Third Normal Form

- BCNF is very retsrictive
- Not always possible to decompose a schema into one in BCNF that has the lossless-join and dependency preservation properties.
- It is always possible to achieve that in 3NF.

Definition: An attribute A of relation R is called *prime* if and only if A is part of some key for R.

Definition: A relation R in a database schema with a set of FDs F is said to be in *third normal form* (3NF) if for any functional dependency of the form $X \rightarrow A$ in F^+ that is preserved in R, where A is a single attribute and $A \notin X$, one of the following two conditions is true:

1. X is a superkey for R
2. A is a prime attribute in R

A database schema is in 3NF if everyone of its relations is in 3NF.

- Every schema that is in BCNF is also in 3NF.

Example: The following database schema is in BCNF, hence it is also in 3NF.

emps

emp_id	emp_name	emp_phone	dept_name

depts

dept_name	dept_phone	dept_mgrname

emp-skills

emp_id	skill_id	skill_date	skill_lvl

skills

skill_id	skill_name

Functional dependencies:

1. `emp_id` $\longrightarrow$ `emp_name emp_phone dept_name`
2. `dept_name` $\longrightarrow$ `dept_phone dept_mgrname`
3. `skill_id` $\longrightarrow$ `skill_name`
4. `emp_id skill_id` $\longrightarrow$ `skill_date skill_lvl`

Example: The following schema is in 3NF but not in BCNF.

emps

emp_id	emp_name	emp_phone	dept_name	emp_city	emp_straddr

empadds

emp_city	emp_zip	emp_straddr

Functional dependencies:

1. **emp_id** → **emp_name emp_phone dept_name**
2. **emp_city emp_straddr** → **emp_zip**
3. **emp_zip** → **emp_city**

The functional dependency **emp_zip** → **emp_city** is preserved in the relation **empadds** but **emp_zip** is not a key. The schema is not in BCNF.

The attribute **emp_city** is prime (key is **emp_city emp_straddr**). Hence the schema is in 3NF.

Second Normal Form

Definition: A relation R in a database schema with functional dependencies F is said to be in *second normal form* (2NF) if for any functional dependency $X \rightarrow A$ in F^+ that is preserved in R, where A is a single attribute, $A \notin X$ and A is non-prime, X is not properly contained in any key of R.

Example: The following schema is in 2NF

`emps`

emp_id	emp_name	emp_phone	dept_name	dept_phone	dept_mgrname

`emp-skills`

emp_id	skill_id	skill_date	skill_lvl

`skills`

skill_id	skill_name

Functional dependencies:

1. `emp_id` → `emp_name emp_phone dept_name`
2. `dept_name` → `dept_phone dept_mgrname`
3. `skill_id` → `skill_name`
4. `emp_id skill_id` → `skill_date skill_lvl`

The superkey for the relation `emp-skills` is `emp_id skill_id`.

The dependencies **emp_id skill_id** $\rightarrow$ **skill_date** and **emp_id skill_id** $\rightarrow$ **skill_lvl** are implied by F and are preserved in **emp-skills**. Neither of **skill_date** and **skill_lvl** is prime and the lhs is not a proper subset of the superkey.

• 2NF is only interesting for historical purposes.

3NF decomposition

Given a universal relation R and a set of FDs F, the following algorithm generates a lossless-join dependency preserving decomposition of R in 3NF. The output is a set of headings for the resulting schema.

replace F by its minimal cover;
$S := \emptyset$
For all $X \rightarrow Y$ in F
    If there does not exist Z in S s.t. $X \cup Y \subseteq Z$
        $S := S \cup (X \cup Y)$
For all candidate keys K for R
    If K is not contained in any Z in S
        $S := S \cup K$;

• Decomposition into 3NF eliminates update anomalies and makes it possible to verify efficiently that desirable functional dependencies remain satisfied when the database is updated.

Example: Assume the following schema:

$Head(R) = \{\texttt{instructor class\_no class\_room text}\}$

$F = \{\texttt{class\_no} \rightarrow \texttt{class\_room text}\}$

The application of the algorithm on this schema will create relations R_1 and R_2 s.t. :

$Head(R_1) = \{\texttt{class\_no class\_room text}\}$

$Head(R_2) = \{\texttt{class\_no instructor}\}$

Summary

- The E-R approach to database design and normalization complement each other.
- A number of commercial products are available for supporting database designers in performing the design task.
- Database design tools:
  1. E-R design editor
  2. E-R to Relational transformation tools
  3. FD to E-R Design transformation tools
  4. Design Analyzers

More Examples

1. Consider the relation r with schema $R = \{A, B, C, D\}$, functional dependencies $F = \{A \rightarrow B, C \rightarrow B, D \rightarrow ABC, AC \rightarrow D\}$, and its decomposition into relations r_1, r_2 with schemas $R_1 = \{A, B\}$ and $R_2 = \{B, C, D\}$ respectively.

   (a) Is the decomposition lossless-join? Is it dependency preserving?

   $R_1 \cap R_2 = \{B\}$

   None of $B \rightarrow AB$, $B \rightarrow BCD$ is in F^+. Hence, it is not lossless-join.

   Dependency $D \rightarrow ABC$ is not preserved in any of R_1, R_2

   (b) Find a decomposition of R into R'_1, R'_2 that is lossless-join.

   $R'_1 = \{A, C, D\}$, $R'_2 = \{B, D\}$

   $R'_1 \cap R'_2 = \{D\}$

   Both $D \rightarrow ACD$ and $D \rightarrow B$ are in F^+. Hence, the decomposition is lossless-join.

2. Consider the schema $R(A, B, C, D)$ and the functional dependencies $F = \{AC \rightarrow B, AB \rightarrow D\}$.

   (a) What is the candidate key and what is the highest normal form of this schema and dependency set?
   $A^+ = \{A\}, B^+ = \{B\}, C^+ = \{C\}, D^+ = \{D\}$
   $AC^+ = \{ABCD\}$, $AB^+ = \{ABD\}$ Hence, AC is the candidate key.

   For dependencies $AC \rightarrow B$ and $AB \rightarrow D$, B and D are non-prime, they're not contained in any candidate key and the lhs's of the dependencies are not proper subsets of the candidate key. Hence, R is in 2NF.

   R is not in 3NF because $AB \rightarrow D$ is preserved in R, D is non-prime and AB is not a superkey.

   Hence, the highest normal form of R is 2NF.

   (b) Does the schema have a decomposition into a higher normal form that is lossless-join and dependency preserving?
   Apply the algorithm for 3NF decomposition:
   F is minimal. The algorithm yields the decomposition:
   $R_1 = \{A, B, C\}$ and $R_2 = \{A, B, D\}$.
   This decomposition is also in BCNF: AC is the superkey for R_1 and AB is the superkey for R_2.

Transaction Processing

- In modern applications databases are shared by more than one user that can query and update them.
- It is not possible to provide each user with their own copy of the database.
- The database needs to be accessed by more than one user at the same time (user processes must execute concurrently).
- A database management system must ensure that
  - concurrent access is provided
  - each user has a consistent view of data
- These properties are guaranteed by means of a *transaction management* strategy.
- A *transaction* is a sequence of database operations (reads and writes).
- Transaction management guarantees that user supplied transactions appear to execute in *isolation*, although they may execute concurrently.

Example: Inconsistent View of Data

Consider the relation **`Accounts`** shown below:

`Account#`	`Lname`	`Fname`	`Type`	`Balance`
123456	Doe	John	Checking	$900.00
654321	Doe	John	Savings	$100.00
...	...	...	...	...

Process P_1 transfers $400 from account 123456 to account 654321. The transfer is implemented by subtracting $400 from the balance of the first account, followed by adding $400 to the balance of the second account.

3 states are distinguishable:

	Balance of acc1:	Ballance of acc2:
Before P_1:	$900	$100
After subtraction:	$500	$100
After addition:	$500	$500

Process P_2 performs a credit check on the account holder and requires a minimum of $900 as the total balance of the accounts to approve the issuance of a credit card. The process reads the balance values of the two tuples and computes their sum.

P_2 is running simultaneously with P_1.

Process Interleaving

P_1	P_2
	sum:=0
subtract \$400 from acc1 balance	
	add acc1 balance to sum (sum=\$500)
	add acc2 balance to sum (sum=\$600)
	issue rejection
add \$400 to acc2 balance	

P_2 ends by issuing a rejection whereas the actual total balance is \$1000. Hence, P_2 has an inconsistent view of the data in the database.

If P_1 executed before P_2 or P_2 executed before P_1, then both processes would have the correct view of data.

Not all interleaved executions of transactions are correct: only those that are equivalent to some serial execution.

Another interleaved execution:

P_1	P_2
	sum:=0
	add acc1 balance to sum (sum=$900)
subtract $400 from acc1 balance	
	add acc2 balance to sum (sum=$1000)
add $400 to acc2 balance	
	issue approval

This execution is correct: both processes see the correct data.

It is equivalent to the serial executions P_1, P_2 and P_2, P_1.

Transaction management has to ensure that only correct interleaving of processes takes place.

Transaction Management

- The problems encountered in the development of large database applications led to the development of transaction management techniques.

  E.g., the following problems arise:

  1. Creation of inconsistent results: the machine crashes in the middle of executing a process.
  2. Errors of concurrent execution: arbitrary concurrent execution of processes lead to inconsistent views of data
  3. Uncertainty as to when changes become permanent: Can we be confident about the results residing in secondary storage even if processes have completed successfully?

- The concept of a transaction was invented to solve these problems.

- A transaction is a series of database operations (reads and writes) that form a single logical entity with respect to the application being modeled.

  E.g. a transfer of funds between accounts is considered a single logical entity

- A transaction *commits* when it finishes execution normally; otherwise it *aborts*.

- Transactions guarantee the following properties (known as the ACID properties): *atomicity, consistency, isolation* and *durability*.

Atomicity

Transactions are considered atomic as far as their effect on the database is concerned. That is, either all operations that make up the transaction are executed or none is. The set of operations that make up the transaction is considered indivisible.

- Atomicity is preserved even when crashes occur: a *database recovery* procedure performs a *rollback* to bring the database back to its state prior to transaction execution.

Consistency

If a transaction preserves a domain-specific consistency constraint when it is executed in isolation of any other transaction, then it should preserve the same constraint when excuted concurrently with other transactions.

Isolation

Also called **serializability**. Any schedule of interleaved execution of transactions is equivalent to some *serial* schedule, where transactions are executed in isolation, one after the other.

Durability

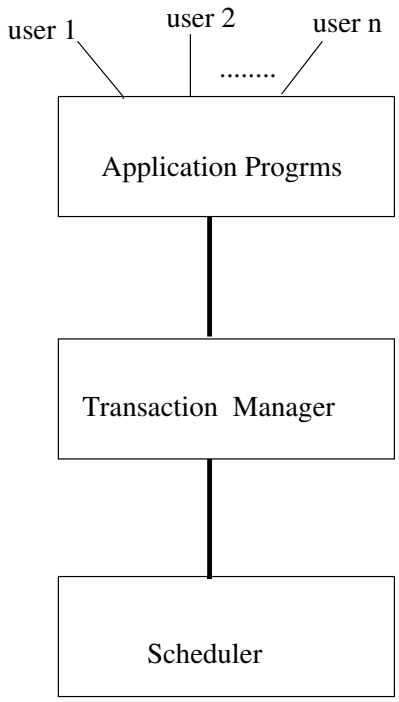
After a transaction commits, it is guaranteed to be recoverable, i.e., transactions are durable to crashes.

- The properties of atomicity and durability are trivially satisfied by any transaction that performs only read operations.
- Notation:
  - $T_1, T_2, \ldots, T_k$: transactions
  - $R_i(X)$: transaction T_i reads database item X
  - $R_i(X, v)$: transaction T_i reads database item X; v is the value read
  - $W_i(X)$: transaction T_i writes database item X
  - $W_i(X, v)$: transaction T_i writes database item X; v is the value written
  - C_i: transaction T_i commits
  - A_I: transaction T_i aborts
- A *schedule* or *history* is an interleaved sequence of operations.

  E.g. a schedule for transactions T_1, T_2 may be

$$R_2(A)\ W_2(A)\ R_1(A)\ R_1(B)\ R_2(B)\ W_2(B)\ C_1\ C_2$$

- A schedule is the result of the translation of processes (specified in some high-level language) into a series of primitive operations.
- The *scheduler* component of the transaction processing component of a DBMS ensures that only "correct" schedules are executed.



- Given a set of transaction specifications, the scheduler component produces a schedule that is equivalent to some serial execution of the transaction.
- If no such schedule is possible, the transaction manager aborts or delays some of the transactions.
- The scheduler also detects *deadlocks*, i.e., situations in which none of the transactions participating in the schedule can proceed unless one of them is aborted.

Example: The schedule

$$R_2(A)\ W_2(A)\ R_1(A)\ R_1(B)\ R_2(B)\ W_2(B)\ C_1\ C_2$$

is illegal. It is not equivalent to any serial execution of the two transactions.

Interpretation of the schedule:

$T_1 = \{R_1(A), R_1(B), C_1\}$
$T_2 = \{R_2(A), W_2(A), R_1(B), W_2(B), C_2\}$

The schedule $S = R_2(A)\, W_2(A)\, R_1(A)\, R_1(B)\, R_2(B)\, W_2(B)\, C_1\, C_2$ is correct only if it is equivalent to one of the serial schedules T_1, T_2 or T_2, T_1.

In T_1, T_2, the values of A and B read by T_1 have not been modified by T_2. But in S, T_1 reads A after T_2 has modified it.

In T_2, T_1, the values read by T_1 are those written by T_2. But in S, T_1 reads B before T_2 writes it.

Hence, the schedule has different effects than any serial execution.

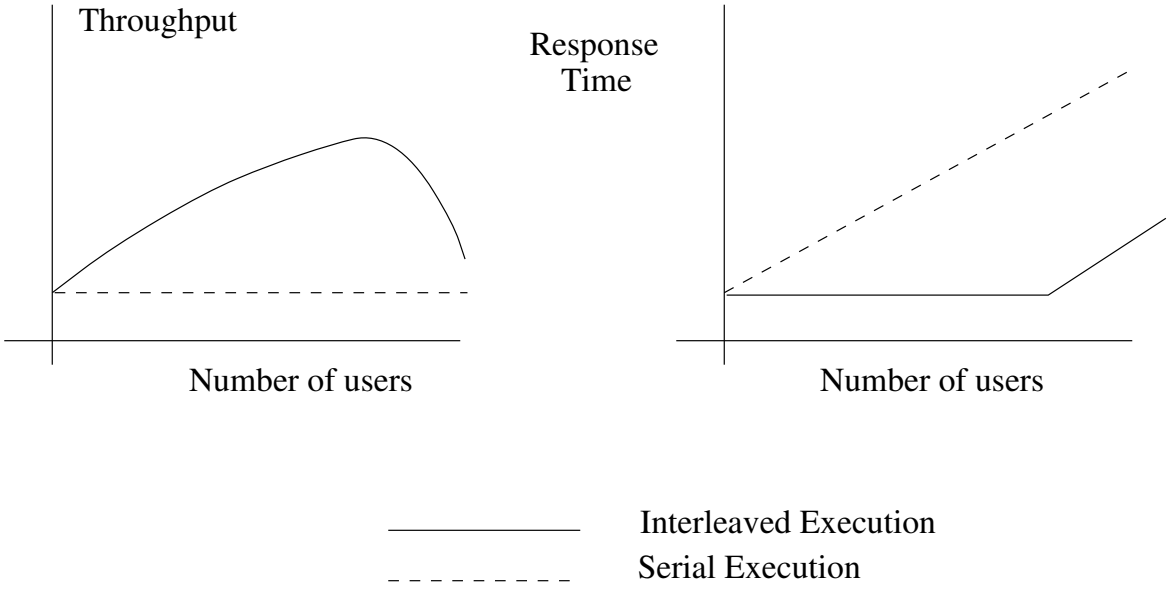
- Two schedules are called *equivalent* if, for any initial state, they have the same effects.
- A schedule is called *serializable* if it can be shown to be equivalent to some serial execution of the same transactions.
- Only serializable schedules are acceptable.

  E.g. $T_1 = \{R_1(A), R_1(B), W_1(A)\}$
  $T_2 = \{W_2(A), R_2(A)\}$
  $S = W_2(A), R_1(A), R_1(B), R_2(A), W_1(A)$

  S is serializable: it is equivalent to $T_2\ T_1$.

Interleaving of DB Operations

- Inrterleaving of database operations can yield large performance gains.
- While some transaction is performing I/O, another transaction can be using the CPU.
- System *throughput* (i.e., the number of transactions that can finish execution in a given period of time) increases whereas *response time* remains constant.



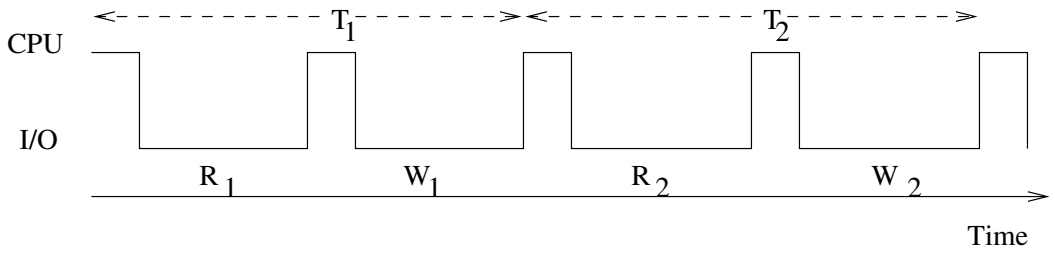
Example: Serial vs Concurrent Execution

A number of database transactions need to be serviced by the transaction manager. Each transaction uses both CPU and I/O resources.

$$T_i : (cpu\ op.)\ R_i()\ (cpu\ op.)\ W_i()\ C_i$$

The system has a single CPU with a 5ms interval and a single disk. Each I/O operation requires 50ms of wait time.

Resourse usage of the system in serial execution:

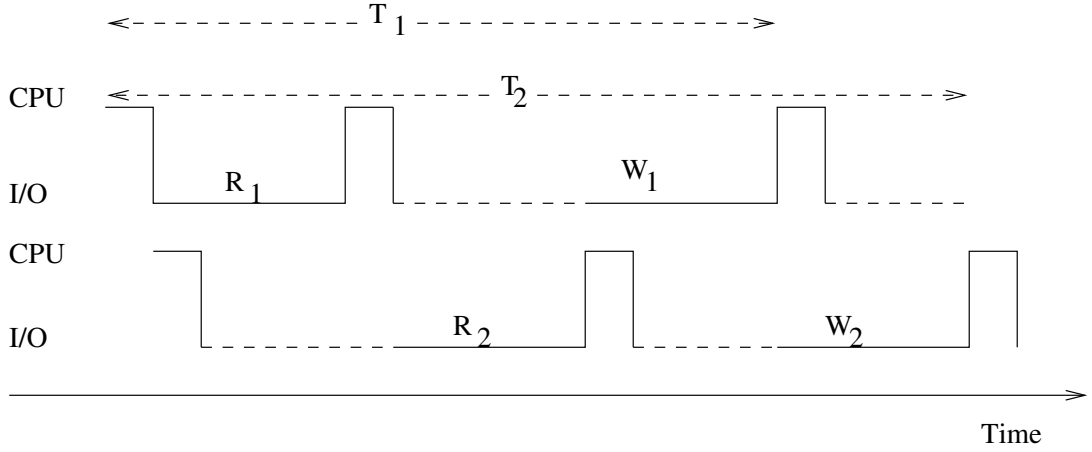


Each transaction requires 110ms. Hence, throughput is 1 transaction per 110ms or 9.09 transactions per sec.

CPU is underutilized: active 9.09% of the time.

Interleaved execution of transactions can increase CPU utilization and thus the system throughput.

Resource usage of the system in interleaved execution:



System throughput has already increased.

Throughput will increase with the number of transactions processed concurrently.

Additional improvements will be incurred when more than one I/O devices are used.

Testing Serializability

- We need criteria for determining whether schedules are equivalent to some serial execution of the same transactions.
- Two database operations are said to be *conflicting* if they belong to different transactions, they reference the same data item and at least one of them is a write operation.
- The order in which conflicting operations occur in a schedule is important.
- Two schedules are called *equivalent* if all pairs of conflicting operations occur in the same order.
- A schedule is serializable if it is equivalent to a serial schedule.
- There may be more than one serial schedules equivalent to some serializable schedule.
- Notation: $op_i(X) <<_S op_j(X)$ means that operation op_i precedes operation op_j in schedule S.
- If $op_i(X) <<_S op_j(X)$ then it must also be the case that $op_i(X) <<_{S'} op_j(X)$, where S' is a serial schedule equivalent to S.
- If $op_i(X) <<_S op_j(X)$ and $op_j(Y) <<_S op_i(Y)$, then S is not serializable. If it were, then in the equivalent serial schedule S', transaction T_i should both precede and follow transaction T_j.

Example: The lost update problem.

Consider the schedule $S_1 = R_1(A)\ R_2(A)\ W_1(A)\ W_2(A)\ C_1\ C_2$

Conflicting operations: $R_1(A), W_2(A)$
$R_2(A), W_1(A)$

Assume there is a serial schedule S_1' equivalent to S_1.

Because of $R_1(A) <<_{S_1} W_2(A)$, it should also be the case that $R_1(A) <<_{S_1'} W_2(A)$, i.e., T_1 must precede T_2.

Because of $R_2(A) <<_{S_1} W_1(A)$, it should also be the case that $R_2(A) <<_{S_1'} W_1(A)$, i.e., T_2 must precede T_1.

Hence, the schedule is non-serializable.

Example: The blind write problem.

Consider the schedule $S_2 = W_1(A)\ W_2(A)\ W_2(B)\ W_1(B)\ C_1\ C_2$

Conflicting operations: $W_1(A), W_2(A)$
$W_2(B), W_1(B)$

Assume there is a serial schedule S_2' equivalent to S_2.

Because of $W_1(A) <<_{S_2} W_2(A)$, it should also be the case that $W_1(A) <<_{S_2'} W_2(A)$, i.e., T_1 must precede T_2.

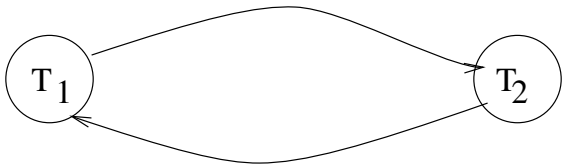
Because of $W_2(B) <<_{S_2} W_1(B)$, it should also be the case that $W_2(B) <<_{S_2'} W_1(B)$, i.e., T_2 must precede T_1.

Hence, the schedule is non-serializable.

Precedence Graphs

Given a schedule S, a *precedence graph* $PG(S)$ for S is a directed graph whose vertices correspond to the transactions in the schedule and whose set of edges consists of an edge $T_i \rightarrow T_j$ whenever there exists two conflicting operations op_i, op_j in S and $op_i <<_s op_j$.

Example: The schedules $S_1 = R_1(A)\ R_2(A)\ W_1(A)\ W_2(A)\ C_1\ C_2$ and $S_2 = W_1(A)\ W_2(A)\ W_2(B)\ W_1(B)\ C_1\ C_2$ both correspond to the precedence graph:



Theorem: A schedule S is serializable if and only if the precedence graph $PG(S)$ contains no cycle.

Proof: The following lemma is needed first

> **Lemma:** In any finite directed acyclic graph G, there is always a vertex v with no incoming edges.

(A) If $PG(S)$ has no cycle, S is serializable.

Assume there are m transactions $T_1, T_2, \ldots, T_m$ in S. We need to find a reordering $T_{i_1}, T_{i_2}, \ldots, T_{i_m}$ of the transactions in order to construct an equivalent serial schedule.

By the previous lemma, in the precedence graph $PG(S)$ there will be some vertex T_k with no incoming edges. Let T_{i_1} be T_k.

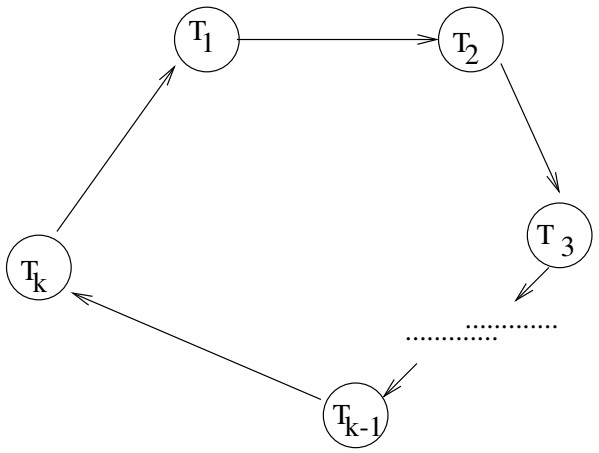
Since T_k has no incoming edges in $PG(S)$ there is no pair of conflicting operations of T_k and some other transaction T_j such that the operation of T_j should precede that of T_k. Hence, in the equivalent serial schedule, T_k should the first to be excuted.

Remove T_k from $PG(S)$ along with all its incident edges. The resulting graph is still acyclic. Hence we can find a vertex T_l that has no incoming edges. Let T_{i_2} be T_l. Then T_l should follow T_k in the serial schedule.

Continue this process until the precedence graph contains one vertex. The corresponding transaction is the last one in the serial schedule.

(B) If S is serializable, then $PG(S)$ is acyclic.

Let S be a serializable schedule and let $PG(S)$ contain a cycle:



$T_1 <<_S T_2 <<_S T_3 <<_S \ldots T_{k-1} <<_S T_K <<_S T_1$ (contradiction)

Locking to Ensure Serializability

- Concurrent access to database items is controlled by strategies based on *locking*, *timestamping* or *certification*.
- A *lock* is an access privilege to a single database item.
- A transaction can lock an object by passing request to the Lock Manager.
- The locks obtained by transactions are stored in a *lock table* which consists of entries of the form : $(item, lock\ type, transaction)$.
- Locks can be *shared* or *exclusive*.
- When a transaction holds an exclusive lock on a database item, no other transaction can read or write the item (used for writing).
- When it holds a shared lock (for reading a DB item), other transactions can obtain a shared lock on the same item as well.
- For simplicity, assume there is a single type of lock.
- Every transaction must obtain a lock before accessing a database item.
- All items locked by a transaction must be unlocked, otherwise no other transaction may gain access them.
- A transaction requesting access to an item on which another transaction holds a lock must wait until the lock is released.

Example: Locking can prevent the lost update problem.

$T_1 = Lock(A)\ R_1(A)\ W_1(A)\ Unlock(A)\ C_1$
$T_2 = Lock(A)\ R_2(A)\ W_2(A)\ Unlock(A)\ C_2$

Under the locking policy, only serial execution of the transactions is permitted.

Livelock

- Undesirable phenomena may occur if locks are granted arbitrarily.

  E.g., while T_2 is waiting for T_1 to release the lock on A, another transaction T_3 that has also requested a lock on A is granted the lock instead of T_2. When T_3 releases the lock on A the lock is granted to T_4 etc.

- The situation where a transaction may wait for ever while other transactions obtain a lock on a DB item is called a *livelock*.

- Livelocks can be avoided by using a first-come-first-served lock granting strategy.

- FCFS can still cause a deadlock though.