatedly rolled back due to deadlocks.
Concurrency control manager can be designed to prevent starvation.

The Two-Phase Locking Protocol:
This is a protocol which ensures conflict-serializable schedules.
Phase 1: Growing Phase

- transaction may obtain locks
- transaction may not release locks

Phase 2: Shrinking Phase

- transaction may release locks
- transaction may not obtain locks

The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their lock points (i.e. the point where a transaction acquired its final lock).

Drawbacks:

Two-phase locking *does not* ensure freedom from deadlocks

Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called strict two-phase locking. Here a transaction must hold all its exclusive locks till it commits/aborts.

Rigorous two-phase locking is even stricter: here *all* locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.

There can be conflict serializable schedules that cannot be obtained if two-phase locking is used.

However, in the absence of extra information (e.g., ordering of access to data), two-phase locking is needed for conflict serializability in the following sense:

Given a transaction T_i that does not follow two-phase locking, we can find a transaction T_j that uses two-phase locking, and a schedule for T_i and T_j that is not conflict serializable.

Lock Conversions:

Two-phase locking with lock conversions:

– First Phase:

- 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:

- can release a lock-S
- can release a lock-X
- can convert a lock-X to a lock-S (downgrade)
- This protocol assures serializability. But still relies on the programmer to insert the
- various locking instructions.

Automatic Acquisition of Locks (conversion of locks)

A transaction T_i issues the standard read/write instruction, without explicit locking calls.

Types:

Upgrade mode—conversion of shared mode lock to exclusive lock mode

Shared mode--- conversion of exclusive mode to shared mode

The operation read (D) is processed as:

if T_i has a lock on D
then
read(D)
else begin

if necessary wait until no other
transaction has a **lock-X** on *D*
grantT_i a **lock-S** on *D*;
read(*D*)
end

Write *(D)* is processed as:

ifT_i has a **lock-X** on *D*
then
write(*D*)
else begin
if necessary wait until no other trans. has any lock on *D*,
ifT_i has a **lock-S** on *D*
then
upgrade lock on *D* to **lock-X**
else
grantT_i a **lock-X** on *D*
write(*D*)
end;

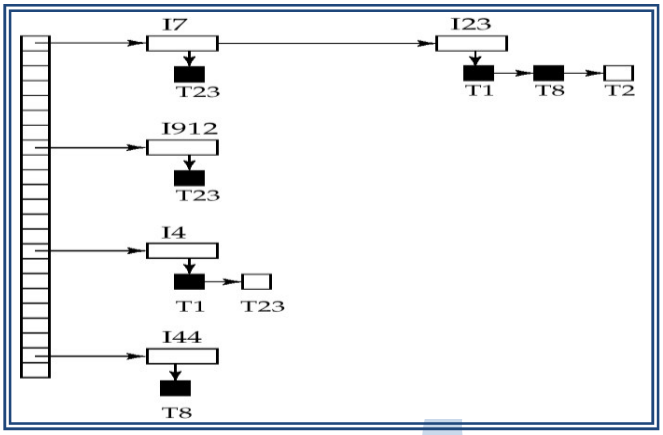
All locks are released after commit or abort

Implementation of Locking

- 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 sending 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
- The lock table is usually implemented as an in-memory hash table indexed on the name of the data item being
- locked

Lock Table

- Black rectangles indicate granted locks, white ones indicate waiting requests
- Lock table also records the type of lock granted or requested
- New request is added to the end of the queue of requests for the data item, and granted if it is compatible with
- all earlier locks
- Unlock requests result in the request being deleted, and later requests are checked to see if they can now be
- granted
- If transaction aborts, all waiting or granted requests of the transaction are deleted
- lock manager may keep a list of locks held by each transaction, to implement this efficiently



Intention Lock Modes

In addition to S and X lock modes, there are three additional lock modes with multiple granularity:

***Intention-shared* (IS):** indicates explicit locking at a lower level of the tree but only with shared locks.

***Intention-exclusive* (IX):** indicates explicit locking at a lower level with exclusive or shared locks

***Shared and intention-exclusive* (SIX):** the sub tree rooted by that node is locked explicitly in shared mode and explicit locking is being done at a lower level with exclusive-mode locks.

- Intention locks allow a higher level node to be locked in S or X mode without having to check all descendent nodes.

Compatibility Matrix with Intention Lock Modes

The compatibility matrix for all lock modes is:

Multiple Granularity Locking Scheme:

Transaction Ti can lock a node Q, using the following rules:

1. The lock compatibility matrix must be observed.
2. The root of the tree must be locked first, and may be locked in any mode.
3. A node Q can be locked by Ti in S or IS mode only if the parent of Q is currently locked by Ti in either IX or
4. IS mode.
5. A node Q can be locked by Ti in X, SIX, or IX mode only if the parent of Q is currently locked by Ti in either
6. IX or SIX modes.
7. Ti can lock a node only if it has not previously unlocked any node (that is, Tiis two-phase).
8. Tican unlock a node Q only if none of the children of Q are currently locked by Ti.

Observe that locks are acquired in root-to-leaf order, whereas they are released in leaf-to-root order.

Deadlock Handling:

Consider the following two transactions:

***T*1: write (*X*)** ***T*2: write(*Y*)**

write(*Y*) **write(*X*)**

Schedule with deadlock

lock-X on X ***lock-X on Y***

write (X) ***wait for lock-X on X***

write (X)

wait for lock-X on Y

System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.

Deadlock prevention protocols ensure that the system will *never* enter into a deadlock state. Some prevention strategies :

Require that each transaction locks all its data items before it begins execution (predeclaration).

Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order (graph-based protocol).

More Deadlock Prevention Strategies

Following schemes use transaction timestamps for the sake of deadlock prevention alone.

Wait-die scheme — non-preemptive

- older transaction may wait for younger one to release data item. 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.
- May be fewer rollbacks than *wait-die* scheme.
- Both in *wait-die* and in *wound-wait* schemes, a rolled back transactions is restarted with its original timestamp.
- Older transactions thus have precedence over newer ones, and starvation is hence avoided.

Timeout-Based Schemes :

- a transaction waits for a lock only for a specified amount of time. After that, the wait times out
- and the transaction is rolled back.

- thus deadlocks are not possible

- Simple to implement; but starvation is possible. Also difficult to determine good value of the timeout

- interval.

Deadlock Detection

Deadlocks can be described as a *wait-for graph*, which consists of a pair $G = (V,E)$,

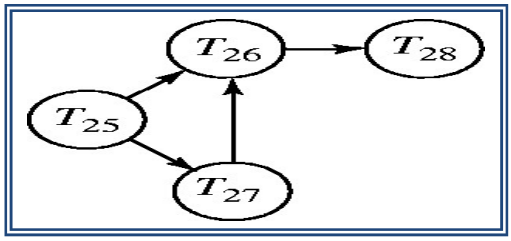
V is a set of vertices (all the transactions in the system)

E is a set of edges; each element is an ordered pair $T_i \rightarrow T_j$.

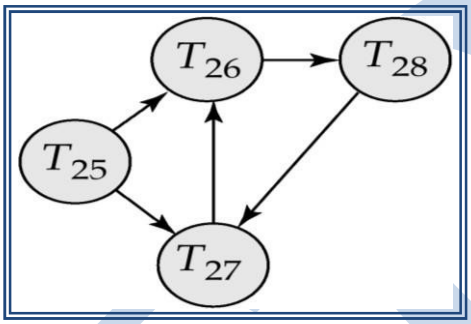
If $T_i \rightarrow T_j$ is in E, then there is a directed edge from T_i to T_j, implying that T_i is waiting for T_j to release a data item.

When T_i requests a data item currently being held by T_j, then the edge T_iT_j is inserted in the wait-for graph. This edge is removed only when T_j is no longer holding a data item needed by T_i.

The system is in a deadlock state if and only if the wait-for graph has a cycle. Must invoke a deadlock-detection algorithm periodically to look for cycles.



Wait-for graph without a cycle



Wait-for graph with a cycle

Deadlock Recovery

When deadlock is detected:

Selection of the victim-Some transaction will have to rolled back (made a victim) to break deadlock. Select that transaction as victim that will incur minimum cost.

Rollback -- determine how far to roll back transaction

Total rollback: Abort the transaction and then restart it.

More effective to roll back transaction only as far as necessary to break deadlock.

Starvation happens if same transaction is always chosen as victim. Include the number of rollbacks in the cost factor to avoid starvation