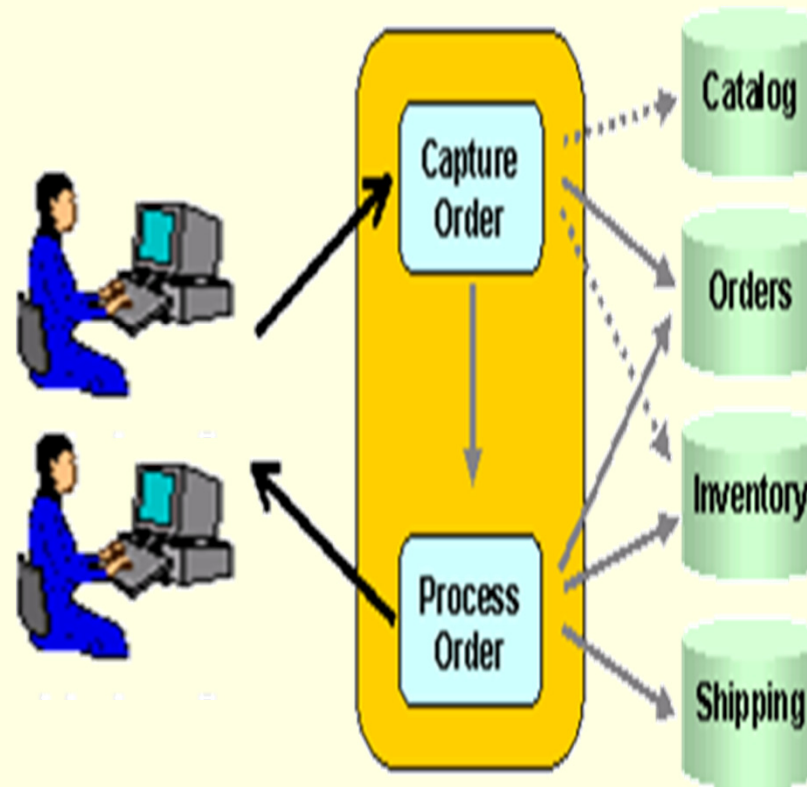


SCS 2109 Database II

Transaction Management

Objectives

- demonstrate concurrent transaction processing of database systems and associated problems



order capture and order process

Objectives

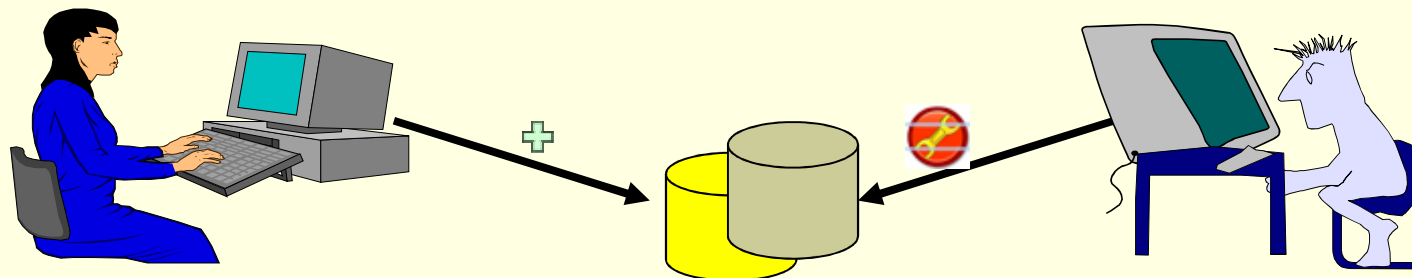
- demonstrate transaction processing of database systems and associated problems
 - Introduction to transaction processing (pg. 744)
 - Stabilizability (pg. 759)
 - Transaction support in SQL (pg. 770)

Course Content

- **Transaction management:** Why concurrency control required; transaction processing; transaction properties.
- **Concurrency:** Serial, Non-Serial, Conflict Serializable schedules; serializability, precedence graphs.

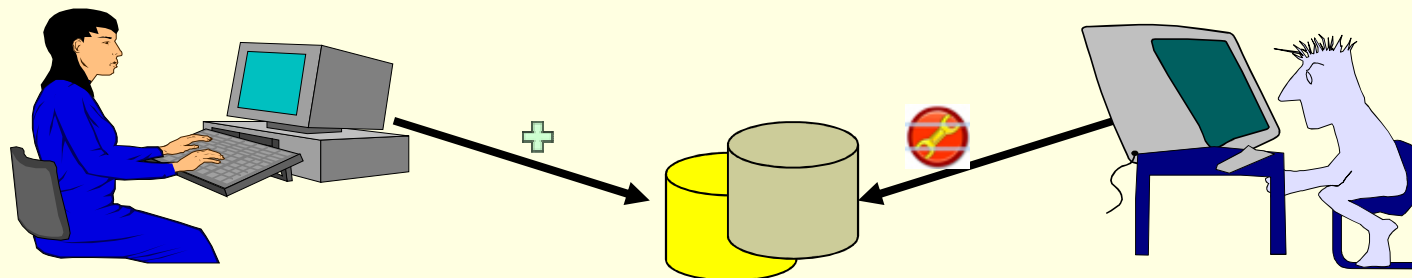
Concurrent Execution of Transactions

- Concurrency control deals with influencing how data can be viewed and updated by users accessing the same information at one time.
 - Many airlines / destinations / flights
 - Same flight / day / time / seat



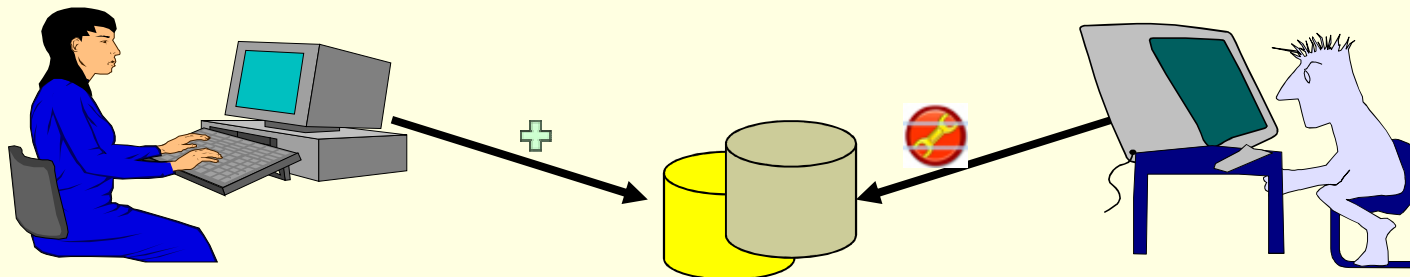
Concurrent Execution of Transactions

- Concurrency control allows users to use the database concurrently without damaging the transactions of other users.
- It supports and ensures the availability and correct operations of simultaneous multiple access in the database system.



Concurrent Execution of Transactions

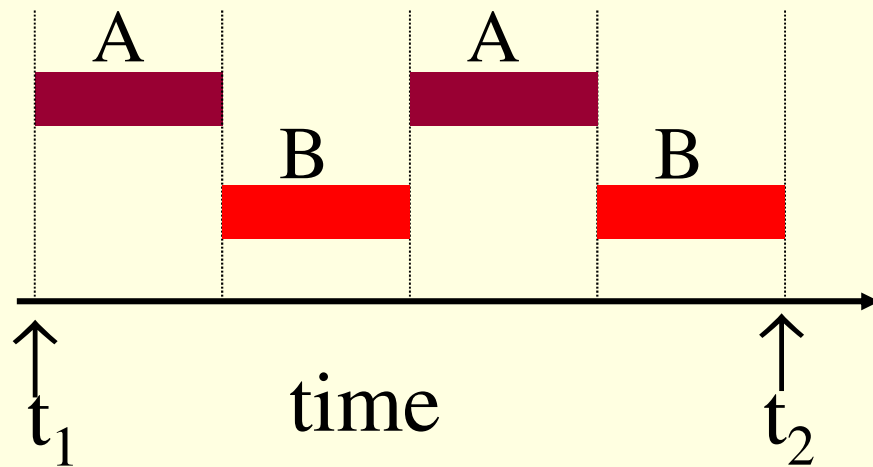
- **Single user** – at most one user at a time can use the system. Restricted to some PC DBMS.
- **Multi-user** – many users can use the system concurrently (at the same time). Most DBMS are multi-user.
 - Airline reservations systems / banks /
 - insurance agencies / stock exchanges



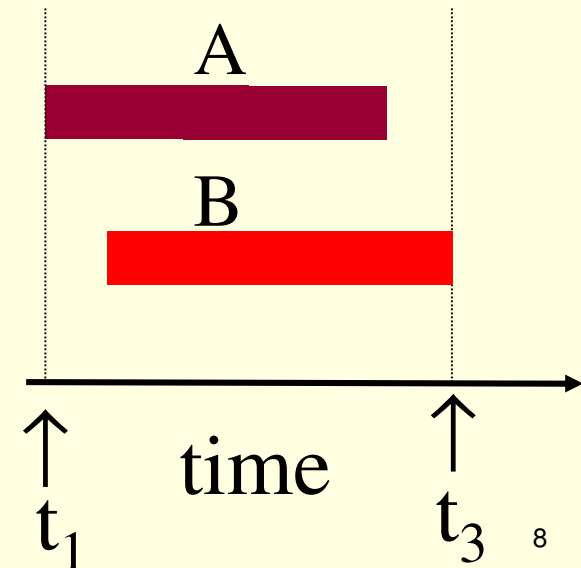
Interleaved model

- Concurrent execution of the program is actually **interleaved**.

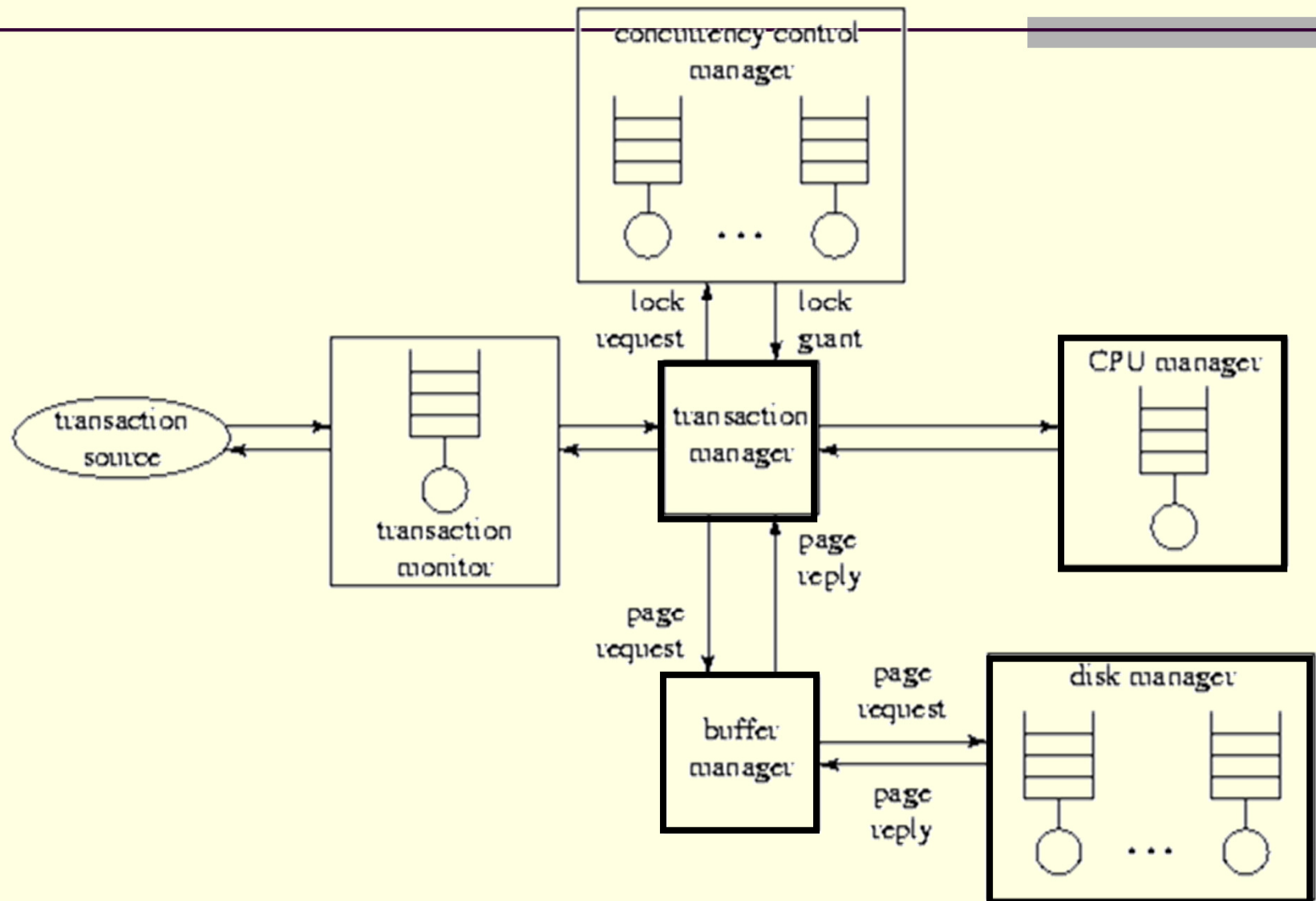
Single CPU



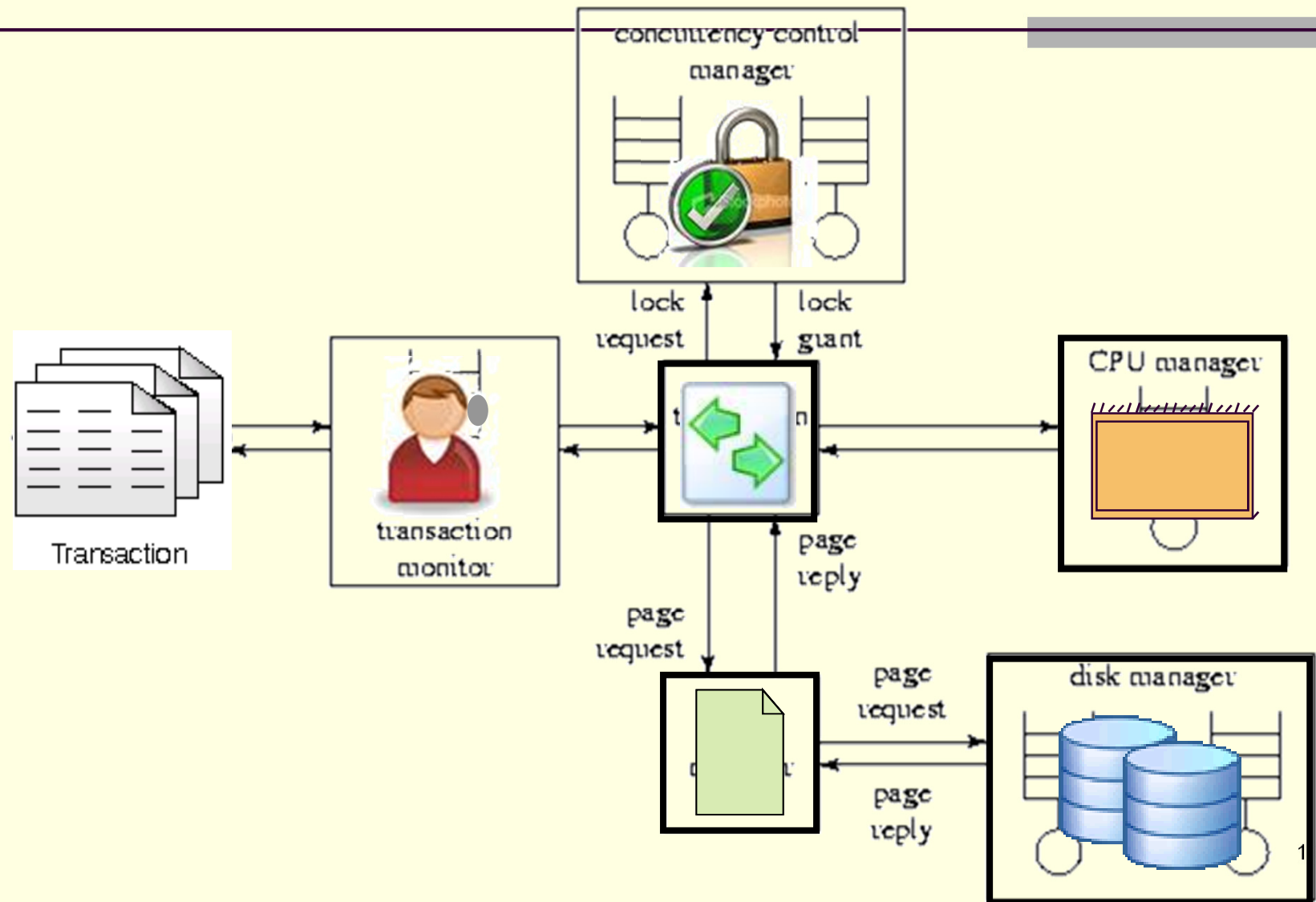
Multiple CPUs



Transaction Management



Transaction Management



Database Access Operations

The database access operations that a transaction can include are

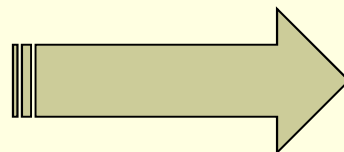
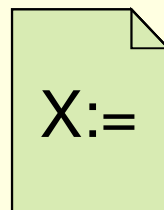
- **READ (X)** - reads database item **X** into a program variable **X**;
- Executing a **READ (X)**
 - Find the address of the disk block that contains item **X**.
 - Copy that disk block into a buffer in main memory (if not already in some main memory buffer).
 - Copy item **X** from the buffer to the program variable named **X**.

```
SELECT labmark INTO old_mark FROM enrol
WHERE studno = sno and courseno = cno
FOR UPDATE OF labmark;
```

Database Access Operations

```
UPDATE enrol SET labmark = new_mark  
WHERE studno = sno and courseno = cno;
```

- **WRITE (X)** - write the value of program variable **x** into the database item **x**.
- Executing a **WRITE (X)**
 - Find the address of the disk block that contains item **x**.
 - Copy that disk block into a buffer in main memory (if not already in some main memory buffer).
 - Copy item **x** from the program variable named **x** into its correct location in the buffer.
 - Store the updated block from the buffer back to disk (either immediately or at some later point of time)



Why Concurrency Control Needed?

E.g. **Transaction T1** - No of reservations for airline A is X;
No of reservation for airline B is Y; N reservation from A
is cancelled and booked for B.

Transaction T2 - M reservations to airline A.

T1

```
READ (X)
X = X - N
WRITE (X)
READ (Y)
Y = Y + N
WRITE (Y)
```

T2

```
READ (X)
X = X + M
WRITE (X)
```

Several problems can occur when
concurrent transactions execute in an
uncontrolled manner.

Concurrency Control Execution

$X = 80; Y = 100; N = 5; M = 4$

(i)

```
READ (X)
X = X - N
WRITE (X)
READ (Y)
Y = Y + N
WRITE (Y)
READ (X)
X = X + M
WRITE (X)
```

$X = ?$

$Y = ?$

(ii)

```
READ (X)
X = X + M
WRITE (X)
READ (X)
X = X - N
WRITE (X)
READ (Y)
Y = Y + N
WRITE (Y)
```

$X = ?$

$Y = ?$

(iii)

```
READ (X)
X = X - N
READ (X)
X = X + M
WRITE (X)
READ (Y)
WRITE (X)
Y = Y + N
WRITE (Y)
```

$X = ?$

$Y = ?$

(iv)

```
READ (X)
X = X - N
WRITE (X)
READ (X)
X = X + M
WRITE (X)
READ (Y)
Y = Y + N
WRITE (Y)
```

$X = ?$

$Y = ?$

1. The lost update problem

(Write – Write Conflict)

This occurs when two transactions that access the same database item have their operations interleaved in a way that makes the value of some database item incorrect. $X=80$; $Y=100$

| T1 | T2 | $N = 5, M = 4$ |
|-------------|-------------|----------------|
| READ (X) | | $X = 80$ |
| $X = X - N$ | | $X = 75$ |
| | READ (X) | $X = 80$ |
| | $X = X + M$ | $X = 84$ |
| WRITE (X) | | $X = 75$ |
| READ (Y) | | |
| | WRITE (X) | $X = 84$ |
| $Y = Y + N$ | | |
| WRITE (Y) | | $Y = 105$ |

T1: $X+Y = 84+105=189$

but X should be $80-5+4 = 79$

2. The temporary update (Dirty read) problem

(Write – Write Conflict)

This occurs when one transaction updates a database item and then the transaction fails for some reason.

T1

T2

N = 5, M = 4

| | |
|------------------------------------|------------------------------------|
| READ (X) X = X - N WRITE (X) | |
| | READ (X) X = X + M WRITE (X) |
| READ (Y) ROLLBACK | |

X = 80

X = 75

X = 75

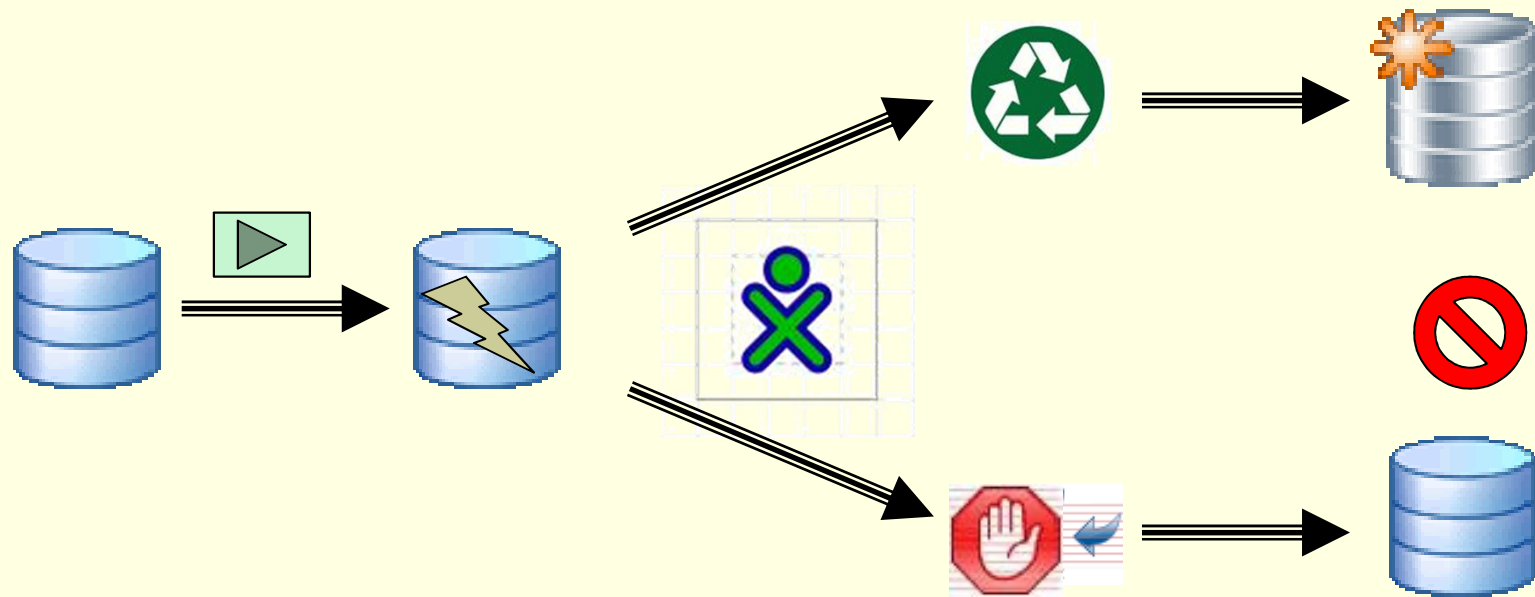
X = 75

X = 79

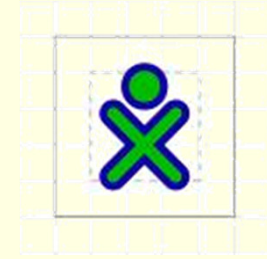
X = 79

- abort - changes X back to its original value gives X = 80
but should be 80+4 = 84

Transaction Processing Activities

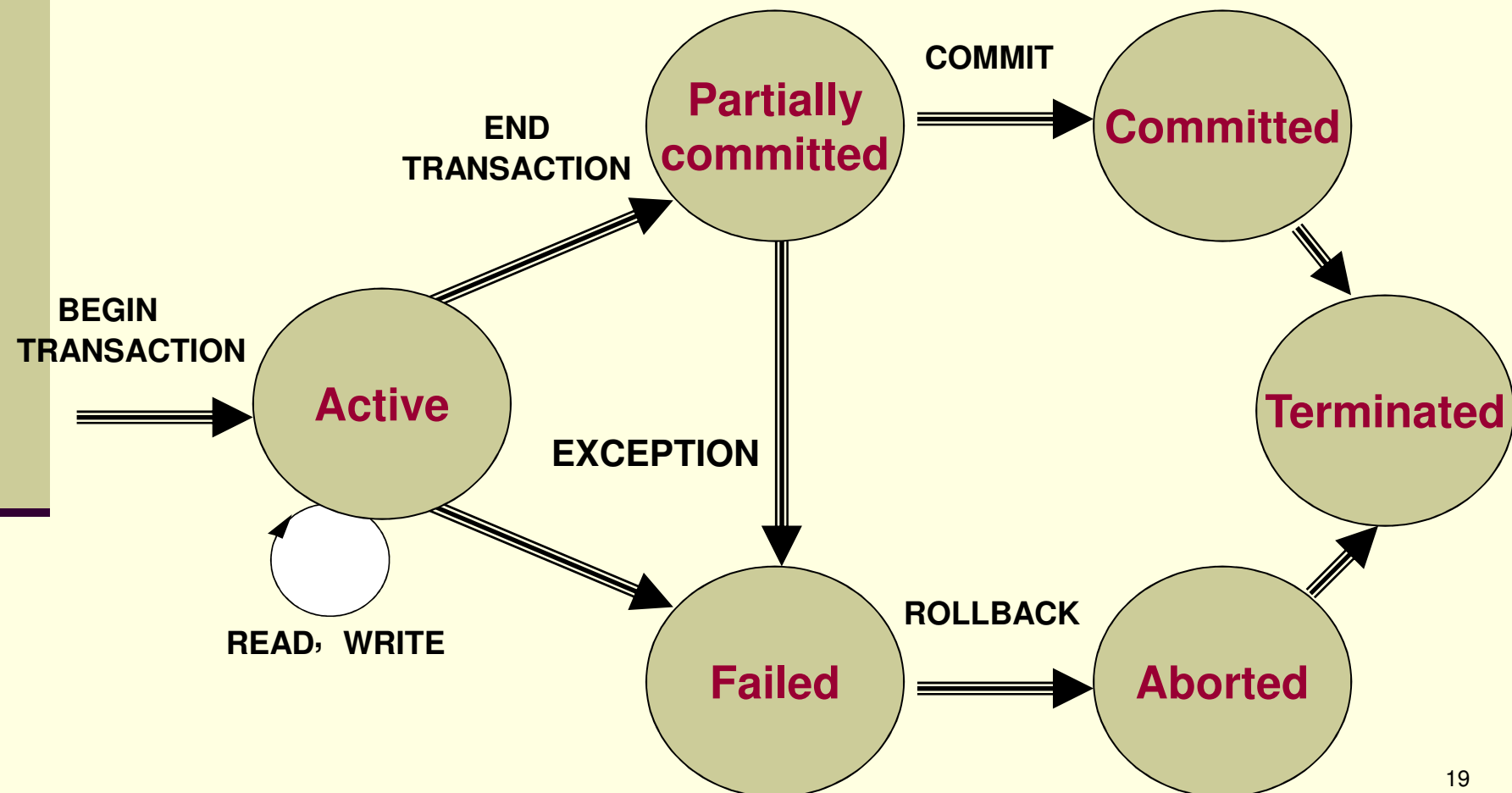


Transaction States



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

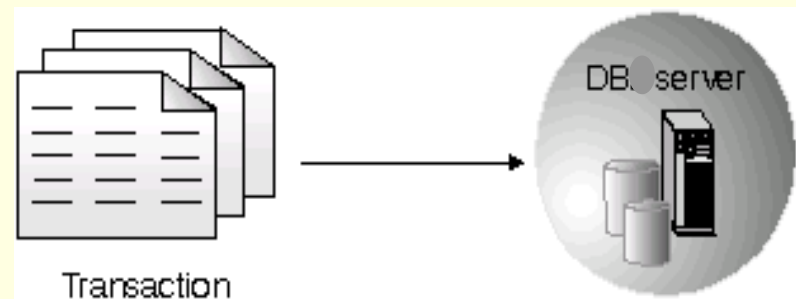
Transaction States




Transaction States


- For recovery purpose, the system needs to keep track of when the transaction starts, terminates and commits or aborts. The recovery manager keeps track of:

- ▶ – **BEGIN** marks the beginning of transaction execution
- ⊘ – **END** specifies that READ and WRITE transaction operations have ended and mark the end of transaction execution.



Transaction States

 **COMMIT** signals a successful end of the transaction so that any changes (updates) executed by the transaction can be safely committed to the database and will not be undone.

 **ROLLBACK (or ABORT)** signals that the transaction has ended unsuccessfully so that any changes or effects that the transaction may have applied to the database must be undone.

Properties of Transactions

Atomicity

A transaction is an atomic unit of processing. It is either performed in its entirety or not performed at all

Consistency preservation

A correct execution of the transaction must take the database from one consistent state to another

Isolation

A transaction should not make its updates visible to other transactions until it is committed

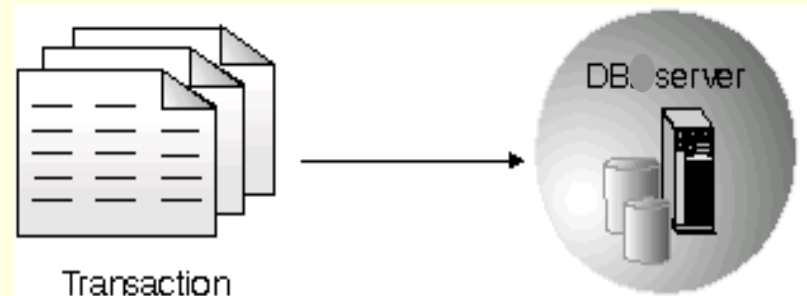
Durability (Permanency)

Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failures

Transaction Atomicity

- Is a unit of program execution that accesses and possibly updates various data items.
- It is an atomic unit of work that is either completed in its entirety or not done at all.

```
BEGIN  
    READ (A) ;  
    A = A - 50 ;  
    WRITE (A) ;  
    READ (B) ;  
    B = B + 50 ;  
    WRITE (B) ;  
END;
```



Transaction Atomicity

BEGIN

READ

WRITE

COMMIT

ROLLBACK

END

Credit_labmark (sno NUMBER, cno CHAR, credit NUMBER)

old_mark NUMBER;

new_mark NUMBER;

SELECT labmark INTO old_mark FROM enrol

WHERE studno = sno and courseno = cno FOR UPDATE OF labmark;

new_mark := old_mark + credit;

UPDATE enrol SET labmark = new_mark WHERE studno = sno and
courseno = cno ;

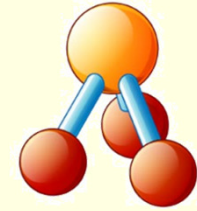
COMMIT;

EXCEPTION

WHEN OTHERS THEN ROLLBACK;

END credit_labmark;

Transaction Properties ...



e.g. Transfer 50 from
account A
(**A=1000**) to B (**B=2000**)

```
T1: BEGIN
    READ (A) ;
    A = A - 50;
    WRITE (A) ;
    READ (B) ;
    B = B + 50;
    WRITE (B) ;
END;
```

A=950; B=2050;

Consistency - take the database from one consistent state to another.

Value of **A+B** (**3000**) should be same before transaction and after transaction

Atomicity - either performed in its entirety or not performed at all.

Transaction failure after **WRITE (A)**, but before **WRITE (B)**, then **A=950; B=2000;** i.e. 50 is lost

Data is now inconsistent as **A+B** is now **2950**

Transaction Properties ...



e.g. Transfer 50 from
account A
(**A=1000**) to B (**B=2000**)

```
T1: BEGIN
      READ (A) ;
      A = A - 50;
      WRITE (A) ;
      READ (B) ;
      B = B + 50;
      WRITE (B) ;
END;
```

A=950; B=2050;

Durability - changes must never be lost because of subsequent failures (e.g. power failure)

Recover database: remove changes of a partially done transaction (**A=1000; B=2000**); reconstruct completed transactions (**A=950; B=2050**)

Isolation - updates not visible to other transactions until committed

Between **WRITE (A)** and **WRITE (B)** if second transaction reads **A** and **B** it sees inconsistent data as **A+B = 2950**

Serializability of Schedules

Schedule (History) - A schedule S of n transactions T1, T2, ..., Tn is an order of the operations of the transactions subject to the constraint that operation of Ti in S must appear in the same order in which they occur in Ti.



T1

T11 **READ(X)**
T12 **WRITE(X)**
T13 **READ(Y)**
T14 **WRITE(Y)**

T2

T21 **READ(X)**
T22 **WRITE(X)**
T23 **READ(Y)**
T24 **WRITE(Y)**

S1

T11 **READ(X)**
T12 **WRITE(X)**
T21 **READ(X)**
T22 **WRITE(X)**
T13 **READ(Y)**
T23 **READ(Y)**
T24 **WRITE(Y)**
T14 **WRITE(Y)**

S2


T11 **READ(X)**
T21 **READ(X)**
T12 **WRITE(X)**
T22 **WRITE(X)**
T13 **READ(Y)**
T23 **READ(Y)**
T14 **WRITE(Y)**
T24 **WRITE(Y)**

S1 & S2 are Schedules

Serializability of Schedules

Serial Schedules - For every transaction T participating in the schedule, all the operations of T are executed consecutively in the schedule. Otherwise the schedule is called non-serial. Serial schedules are always correct.

T1



T11 **READ(X)**
T12 **WRITE(X)**
T13 **READ(Y)**
T14 **WRITE(Y)**

T2

T21 **READ(X)**
T22 **WRITE(X)**
T23 **READ(Y)**
T24 **WRITE(Y)**

S3

T11 **READ(X)**
T12 **WRITE(X)**
T13 **READ(Y)**
T14 **WRITE(Y)**
T21 **READ(X)**
T22 **WRITE(X)**
T23 **READ(Y)**
T24 **WRITE(Y)**

S4

T21 **READ(X)**
T22 **WRITE(X)**
T23 **READ(Y)**
T24 **WRITE(Y)**
T11 **READ(X)**
T12 **WRITE(X)**
T13 **READ(Y)**
T14 **WRITE(Y)**

S3 & S4 Serial Schedules – Always correct!

S1 & S2 Non-Serial Schedules – When are they correct?

Question

Suppose client C1 issues transactions **T1;T2** and client C2 concurrently issues transactions T3;T4.

How many different "equivalent sequential orders" are there for these four transactions?

- a) 2
- b) 4
- c) 6
- d) 16
- e) 24

1 T1 T2 T3 T4

2 T1 T3 T2 T4

3 T1 T3 T4 T2

4 T3 T1 T2 T4

5 T3 T1 T4 T2

6 T3 T4 T1 T2

Dirty Reads

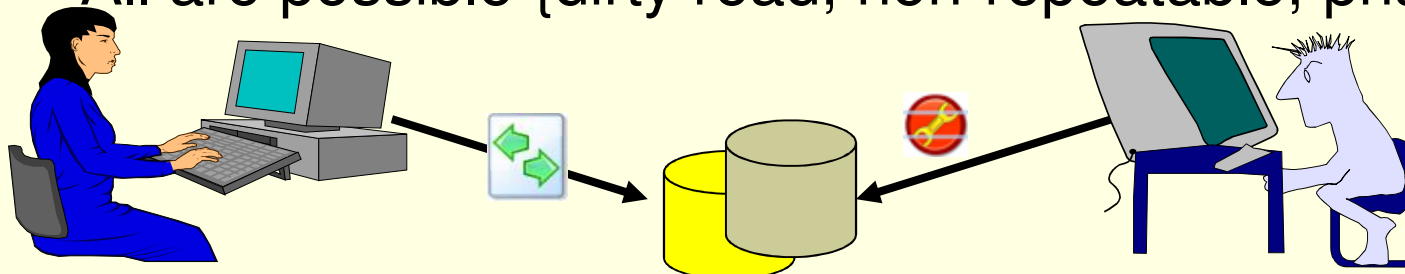


- Database server process reads from the database table without checking for locks (let this process look at dirty data). This can be useful when the table is static; 100% accuracy is not as important as speed and freedom from contention; you cannot wait for locks to be released.

SQL Syntax:

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
```

All are possible {dirty read, non-repeatable, phantom}



Consider a table R(A) containing {(1), (2)}, and two transactions T1 and T2:

T1: **update** R **set** $A = 2 * A$

T2: **select** avg (A) **from** R

If transaction T2 executes using *Read Uncommitted*, what are the possible values it returns?

- a) 1.5
- b) 1.5, 3
- c) 1.5, 2, 3
- d) 1.5, 2.5, 3
- e) 1.5, 2, 2.5, 3

Serial Schedule

T1 T2 3
T2 T1 1.5

T2 1.5

T1 {(2),(4)} T2 3

T1 {(2),(2)} T2 2

T1 {(1),(4)} T2 2.5

3. The incorrect summary problem

(Read– Write Conflict)

If one transaction is calculating an aggregate summary function on a number of records while other transactions are updating some of these records, the aggregate function may calculate some values before they are updated and

X=80; Y=100

T1

T2

READ (X)
X = X - N
WRITE (X)

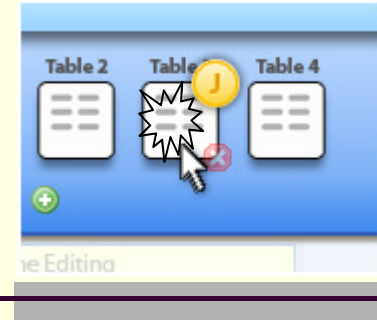
READ (Y)
Y = Y + N
WRITE (Y)

sum = 0

READ (X)
sum = sum + X
READ (Y)
sum = sum + Y

sum=X+Y=75+100=175

Committed Reads

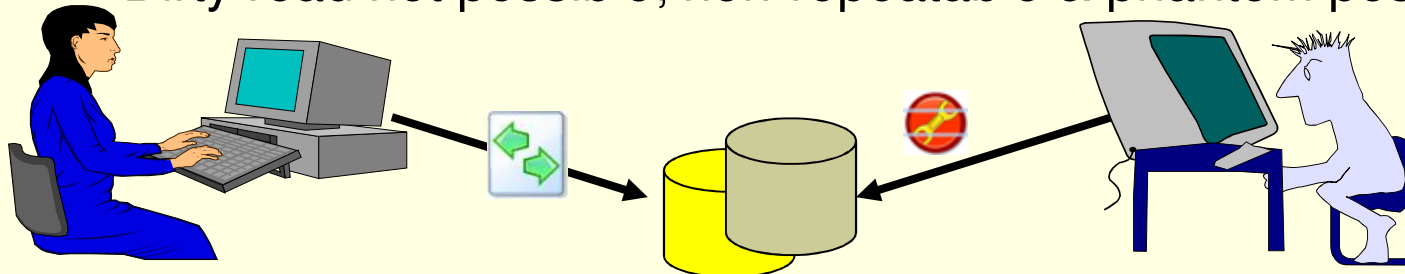


- In this isolation level, read locks are acquired on selected data but they are released immediately whereas write locks are released at the end of the transaction. Data records retrieved by a query are not prevented from modification by some other transaction.
- This can be useful for lookups; queries; reports yielding general information (e.g. month-ending sales analyses).

SQL Syntax:

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

Dirty read not possible; non-repeatable & phantom possible



Consider tables R(A) and S(B), both containing {(1), (2)}, and two concurrent transactions T1 and T2:

T1: **update** R **set** A = 2*A; **update** S **set** B = 2*B **commit**

T2: **select** avg(A) **from** R; **select** avg(B) **from** S

If transaction T2 executes using **Read Committed**, is it possible for T2 to return two different values?

a) Yes

b) No

T2 1.5 1.5

T1 T2 3 3

X T1 {(2),(4)} {(1),(2)} **T2** 3 1.5

Consider tables R(A) and S(B), both containing {(1), (2)}, and two transactions:

T1: **update** R **set** A = 2*A; **update** S **set** B = 2*B

T2: **select** avg(A) **from** R; **select** avg(B) **from** S

If transaction T2 executes using **Read Committed**, is it possible for T2 to return a smaller avg(B) than avg(A)?

a) Yes

b) No

T2 1.5 1.5

T1 T2 3 3

X T1 {(2),(4)} {(1),(2)} **T2** 3 1.5

4. Unrepeatable Read problem

(Read– Write Conflict)

Another problem that may occur is the **unrepeatable read** where a transaction T2 reads an item twice (i.e. X) and the item is changed by another transaction (i.e. T1) between the two reads.

T1

T2

| | |
|------------------------------------|------------------------------|
| | READ (X) |
| READ (X) X = X - N WRITE (X) | |
| | READ (X) |

X=80

X=75

Repeatable Reads

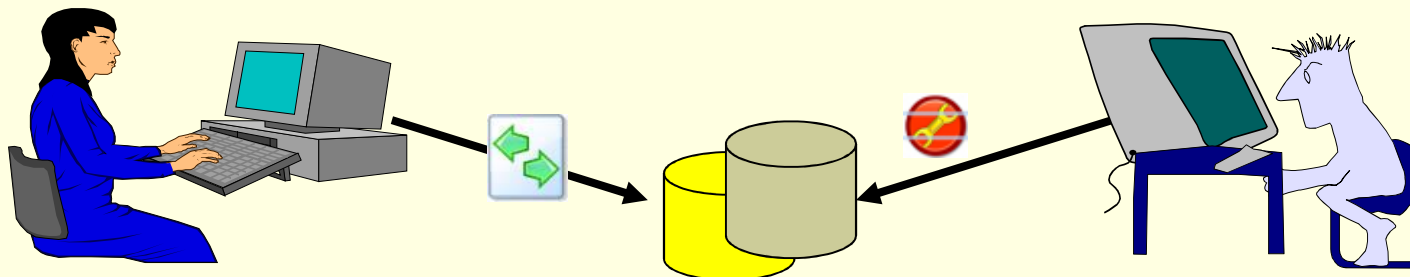


- Database server process puts locks on all rows examined to satisfy the query (do not let other processes change any of the rows I have looked at until I am done). It can be used for critical, aggregate arithmetic (e.g. account balancing); coordinated lookups from several tables (e.g. reservation systems).

SQL Syntax:

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

Dirty read and non-repeatable not possible; phantom possible



Consider a relation $R(A)$ containing $\{(5), (6)\}$ and two transactions:

— T1: **update** R **set** $A = A + 1$

T2: **update** R **set** $A = 2 * A$

Suppose both transactions are submitted under the isolation and atomicity properties. Which of the following is NOT a possible final state of R?

a) $\{(6), (7)\}$

T1 c T2 r

b) $\{(10), (12)\}$

T1 r T2 c

c) $\{(12), (14)\}$

T1 c T2 c

d) $\{(11), (13)\}$

T2 c T1 c

e) $\{(11), (12)\}$

X

Consider table R(A) containing {(1), (2)}, and two concurrent transactions T1 & T2:

T1: **update** R **set** $A=2*A$; **insert into** R **values** (6)

T2: **select** avg(A) **from** R; **select** avg(A) **from** R

If transaction T2 executes using **Repeatable Read**, what are the possible values returned by its SECOND statement?

- a) 1.5, 3
- b) 1.5, 4
- c) 1.5, 2, 4
- d) 1.5, 3, 4
- e) 1.5, 2, 3, 4

T2 1.5

T1 {(2)(4)(6)} **T2** 3

T1 {(2)(4)(6)} **T2** 4

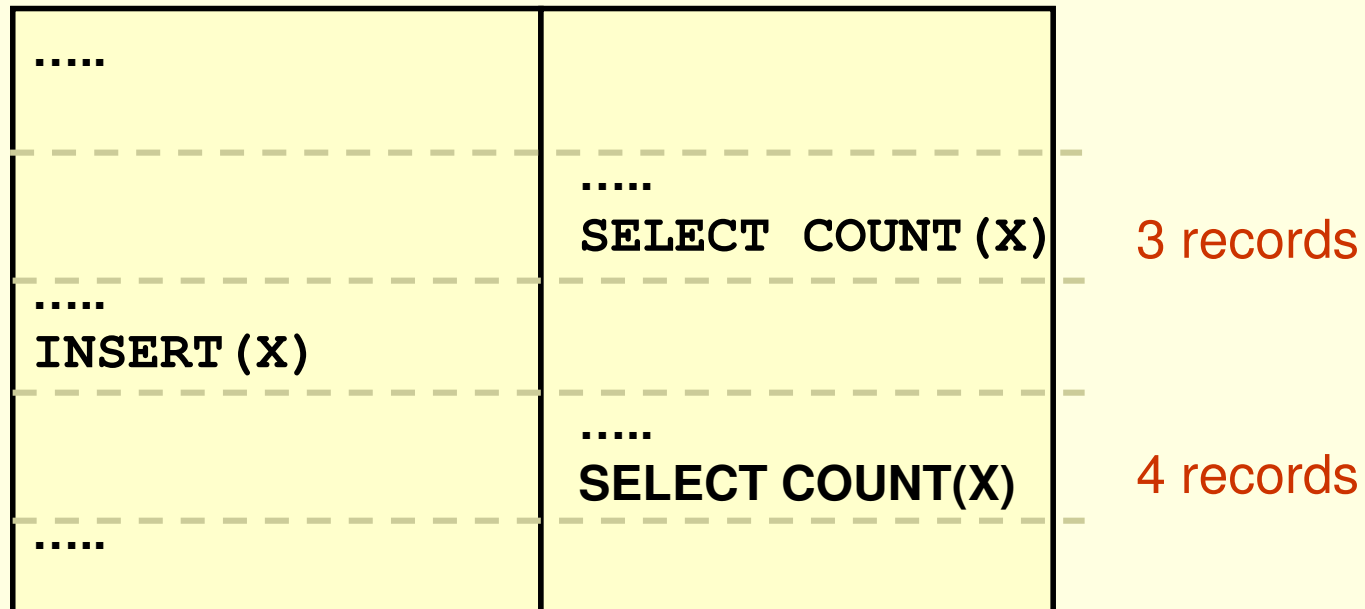
Phantom Phenomenon

(Read– Write Conflict)

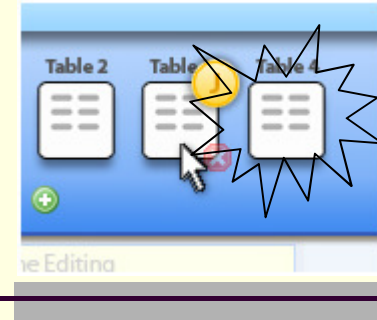
Set of rows that is read once might be different due to insert of new record.

T1

T2



Serializable

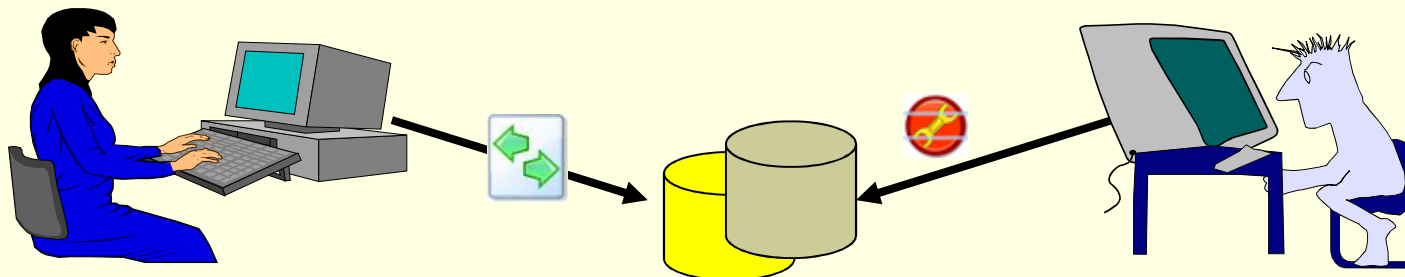


- Always guarantee correct execution of transaction.

SQL Syntax:

SET ISOLATION TO SERIALZABLE;

All are not possible {Dirty read, non-repeatable, phantom}



SQL Syntax

- SET TRANSACTION ISOLATION LEVEL {
 REPEATABLE READ |
 READ COMMITTED |
 READ UNCOMMITTED |
 SERIALIZABLE }

Read Uncommitted / Dirty Read

- To maintain the highest level of isolation, a DBMS usually acquires locks on data, which may result in a loss of concurrency and a high locking overhead. This isolation level relaxes this property.

Last Week Summary

- Transaction
 - Properties
 - States
 - Isolation Levels
 - Read Uncommitted
 - Read Committed
 - Repeatable Read
 - Serializable

Serializability of Schedules


Serializable - If two disjoint groups of the non-serial schedules are equivalent to one of the serial schedules. Otherwise non-serializable.

- Serializable means that concurrency has added no effect
- Unserializable schedules introduce inconsistencies (e.g. Dirty Read, Non-repeatable read)

Serializability of Schedules

Serial Schedules - For every transaction T participating in the schedule, all the operations of T are executed consecutively in the schedule. Otherwise the schedule is called non-serial. Serial schedules are always correct.

T1



T11 READ(X)
T12 WRITE(X)
T13 READ(Y)
T14 WRITE(Y)

T2

T21 READ(X)
T22 WRITE(X)
T23 READ(Y)
T24 WRITE(Y)

S3

T11 READ(X)
T12 WRITE(X)
T13 READ(Y)
T14 WRITE(Y)
T21 READ(X)
T22 WRITE(X)
T23 READ(Y)
T24 WRITE(Y)

S4

T21 READ(X)
T22 WRITE(X)
T23 READ(Y)
T24 WRITE(Y)
T11 READ(X)
T12 WRITE(X)
T13 READ(Y)
T14 WRITE(Y)

S3 & S4 Serial Schedules – Always correct!

S1 & S2 Non-Serial Schedules – When are they correct?

Serializability of Schedules

Serializable - If two disjoint groups of the non-serial schedules are equivalent to one of the serial schedules. Otherwise non-serializable.

T1

T11 **READ(X)**
T12 **WRITE(X)**
T13 **READ(Y)**
T14 **WRITE(Y)**

T2

T21 **READ(X)**
T22 **WRITE(X)**
T23 **READ(Y)**
T24 **WRITE(Y)**

S3

T11 **READ(X)**
T12 **WRITE(X)**
T13 **READ(Y)**
T14 **WRITE(Y)**
T21 **READ(X)**
T22 **WRITE(X)**
T23 **READ(Y)**
T24 **WRITE(Y)**

S5

T21 **READ(X)**
T22 **WRITE(X)**
T11 **READ(X)**
T12 **WRITE(X)**
T23 **READ(Y)**
T24 **WRITE(Y)**
T13 **READ(Y)**
T14 **WRITE(Y)**

Results of S5 is equal to that of S3

Thus S5 is correct & is Serializable

How to identify which non-serial schedules are correct?

Serializability

- Two major types of serializability exist
 - *View-serializability*
 - *Conflict-serializability.*
- View-serializability matches the general definition of serializability given above.
- Conflict-serializability is a broad special case, i.e., any schedule that is conflict-serializable is also view-serializable, but not necessarily the opposite.

Venn Diagram



All schedules

View Serializable

Conflict
Serializable

Conflict-serializability

- Conflict-serializability is defined by equivalence to a serial schedule (no overlapping transactions) with the same transactions, such that both schedules have the same sets of respective chronologically ordered pairs of conflicting operations (same precedence relations of respective conflicting operations).

Check for Conflict Serializable?

- Check for the conflicting actions.
- Check for a cycle in the Precedence Graph.

Check for Conflicting Actions

Two or more actions are said to be in conflict if:

1. The actions belong to different transactions.
2. At least one of the actions is a write operation.
3. The actions access the same object (read or write).

■ The following set of actions is conflicting:

T1:R(X), T2:W(X), T3:W(X)

■ While the following two sets of actions are not:

T1:R(X), T2:R(X), T3:R(X)

T1:R(X), T2:W(Y), T3:R(X)

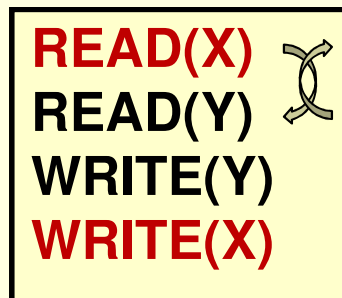
Conflict serialisability

A Schedule (S) whose consecutive instructions (T1i & T2j) can be swapped (T1j & T2i) without affecting the result of the schedule.

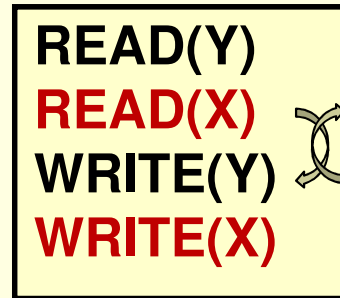
If T1i & T2j refers to different data items (X & Y) then swapping the instructions will not affect the results.



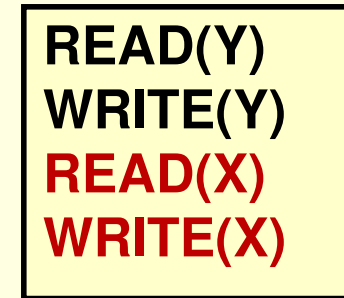
S



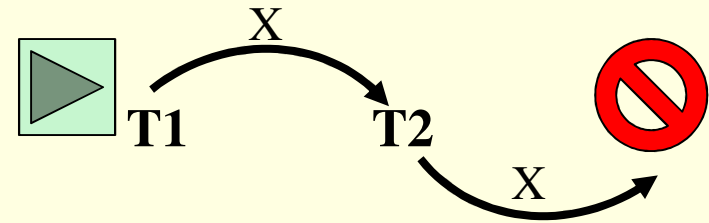
S'



S''



Conflict equivalence

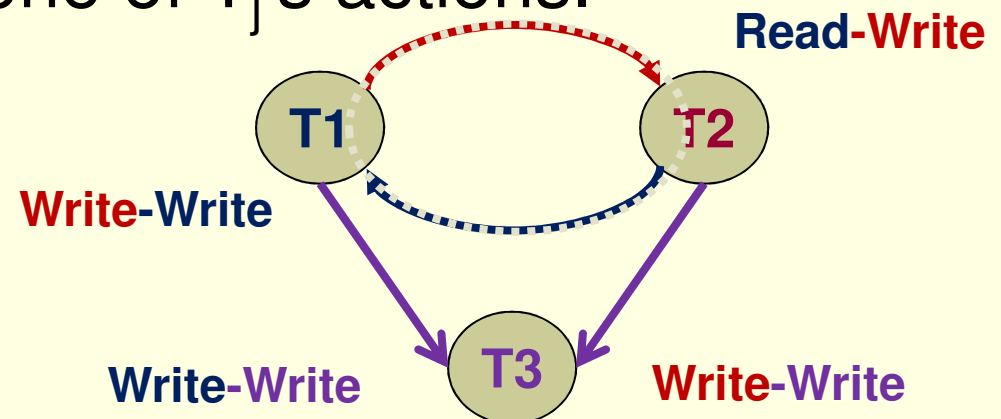
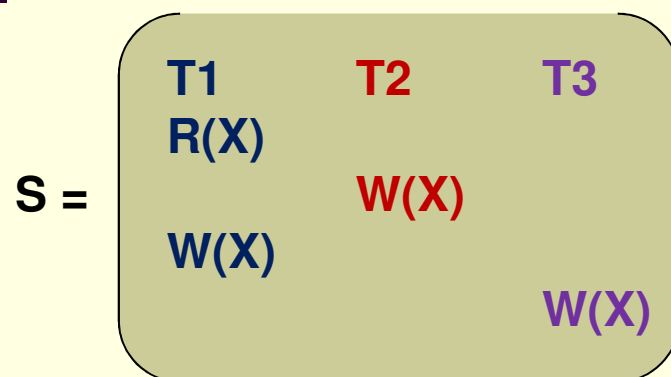


- A simple and efficient method for determining conflict serializability of a schedule is construction of a directed graph called a **precedence / conflict / serializability graph**.
- Graph consists of a pair $G=(V,E)$ where V is a set of vertices (each transaction is a vertex) and E is a set of edges (arc joining vertices). E consists of all the transactions participating in the schedule such that for all E , $T1 \rightarrow T2$ (arc from $T1$ to $T2$) holds such that
 - $T1$ execute $WRITE(X)$ before $T2$ executes $READ(X)$
 - $T1$ execute $READ(X)$ before $T2$ executes $WRITE(X)$
 - $T1$ execute $WRITE(X)$ before $T2$ executes $WRITE(X)$

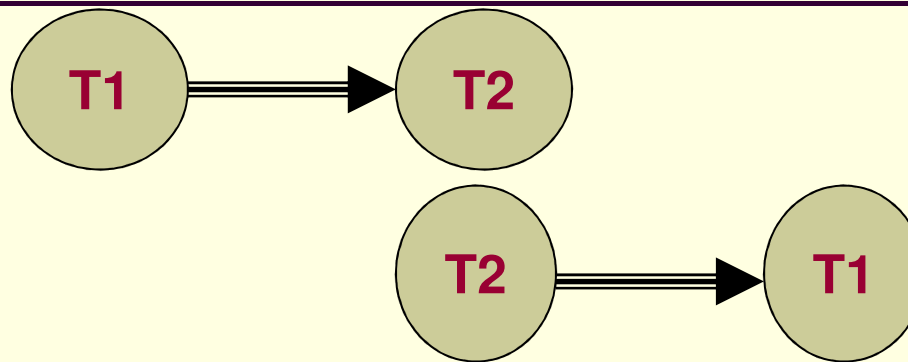
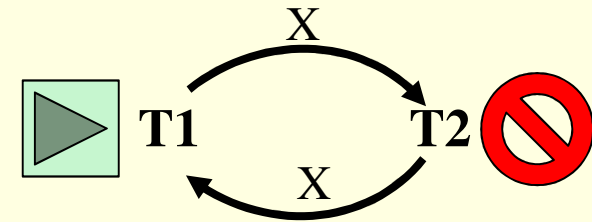
Precedence graph

(serializability graph, serialization graph, conflict graph)

- A node for each committed transaction in S
- For each transaction T_i participating in schedule S, create a node labelled T_i in the precedence graph. So the precedence graph contains T_1, T_2, T_3
- An arc from T_i to T_j if an action of T_i precedes and conflicts with one of T_j 's actions.

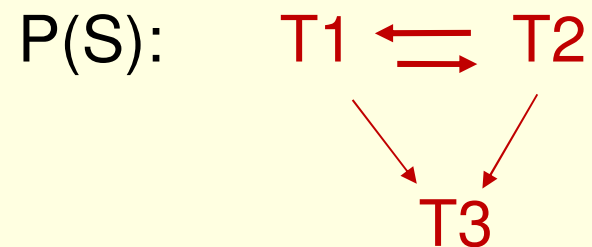


Precedence graphs

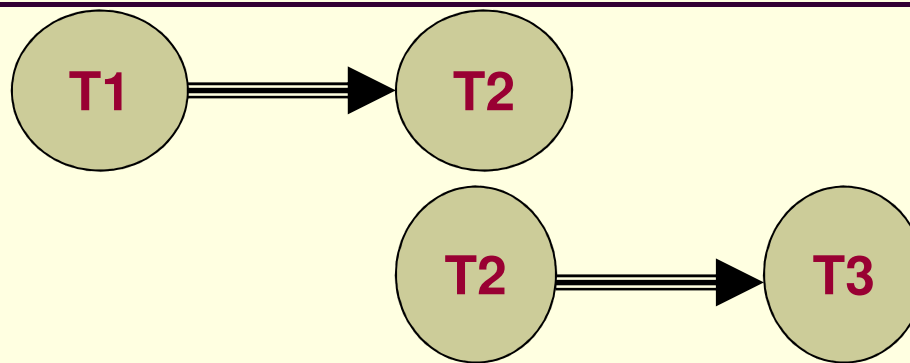


If the precedence graph has a cycle, then the schedule is **Not conflict serializable**. Incorrect schedule

E.g. $S = r1(X) \ w2(X) \ w1(X) \ w3(X)$

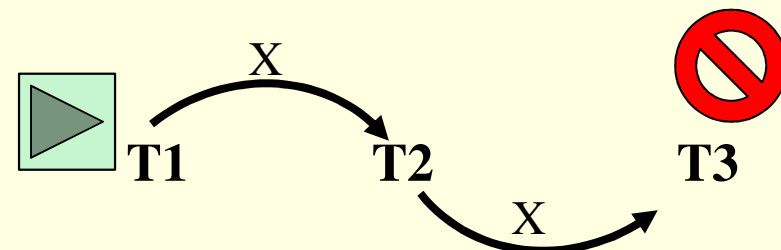
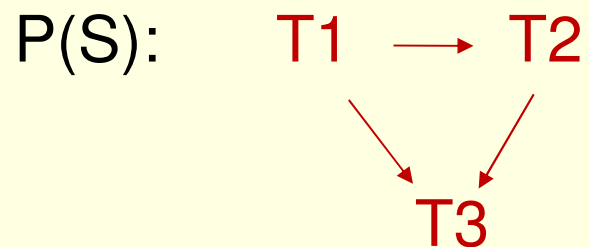


Precedence graphs



If the precedence graph hasn't a cycle, then the schedule is conflict serializable. Correct schedule

E.g. $S = r1(X) \ w2(X) \ w3(X)$



-
- Consider the following transactions:

T1: $r_1(X)$, $w_1(X)$, $r_1(Y)$, $w_1(Y)$

T2: $r_2(X)$, $w_2(X)$, $r_2(Y)$, $w_2(Y)$

How many schedules are there that are conflict equivalent to $(T1, T2)$?

■ 6

- $r1(X), w1(X), r1(Y), w1(Y), r2(X), w2(X), r2(Y), w2(Y)$
- $r1(X), w1(X), r1(Y), r2(X), w1(Y), w2(X), r2(Y), w2(Y)$
- $r1(X), w1(X), r1(Y), r2(X), w2(X), w1(Y), r2(Y), w2(Y)$
- $r1(X), w1(X), r2(X), r1(Y), w1(Y), w2(X), r2(Y), w2(Y)$
- $r1(X), w1(X), r2(X), r1(Y), w2(X), w1(Y), r2(Y), w2(Y)$
- $r1(X), w1(X), r2(X), w2(X), r1(Y), w1(Y), r2(Y), w2(Y)$

-
- Consider the following ordering S of transactions:

T1: R(X), W(Y); T2: R(X), W(Y); T3: R(X), W(Y).

How many schedules, if any, are conflict-equivalent to S?

-
- There is only 1 possible location for T3's WRITE. It must go last. This means that T3's READ has a choice of 5 possible positions: ____ ____ ____ ____ ____ T3: W(Y)
 - After we place T3's READ, this only leaves 4 empty spots. We have only no choice as to where to put T2's WRITE. It MUST go in the last empty space of whichever spaces are currently left. That leaves 3 choices for placing T2's READ.
 - Now, there are only 2 spots left. We have no choices. T1's WRITE MUST go in the last empty spot, and T1's READ goes in the other empty spot.
 - This means that there are a total of $5 \times 3 = 15$ possible conflict-equivalent schedules.

- Consider the following ordering S of transactions:

T2: R(X); T3: W(X); T3: commit; T1: W(Y); T1: commit; T2: R(Y); T2: W(Z); T2: commit; T4: R(X); T4: R(Y); T4: commit.

- Draw a precedence graph for S.
- Is S conflict-serializable according to the precedence graph?

- T1 -> T2 -> T3 -> T4



- S is conflict-serializable because there are no cycles in the graph. And it is also serializable. The serial ordering can be seen from the precedence graph directly.

Consider the following schedule:

S: $r_1(X)$, $w_2(X)$, $r_3(X)$, $r_1(Y)$, $w_2(Z)$, $r_3(Y)$, $w_3(Z)$,
 $w_1(Y)$

- (a) Draw the precedence graph.
- (b) Is S conflict serializable?

■ $T1 \rightarrow T2 \rightarrow T3$



■ Has a cycle, hence, No.