Repeatable Reads

- Database server process puts locks on all rows examined to satisfy the query (do not let other processes change any of the locked rows until the transaction is completed).
- 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;**

Question 2

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); commit;

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

Serializable

- Always guarantee correct execution of transaction.

SQL Syntax

- **SET ISOLATION TO SERIALIZABLE;**

Transaction Isolation Levels

Transaction isolation level	Dirty reads	Nonrepeatable reads	Phantoms
Read uncommitted	X	X	X
Read committed	--	X	X
Repeatable read	--	--	X
Serializable	--	--	--

Need For Recovery

- There are several possible reasons for a transaction to fail in the middle of execution:

- **Transaction**
  - a transaction fails after executing some of its operations but before executing all of them.
  - caused by errors within the transaction processes.

Need For Recovery

- **System (soft crash)**
  - The volatile storage is destroyed (e.g. power failure). This affects all transactions currently in progress but do not cause damage to the database.
  - Caused by failure of network or operating system or physical threats to the system as a whole.

- **Media failure** (physical failures)
  - Failure of hard disk, out of memory errors, out of disk space errors.

Need For Recovery

- Recovery allows a database system to recover from physical or software failures when they occur in the system.
- Through recovery the system makes sure that either
  i) all the operations in the transaction are completed successfully and their effect is recorded permanently in the database, or
  ii) the transaction has no effect whatsoever on the database or on any other transactions.

The Log File

- Holds the information that is necessary for the recovery process.
- Records all relevant operations in the order in which they occur.
- Is an append-only file. Kept on disk.
- The notion of recovery from a transaction failure amounts to either undoing or redoing transaction operations individually from the log.

Log Records

- [start_transaction,T]: Records that transaction T has started execution.
- [write_item,T,X,old_value,new_value]: T has changed the value of item X from old_value to new_value.
- [read_item,T,X]: T has read the value of item X (not needed in many cases).
- [end_transaction,T]: T has ended execution
- [commit,T]: T has completed successfully, and committed.
- [abort,T]: T has been aborted.

Log Files

Transaction Log

- For recovery from any type of failure data values prior to modification (BFIM - BeFore Image) and the new value after modification (AFIM – AFter Image) are required.
- These values and other information is stored in a sequential file called Transaction log. A sample log is given below. Back P and Next P point to the previous and next log records of the same transaction.

T ID	Back P	Next P	Operation	Data item	BFIM	AFIM
T1	0	1	Begin			
T1	1	4	Write	X	X = 100	X = 200
T2	0	8	Begin			
T1	2	5	W	Y	Y = 50	Y = 100
T1	4	7	R	M	M = 200	M = 200
T3	0	9	R	N	N = 400	N = 400
T1	5	nil	End			

Log Files

- **Undo**
  - If a system failure occurs, the log is checked for all transactions T that have written a [start_transaction,T] record into the log but have not written their [commit,T] record yet; these transactions may have to be *rolled back* to undo their effect on the database during the recovery process.
- **Redo**
  - Transactions that have written their commit record in the log must also have recorded all their WRITE operations in the log, so their effect on the database can be *redone* from the log records.

Schedule

- A schedule is a collection of many transactions which is implemented as a unit.
- When transactions are executing concurrently in an interleaved fashion, then the order of execution of operations from all the various transactions is known as a schedule.
- Depending upon how these transactions are arranged in within a schedule, a schedule can be of two types:
  - Serial: The transactions are executed one after another.
  - Concurrent: The transactions are executed in time shared method.

Conflicts

Two operations in a schedule are said to conflict if they satisfy all three of the following conditions:

- they belong to different transactions;
- they access the same item X; and
- at least one of the operations is a write_item(X).

Serial Schedule

- No question of sharing a single data item among many transactions, because not more than a single transaction is executing at any point of time.
- Inefficient
  - Transactions suffer for having a longer waiting time
  - High response time
  - Low amount of resource utilization.

Example

T1
Read A;
A = A – 100;
Write A;
Read B;
B = B + 100;
Write B;

T2
Read A;
Temp = A * 0.1;
Read C;
C = C + Temp;
Write C;

T2
Read A;
Temp = A * 0.1;
Read C;
C = C + Temp;
Write C;

T1
Read A;
A = A – 100;
Write A;
Read B;
B = B + 100;
Write B;

Concurrent Schedules

- CPU time is shared among two or more transactions in order to run them concurrently.
- More than one transaction may need to access a single data item for read/write purpose.
- The database could contain inconsistent value if such accesses are not handled properly.

Example

T1

Read A;

A = A – 100;

Write A;

Read B;

B = B + 100;

Write B;

T2

Read A;

Temp = A * 0.1;

Read C;

C = C + Temp;

Write C;

Example

T1

Read A;

A = A – 100;

Write A;

Read B;

B = B + 100;

Write B;

T2

Read A;

Temp = A * 0.1;

Read C;

C = C + Temp;

Write C;

Context switching has been done appropriately.

Example

T1	T2
Read A;	
A = A – 100;	
	Read A;
	Temp = A * 0.1;
	Read C;
	C = C + Temp;
	Write C;
Write A;	
Read B;	
B = B + 100;	
Write B;	

Context switching has not done appropriately. C makes an unjust gain.

Serializability

- A well-formed rules regarding how to arrange instructions of the transactions to create error free concurrent schedules is required.

Serializability

- When several concurrent transactions are trying to access the same data item, the instructions within these concurrent transactions must be ordered in some way so as there are no problem in accessing and releasing the shared data item.
- A schedule S of n transactions is serializable if it is equivalent to some serial schedule of the same n transactions.
- Two types
  - Conflict Serializability
  - View Serializability

Conflict Equivalent

The schedules S1 and S2 are said to be conflict-equivalent if the following conditions are satisfied:

- Both schedules S1 and S2 involve the same set of transactions (including ordering of actions within each transaction).
- Both schedules have same set of conflicting operations.
- The order of any two conflicting operations is the same in both schedules.

Conflict Serializability

- A schedule S to be conflict serializable if it is conflict equivalent to some serial schedule S'.
- In such a case, the non-conflicting operations in S can be reordered until an equivalent serial schedule S‘ is formed.

Conflict Serializability

- A conflict arises if at least one (or both) of the instructions is a write operation.
- The following rules are important in Conflict Serializability:
  i) If two instructions of the two concurrent transactions are both for read operation, then they are not in conflict, and can be allowed to take place in any order.
  ii) If one of the instructions wants to perform a read operation and the other instruction wants to perform a write operation, then they are in conflict, hence their ordering is important.
    - Read;Write
    - Write;Read

Conflict Serializability

iii) If both the transactions are for write operation, then they are in conflict. The value that persists in the data item after the schedule is the one written by the instruction that performed the last write.

Conflicting Actions

- Two operations in a schedule are said to conflict if they satisfy all three of the following conditions:
  (1) they belong to different transactions;
  (2) they access the same item X; and
  (3) at least one of the operations is a write_item(X).
- The following set of actions is conflicting:
  T1:R(X), T2:W(X), T3:W(X)
- While the following sets of actions are not:
  T1:R(X), T2:R(X), T3:R(X)
  T1:R(X), T2:W(Y), T3:R(X)

Precedence Graph

- A simple and efficient method for determining conflict serializability of a schedule is construction of a directed graph called a **precedence/conflict/ serializability graph**.
- A precedence graph is a directed graph G = (N, E) that consists of a set of nodes N= $\{T_1,T_2,...,T_n\}$ and a set of directed edges E= $\{e_1, e_2,...,e_m\}$.
- There is one node in the graph for each transaction T_i in the schedule. Each edge e_i in the graph is of the form $(T_j \rightarrow T_k)$, $1 \leq j \leq n$, $1 \leq k \leq n$, where T_j is the starting node of e_i and T_k is the ending node of e_i.

Precedence Graph

1. For each transaction T_i participating in schedule S, create a node labeled T_i in the precedence graph.
2. For each case in S where T_j executes a read_item(X) after T_i executes a write_item(X), create an edge $(T_i \rightarrow T_j)$ in the precedence graph.
3. For each case in S where T_j executes a write_item(X) after T_i executes a read_item(X), create an edge $(T_i \rightarrow T_j)$ in the precedence graph.
4. For each case in S where T_j executes a write_item(X) after T_i executes a write_item(X), create an edge $(T_i \rightarrow T_j)$ in the precedence graph.
5. The schedule S is serializable if and only if the precedence graph has no cycles.

Precedence Graph

If there is no cycle in the precedence graph, it is possible to create an equivalent serial schedule S' that is equivalent to S, by ordering the transactions that participate in S as follows:

- Whenever an edge exists in the precedence graph from T, to Tj , T, must appear before Tj in the equivalent serial schedule S‘.
- Notice that the edges (Ti → Tj) in a precedence graph can optionally be labeled by the name(s) of the data item(s) that led to creating the edge.

Example 1

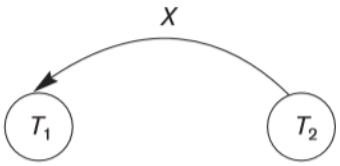
Draw Precedence graph for each schedule. Colours represent two different transactions.

1	2	3	4
READ(X)	READ(X)	READ(X)	READ(X)
X = X - N	X = X + M	X = X - N	X = X - N
WRITE(X)	WRITE(X)	READ(X)	WRITE(X)
READ(Y)	READ(X)	X = X + M	READ(X)
Y = Y + N	X = X - N	WRITE(X)	X = X + M
WRITE(Y)	WRITE(X)	READ(Y)	WRITE(X)
READ(X)	READ(Y)	WRITE(X)	READ(Y)
X = X + M	Y = Y + N	Y = Y + N	Y = Y + N
WRITE(X)	WRITE(Y)	WRITE(Y)	WRITE(Y)

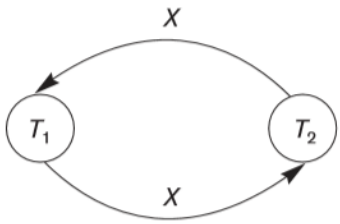
Example 1

- 1

- 2



3



4

Example 2

Transaction T_1
read_item(X);
write_item(X);
read_item(Y);
write_item(Y);

Transaction T_2
read_item(Z);
read_item(Y);
write_item(Y);
read_item(X);
write_item(X);

Transaction T_3
read_item(Y);
read_item(Z);
write_item(Y);
write_item(Z);

Example 2 – Schedule E

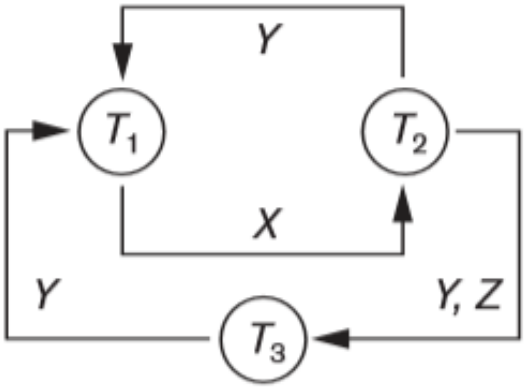
- Draw the serializable graph for the schedule E. Write the equivalent serial schedules if exist.

Time

Transaction T_1	Transaction T_2	Transaction T_3
	read_item(Z);	
	read_item(Y);	
	write_item(Y);	
		read_item(Y);
read_item(X);		read_item(Z);
write_item(X);		
		write_item(Y);
		write_item(Z);
	read_item(X);	
read_item(Y);		
write_item(Y);		
	write_item(X);	

Schedule E

Example 2 – Schedule E



- No Equivalent serial schedules.
- 2 cycles
  - Cycle X(T1→ T2),Y(T2→ T1)
  - Cycle X(T1 →T2),YZ (T2→ T3),Y(T3→ T1)

Example 2 – Schedule F

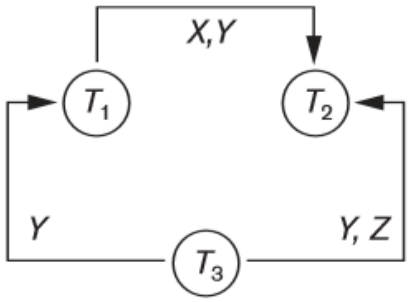
Transaction T_1	Transaction T_2	Transaction T_3
read_item(X); write_item(X);		read_item(Y); read_item(Z);
		write_item(Y); write_item(Z);
	read_item(Z);	
read_item(Y); write_item(Y);		

	read_item(Y);
	write_item(Y);
	read_item(X);
	write_item(X);	

Time

Schedule F

Example 2 – Schedule F

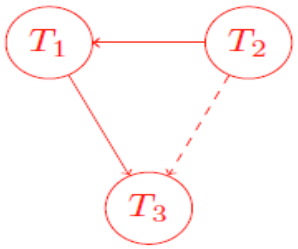


- Serial Schedule

  T3→T1→T2

Precedence Graph

$$R_1(A)W_2(B)R_2(A)W_1(B)W_3(B)W_1(C)R_3(B)W_1(A)$$



Conflict serializable, equivalent to $T_2T_1T_3$