SCHEDULE

Module VI

Schedules Based on Recoverability

- Recoverable schedule
- Non recoverable schedule
- Cascade less schedule
- Strict

Strict £Cascade less schedule £ Recoverable schedule

Schedules classified on recoverability:

- **Recoverable schedule:**

One where no transaction needs to be rolled back.

- A schedule S is recoverable if no transaction T in S commits until all transactions T' that have written an item that T reads have committed.

S1: R1(A),W1(A), R2(A),W2(A),c1,c2;

S2: R1(A),W1(A), R2(A),W2(A),c2,c1;

A*B+C

- **Cascadeless schedule:**
- One where every transaction reads only the items that are written by committed transactions.
- S3: R1(A),W1(A), c1,R2(A),W2(A),c2;

- **Schedules requiring cascaded rollback:**

A schedule in which uncommitted transactions that read an item from a failed transaction must be rolled back.

Strict Schedules:

- A schedule in which a transaction can neither read or write an item until the last transaction that wrote X has committed

example1

- s1:R1(X),W1(X),R1(Y)W1(Y),C1,R2(X),W2(X),c2

example1

- s1:R1(X),W1(X),R1(Y)W1(Y),C1,R2(X),W2(X),c2
- -Strict

example2

- s2:R1(X),W1(X),R1(Y)W1(Y), R2(X), C1,W2(X),c2

example2

- s2:R1(X),W1(X),R1(Y)W1(Y), R2(X), C1,W2(X),c2- recoverable

example3

- s4:R1(X),W1(X),R1(Y)W1(Y), R2(X), W2(X),c2,c1

example3

- s4:R1(X),W1(X),R1(Y)W1(Y), R2(X), W2(X),c2,c1 –Non recoverable

Example4

- s17:R1(X),W1(X), R2(X), W2(X), R1(Y),W1(Y),c1,c2

Example4

- s17:$R_1(X)$,$W_1(X)$, $R_2(X)$, $W_2(X)$, $R_1(Y)$,$W_1(Y)$,c_1,c_2 - Recoverable

Example5

- s18:$R_1(X)$,$W_1(X)$, $R_2(X)$, $W_2(X)$, $R_1(Y)$,$W_1(Y)$,c_2,c_1 -

Example5

- s_{18}:$R_1(X)$,$W_1(X)$, $R_2(X)$, $W_2(X)$, $R_1(Y)$,$W_1(Y)$,c_2,c_1 – Non Recoverable

Example6

- S21:R1(X),R2(X),W1(X),R1(Y),W1(Y),C1,W2(X),C2;

Example6

- S_{21}:$R_1(X)$,$R_2(X)$,$W_1(X)$,$R_1(Y)$,$W_1(Y)$,C_1,$W_2(X)$,C_2;

Example6

- S21:R1(X),R2(X),W1(X),R1(Y),W1(Y),C1,W2(X),C2;-Strict(hence cascadeless)

Example6

- S22:R1(X),R2(X),W1(X),R1(Y),W1(Y),W2(X),c1,C2;-

W2(X),

Example6

- S22:R1(X),R2(X),W1(X),R1(Y),W1(Y),W2(X),c1,C2;

W2(X),

Example6

- S22:R1(X),R2(X),W1(X),R1(Y),W1(Y),W2(X),c1,C2;-cascadeless

More Examples

- S23:R1(X),R2(X),W1(X),R1(Y),W1(Y),W2(X),c2,C1;

More Examples

- S23:R1(X),R2(X),W1(X),R1(Y),W1(Y),W2(X),c2,C1; - Cascadeless

- SS:R1(X),W1(X),C1,R2(X),W2(X) STRICT
- SS1:R1(X),W1(X),W2(X),C1, R2(X), CASCADELESS
- SS2:R1(X)R2(X)W1(X)C1W2(X)C2 STRICT

More Examples

- S22:R1(X),R2(X),W1(X),R1(Y),W1(Y),W2(X),c2,C1;

RECOVERABLE SCHEDULE

- T_j commits after T_i if T_j has read an item written by T_i

Cascadeless schedule

T_j reads X only after Ti has written to X and terminated(committed or aborted)

Strict :

A schedule is strict if the following conditions are saisfied

1.T_j reads X only after Ti has written to X and terminated(committed or aborted)

2.T_j writes X only after Ti has written to X and terminated(committed or aborted)

The System Log

- **Log or Journal:** The log keeps track of all transaction operations that affect the values of database items.
- This information may be needed to permit recovery from transaction failures.
- The log is kept on disk, so it is not affected by any type of failure except for disk or catastrophic failure.
- In addition, the log is periodically backed up to archival storage (tape) to guard against such catastrophic failures.

Log records

- unique **transaction-id** that is generated automatically by the system and is used to identify each transaction:

Types of log record:

- **[start_transaction,T]**: Records that transaction T has started execution.

[write_item,T,X,old_value,new_value]: Records that transaction T has changed the value of database item X from old_value to new_value.

- **[read_item,T,X]:** Records that transaction T has read the value of database item X.
- **[commit,T]**: Records that transaction T has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.
- **[abort,T]:** Records that transaction T has been aborted.

Recovery using log records:

- If the system crashes, we can recover to a consistent database state by examining the log and

1. Because the log contains a record of every write operation
that changes the value of some database item, it is possible
to **undo the effect of these write operations of a transaction T**
by tracing backward through the log and resetting all items
changed by a write operation of T to their old_values.
2. We can also **redo the effect of the write operations of a**
transaction T by tracing forward through the log and setting all
items changed by a write operation of T (that did not get done
permanently) to their new_values

Commit Point of a Transaction:

- **Definition a Commit Point:** A transaction T reaches its **commit point when all its** operations that access the database have been executed successfully *and the effect of all the transaction operations on* the database has been recorded in the log.
- Beyond the commit point, the transaction is said to be

committed, and its effect is assumed to be permanently recorded in the database.

- The transaction then writes an entry [**commit,T**] into the log.
- **Roll Back of transactions:**

Needed for transactions that have a [start_transaction,T] entry into the log but no commit entry [commit,T] into the log.

Commit Point of a Transaction (cont):

- **Redoing transactions:**
- Transactions that have written their commit entry in the log must also have recorded all their write operations in the log; otherwise they would not be committed, so their effect on the database can be redone from the log entries. (Notice that the log file must be kept on disk.
- At the time of a system crash, only the log entries that have been written back to disk are considered in the recovery process because the contents of main memory may be lost.)
- **Force writing a log:**

Before a transaction reaches its commit point, any portion of the log that has not been written to the disk yet must now be written to the disk. This process is called force-writing the log file before committing a transaction.

Two approaches using logs

- Deferred database modification
- Immediate database modification

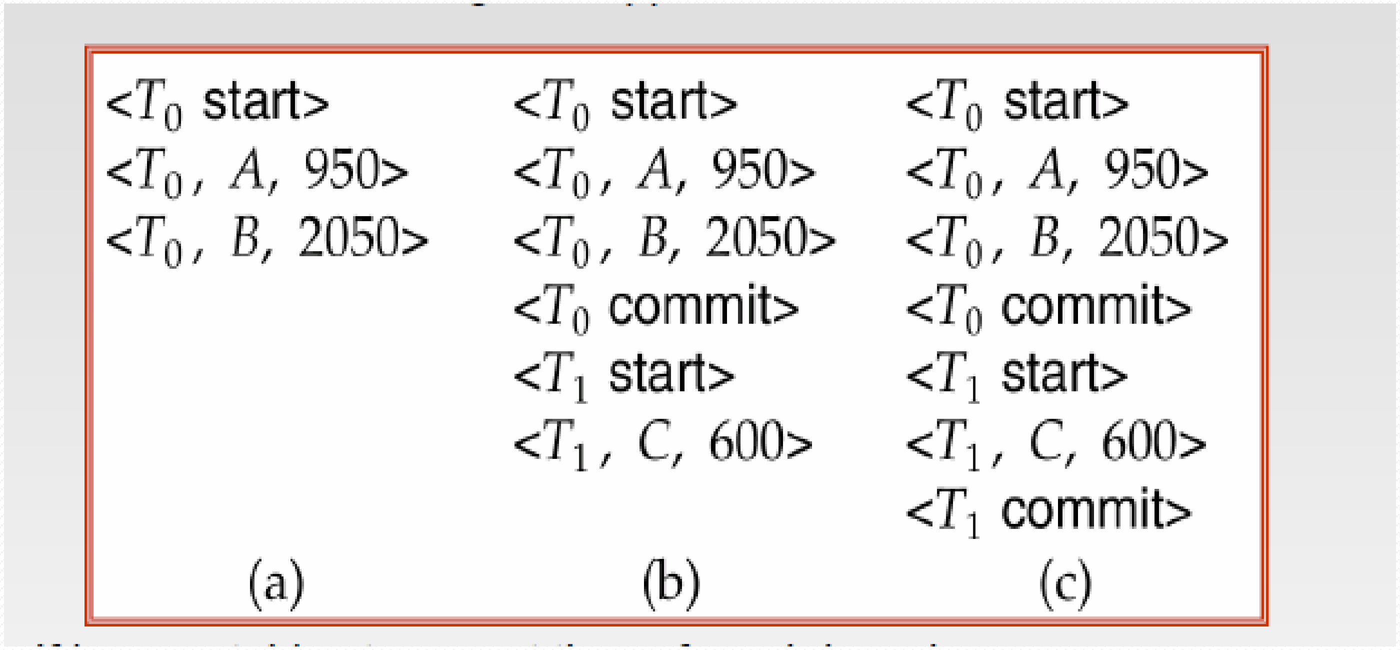
Deferred Database Modification

- The **deferred database modificationscheme records all modifications to the log, but defers all the writes to after partial commit.**
- Assume that transactions execute serially
- Transaction starts by writing <*Ti,**start> record to log.***
- A **write(*X*) *operation results in a log record <Ti, X, V> being written, where V is the new value for X***
  - Note: old value is not needed for this scheme
- The write is not performed on *X at this time, but is deferred.*
- When *Ti partially commits, <Ti,**commit> is written to the log***
- Finally, the log records are read and used to actually execute the previously deferred writes.

- During recovery after a crash, a transaction needs to be redone if and only if both <*Tistart> and<Ti commit> are there in the log.*
- Redoing a transaction *Ti (redoTi) sets the value of all data items updated by the transaction to the new values.*
- Crashes can occur while
  - the transaction is executing the original updates,
  - or while recovery action is being taken

```
$T_0$: read ($A$)
        $A$: - $A$ - 50
        Write ($A$)
        read ($B$)
        $B$:-  $B$ + 50
        write ($B$)
```

```
$T_1$ : read ($C$)
        $C$:-   $C$- 100
        write ($C$)
```



<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 950>	<T_0, A, 950>	<T_0, A, 950>
<T_0, B, 2050>	<T_0, B, 2050>	<T_0, B, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 600>	<T_1, C, 600>
		<T_1 commit>
(a)	(b)	(c)

If log on stable storage at time of crash is as in case:

(a) No redo actions need to be taken

(b) redo(T_0) must be performed since <T_0 **commit**> is present

(c) **redo**(T_0) must be performed followed by redo(T_1) since <T_0 **commit**> and <T_i commit> are present

Immediate Database Modification

The **immediate database modification** scheme allows database updates of an uncommitted transaction to be made as the writes are issued

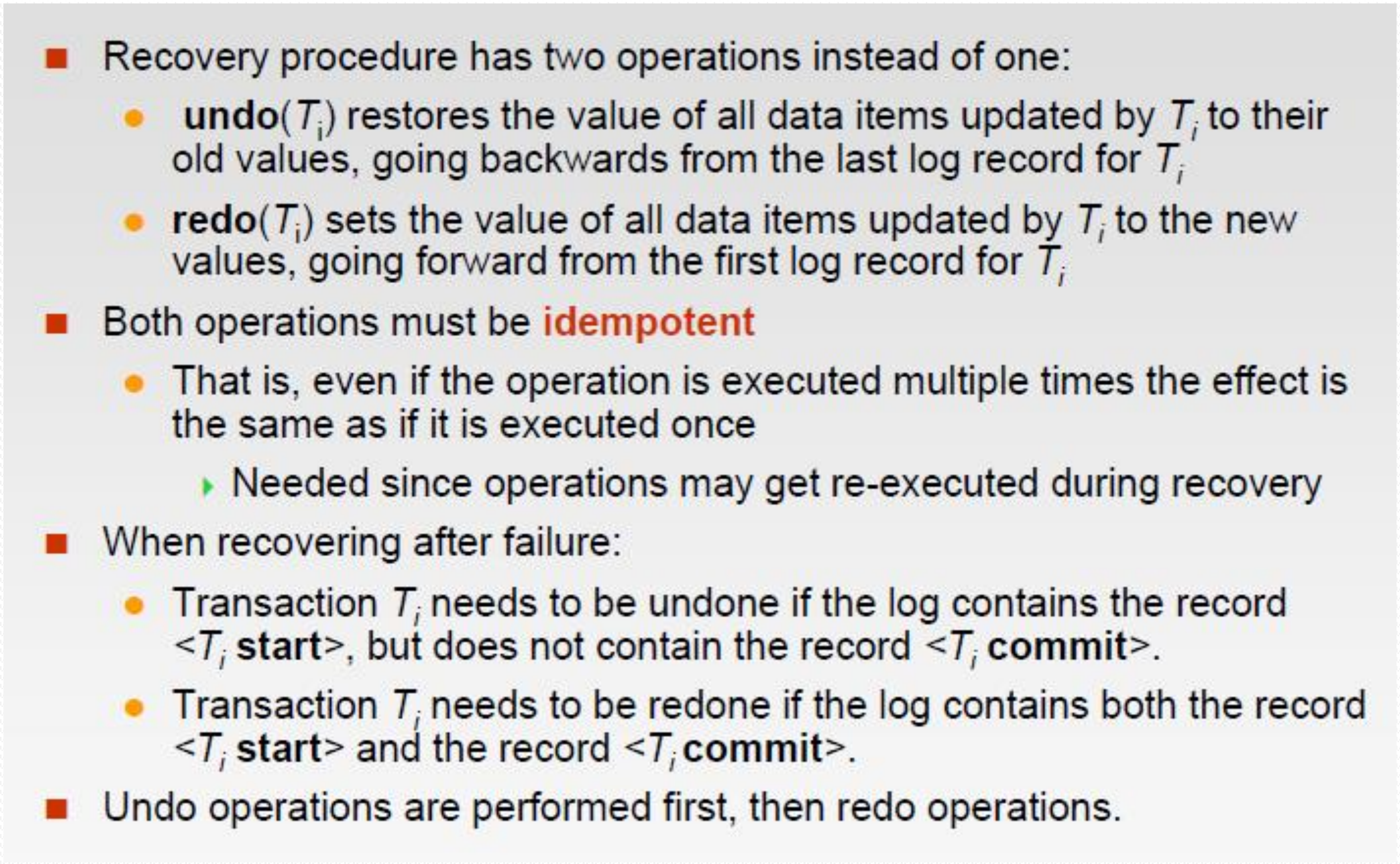
- since undoing may be needed, update logs must have both old value and new value

Update log record must be written *before* database item is written

- We assume that the log record is output directly to stable storage
- Can be extended to postpone log record output, so long as prior t execution of an **output**(*B*) operation for a data block B, all log records corresponding to items *B* must be flushed to stable storage

Output of updated blocks can take place at any time before or after transaction commit

Order in which blocks are output can be different from the order in which they are written.



- Recovery procedure has two operations instead of one:
  - **undo**(T_i) restores the value of all data items updated by T_i to their old values, going backwards from the last log record for T_i
  - **redo**(T_i) sets the value of all data items updated by T_i to the new values, going forward from the first log record for T_i
- Both operations must be **idempotent**
  - That is, even if the operation is executed multiple times the effect is the same as if it is executed once
    - Needed since operations may get re-executed during recovery
- When recovering after failure:
  - Transaction T_i needs to be undone if the log contains the record <T_i **start**>, but does not contain the record <T_i **commit**>.
  - Transaction T_i needs to be redone if the log contains both the record <T_i **start**> and the record <T_i **commit**>.
- Undo operations are performed first, then redo operations.

Below we show the log as it appears at three instances of time.

<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 1000, 950>	<T_0, A, 1000, 950>	<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 700, 600>	<T_1, C, 700, 600>
		<T_1 commit>
(a)	(b)	(c)

Recovery actions in each case above are:

(a) undo (T_0): B is restored to 2000 and A to 1000.

(b) undo (T_1) and redo (T_0): C is restored to 700, and then *A* and *B* are set to 950 and 2050 respectively.

(c) redo (T_0) and redo (T_1): A and B are set to 950 and 2050 respectively. Then *C* is set to 600