Module 6

Transaction Processing Concepts

A **transaction** is a logical unit of database processing that includes one or more database access operations—these can include insertion, deletion, modification, or retrieval operations.

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. To simplify our notation, we assume that the program variable is also named X.
- **write_item(X)**: Writes the value of program variable X into the database item named X.

Concurrency control

KTU STUDENTS

Why we need concurrency control?

The *Lost Update* Problem

This problem occurs when two transactions that access the same database items have their operations interleaved in a way that makes the *update* operation of one of the transactions *lost* .

The *Temporary Update* (or *Dirty Read*) Problem

This problem 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.

The U*nrepeatable Read* Problem

For more study materials>www.ktustudents.in

When a transaction T reads an item twice and the item is changed by another transaction T' between the two reads; hence, T receives different values for its two reads of the same item.

Recovery

Whenever a failure occurs, the system must keep sufficient information to recover from the failure.

The **recovery manager** keeps track of the following operations:

1. BEGIN_TRANSACTION
2. READ or WRITE
3. END_TRANSACTION
4. 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.
5. 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.

A **system log** is a disk file that keeps track of all transaction operations that affect the values of database items.

KTU STUDENTS

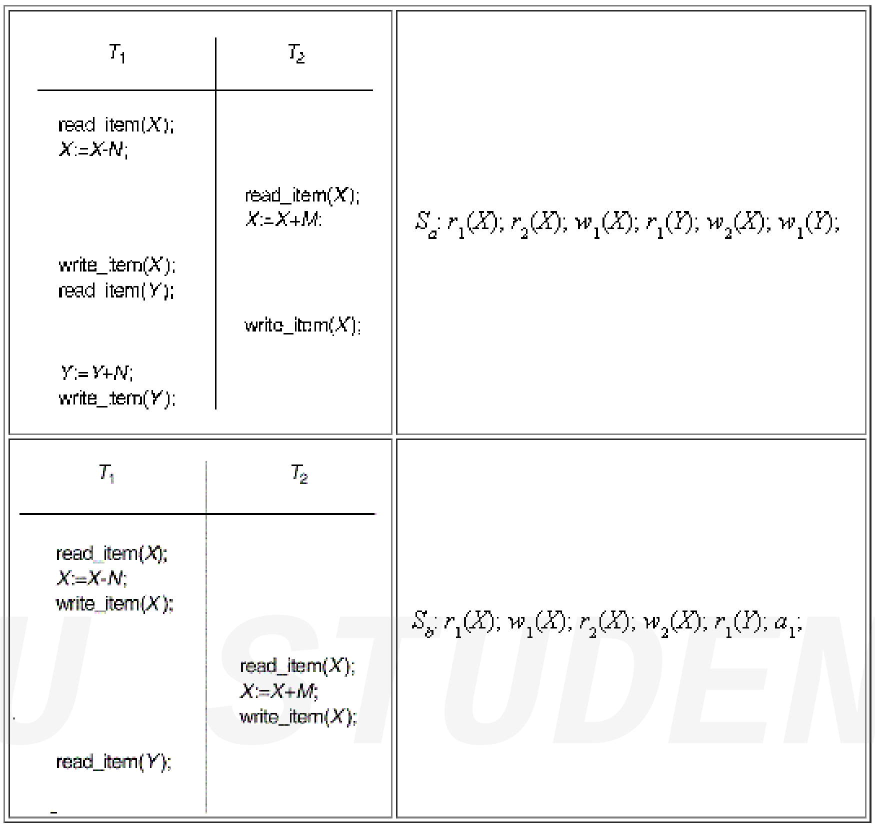
Acid properties

1. **Atomicity**: A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all.
2. **Consistency preservation** : A transaction is consistency preserving if its complete execution take(s) the database from one consistent state to another.
3. **Isolation**: A transaction should appear as though it is being executed in isolation from other transactions. That is, the execution of a transaction should not be interfered with by any other transactions executing concurrently.
4. **Durability** or **permanency**: The changes applied to the database by a committed transaction must persist in the database. These changes must not be lost because of any failure.

For more study materials>www.ktustudents.in

Serial and concurrent schedules

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



T_1	T_2
read_item(X);	
X:=X-N;	
	read_item(X);
	X:=X+M;
write_item(X);	
read_item(Y);	
	write_item(X);
Y:=Y+N;	
write_item(Y);	

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

T_1	T_2
read_item(X);	
X:=X-N;	
write_item(X);	
	read_item(X);
	X:=X+M;
	write_item(X);
read_item(Y);	

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

KTU STUDENTS

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
3. At least one of the operations is a write_item(X).

In a recoverable schedule, no committed transaction ever needs to be rolled back.

Cascading rollback (or cascading abort) is where an *uncommitted* transaction has to be rolled back because it read an item from a transaction that failed.

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

For more study materials>www.ktustudents.in

A **strict schedule** are composed of transactions that can neither read nor write an item X until the last transaction that wrote X has committed (or aborted).

A *strict* schedule is both *cascadeless* and *recoverable*. A *cascadeless* schedule is a *recoverable* schedule.

Serializability of Schedules

In a serial schedule, the operations of each transaction are executed consecutively, without any interleaved operations from other transactions.

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

Conflict serializability

- Two schedules are said to be **conflict equivalent** if the order of any two *conflicting* operations is the same in both schedules.
- A schedule S is **conflict serializable** if it is *conflict equivalent* to some serial schedule S´.

KTU STUDENTS

ALGORITHM 19.1 Testing conflict serializability of a schedule S.

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

If there is a cycle in the precedence graph, schedule S is not (conflict) serializable; if there is no cycle, S is serializable.

In the precedence graph, an edge from T_i to T_j means that transaction T_i must come before transaction T_j in any serial schedule that is equivalent to S, because two conflicting operations appear in the schedule in that order.

For more study materials>www.ktustudents.in

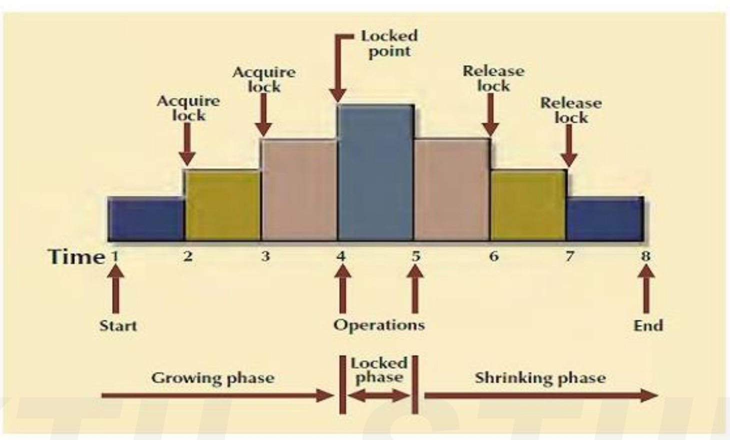
Two phase locking

Two-phase locking guarantees serializability, but it does not prevent deadlocks. The two phases are:

1. A growing phase, in which a transaction acquires all required locks without unlocking any data. Once all locks have been acquired, the transaction is in its locked point.

2. A shrinking phase, in which a transaction releases all locks and cannot obtain any new lock.

The two-phase locking protocol is governed by the following rules:
• Two transactions cannot have conflicting locks.



The two-phase locking protocol

• No unlock operation can precede a lock operation in the same transaction.

• No data are affected until all locks are obtained—that is, until the transaction is in its locked point.

The two-phase locking protocol

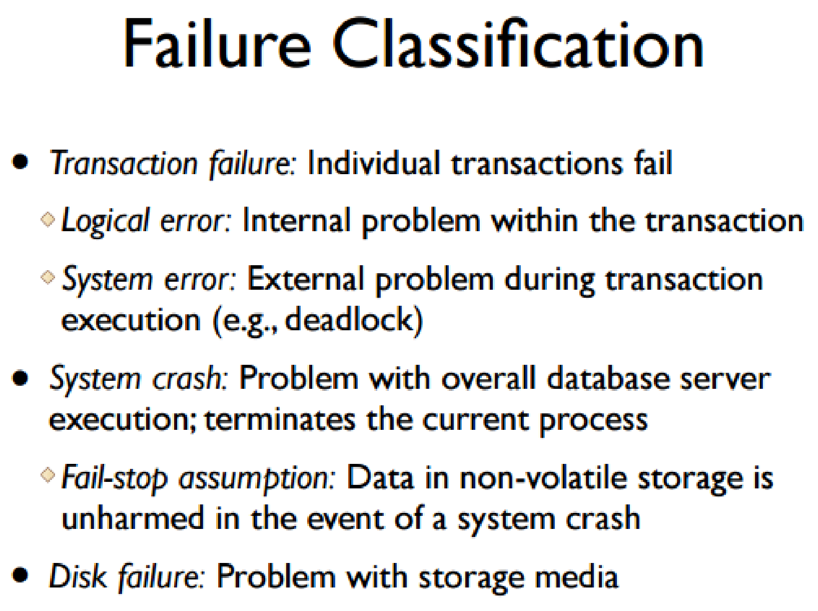
KTU STUDENTS

In this example, the transaction acquires all of the locks it needs until it reaches its locked point. (In this example, the transaction requires two locks.) When the locked point is reached, the data are modified to conform to the transaction’s requirements. Finally, the transaction is completed as it releases all of the locks it acquired in the first phase.

Two-phase locking increases the transaction processing cost and might cause additional undesirable effects. One undesirable effect is the possibility of creating deadlocks.

For more study materials>www.ktustudents.in

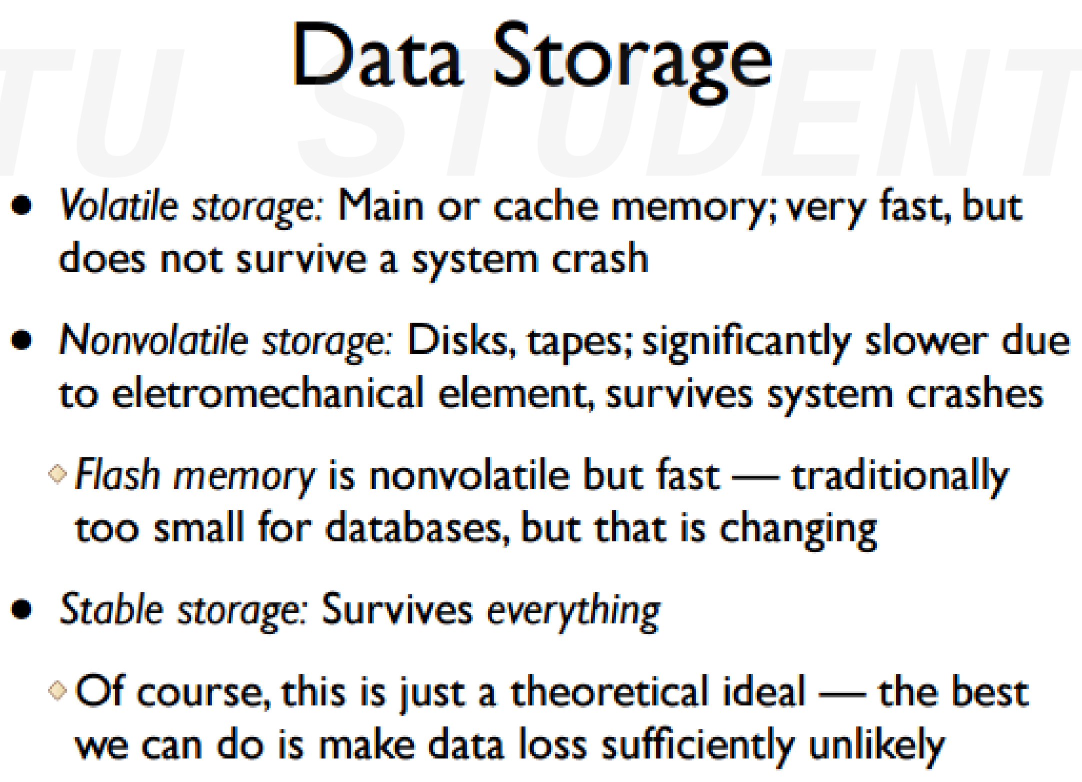
Failure classification



Failure Classification

- *Transaction failure:* Individual transactions fail
  - *Logical error:* Internal problem within the transaction
  - *System error:* External problem during transaction execution (e.g., deadlock)
- *System crash:* Problem with overall database server execution; terminates the current process
  - *Fail-stop assumption:* Data in non-volatile storage is unharmed in the event of a system crash
- *Disk failure:* Problem with storage media

Storage structure



Data Storage

- *Volatile storage:* Main or cache memory; very fast, but does not survive a system crash
- *Nonvolatile storage:* Disks, tapes; significantly slower due to eletromechanical element, survives system crashes
  - *Flash memory* is nonvolatile but fast — traditionally too small for databases, but that is changing
- *Stable storage:* Survives *everything*
  - Of course, this is just a theoretical ideal — the best we can do is make data loss sufficiently unlikely

KTU STUDENTS

For more study materials>www.ktustudents.in

Stable storage

- We can approximate stable storage through redundant copies, copies at different physical locations, and controlled updates of these copies
  - ◇ Write to one copy first; upon successful initial write, then write to second copy
  - ◇ Recovery involves comparing the copies, and replacing the copy with a detectable error (i.e., checksum) or replacing the copy that was written first if no detectable error
- Data access involves different layers of storage space: *physical blocks* on disk, *buffer blocks* in a designated main memory *disk buffer*, and private *work areas* for each transaction, also in main memory
  - ◇ *input(B)* and *output(B)* transfer between physical and buffer blocks
  - ◇ *read(X)* and *write(X)* transfer between buffer blocks and the work area, possibly triggering *input(B_X)* automatically
  - ◇ A *buffer manager* decides when buffer blocks should be written back to physical blocks
  - ◇ When needed, the database system can perform a *force-output* with a direct *output(B)* call

KTU STUDENTS

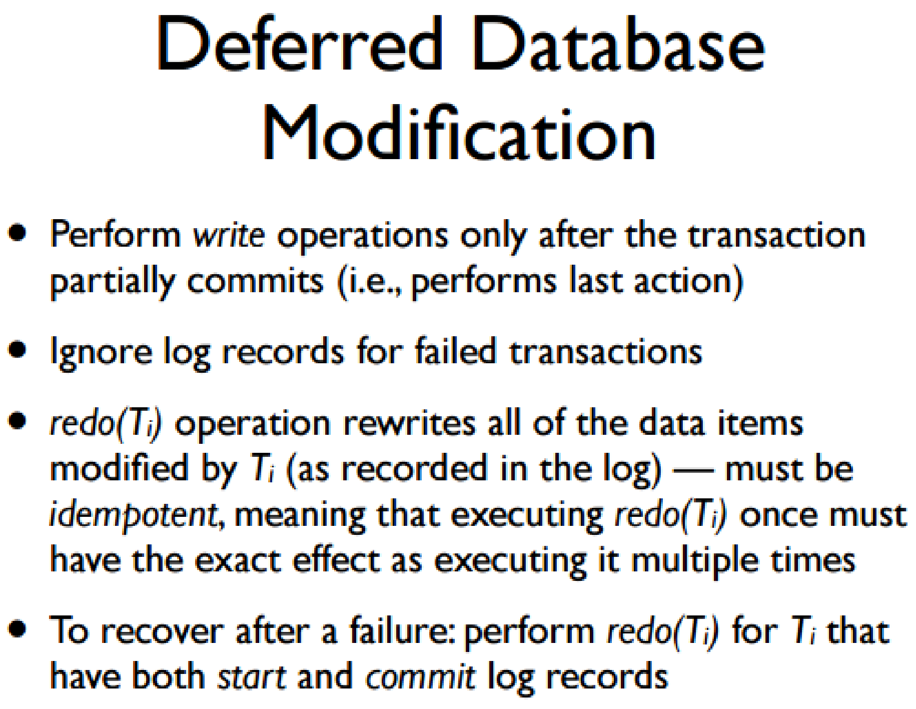
Log based recovery

Log-Based Recovery

- *Logs* are the most widely-used approach for recording database modifications
- A log consists of a sequence of *log records*
- A database write is recorded in an *update log record*, which consists of:
  - ◇ *Transaction identifier* indicates the writing transaction
  - ◇ *Data-item identifier* indicates what was written (typically on-disk location of item)
  - ◇ *Old, new values* of the item before and after the write

For more study materials>www.ktustudents.in

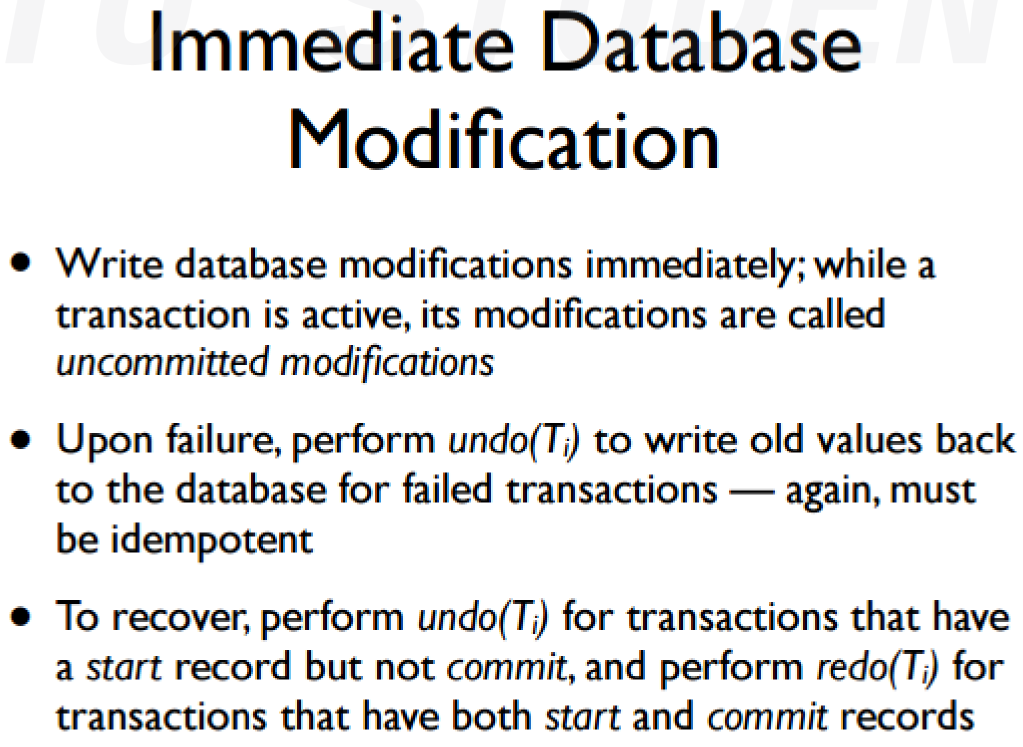
Deferred database modification



Deferred Database Modification

- Perform *write* operations only after the transaction partially commits (i.e., performs last action)
- Ignore log records for failed transactions
- *redo(T_i)* operation rewrites all of the data items modified by T_i (as recorded in the log) — must be *idempotent*, meaning that executing *redo(T_i)* once must have the exact effect as executing it multiple times
- To recover after a failure: perform *redo(T_i)* for T_i that have both *start* and *commit* log records

KTU STUDENTS

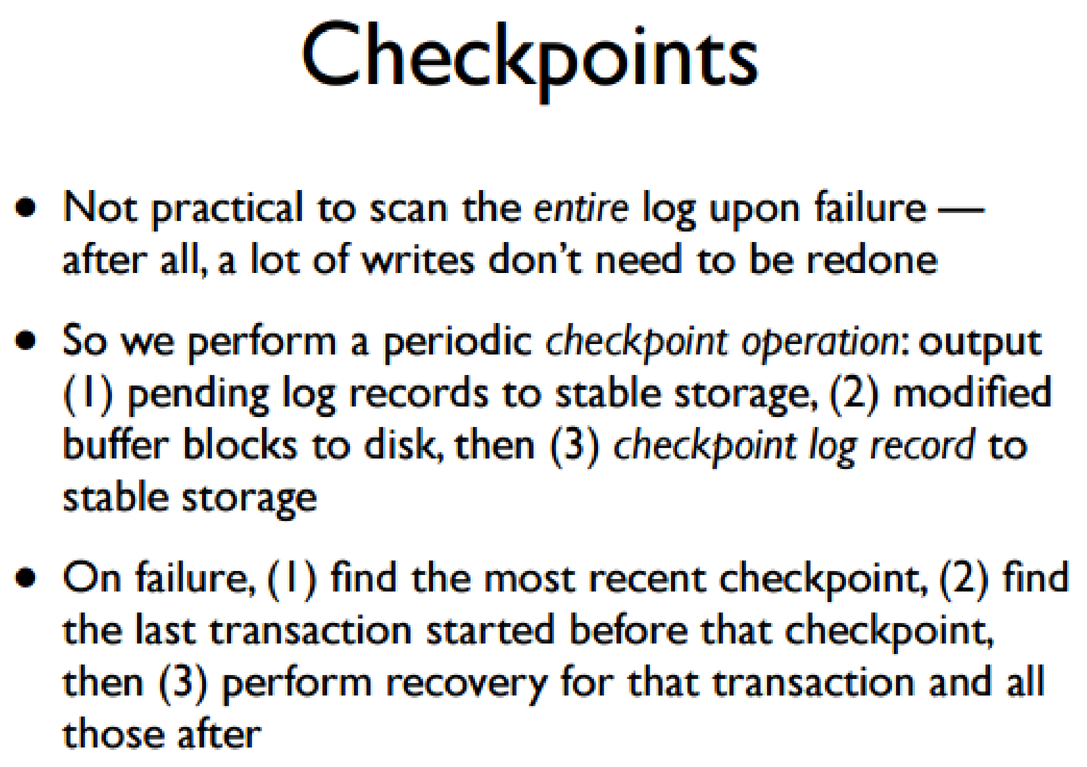


Immediate Database Modification

- Write database modifications immediately; while a transaction is active, its modifications are called *uncommitted modifications*
- Upon failure, perform *undo(T_i)* to write old values back to the database for failed transactions — again, must be idempotent
- To recover, perform *undo(T_i)* for transactions that have a *start* record but not *commit*, and perform *redo(T_i)* for transactions that have both *start* and *commit* records

For more study materials>www.ktustudents.in

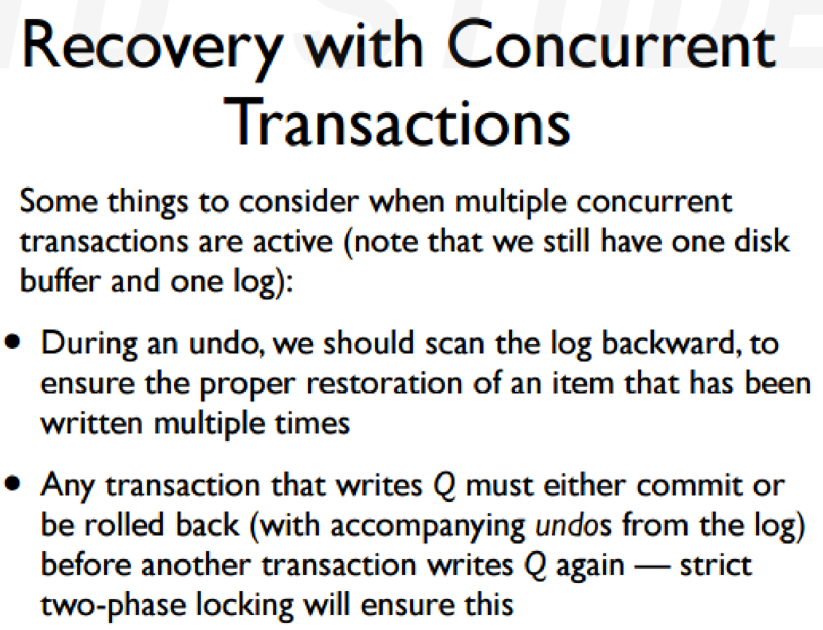
Check pointing



Checkpoints

- Not practical to scan the *entire* log upon failure — after all, a lot of writes don't need to be redone
- So we perform a periodic *checkpoint operation*: output (1) pending log records to stable storage, (2) modified buffer blocks to disk, then (3) *checkpoint log record* to stable storage
- On failure, (1) find the most recent checkpoint, (2) find the last transaction started before that checkpoint, then (3) perform recovery for that transaction and all those after

KTU STUDENTS



Recovery with Concurrent Transactions

Some things to consider when multiple concurrent transactions are active (note that we still have one disk buffer and one log):

- During an undo, we should scan the log backward, to ensure the proper restoration of an item that has been written multiple times
- Any transaction that writes Q must either commit or be rolled back (with accompanying *undos* from the log) before another transaction writes Q again — strict two-phase locking will ensure this

For more study materials>www.ktustudents.in

- Checkpoint log records must include a list of active transactions at the time of the checkpoint — otherwise, we wouldn't know how when to stop scanning the log
- Recovery algorithm then requires two passes: one to determine the transactions to undo and redo, and another to perform the actual undo and redo
  - ◇ Scan the log backward, up to the most recent checkpoint log record
  - ◇ Every committed transaction goes to the *redo-list*
  - ◇ Every started transaction not already in the redo-list goes to the undo-list
  - ◇ Transactions in the checkpoint log record not already in the redo-list go to the undo-list
- The second pass then performs the data modifications:
  - ◇ Scanning backward, undo all updates by transactions in the undo-list
  - ◇ Going forward from the most recent checkpoint log record, redo all updates made by transactions in the redo-list

KTU STUDENTS

For more study materials>www.ktustudents.in