APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

STUDY MATERIALS

MODULE V

Transaction Processing Concepts - Overview of Concurrency Control, Transaction Model, Significance of Concurrency Control & Recovery, Transaction States, System Log, Desirable Properties of Transactions.

Serial Schedules, Concurrent and Serializable Schedules, Conflict Equivalence and Conflict Serializability, Recoverable and Cascade-Less Schedules, Locking, Two-Phase Locking and Its Variations. Log-Based Recovery, Deferred Database Modification, Check-Pointing.

Introduction to NoSQL Databases, Main Characteristics of Key-Value DB (Examples From: Redis), Document DB (Examples From: Mongodb)

Main Characteristics Of Column - Family DB (Examples From: Cassandra) And Graph DB (Examples From : Arangodb)

References:

1. Elmasri R. and S. Navathe, Database Systems: Models, Languages, Design and Application Programming, Pearson Education, 2013.
2. Web Resource: https://www.w3resource.com/redis/
3. Web Resource: https://www.w3schools.in/category/mongodb/
4. Web Resource: https://www.tutorialspoint.com/cassandra/cassandra_introduction.htm
5. Web Resource: https://www.tutorialspoint.com/arangodb/index.htm

Transaction Processing Concepts

Single-User System: At most one user at a time can use the system.

Multiuser System: Many users can access the system concurrently.

Overview of Concurrency Control and Recovery

Concurrency may be achieved by **Interleaved Processing**, where processes are executed in an interleaved fashion with a single CPU or by **Parallel Processing**, where processes are concurrently executed in multiple CPUs.

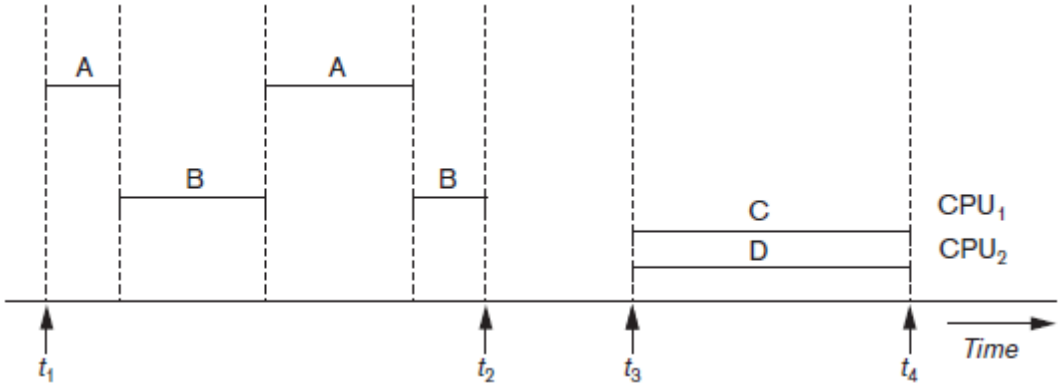


Figure 6.1: Interleaved processing versus parallel processing of concurrent transactions.

A Transaction is a Logical unit of database processing that includes one or more access operations. These can include insertion, deletion, modification, or retrieval operations. If the database operations in a transaction do not update the database but only retrieve data, the transaction is called a **read-only transaction**; otherwise it is known as a **read-write transaction**. A **database** is basically represented as a collection of *named data items.* The size of a data item is called its **granularity**. A **data item** can be a *database record*, but it can also be a larger unit such as a whole *disk block* or even a smaller unit such as an individual *field (attribute) value* of some record in the database. The transaction processing concepts are independent of the data item granularity (size) and apply to data items in general.

The basic database access operations that a transaction can include are as follows:

■ **read_item(*X*).** Reads a database item named *X* into a program variable *X*.

■ **write_item(*X*).** Writes the value of program variable *X* into the database item named *X*.

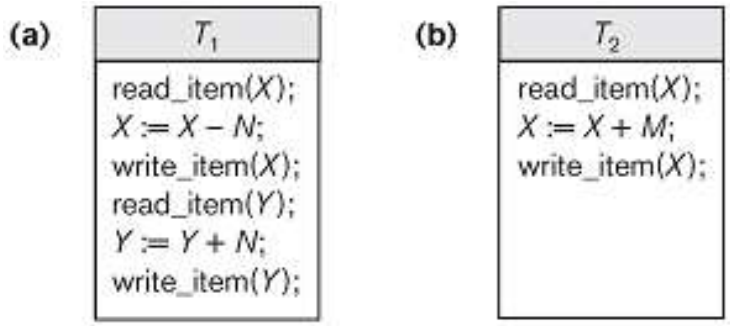
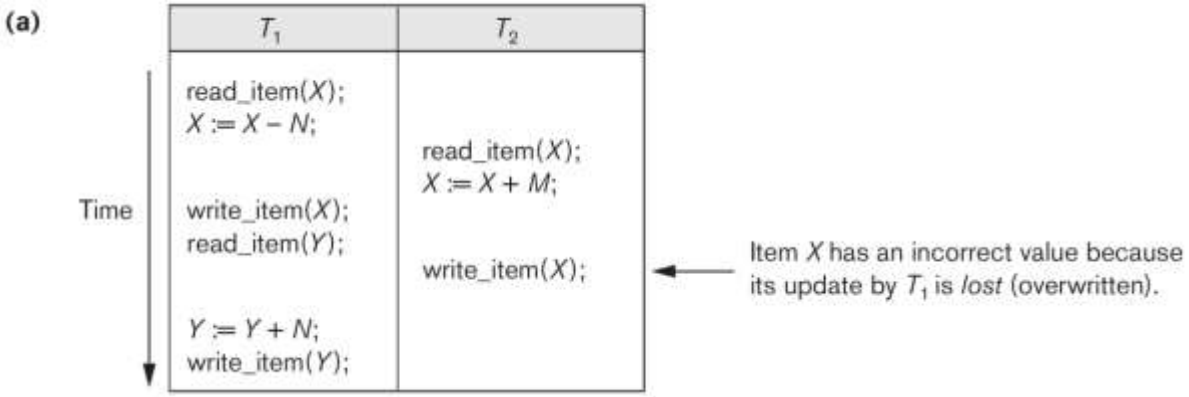


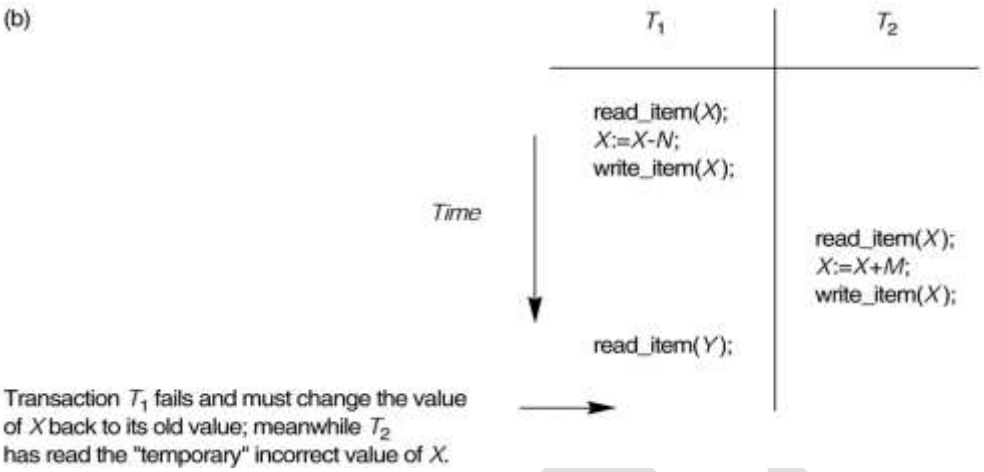
Figure 6.2: Two sample transactions. (a) Transaction *T*1. (b) Transaction *T*2.

Why Concurrency Control is Needed:

- **The Lost Update Problem:** This occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database item incorrect.



- **The Temporary Update (or Dirty Read) Problem:** This occurs when one transaction updates a database item and then the transaction fails for some reason. The updated item is accessed by another transaction before it is changed back to its original value.



- **The Incorrect Summary Problem**: 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 others after they are updated.

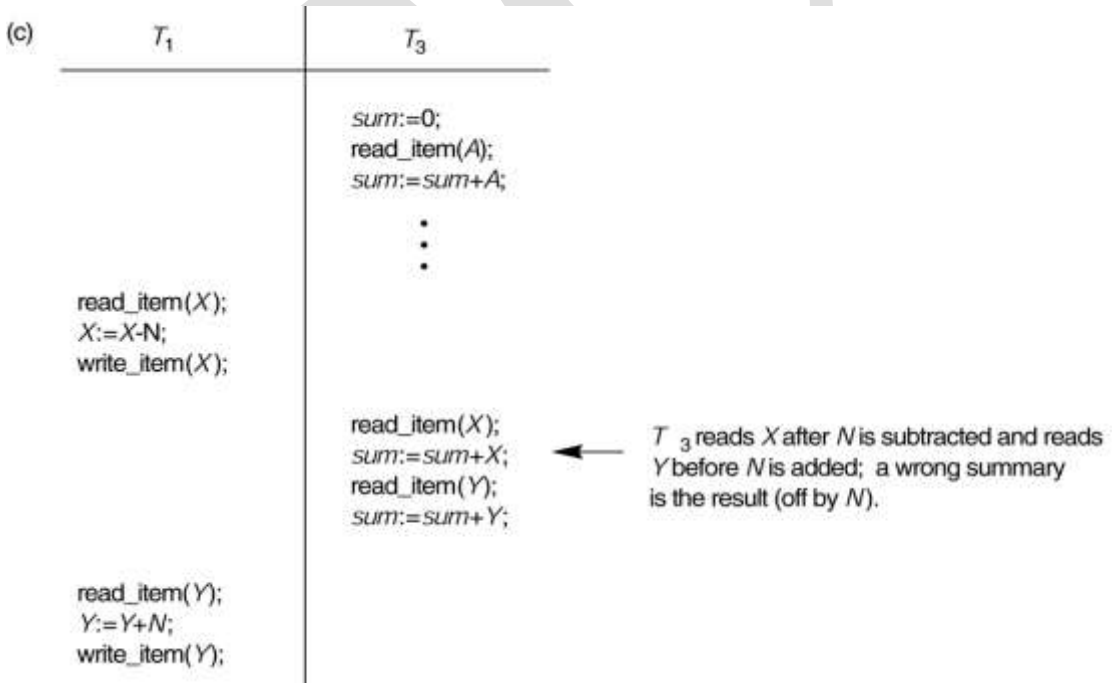


Figure 6.3: Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.

Why Recovery is Needed: (What Causes a Transaction to Fail?)

Types of Failures

1. A computer failure (system crash): A hardware or software error occurs in the computer system during transaction execution. If the hardware crashes, the contents of the computer's internal memory may be lost.

2. A transaction or system error: Some operation in the transaction may cause it to fail, such as integer overflow or division by zero. Transaction failure may also occur because of erroneous parameter values or because of a logical programming error. In addition, the user may interrupt the transaction during its execution.

3. Local errors or exception conditions detected by the transaction: Certain conditions necessitate cancellation of the transaction. For example, data for the transaction may not be found. A condition, such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal from that account, to be canceled. A programmed abort in the transaction causes it to fail.

4. Concurrency control enforcement: The concurrency control method may decide to abort the transaction, to be restarted later, because it violates serializability or because several transactions are in a state of deadlock.

5. Disk failure: Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash. This may happen during a read or a write operation of the transaction.

6. Physical problems and catastrophes: This refers to an endless list of problems that includes power or air-conditioning failure, fire, theft, sabotage, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator.

Transaction States:

- **Active state:** the initial state; the transaction stays in this statewhile it is executing.
- **Partially committed state:** after the final statement has been executed
- **Committed state:** after successful completion
- **Failed state:** after the discovery that normal execution can no longer proceed.

- **Terminated State:** end of the transaction and leaving the system.

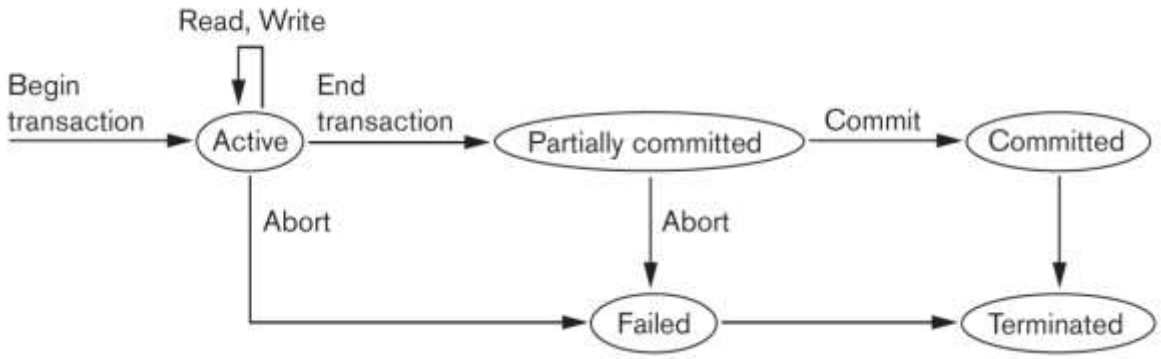


Figure 6.4: State transition diagram illustrating the states for transaction execution.

- A transaction goes into an **active state** immediately after it starts execution, where it can execute its READ and WRITE operations.
- When the transaction ends, it moves to the **partially committed state**. At this point, some recovery protocols need to ensure that a system failure will not result in an inability to record the changes of the transaction permanently.
- Once this check is successful, the transaction is said to have reached its commit point and enters the **committed state**.
- A transaction can go to the **failed state** if one of the checks fails or if the transaction is aborted during its active state.
- The transaction may then have to be rolled back to undo the effect of its WRITE operations on the database.
- The **terminated state** corresponds to the transaction leaving the system.

Operations in a Transaction

begin_transaction: This marks the beginning of transaction execution.

read or write: These specify read or write operations on the database items that are executed as part of a transaction.

end_transaction: This specifies that read and write transaction operations have ended and marks the end limit of transaction execution.

At this point it may be necessary to check whether the changes introduced by the transaction can be permanently applied to the database or whether the transaction has to be aborted because it violates concurrency control or for some other reason.

commit_transaction: This 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): This 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.

ACID Properties (Desirable 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 appear as though it is being executed in isolation from other transactions, even though many transactions are executing concurrently. That is, the execution of a transaction should not be interfered with by any other transactions executing concurrently. A transaction should not make its updates visible to other transactions until it is committed; this property, when enforced strictly, solves the temporary update problem and makes cascading rollbacks of transactions unnecessary.
- **Durability or permanency:** Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.

The System Log (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.

- T in the following discussion refers to a 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.

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.

Definition of 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.

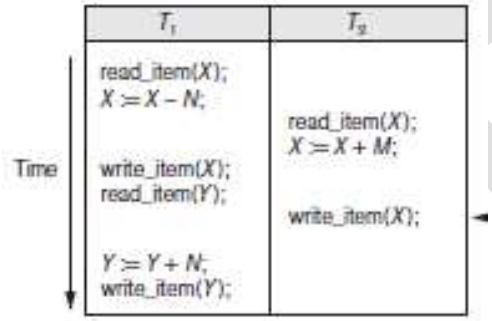
Redoing Transactions: Transactions that have written their commit entry in the log must also have recorded all their write operations in the log; otherwise they would not be committed, so their effect on the database can be *redone* from the log entries.

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.

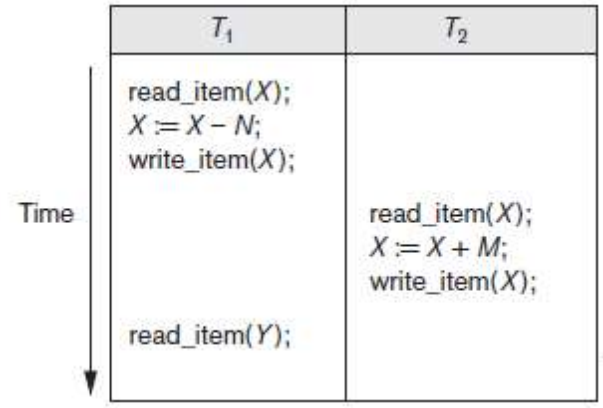
Transaction Schedule or History:

When transactions are executing concurrently in an interleaved fashion, the order of execution of operations from the various transactions forms what is known as a transaction schedule (or history). Operations from different transactions can be interleaved in the schedule S. However, for each transaction Ti that participates in the schedule S, the operations of Ti in S must appear in the same order in which they occur in Ti.

Example:



***Sa*: *r*1(*X*); *r*2(*X*); *w*1(*X*); *r*1(*Y*); *w*2(*X*); *w*1(*Y*);**



***Sb*: *r*1(*X*); *w*1(*X*); *r*2(*X*); *w*2(*X*); *r*1(*Y*); *a*1;**

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*).

For example, in schedule *Sa*, the operations *r*1(*X*) and *w*2(*X*), *r*2(*X*) and *w*1(*X*), *w*1(*X*) and *w*2(*X*) are conflict operations.

The operations $r_1(X)$ and $r_2(X)$ do not conflict, since they are both read operations; the operations $w_2(X)$ and $w_1(Y)$ do not conflict because they operate on distinct data items X and Y; and the operations $r_1(X)$ and $w_1(X)$ do not conflict because they belong to the same transaction.

Characterizing Schedules Based on Recoverability

Recoverable schedule ensure that, once a transaction *T* is committed, it should *never* be necessary to roll back *T*. A schedule *S* is recoverable if no transaction *T* in *S* commits until all transactions *T* ' that have written some item *X* that *T* reads have committed.

Example *Sa'*: *r*1(*X*); *r*2(*X*); *w*1(*X*); *r*1(*Y*); *w*2(*X*); *c*2; *w*1(*Y*); *c*1;

Sa' is recoverable, even though it suffers from the lost update problem

Example *Sc*: *r*1(*X*); *w*1(*X*); *r*2(*X*); *r*1(*Y*); *w*2(*X*); *c*2; *a*1;

Sc is not recoverable because *T*2 reads item *X* from *T*1, but *T*2 commits before *T*1 commits. The problem occurs if *T*1 aborts after the *c*2 operation in *Sc*, then the value of *X* that *T*2 read is no

longer valid and *T*2 must be aborted *after* it is committed, leading to a schedule that is *not recoverable*.

1. **Cascading Rollback** (or **cascading abort**) is a phenomenon occurring in some recoverable schedules where an *uncommitted* transaction has to be rolled back because it read an item from a transaction that failed.

Example Se: r1(X); w1(X); r2(X); r1(Y); w2(X); w1(Y); a1; a2;

In schedule Se, where transaction T2 has to be rolled back because it read item X from T1, and T1 then aborted.

2. A schedule is said to be **Cascadeless**, or to **avoid cascading rollback**, if every transaction in the schedule reads only items that were written by committed transactions.

To satisfy this criterion, the *r*2(*X*) command in schedules *Se* must be postponed until after *T*1 has committed (or aborted), thus delaying *T*2 but ensuring no cascading rollback if *T*1 aborts.

3. A **Strict Schedule**, is in which transactions can *neither read nor write* an item *X* until the last transaction that wrote *X* has committed (or aborted).

Characterizing Schedules Based on Serializability

Serializable schedules are always considered to be *correct* when concurrent transactions are executing.

Serial Schedules: A schedule S is serial if, for every transaction T participating in the schedule, all the operations of T are executed consecutively in the schedule. Otherwise, the schedule is called **Nonserial Schedule**.

- In a serial schedule, only one transaction at a time is active—the commit (or abort) of the active transaction initiates execution of the next transaction.
- No interleaving occurs in a serial schedule.
- Every serial schedule is considered correct.

If no interleaving of operations is permitted, there are only two possible outcomes:

1. Execute all the operations of transaction T1 (in sequence) followed by all the operations of transaction T2 (in sequence).

2. Execute all the operations of transaction T2 (in sequence) followed by all the operations of transaction T1 (in sequence).

These two schedules are called ***serial schedules.***

- If interleaving of operations is allowed, there will be many possible orders in which the system can execute the individual operations of the transactions.

Problems with Serial Schedules

- Limit concurrency by prohibiting interleaving of operations.
- Wastage of valuable CPU processing time.
- If some transaction T is quite long, the other transactions must wait for T to complete all its operations before starting.
- Serial schedules are considered unacceptable in practice.

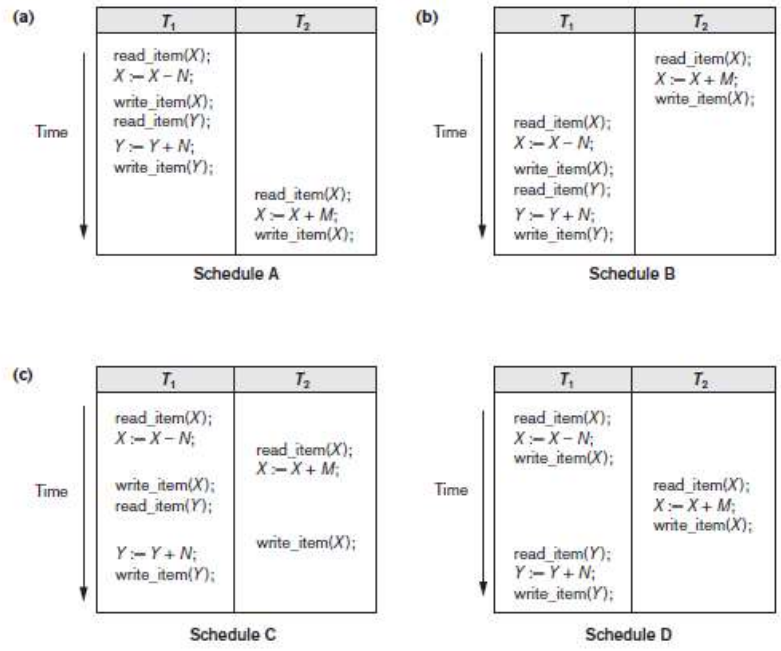


Figure 6.5: Examples of serial and nonserial schedules involving transactions *T*1 and *T*2. (a) Serial schedule A: *T*1 followed by *T*2. (b) Serial schedule B: *T*2 followed by *T*1. (c) Two nonserial schedules C and D with interleaving of operations.

Serializability

The concept of serializability of schedules is used to identify which schedules are correct when transaction executions have interleaving of their operations in the schedules.

Serializable schedule: A schedule S is serializable if it is equivalent to some serial schedule of the same n transactions.

Result equivalent: Two schedules are called result equivalent if they produce the same final state of the database.

Being serializable is not the same as being serial. Being serializable implies that the schedule is a correct schedule.

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

Conflict serializable: A schedule S is said to be conflict serializable if it is conflict equivalent to some serial schedule S'. In such a case, we can reorder the non-conflicting operations in S until we form the equivalent serial schedule S'.

Testing for Conflict Serializability of a Schedule

1. For each transaction *Ti* participating in schedule *S*, create a node labeled *Ti* in the precedence graph.

2. For each case in *S* where *Tj* executes a read_item(*X*) after *Ti* executes a write_item(*X*), create an edge (*Ti*→ *Tj*) in the precedence graph.

3. For each case in *S* where *Tj* executes a write_item(*X*) after *Ti* executes a read_item(*X*), create an edge (*Ti*→*Tj*) in the precedence graph.

4. For each case in *S* where *Tj* executes a write_item(*X*) after *Ti* executes a write_item(*X*), create an edge (*Ti*→ *Tj*) in the precedence graph.

5. The schedule *S* is serializable if and only if the precedence graph has no cycles.

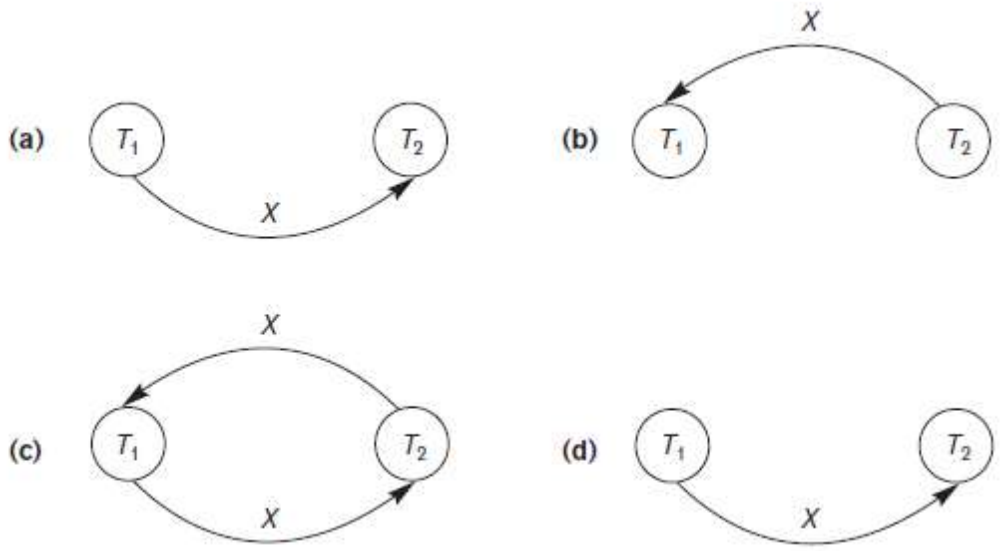


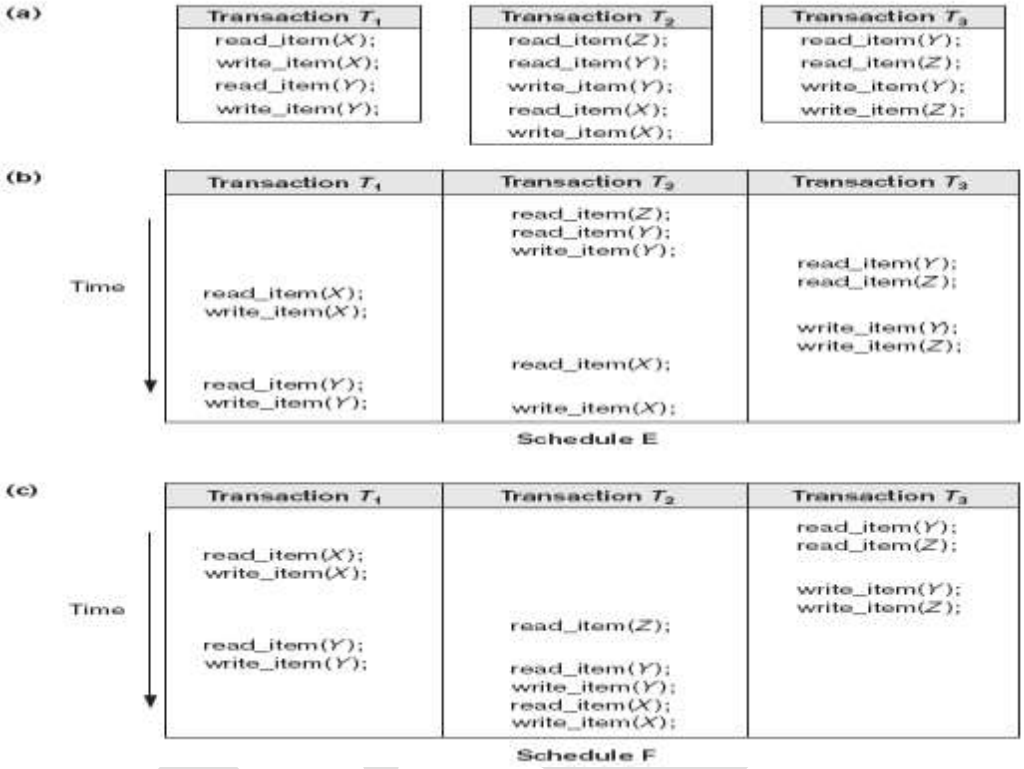
Figure 6.6: Constructing the precedence graphs for schedules A to D from Figure 6.5 to test for conflict serializability.

(a) Precedence graph for serial schedule A. (b) Precedence graph for serial schedule B.

(c) Precedence graph for schedule C (not serializable).

(d) Precedence graph for schedule D (serializable, equivalent to schedule A).

Another Example



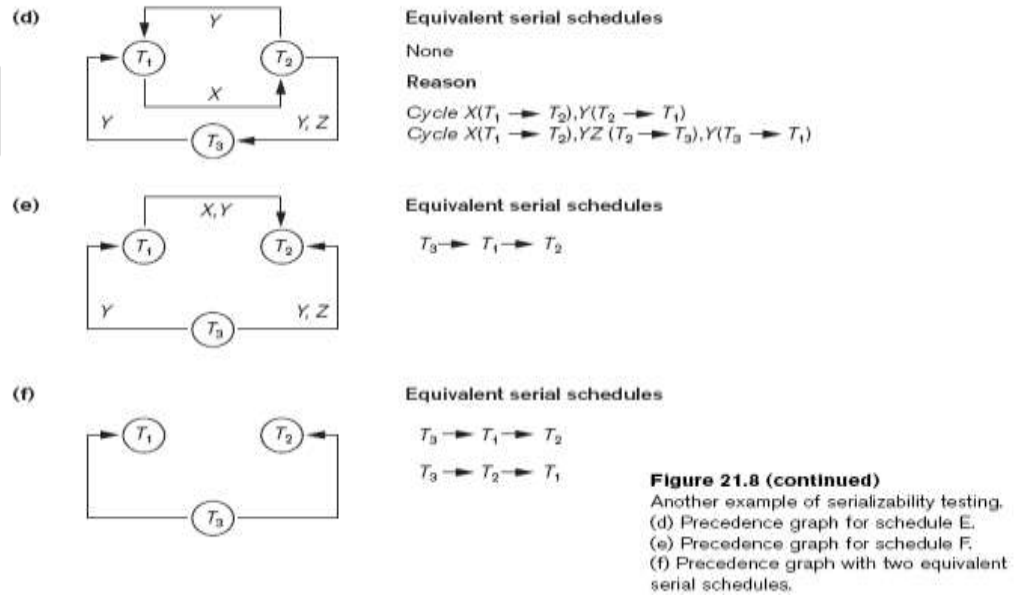


Figure 21.8 (continued)
Another example of serializability testing.
(d) Precedence graph for schedule E.
(e) Precedence graph for schedule F.
(f) Precedence graph with two equivalent serial schedules.

Concurrency Control Techniques

Employ the technique of **locking** data items to prevent multiple transactions from accessing the items concurrently. 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.

- Lock guarantees current transaction exclusive use of data item, i.e., transaction T2 does not have access to a data item that is currently being used by transaction T1.
- Acquires lock prior to access.
- Lock released when transaction is completed.
- DBMS automatically initiates and enforces locking procedures.
- All lock information is managed by lock manager.

Binary Locks: A **binary lock** can have two **states** or **values:** locked and unlocked (or 1 and 0, for simplicity). A distinct lock is associated with each database item *X*. If the value of the lock on *X* is 1, item *X cannot be accessed* by a database operation that requests the item. If the value of the lock on *X* is 0, the item can be accessed when requested, and the lock value is changed to 1. Two operations, lock_item and unlock_item, are used with binary locking. A binary lock enforces **mutual exclusion** on the data item.

If the simple binary locking scheme described here is used, every transaction must obey the following rules:

1. A transaction *T* must issue the operation lock_item(*X*) before any read_item(*X*) or write_item(*X*) operations are performed in *T*.

2. A transaction *T* must issue the operation unlock_item(*X*) after all read_item(*X*) and write_item(*X*) operations are completed in *T*.

3. A transaction *T* will not issue a lock_item(*X*) operation if it already holds the lock on item *X*.

4. A transaction *T* will not issue an unlock_item(*X*) operation unless it already holds the lock on item *X*.

Shared/Exclusive (or Read/Write) Locks: Here several transactions to access the same item *X* if they all access *X* for *reading purposes only*. This is because read operations on the same item by different transactions are not conflicting. However, if a transaction is to write an item *X*, it must have exclusive access to *X*. For this purpose, a different type of lock called a **multiple-mode lock** is used. In this scheme, there are three locking operations: read_lock(*X*), write_lock(*X*), and unlock(*X*). A **read-locked item** is also called **share-locked** because other transactions are allowed to read the item, whereas a **write-locked item** is called **exclusive-locked** because a single transaction exclusively holds the lock on the item.

When we use the shared/exclusive locking scheme, the system must enforce the following rules:

1. A transaction *T* must issue the operation read_lock(*X*) or write_lock(*X*) before any read_item(*X*) operation is performed in *T*.

2. A transaction *T* must issue the operation write_lock(*X*) before any write_item(*X*) operation is performed in *T*.

3. A transaction *T* must issue the operation unlock(*X*) after all read_item(*X*) and write_item(*X*) operations are completed in *T*.

4. A transaction *T* will not issue a read_lock(*X*) operation if it already holds a read (shared) lock or a write (exclusive) lock on item *X*.

5. A transaction *T* will not issue a write_lock(*X*) operation if it already holds a read (shared) lock or write (exclusive) lock on item *X*.

6. A transaction *T* will not issue an unlock(*X*) operation unless it already holds a read (shared) lock or a write (exclusive) lock on item *X*.

Conversion of Locks. Sometimes it is desirable to relax conditions 4 and 5 in the preceding list in order to allow **lock conversion**; that is, a transaction that already holds a lock on item *X* is allowed under certain conditions to **convert** the lock from one locked state to another. For example, it is possible for a transaction *T* to issue a read_lock(*X*) and then later to **upgrade** the lock by issuing a write_lock(*X*) operation. If *T* is the only transaction holding a read lock on *X* at the time it issues the write_lock(*X*) operation, the lock can be upgraded; otherwise, the transaction

must wait. It is also possible for a transaction T to issue a write_lock(X) and then later to **downgrade** the lock by issuing a read_lock(X) operation.

Two Phase Locking

A transaction is said to follow the **two-phase locking protocol** if *all* locking operations (read_lock, write_lock) precede the *first* unlock operation in the transaction. Such a transaction can be divided into two phases: an **expanding** or **growing (first) phase**, during which new locks on items can be acquired but none can be released; and a **shrinking (second) phase**, during which existing locks can be released but no new locks can be acquired. If lock conversion is allowed, then upgrading of locks (from read-locked to write-locked) must be done during the expanding phase, and downgrading of locks (from write-locked to read-locked) must be done in the shrinking phase. Hence, a read_lock(X) operation that downgrades an already held write lock on X can appear only in the shrinking phase.

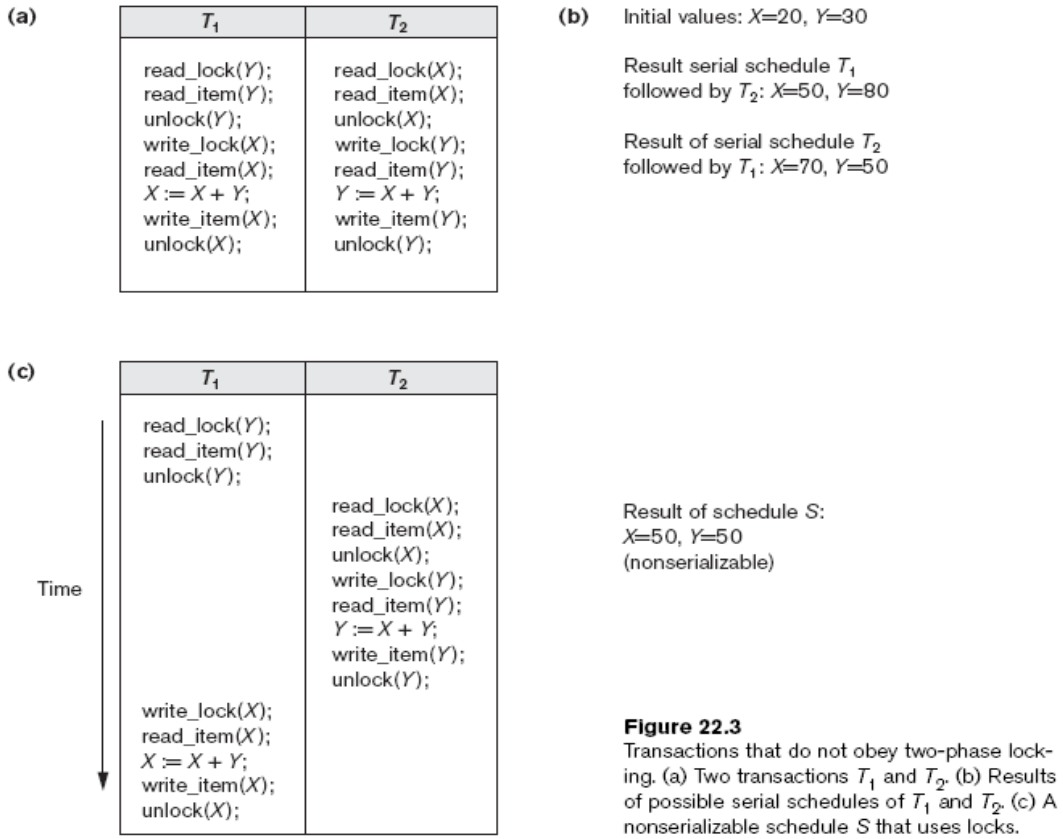


Figure 22.3
Transactions that do not obey two-phase locking. (a) Two transactions T_1 and T_2. (b) Results of possible serial schedules of T_1 and T_2. (c) A nonserializable schedule S that uses locks.

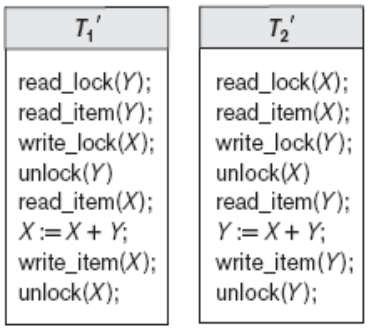


Figure 22.4
Transactions T_1' and T_2', which are the same as T_1 and T_2 in Figure 22.3, but follow the two-phase locking protocol. Note that they can produce a deadlock.

Basic, Conservative, Strict, and Rigorous Two-Phase Locking:

There are a number of variations of two-phase locking (2PL). The technique just described is known as **basic 2PL**.

A variation known as **conservative 2PL** (or **static 2PL**) requires a transaction to lock all the items it accesses *before the transaction begins execution*, by **predeclaring** its *read-set* and *write-set*. If any of the predeclared items needed cannot be locked, the transaction does not lock any item; instead, it waits until all the items are available for locking. Conservative 2PL is a deadlock-free protocol.

In **strict 2PL**, a transaction *T* does not release any of its exclusive (write) locks until *after* it commits or aborts. Hence, no other transaction can read or write an item that is written by *T* unless *T* has committed, leading to a strict schedule for recoverability. Strict 2PL is not deadlock-free.

In **rigorous 2PL** a transaction *T* does not release any of its locks (exclusive or shared) until after it commits or aborts, and so it is easier to implement than strict 2PL. Notice the difference between conservative and rigorous 2PL: the former must lock all its items *before it starts*, so once the transaction starts it is in its shrinking phase; the latter does not unlock any of its items until *after it terminates* (by committing or aborting), so the transaction is in its expanding phase until it ends.

Storage Structure

- **Volatile storage**. Information residing in volatile storage does not usually survive system crashes. Examples of such storage are main memory and cache memory. Access to volatile storage is extremely fast, both because of the speed of the memory access itself, and because it is possible to access any data item in volatile storage directly.

- **Nonvolatile storage**. Information residing in nonvolatile storage survives system crashes. Examples of nonvolatile storage include secondary storage devices such as magnetic disk and flash storage, used for online storage, and tertiary storage devices such as optical media, and magnetic tapes, used for archival storage. At the current state of technology, nonvolatile storage is slower than volatile storage, particularly for random access. Both secondary and tertiary storage devices, however, are susceptible to failure which may result in loss of information.
- **Stable storage**. Information residing in stable storage is *never* lost. Although stable storage is theoretically impossible to obtain, it can be closely approximated by techniques that make data loss extremely unlikely. To implement stable storage, we replicate the information in several non-volatile storage media (usually disk) with independent failure modes. Updates must be done with care to ensure that a failure during an update to stable storage does not cause a loss of information.

Log Based Recovery

The most widely used structure for recording database modifications is the **log**. The log is a sequence of **log records**, recording all the update activities in the database. There are several types of log records. An **update log record** describes a single database write. It has these fields:

- **Transaction identifier**, which is the unique identifier of the transaction that performed the write operation.
- **Data-item identifier**, which is the unique identifier of the data item written.
- **Old value**, which is the value of the data item prior to the write.
- **New value**, which is the value that the data item will have after the write.

We represent an update log record as $<Ti , Xj , V1, V2>$, indicating that transaction Ti has performed a write on data item Xj . Xj had value $V1$ before the write, and has value $V2$ after the write.

Whenever a transaction performs a write, it is essential that the log record for that write be created and added to the log, before the database is modified. Once a log record exists, we can output the

modification to the database if that is desirable. Also, we have the ability to *undo* a modification that has already been output to the database. We undo it by using the old-value field in log records. For log records to be useful for recovery from system and disk failures, the log must reside in stable storage. For now, we assume that every log record is written to the end of the log on stable storage as soon as it is created. There are two approaches of recovery using logs, they are Deferred database modification and Immediate database modification.

Deferred Database Modification

If a transaction does not modify the database until it has committed, it is said to use the **deferred-modification** technique. The deferred database modification scheme records all modifications to the log, but defers all the **write**s to after partial commit. Deferred modification has the overhead that transactions need to make local copies of all updated data items; further, if a transaction reads a data item that it has updated, it must read the value from its local copy.

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, V1,V2*> being written, where *V2* is the new value for *X*.

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 <*Ti* **start**> and<*Ti* **commit**> are there in the log, otherwise ignore the log records.

Immediate Database Modification

If database modifications occur while the transaction is still active, the transaction is said to use the **immediate-modification** technique. Here the recovery procedure has two operations instead of one:

- undo(*T*i) restores the value of all data items updated by *Ti* to their old values, going backwards from the last log record for *Ti*

- redo(*T*i) sets the value of all data items updated by *Ti* to the new values, going forward from the first log record for *Ti*

When recovering after failure:

- Transaction *Ti* needs to be undone if the log contains the record <*Ti* **start**>, but does not contain the record <*Ti* **commit**>.
- Transaction *Ti* needs to be redone if the log contains both the record <*Ti* **start**> and the record <*Ti* **commit**>.

Undo operations are performed first, then redo operations.

Check-Pointing

The problems in recovery procedure are:

- searching the entire log is time-consuming
- we might unnecessarily redo transactions which have already output their updates to the database.

We can streamline recovery procedure by periodically performing checkpointing

- Output all log records currently residing in main memory onto stable storage.
- Output all modified buffer blocks to the disk.
- Write a log record < checkpoint> onto stable storage.

During recovery we need to consider only the most recent transaction Ti that started before the checkpoint, and transactions that started after *Ti*.

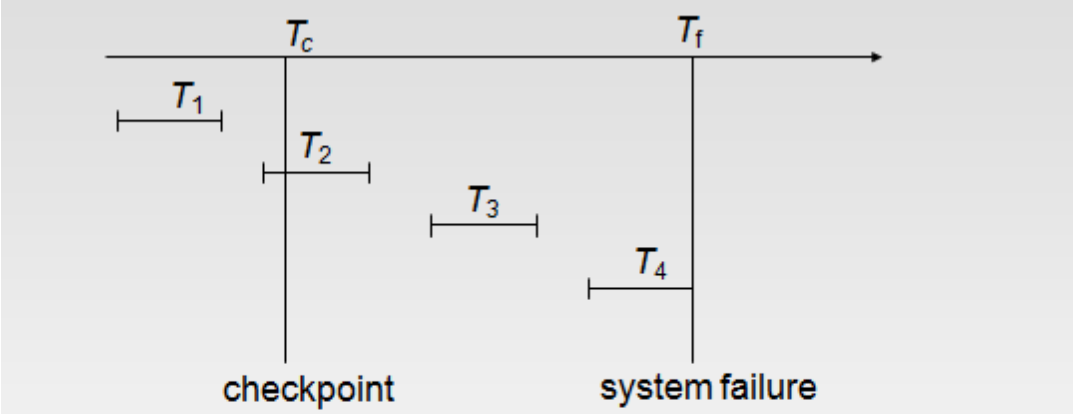
1. Scan backwards from end of log to find the most recent <checkpoint> record

2. Continue scanning backwards till a record <*Ti* start> is found.

3. Need only consider the part of log following above start record. Earlier part of log can be ignored during recovery, and can be erased whenever desired.

4. For all transactions (starting from *Ti* or later) with no <*Ti* commit>, execute undo*(Ti).* (Done only in case of immediate modification.)

5. Scanning forward in the log, for all transactions starting from *Ti* or later with a <*Ti* commit>, execute redo*(Ti)*.



- T_1 can be ignored (updates already output to disk due to checkpoint)
- T_2 and T_3 redone.
- T_4 undone

try it now

A KTU STUDENTS PLATFORM

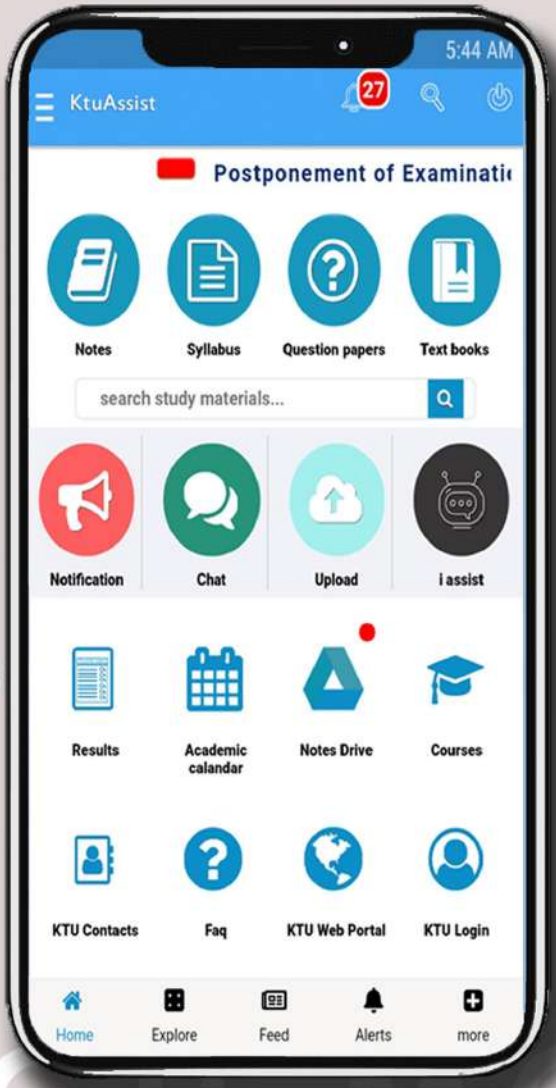
SYLLABUS

NOTES

TEXT BOOKS

QUESTION PAPERS

KTU NOTIFICATION



DOWNLOAD IT FROM GOOGLE PLAY

CHAT

FAQ

LOGIN

CALENDAR

MUCH MORE

DOWNLOAD APP

ktuassist.in

instagram.com/ktu_assist

facebook.com/ktuassist