# Module - 6

## Transaction Management, Concurrency Control and Recovery protocols

### 6.1

### # Transaction Introduction:

Transaction: This term refers to a collection of operations that form a single logical unit of work. For instance, transfer of money from one account to another is a transaction consisting of two updates, one to each account.

Every transaction should follow the ACID properties where
A - Atomicity
C - Consistency
I - Isolation
D - Durability

The concepts in this module allows application developers to focus on the implementation of individual transactions, ignoring the issues of concurrency and fault tolerance.

# Fault Tolerance: A process that enables an operating system to respond to a failure in hardware or software.

6.1.1 Transaction Concept:

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

2) A transaction is delimited by statements of the form 'Begin Transaction' and 'End transaction'.

3) The transaction consists of all operations executed between the 'begin transaction' and 'end transaction'.

4) Let's see an example on the transaction concept using a simple bank application consisting of several accounts & transactions that access (read) and update (write) ~~those~~ records in accounts.

Transactions access data using two operations:

a) READ(X): Transfers the data item X from the database to a variable X in a buffer in main memory, belonging to the transaction that executed the read operation.

(It's stored in a buffer in main memory because it provides a faster access to frequently accessed data items. This will help make the transaction efficient).

b) WRITE(X): Transfers the value of the variable X in the main memory buffer

of the transaction, that executed the 'write', to the data item X in the database. Opposite of READ)

Let $T_i$ be a transaction that transfers Rs. 1 L from account A to account B. This transaction can be defined as:

$T_i$ :

```
READ (A);          // taking current value of A in var
A = A - 100000;
WRITE (A); // transfering the new value to account
READ (B); // taking current value of B in Variable
B = B + 100000;
WRITE (B); // transferring the new value
```

The ACID property for the above transaction can be given by:

a) Consistency: The consistency requirement here is that the sum of A and B be unchanged by the execution of transaction. (Without consistency requirement, money could be created or destroyed by the transaction.

If the database is consistent before the execution of a transaction, it should remain consistent after the execution of a transaction.

Ensuring consistency is the application programmers responsibility.

b) Atomicity: If 1L is deduced from account A, it should be credited to account B, otherwise the transaction is not completed, and ~~is~~ in the database is in an inconsitent state.

Therefore, atomicity ensures that all actions of the transaction are reflected in the database or none at all.

Ensuring atomicity is done by the system by keeping track of the old values of all data on which the transaction performs a WRITE (eg: Old values of A and B). This information is stored in a file called 'log'.
If the transaction does not complete its execution, the database system restores the old values from the log (no change in database). This is handled by the Recovery System.

However, any transaction will briefly be in an incosistent ~~state~~, when for example, the 1L is deduced but not yet credited, but this state should eventually be replaced by a consitent state.

Note: Consistency rules can be specified for a database system, apart from the basic consistency rules. A consistency rule could be that account 'A' (in above example) has no loans yet to be paid, apart from the sum of balance being same before & after. The transaction

can only be consistent if all the consistency rules are followed.

c) Durability: Once the execution of the transaction executes successfully, no system-failure should result in the loss of data corresponding to this transfer of money.

The durability property ensures that once a transaction completes successfully, all the updates that it carried out on the database persists, even if there's a system failure after the transaction completes execution.

The update in a transaction is first saved in the database in the main memory then it is saved on a disk, making the update permanent. The system ensures durability by saving the update on the disk before this transaction is about to end.

If the transaction causes a system failure, the updates can be reconstructed after restarting the database.

(If you have written TCL commands, we basically wrote `committed;` before the transaction ended)

The recovery system (6.4) is responsible for ensuring durability and atomicity.

d) Isolation: If several transactions are executed concurrently, their operation may interleave in some undesirable way, resulting in an inconsistent state.

If let's say that the amount is deducted from account A but not yet credited to the account B and is in temporary inconsistent state and while it is in this state, another transaction is occurring parallely. It sees this temporary consistent state & tries to make it consistent by performing updates (WRITE operation) on both accounts.
After the first transaction is completed, the database remains inconsistent because of the unwanted WRITE operations.

Thus, the isolation property ensures that the concurrent execution of transactions results in a system state that is equivalent to a state that could have been obtained had these transactions executed one at a time.

The Concurrency-Control-System ensures isolation (6.3).

6.1.2) Transaction States:

A transaction at any stage should be in one of these states:

1) Active: The initial state. The transaction stays in this state while it is executing.

2) Partially Committed: After the transaction's final statement has been executed. At this point it may be possible that the transaction fails, in case of a system failure while the update information was still in the main memory and wasn't permanently saved, or in case if the user tries sending more funds than they have in account.

3) ~~Aborted:~~

3) Failed: Enters failed state after the system determines that the transaction can no longer proceed with its normal execution because of hardware (system failure) or logical (invalid input or incorrect application program) errors.

4) Aborted: After failure, the transaction is aborted. Any changes done by the aborted transaction must be undone. Once it is undone, we say that the transaction has been rolled back.

5) Committed: If there was no failure, the transaction is successfully committed & we land on the committed state.

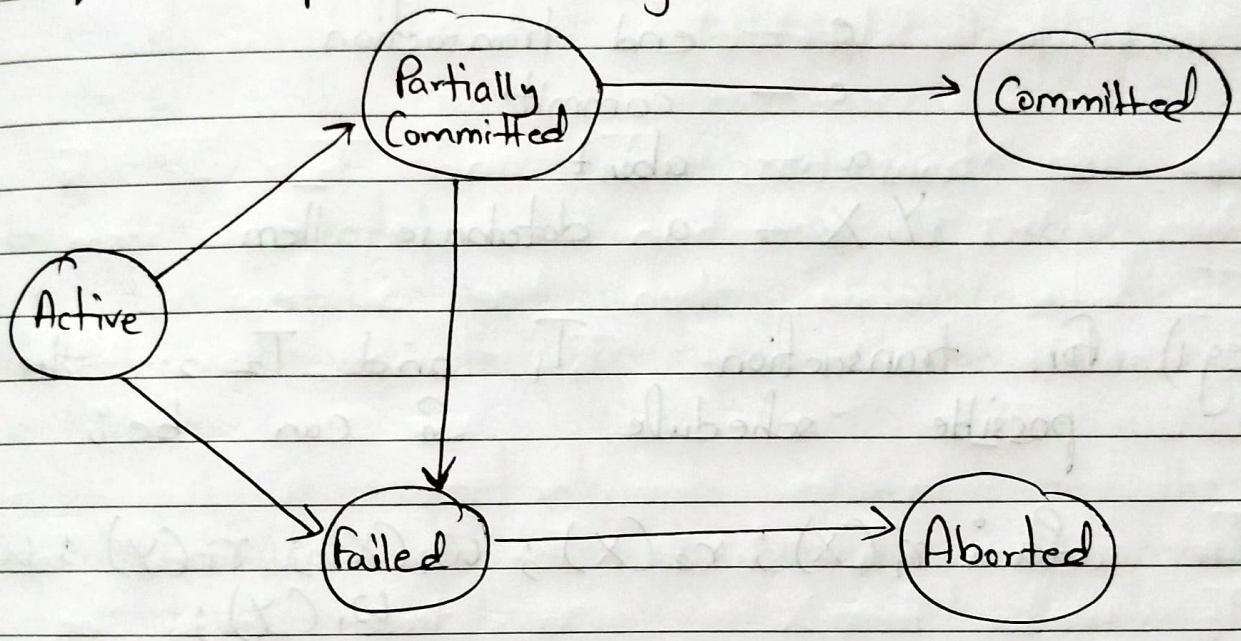
Note: After abortion of a transaction, the system has 2 options depending on the kind of failure that occured:

a) Restart the transaction: if the ~~failure~~ failure

occurred due to a system failure and there was no logical error in the transaction.

b) Kill the transaction: If the transaction failed because of a logical error that can not be fixed at the system level

## Important Diagram:




## ACID Properties

## 6.2

1) Characterizing Schedule based on recoverability:

When transactions are executing concurrently in an interleaved fashion, then the order of execution of operations from all the various transactions is known as schedule (or history).

6.1.3) ACID properties: we require that the database system maintain the following properties of the transactions:

- **Atomicity**. Either all operations of the transaction are reflected properly in the database, or none are.
- **Consistency**. Execution of a transaction in isolation (i.e., with no other transaction executing concurrently) preserves the consistency of the database.
- **Isolation**. Even though multiple transactions may execute concurrently, the system guarantees that, for every pair of transactions $T_i$ and $T_j$, it appears to $T_i$ that either $T_j$ finished execution before $T_i$ started or $T_j$ started execution after $T_i$ finished.

Thus, each transaction is unaware of other transactions executing concurrently in the system.

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

# 6.2

1) Characterizing Schedule based on recoverability:

When transactions are executing concurrently in an interleaved fashion, then the order of execution of operations from all the various transactions is known as schedule (or history).

a) A schedule S of n transactions $T_1, T_2, \ldots, T_n$ is an ordering of the operations of the transactions. Operations from different transactions can be interleaved in schedule S.

b) Notations:
S – Schedule
r – Read
w – write
e – end transaction
c – commit
a – abort
Y, X – a database item

c) eg1) For transaction $T_1$ and $T_2$, ~~the~~ a possible schedule $S_a$ can be:

$S_a$: $r_1(X)$; $r_2(X)$; $w_1(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$;

eg2) Another transaction $S_b$ could be:

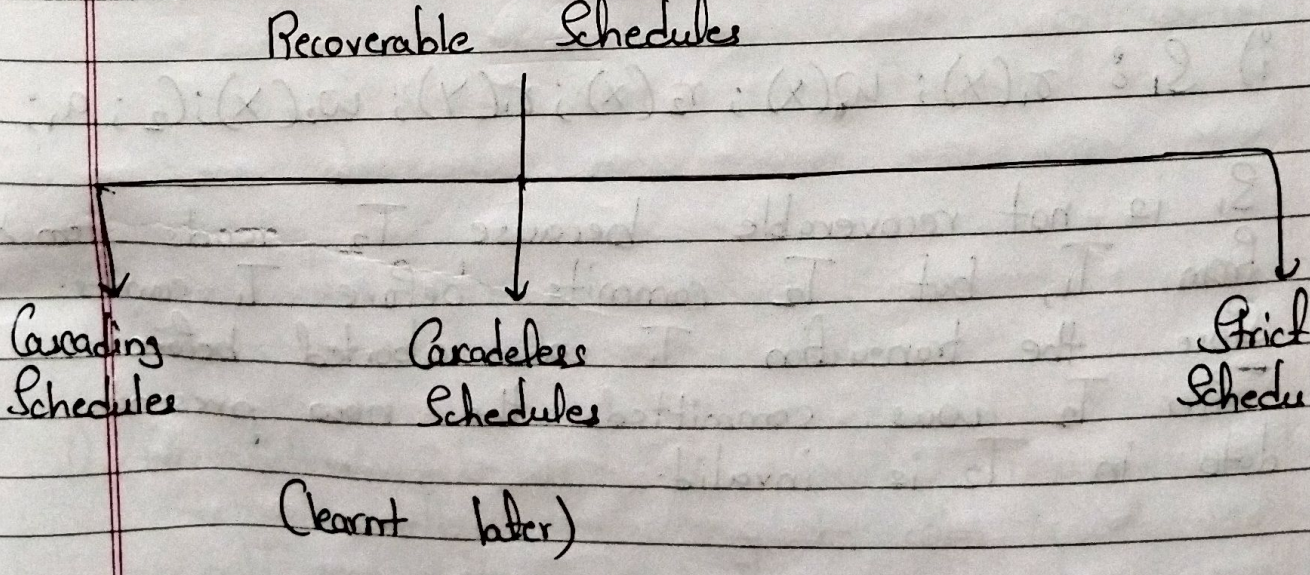
$S_b$: $r_1(X)$; $w_1(X)$; $r_2(X)$; $w_2(X)$; $r(Y)$; $a_1$;

d) Two operations in a schedule are said to be conflicting if the following 3 conditions are true:
i) They belong to different transactions
ii) They access the same item X.
iii) At least one of the operations is a write(X).

eg: for $S_a$, the operations $r_1(X)$ and $w_2(X)$ conflict along with $r_2(X)$ and $w_1(X)$ & $w_1(X)$ and $w_2(X)$.
However, the operations $r_1(X)$ and $r_2(X)$ do not conflict.
Operations $w_2(X)$ and $w_1(Y)$ also do not conflict because they operate on distinct items & operations, $w_1(X)$ and $r_1(X)$ do not conflict because they are part of the same transactions.

e) For some schedules it is easy to recover from transaction and system failures, whereas for other schedules the recovery process can be difficult or impossible.
Hence, it is important to characterize the types of schedules for which recovery is possible.

These characterization do not actually provide the recovery algorithm; they only attempt to theoretically characterize the different types of schedules.



f) Once a transaction is committed, it should not be possible to rollback the transaction. This ensures the durability of the transaction.

The schedules that meet above criterion are called recoverable schedules, while the schedules that do not meet the criterion are non-recoverable schedules.

g) Recoverable Schedule:

Take 2 transactions $T_1$ and $T_2$. If $T_1$ does WRITE operation on a data item that $T_2$ later reads, then the system is recoverable only if $T_1$ is committed before $T_2$.

In addition, $T_1$ should not be aborted before $T_2$ reads item X and there should be no transactions that write X after $T_1$ writes it and before $T_2$ reads it.

h) Let's see a few transactions:

i) $S_1$: $r_1(X); w_2(X); r_2(X); r_1(Y); w_2(X); c_2; a_1;$

$S_1$ is not recoverable because $T_2$ reads item X from $T_1$, but $T_2$ commits before $T_1$ commits. Since the transaction $T_1$ was aborted before after $T_2$ was committed, the now present data in $T_2$ is invalid.

ii) $S_2$: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$; $C_1$; $C_2$;

In $S_2$, $w_1(X)$ is followed by $r_2(X)$, thus $C_1$ should be done before $C_2$ for transaction to be recoverable, and that is what's happening. Thus, $S_2$ is recoverable.

iii) $S_3$: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; $w_2(Y)$; $a_1$; $a_2$;

In $S_3$, $w_1(X)$ is followed by $r_2(X)$. Thus, as illustrated, if $T_1$ aborts before committing, then $T_2$ should also abort before committing for the transaction to be recoverable.

Now let's look into the 3 types of recoverable schedules:

i) Cascading Rollback: When an uncommitted transaction has to be rolled back because it read an item from a transaction that failed and aborted, the phenomenon is known as cascading rollback. This is illustrated in $S_3$ above.

This can be costly if numerous transactions have to be rolled back.

ii) Cascadeless Schedule: A schedule is said to be cascadeless if every transaction in the schedule only read items that were written by committed

statements.

If we had to make $S_2$ cascadeless:

$S_2 =$ $r_1(X); w_1(X); c_1; r_2(X); r_1(Y); w_2(X); w_1(Y); c_2$

This delays $T_2$ but ensures no cascading rollback needed after $T_1$ aborts.

k) Strict Schedule: Most restricted scheduling in which transactions can neither read nor write an item X until the last transaction that wrote X has committed or aborted.

$S_4 = r_1(X); w_1(X); c_1; r_2(X); r_2(Y); c_2;$

Strict Schedules simplify the recovery process.
It is important to note that any strict schedule is also cascadeless.

2) Characterizing Schedules Based on Serializability

In previous section, schedules were characterized based on their recoverability properties.

Now we are characterizing schedules based on that are always considered to be correct when concurrent transactions are executing.

a) Serial Schedules: If let's say there are 2 transactions $T_1$ and $T_2$. A schedule for these transactions can only be called serial if all the operations in $T_1$ are strictly followed by

$T_2$ and Vice Versa.

If the operations in both transactions happen to interleave, then they are called nonserial schedules.

∴ in serial schedules only 1 transaction is active at a time. The commit or abort of the active transaction initiates execution of the next transaction.

Problem ∴ They limit the concurrency by prohibiting interleaving of operations.

However, serial schedules are sure to give the correct output.

Eg∴ a)

| $T_1$ | $T_2$ |
|---|---|
| read(X); | |
| X = X − N; | |
| write(X); | |
| read(X); | |
| Y = Y + N | |
| write(Y) | |
| | read(X); |
| | X = X + M |
| | write(X); |

time ↓

b)

| $T_1$ | $T_2$ |
|---|---|
| | read(X); |
| | X = X + M; |
| | write(X); |
| read(X); | |
| X = X − N; | |
| write(X); | |
| read(Y); | |
| Y = Y + N; | |
| write(Y); | |

b) Non Serial Schedules∴ If interleaving of operations is allowed then the schedule is non serial.

Eg: c)

| | $T_1$ | $T_2$ |
|---|---|---|
| | read (X); | |
| | X = X-N; | |
| | | read(X); |
| ↓ | | X = X+M; |
| Time | Write(X); | |
| | read (Y); | |
| | | Write (X); |
| | Y = Y+N | |
| | Write (Y); | |

d)

| $T_1$ | $T_2$ |
|---|---|
| read (X); | |
| X = X-N; | |
| write(N); | |
| | read (X); |
| | X = X+M; |
| | write (X); |
| read (Y); | |
| Y = Y+N; | |
| write (Y); | |

e) Serializable Schedules:

A non-serial schedule is serializable if it is equivalent to a serial schedule.

Meaning that the output given by the non-serial schedule should be same as

i) If there are n transactions $T_1, T_2, T_3, \ldots T_n$, then a non-serial schedule of the transactions should give the same output that a serial schedule of the trasactions would give ($T_1 \rightarrow T_2 \rightarrow \ldots T_n$) only then the non-serial transaction is serializable.

In the above example, c) coas not serializable. This is because it gives an ~~ico~~ incorrect output, which is not same as either a) or b). The reason it gives incorrect output is the lost update Problem. This problem occcurs when multiple transactions perform operations on the

same record simultaneously.

In c), $T_2$ reads the data item X before $T_1$ writes it. Thus, only the effect of $T_2$ on X is reflected on the database.

d) however, is equivalent to a serial schedule & gives the correct output, thus its serializable.

ii) When are 2 schedules equivalent?

Two definitions of equivalence of schedules are generally we:

- Conflict Equivalence
- View Equivalence

- Conflict Equivalence:

- First lets define what conflicting instructions are:

If instructions $I_i$ and $I_j$ are consecutive in a schedule & they do not conflict, their results would remain same or even if they had been interchanged in the schedule.

* $I_i$ = read(X), $I_j$ = read(X)
$I_i$ and $I_j$ don't conflict.

* $I_i$ = read(X), $I_j$ = write(X)
They Conflict.

* $I_i$ = write(X), $I_j$ = read(X)
They Conflict

If $I_i$ = write(X), $I_j$ = write(X)

They Conflict.

- Two schedules are said to be conflict equivalent if the order of any two conflicting operations is the same in both schedules.

- Eg)

| $T_1$ | $T_2$ |
|---|---|
| read(X); | |
| X = X - N; | |
| write(X); | |
| | read(X); |
| | X = X + M; |
| | write(X); |
| read(Y); | |
| Y = Y + N; | |
| write(Y); | |

Time ↓

The order of the conflicting schedules (read → write) is same in both the schedules.

- View Equivalence:

Two schedules S and S' are said to be view equivalent if following 3 conditions hold:

- Same set of transactions participate in S and S', and S and S' include the same operations of these transactions.

- Each read operation of a transaction should ~~read~~ read the result of the same write operation in both schedules.

In other words, for any operation $r_1(x)$ of $T_1$ in $S$, if it is reading X after $w_2(x)$ of $T_2$, then in $S'$ also $r_1(x)$ should read after $w_2(x)$.

- If the operation $w_3(y)$ from $T_3$ is the last operation to write item Y in $S$, then $w_3(y)$ of $T_3$ should also be the last operation ~~to~~ to write item Y in $S'$.

→ If two schedules have conflict equivalence, then they are called conflict serializable.

→ If two schedules have view equivalence, then they are called view ~~equivalent~~ serializable.

Remember that one of these schedules should be serial schedule so that ~~if~~ the other can schedule can be called serializable.

If we observe, we can point out that example c) was neither conflict nor view serializable with either a) or b).

c) Testing for Conflict Serializability of a Schedule.

For testing whether a particular schedule is conflict serializable or not we can construct a precedence graph or serialization graph.

- Nodes $N = \{T_1, T_2, \cdots T_n\}$

  Edges $E = \{e_1, e_2, \cdots e_n\}$

- for each transaction $T_i$, there is a node $T_i$.

- Algorithm for testing conflict serializability with a schedule $S$:

  i) For each transaction $T_i$ participating in schedule $S$, create a node labeled $T_i$ in the precedence graph.

  ii) For each case in $S$ where $T_i$ performs a write(X) operation followed by $T_j$ performing read(X) operation, create an edge $(T_i \rightarrow T_j)$

  iii) If $T_i$ performs a read(X) operation followed by $T_j$ performing write(X) operation, create an edge $(T_i \rightarrow T_j)$.

  iv) If $T_i$ performs a write(X) operation followed

by $T_j$ performing write(X) operation, create an edge $(T_i \rightarrow T_j)$

v) The schedule S is serializable if and only if the precedence graph has no cycles.

eg)

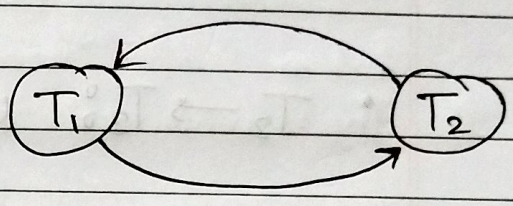
| $T_1$ | $T_2$ |
|---|---|
| read(X); | |
| X = X - N; | |
| | read(X); |
| | X = X + M |
| write(X); | |
| read(Y); | |
| | write(X); |
| Y = Y + N; | |
| write(Y); | |

Don't care: R → R

So we see this

Time ↓

Precedence Graph:




- $T_1 \rightarrow T_2$ because $write_1(X)$ is followed by $write_2(X)$.
- $T_2 \rightarrow T_1$ because $read_2(X)$ is followed by $write_1(X)$.

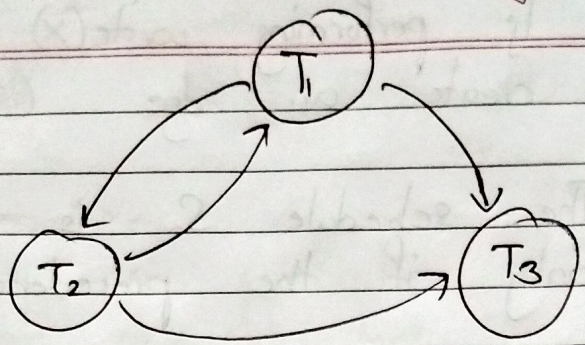
There is a cycle in the precedence graph, thus it is not conflict serializable.

eg)

| $T_1$ | $T_2$ | $T_3$ |
|---|---|---|
| read(A) | | |
| | read(B) | |
| | write(A) | |
| | Commit | |
| Write(A) | | |
| Commit | | |
| | | Write(A) |

(ignore statements & abort) ~~Commit~~ commit

Precedence Graph:




- $T_1 \rightarrow T_2$: $read_1(A)$ followed by $write_2(A)$ (ignore read → read)
- $T_2 \rightarrow T_1$: $write_2(A)$ followed by $write_1(A)$
- $T_2 \rightarrow T_3$: $write_1(A)$ followed by $write_3(A)$
- $T_1 \rightarrow T_3$: $Read_1(A)$ followed by $write_3(A)$

AND

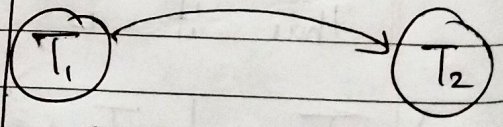
$write_1(A)$ followed by $write_3(A)$.

Not conflict serializable because loop exists.

eg)

| $T_1$ | $T_2$ |
|---|---|
| read(X); | |
| X = X-N; | |
| write(X); | |
| | read(X); |
| | X = X+M; |
| | write(X); |
| read(Y); | |
| Y = Y+N; | |
| write(Y); | |

Precedence Graph:




$T_1 \rightarrow T_2$ because $read_1(X)$ followed by $write_2(X)$.

Note:- Important definition : Blind ~~Read~~ Write:-
When a write a write operation is performed without a corresponding read operation before it:

S: $w_1(X)\ w_1(Y)\ r_2(X)\ r_2(Y)\ w_2(Y)$;

{ $w_1(X)$ & $w_1(Y)$ are blind writes }

6.3

6.3.1 ~~Two phase Lock based~~ Concurrency Control

Why do we need concurrency control?

1) Lost Update Problem:
Occurs when 2 transactions update the same data item, but both read the same original value before update.

2) Dirty Read (or temporary update) problem:
This occurs when a transaction $T_1$ updates a database item X which is accessed by another transaction $T_2$, then $T_1$ aborts.
X was read by $T_2$ before its value is changed after $T_1$ aborts.

3) The incorrect summary problem:

One transaction is calculating an aggregate summary while other transactions are updating some of these records.
The aggregate function may read some old values before they are updated and others after they are updated.

eg: $T_1$ finding average of 10 data items and while $T_1$ is on data item 5, some transaction $T_2$ updates the value of the 4th data item and 7th data item. The transaction $T_1$ will thus calculate incorrect result.

4) The unrepeatable read problem:

The transaction $T_1$ may read an item and later may read the same item again & get a different value because transaction $T_2$ has updated the item between the two reads by $T_1$.

eg) → $T_1$ reads the number of seats available.
→ $T_2$ reduces the available no. of seats by reserving the seat.
→ $T_1$ again reads the no. of seat available, but the value is different.

G.3.1 Two-phase : Lock Based Concurrency Control

Locking data items is a technique used to control concurrent execution of transactions.

1) Lock : A lock is a variable associated with a data item that describes the status of the item with respect to possible operations that can be applied to it.

Generally, there is one lock for each data item in the database.

2) The are 2 types of Locks that can be used in a system :

- Binary Locks
- Shared / Exclusive (or Read / Write) locks

3) Binary Locks : A Binary Lock can have two states or values : locked (1) and unlocked (0).

a) If the value of the lock on data item X

is 1, item X cannot be accessed by a database operation that requests the item.

b) If the value of the lock on X is 0, the data item can be accessed when requested and the lock value is changed to 1.

c) lock(X) : Current state of the lock on X.
unlock_item(X): sets lock(X) to 0 (unlocked).
lock_item(X): sets lock(X) to 1 (locked).

d) When a transaction requests a data item X whose Lock(X) = 0, the transaction can access X after locking X and unlocking it after its done.

If the transaction requests X when its Lock(X) = 1 (locked), the transaction is forced to wait till lock(X) becomes 0 (unlocked). There can be many transactions waiting for an item.

e) lock_item (X):
B: if lock(X) == 0
then lock(X) = 1
else
begin
wait (until lock(X) = 0 & the lock manager wakes up the transaction);
go to B
end;

the algorithm above is for locking a data item. If it is unlocked when a transaction requests it, it is directly locked. If it is locked, the transaction has to wait till its unlocked, after which it can access the item & lock it.

f) unlock_item(X):

```
Lock(X) = 0;
if any transactions are waiting the wake one
    of them up;
```

The algorithm is for unlocking a ~~transaction~~ data item after a transaction has ~~committed~~ completed its operations on the data item X.

Once its unlocked, the other transactions that were waiting for it can be ~~wok~~ woken up by one of the system.

g) Lock table: The system needs to record maintain some records for the items that are currently locked, in a lock table:

a) Data item's name
b) LOCK (0 or 1)
c) The transaction that has locked the item.
d) The queue for all the transactions waiting to access the item.

h) Lock manager Subsystem: Each ~~DBMS~~ DBMS has

a lock manager subsystem to keep track of & control access to locks.

i) In binary locking, every transaction must obey the following rules:

- A transaction must lock data item before performing any read or write operation to it.
- A transaction must unlock data item before after all read and write operations to it are completed.
- A transaction can not issue Lock to a data item if it already holds the lock on item.
- A transaction can not issue an unlock to a data item unless it already holds the lock for it.

Note: Between lock_item(X) and unlock_item(X), a transaction T is said to hold the lock on item X.

3) Shared / Exclusive (or Read/Write) Locks:

Binary Locks can be very restrictive because at most, one transaction can hold a lock on a given time.

To make it less restrictive, the logic is to allow

several transactions to access the same item X if they all access the item X for reading purposes only because read operations are not conflicting. However, if a transaction is to write an item X then only that transaction should have the (exclusive) access to X.

a) Shared / exclusive or Read / Write locks:
In this scheme, there are 3 locking operations:
- read_lock(X)
- write_lock(X)
- unlock(X).

When a transaction has

- read_lock(X) (or shared_lock(X))
  then other transactions are allowed to read the item.

- write_lock(X)
  then that transaction exclusively holds the lock on the item & no other transaction can access it in any way.

b) The lock table keeps the following records:
- Data item's name
- Lock (read or write locked)
- no. of reads to the data item
- Locking transaction(s).

If the variable -Lock is read-lock, then

the list of all transactions accessing data item for reading are listed in - locking transactions.

Else if - lock is write locked, then the transaction with the exclusive access is written for - locking transaction.

c) The system must enforce the following rules for shared / exclusive locking scheme:

i - A transaction must issue a read_lock(X) or write_lock(X) before it can read_item(X).

ii - A transaction must issue a write_lock(X) before it can write_item(X).

iii - A transaction must issue operation unlock(X) after all operations read_item(X) and write_item(X)

iv - A transaction will not issue a read_lock(X) if it already holds the ~~op~~ read_lock(X) or the write_lock(X).

v - A transaction will not issue a write_lock(X) if it already holds the read_lock(X) or the write_lock(X).

vi - A transaction will not issue an unlock(X) ~~if~~ unless it already holds a read_lock(X) or write_lock(X).

d) Sometimes it is desirable to relax points iv) and v) to allow lock conversion.
In lock conversion, the transaction is allowed to convert its lock from read_lock(X) to write_lock(X) and vice versa.

Upgrade of lock – issuing write_lock(X) from read_lock(X).

Downgrade of lock – issuing read_lock(X) from write_lock(X).

The Lock table, thus, has to update correspondingly.

e) Algorithms for all 3 operations in shared/exclusive locks:

```
read_lock(X):
B:  if LOCK(X) = "unlocked"
        then begin LOCK(X) ← "read-locked";
                   no_of_reads(X) ← 1
                   end
    else if LOCK(X) = "read-locked"
        then no_of_reads(X) ← no_of_reads(X) + 1
    else begin
             wait (until LOCK(X) = "unlocked"
                 and the lock manager wakes up the transaction);
             go to B
             end;
write_lock(X):
B:  if LOCK(X) = "unlocked"
        then LOCK(X) ← "write-locked"
    else begin
             wait (until LOCK(X) = "unlocked"
                 and the lock manager wakes up the transaction);
             go to B
             end;
unlock (X):
    if LOCK(X) = "write-locked"
        then begin LOCK(X) ← "unlocked";
                   wakeup one of the waiting transactions, if any
                   end
    else it LOCK(X) = "read-locked"
        then begin
                   no_of_reads(X) ← no_of_reads(X) –1;
                   if no_of_reads(X) = 0
                       then begin LOCK(X) = "unlocked";
                                  wakeup one of the waiting transactions, if any
                                  end
                   end;
```

**Figure 22.2**
Locking and unlocking operations for two-mode (read-write or shared-exclusive) locks.

## e) Two-phase Locking:

Two-phase locking is used to guarantee serializability as using the locks above do not guarantee serializability of schedules on their own.

A transaction is said to follow the two-phase locking protocol if all locking operations are followed by the first unlocking operation in the transaction.

a) The two phases are, thus;:

i) Expanding or growing (first phase):

New locks on items can be acquired but none can be released.

ii) Shrinking (second) phase:

Existing ~~blue~~ locks can be released but no new locks can be acquired.

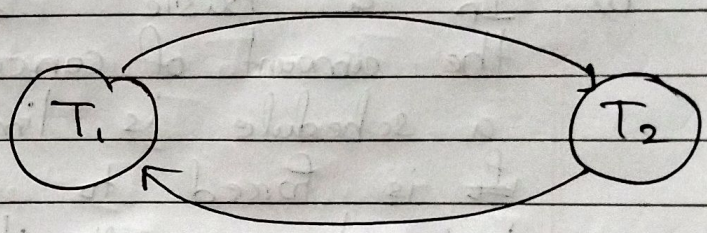
If lock conversion is allowed, upgrading (read_lock to write_lock) should be done in expanding phase and downgrading (write_lock to read_lock) should be done in shrinking phase.

- Example of non-serializable schedule that does not follow 2-phase locking:

| $T_1$ | $T_2$ |
|---|---|
| read_lock(X); | |
| read_item(Y); | |
| unlock(Y); | |
| | read_lock(X); |
| | read_item(X); |
| | unlock(X); |
| | write_lock(Y); |
| | read_item(Y); |
| | Y = X + Y; |
| | write_item(Y); |
| | unlock(Y); |
| write_lock(X); | |
| read_item(X); | |
| X = X + Y; | |
| write_item(X); | |
| unlock(X); | |

Time ↓

Precedence Graph:




- $T_1 \rightarrow T_2$ because read_item$_1$(X) followed by write_item(Y).
- $T_2 \rightarrow T_1$ because read_item$_2$(X) followed by write_item$_1$(X)

Rewriting the above example so that it is 2-phase locked, we get:

| $T_1'$ | $T_2'$ |
|---|---|
| read_lock(Y); | read_lock(X); |
| read_item(Y); | read_item(X); |
| write_lock(X); | write_lock(Y); |
| unlock(Y); | unlock(X); |
| read_item(X); | read_item(Y); |
| X = X+Y; | Y = X+Y; |
| write_item(X); | write_item(Y); |
| unlock(X); | unlock(Y); |

↓ Time

Above schedule is also serializable.

b) In a basic 2 ~~phase~~ phase locking (2PL) the amount of concurrency that can occur in a schedule is limited, because a transaction ~~is~~ is forced to hold an item even after it has done all its operation with it, while another transaction is waiting for that data item.
There are 3 different types of 2PL that increase concurrency:-

- Conservative 2PL
- Strict 2PL
- Rigorous 2PL.

c) Conservative 2PL: In conservative 2PL, the transaction is required ~~to~~ to predeclare its read-set (set of all items it needs to read)

and write-set (set of all items it needs to write). If any of the predeclared items needed cannot be locked, the transaction does not lock any items & waits until all the items are available for locking.

d) Strict 2PL: Guarantees strict schedules. In this scheme, the transaction holding a write-lock cannot release that write-lock before it has committed or aborted. This leads to a strict schedule (no read or write of X before commit/abort) for recoverability.

e) Rigorous 2PL: Also guarantees strict schedule. In this variation, the transaction T does not release any of its locks (read_lock or write_lock) until after it commits or aborts.

6.3.5) Deadlock Handling

What is a deadlock?

Deadlock occurs when each transaction T in a set of 2 or more transactions is waiting for some item that is locked by some other transaction T' in the set.

Example: A transaction $T_1$ is waiting for the data item X while holding the data item Y. Another transaction $T_2$ is waiting for Y from transaction $T_1$ while holding X.

Hence, each transaction is in a waiting queue

waiting for the other transaction to release the lock on an item.

a) Deadlock Prevention Protocols:

i) Transaction Timestamp TS(T): Unique identifier assigned to each transaction. The ~~stamp~~ timestamps are based on the order in which the ~~st~~ transactions started. Hence, if transaction $T_1$ starts before $T_2$ then $TS(T_1) < TS(T_2)$

ii) Two schemes that prevent deadlock:

- Wait - Die: $T_1$: older transaction
  $T_2$: younger transaction
  $TS(T_1) < TS(T_2)$
  - If $T_1$ requests access to a data item held by $T_2$, $T_1$ is allowed to wait.
  - But if $T_2$ requests access to a data item held by $T_1$, $T_2$ is killed and restarted later ~~aft~~ with the same timestamp value $TS(T_2)$.

- Wound - Die:
  - If $T_2$ requests a data item held by $T_1$, then $T_2$ ~~e~~ is allowed to wait.
  - However, If $T_1$ requests a data item held by $T_2$, then $T_2$ is killed & restarted with the timestamp $TS(T_2)$.

Notice that in both the methods, the younger

of the 2 transactions has to die.

iii) Another schemes for deadlock detection:

- No waiting: If a transaction is unable to obtain a lock, it is immediately aborted & restarted later without checking if a deadlock will actuall occur or not.

- Cautious Waiting: If $T_1$ holds a data item X that $T_2$ requests then:

  - Block $T_2$, if $T_1$ is not blocked
  - Kill $T_2$, if $T_1$ is blocked,

  where 'Blocked' means that the transaction is waiting of a data item held by another transaction.
  If $T_2$ is blocked, it is allowed to wait for $T_1$ to release the lock on X.

* b) Deadlock Detection:

→ In deadlock detection, the system checks if the state of deadlock actually exists to deal with deadlocks.

→ A simple way to detect a state of deadlock is for the system to construct & maintain a Wait-for Graph.

→ Algorithm to create the wait-for graph

- One node is created in the wait-for graph for each transaction that is currently executing.
- Whenever a transaction $T_2$ is waiting to lock an item X that is currently locked by the item $T_1$, a directed edge $T_2 \rightarrow T_1$ is created in the wait-for graph.
- When $T_1$ releases the lock on the item that $T_2$ was waiting for, the directed edge is dropped from the wait-for graph.

* We have a state of deadlock if & only if the wait-for graph has a cycle.

→ If the system is in a state of deadlock, some of the transactions causing the deadlock must be aborted. Choosing which transactions to abort is known as victim selection.

The algorithm for victim selection should avoid transactions that have been running for a long time that have performed many updates & instead select transactions that have not many changes.

→ Another simple method for ~~dead~~ dealing with deadlock is timeouts.
In this method, if a transaction waits for a

period longer than a system-defined timeout period, the system assumes that the transaction may be deadlocked and aborts it.

c) Starvation: This problem occurs when a transaction cannot proceed for ~~an~~ a long period of time, ~~because~~ while the other transactions continue normally.

This can happen if the system prioritizes some transaction over the other & one transaction being overlooked.

Solution: Using a first-come-first-served queue, transactions get access to the data item in the order that they requested.

→ ~~Dea~~ Starvation due to Victim Selection:
The algorithm of victim selection may choose the ~~tr~~ same transaction as the victim multiple times and aborting it.
Solution: Prioritize transactions ~~there~~ that have been aborted multiple times.

6.3.2) Concurrency Control Based on Timestamp:

A different approach to 2PL transactions that guarantees serializability is involves using timestamp to order transaction execution for an equivalent serial schedule.

1) Time Stamp: A timestamp is a unique identifier created by DBMs to identify a transaction.

Typically, a timestamp values are assigned in the order in which the transactions are submitted to the system.
Timestamp of transaction T = TS(T)

2) Timestamps can be generated is several ways:

a) Use a counter that is incremented each time a new transaction is assigned a timestamp value.
The counter has to reset when no transactions are executing, because there is a max to the count.

b) Use the current date/time of the system to assign timestamps, ensure no two transactions happen at the same tick.

3) Timestamp ordering algorithm.

a) Timestamp Ordering (TO): In this scheme, we order the transactions based on their timestamps. The schedule is the serializable.

A non-serial schedule thus formed will have equivalent serial schedule that is also ordered according to the timestamps of the transactions.

The algorithm must ensure that, in case of conflicting transactions, the access to data items should be given in the order of timestamps.

To do this, the algorithm associates with each database item X 2 timestamp (TS) values:

i) read_TS(X): The read timestamp of a data item $X_{out}$ is equal to the largest timestamp of all the conflicting transactions (youngest transaction, which will have the largest TS(T) value).

ii) write_TS(X): The write_timestamp of a data item X is equal to the largest timestamp$_{out}$ of all the conflicting transactions.

b) Basic Timestamp Ordering:
Whenever a transaction T tries to access data item X, the system checks the timestamp of the transaction T and compares it with read_TS(X) and write_TS(X).

If the timestamp of T is greater, the trasaction is allowed to proceed & if it is not, the transaction is aborted & resubmitted to the system with a new (greater) timestamp.

c) Strict Timestamp Ordering:
This ordering ensures that the schedule is strict (& recoverable) and serializable.

Let's say there are 2 transactions $T_1$ and $T_2$. $T_1$ last did the write operation,

on X, thus write_TS(X) = TS($T_1$).
Now if $T_2$ issues a read or write operation on X, we only let it proceed if $T_1$ has committed or aborted, else it has to wait.
Note that TS($T_2$) > TS($T_1$), should be the case, otherwise $T_2$ will be aborted & resubmitted with new timstamp.

## 6.3.3) Multiversion Concurrency Control:

1) Multiversion Concurrency Control logic -
In this protocol, old values of the data item is recorded when that data item is updated. Thus, several versions of an item are maintained.

2) Multiversion technique based on timestamp ordering:

Several versions $X_1, X_2, \ldots X_k$ of each data item X are maintained.
For each version, the value of version $X_i$ along with following 2 timestamps are kept:

i) read_TS($X_i$): The read timestamp of version $X_i$ is the largest of all the timestamps of transactions that have successfully read version $X_i$.

ii) Write_TS($X_i$): The write timestamp of version $X_i$ is the largest of all the transactions that have successfully wrote the value of $X_i$.

Whenever a transaction T is allowed to execute a write operation on X, a new version X' is created. the read_TS(X') and write_TS(X') is set to TS(T).

If T executed only read operation, read_TS(X') is set to TS(T).

3) Multiversion Two-Phase locking using certify locks:

There are three locking modes for an item in multiversion 2PL:

- read
- write
- certify

Hence, a data item X can be read_locked, write_locked or certify_locked. or unlocked.

In the lock compatibility table given below, Let's say there are 2 transactions $T_1$ and $T_2$. $T_1$ has the access specified by the column headers, to the data item X.
$T_2$ requests the access specified by the row headers, to the data item X.

A Yes indicates that $T_2$ can get access to X that is already accessed by $T_1$.

A No indicates that $T_2$ cannot get the requested access to X.

**(a)**

| | Read | Write |
|---|---|---|
| Read | Yes | No |
| Write | No | No |

**(b)**

| | Read | Write | Certify |
|---|---|---|---|
| Read | Yes | Yes | No |
| Write | Yes | No | No |
| Certify | No | No | No |

**Figure 22.6**
Lock compatibility tables. (a) A compatibility table for read/write locking scheme. (b) A compatibility table for read/write/certify locking scheme.

A transaction T can write the value of X as needed without affecting the value of the committed X.
Once T is ready to commit, it must obtain a certify lock on all items that it currently holds write locks on before it can commit.

Once the certify locks are acquired, the committed version X of the data item is set to the value of the new X written by T.
The certify locks are then released.
Certify locks are exclusive locks.

Thus, certify locks can be defined as:
an exclusive lock obtained only when all the updated values are finalized.

6.3.4) Validation Based Concurrency Control:

In all the previous concurrency control techniques, some checking is done before a data item can be committed to the database.

1) In validation (or optimistic) Concurrency control, no checking is done during the transaction is running.

During transaction execution, updates are applied to the local copies of the data items that are kept for the transaction. These updates are not committed to the database.

At the end of transaction execution, a validation phase checks whether any of the transaction's updates violates ~~serializ~~ serializability.

If serializability is not violated, the transaction is committed to the database, otherwise it is aborted and restarted later.

2) There are 3 phases for this concurrency control protocol:

a) Read Phase: A transaction can read values of committed data items from the database. However, updates are applied only to local copies of the data items kept in the transaction workspace.

b) Validation Phase: Checking is performed to ensure that serializability will not be violated if the transaction updates are applied to the database.

c) Write phase: If the validation phase is successful, the transaction updates are applied to the database, otherwise the updates are discarded & the transaction is restarted.

3) The validation Phase for $T_{random}$, checks that, for each such transition, $T_{random}$ that is either committed or is in its validation phase, one of the following conditions holds:

a) The transaction $T_2$ completes its write phase before $T_1$ starts its read phase.

b) $T_1$ starts its write phase after $T_2$ completes its write phase, & the read-set of $T_1$ has no items in common with the write-set of $T_2$.

c) Bothe the read_set and write_set of $T_1$ have no items in common with the write_set of $T_2$, & $T_2$ completes its read phase before $T_1$ completes its read phase.

In the validation phase, the first condition is checked first. If ~~it does not~~ it's not met then the 2nd condition is checked. If it's not met either then the 3rd condition is checked.

If none of the conditions are met the validation of $T_1$ ~~is~~ fails & it is aborted & ~~et~~ restarted later.

Else, the validation phase is successful.