Advanced DBMS

Unit-1

Unit-1

- SQL
- PL/SQL
- Transaction Serializability
- Locking
- DB Architecture - Transaction model and properties
- Transaction Structure
- Transaction Serialization
- Concurrency Control and Recovery

Transaction and ACID properties

A **transaction** is a unit of program execution that accesses and possibly updates various data items. To preserve the integrity of data the database system must ensure:

Atomicity: Either all operations of the transaction are properly reflected in the database or none are.

Consistency: Execution of a transaction in isolation preserves the consistency of the database.

Isolation: Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.

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

Transaction Processing

- e.g., transaction to transfer $50 from account A to account B:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B)*
- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

ACID properties

Consider the transaction shown in the above slide

- **Atomicity requirement**
  - If the transaction fails after step 3 and before step 6, money will be “lost” leading to an inconsistent database state
    - Failure could be due to software or hardware
  - The system should ensure that updates of a partially executed transaction are not reflected in the database

- **Durability requirement** — once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.

Transaction Processing

- **Consistency requirement** in above example:
  - The sum of A and B is unchanged by the execution of the transaction
- In general, consistency requirements include
    - Explicitly specified integrity constraints such as primary keys and foreign keys
    - Implicit integrity constraints
      - e.g., sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand
- A transaction, when starting to execute, must see a consistent database.
- During transaction execution the database may be temporarily inconsistent.
- When the transaction completes successfully the database must be consistent
  - Erroneous transaction logic can lead to inconsistency

Transaction Processing

- **Isolation requirement** — if between steps 3 and 6 (of the fund transfer transaction) , another transaction **T2** is allowed to access the partially updated database, it will see an inconsistent database (the sum $A + B$ will be less than it should be).

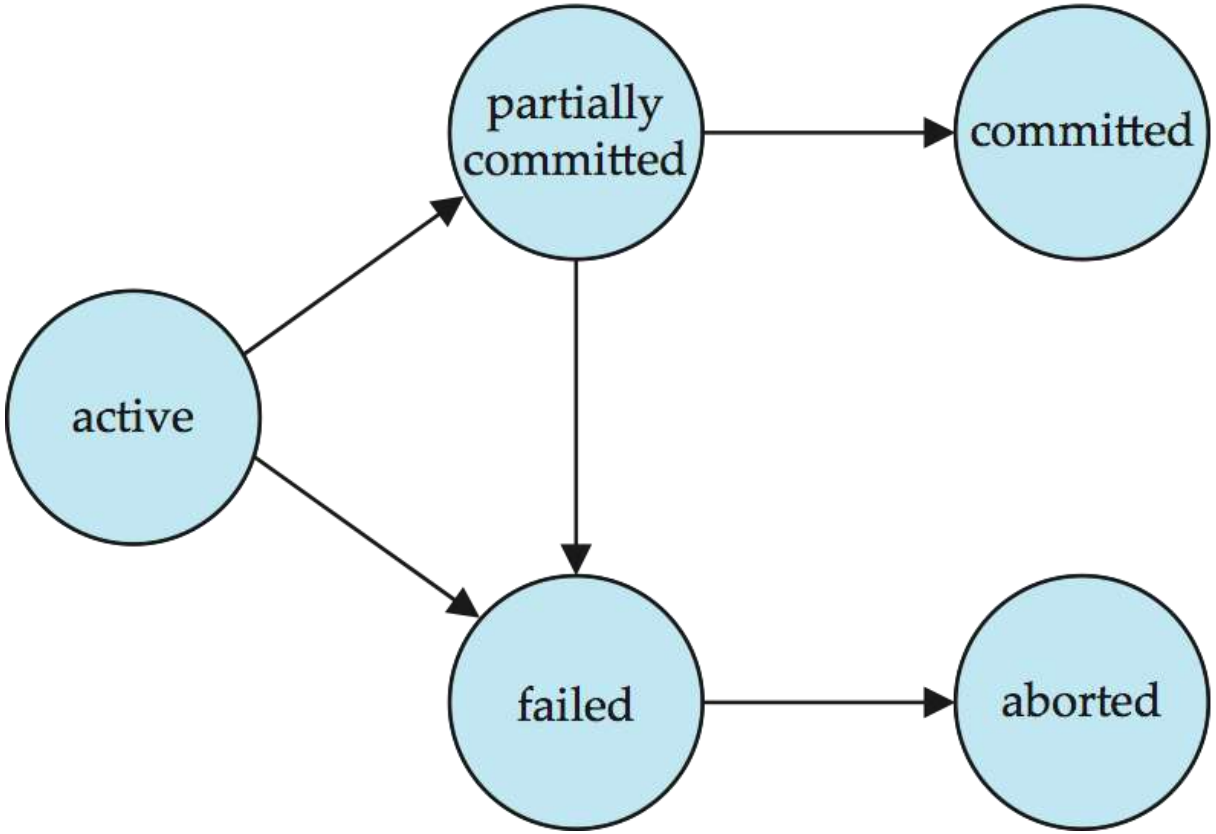
	T1	**T2**
1.	**read**(*A*)	
2.	*A* := *A* – 50	
3.	**write**(*A*)	
		read(A), read(B), print(A+B)
4.	**read**(*B*)	
5.	*B* := *B* + 50	
6.	**write**(*B*)	

- Isolation can be ensured trivially by running transactions **serially**
  - That is, one after the other.

States of a transaction

- **Active** – the initial state; the transaction stays in this state while it is executing
- **Partially committed** – after the final statement has been executed.
- **Failed --** after the discovery that normal execution can no longer proceed.
- **Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - Restart the transaction
    - can be done only if no internal logical error
  - Kill the transaction
- **Committed** – after successful completion.

States of a transaction



Introduction to Concurrency Control

Multiple transactions are allowed to run concurrently in the system. Advantages are:

- **Increased processor and disk utilization**, leading to better transaction *throughput*
  e.g. one transaction can be using the CPU while another is reading from or writing to the disk.
- **Reduced average response time** for transactions: short transactions need not wait behind long ones.

Deadlock

- Deadlock occurs when each transaction T in a set of two or more transactions is waiting for some item that is locked by some other transaction T^1 in the set.
- Hence each transaction in the set is in a waiting queue, waiting for one of the other transactions in the set to release the lock on an item.
- But because the other transaction is also waiting, it will never release the lock.

Deadlock

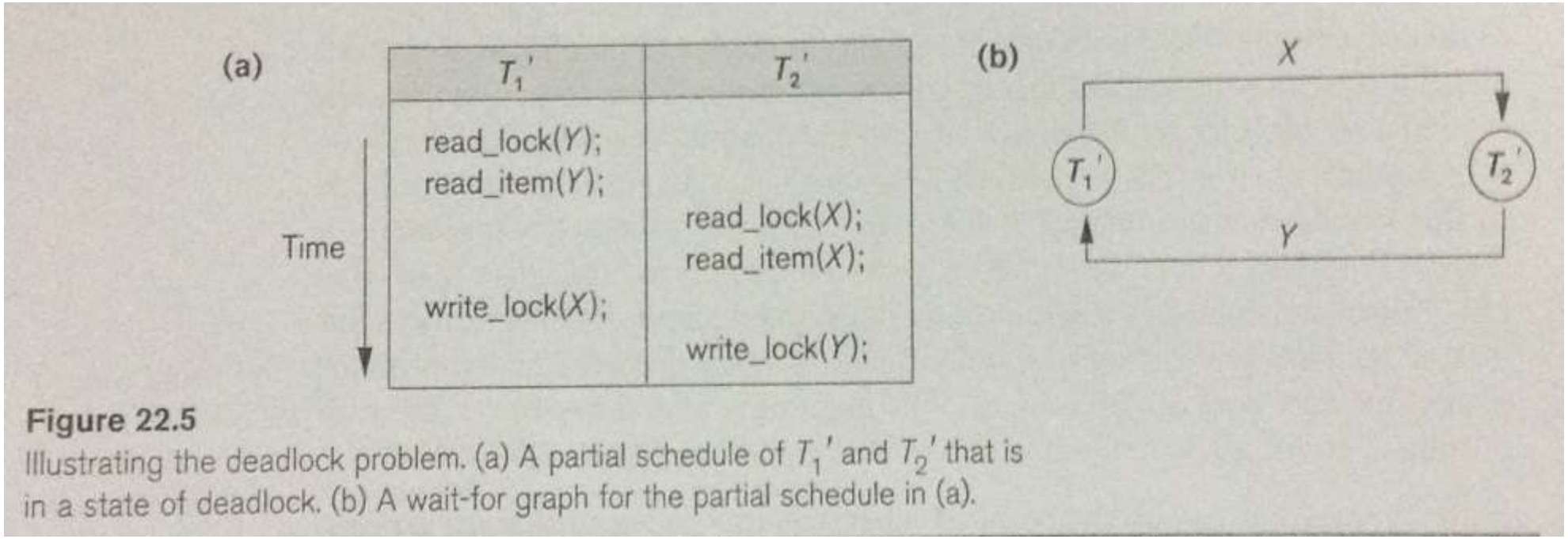


Figure 22.5
Illustrating the deadlock problem. (a) A partial schedule of T_1' and T_2' that is in a state of deadlock. (b) A wait-for graph for the partial schedule in (a).

Recovery

- Recovery from transaction failures usually means that the database is restored to the most recent consistent state just before the time of failure.
- The system store information about the changes that were applied to data items by the various transactions in order to do the recovery.
- This information is typically stored in a system log.

Recovery

If there is extensive damage to a wide portion of the database due to catastrophic failure, such as a disk crash, the recovery method restores a past copy of the database that was backed up to archival storage (tape or other large capacity offline storage media) and reconstructs a more current state by reapplying or redoing the operations of committed transactions from the backed up log, up to time of failure.

Recovery

When the database on disk is not physically damaged, and a noncatastrophic failure, the recovery strategy is to identify any changes that may cause an inconsistency in the database.

For example, a transaction that has updated some database items on disk but has not been committed needs to have its changes reversed by undoing its write operations.

It may also be necessary to redo some operations in order to restore a consistent state of the database. For example if a transaction has committed but some of its write operations have not yet been written to disk.

Recovery

The main two techniques for recovery from noncatastrophic transaction failures are **deferred update** and **immediate update.**

The deferred update techniques do not physically update the database on disk until after a transaction reaches its commit point, then updates are recorded in the database.

Before reaching the commit, all transaction updates are recorded in the local transaction workspace or in the main memory buffers maintained by the DBMS.

Recovery

Before commit, the updates are recorded persistently in the log, and then after commit, the updates are written to the database on disk.

If a transaction fails before reaching its commit point, it will not have changed the database in any way, hence UNDO is not needed. Hence deferred update is also known as the NO-UNDO/REDO algorithm.

Recovery

In the immediate update techniques, the database may be updated by some operations of a transaction before the transaction reaches its commit point.

These operations must also be recorded in the log on disk by force-writing before they are applied to the database on disk, making recovery still possible.

Recovery

If a transaction fails after recording some changes in the database on disk but before reaching its commit point, the effect of its operations on the database must be undone, i.e. the transaction must be rolled back.

In the general case of immediate update, both undo and redo may be required during recovery. This technique is known as the UNDO/REDO algorithm, requires both operations during recovery.