Chapter 20: Introduction to Transaction Processing Concepts and Theory

Database Systems CS219

Introduction to Transaction Processing

Introduction

Transaction

- Describes local unit of database processing

Transaction processing systems

- Systems with large databases and hundreds of concurrent users
- Require high availability and fast response time

20.1 Introduction to Transaction Processing

- Single-user DBMS
  - At most one user at a time can use the system
  - Example: home computer
- Multiuser DBMS
  - Many users can access the system (database) concurrently
  - Example: airline reservations system

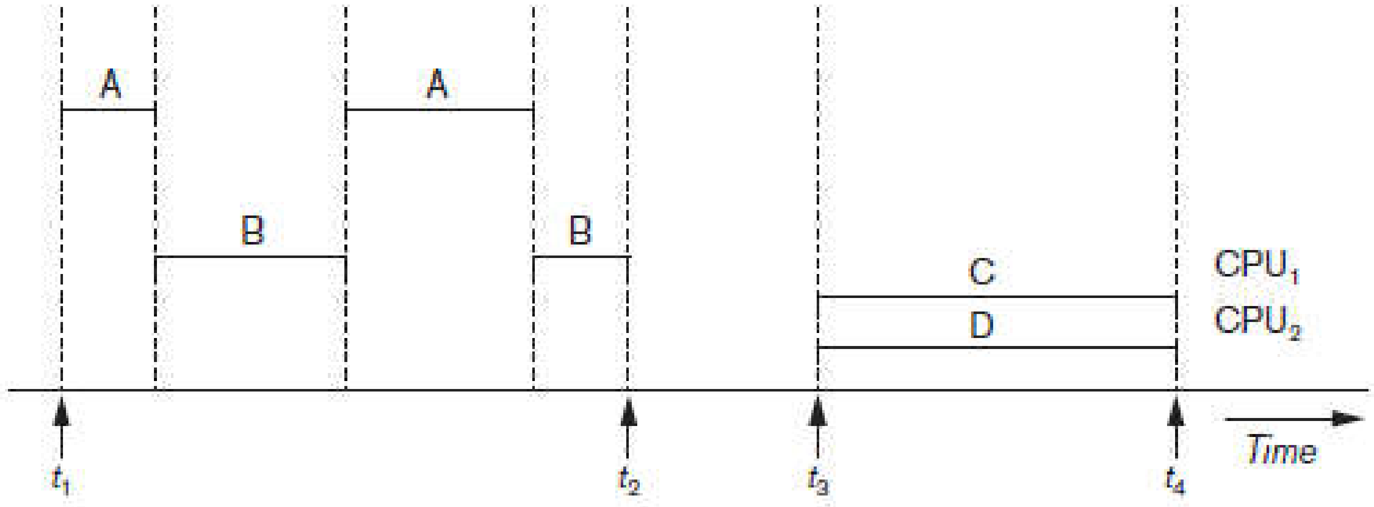
Introduction to Transaction Processing (cont'd.)

Multiprogramming

- Allows operating system to execute multiple processes concurrently
- Executes commands from one process, then suspends that process and executes commands from another process, etc.

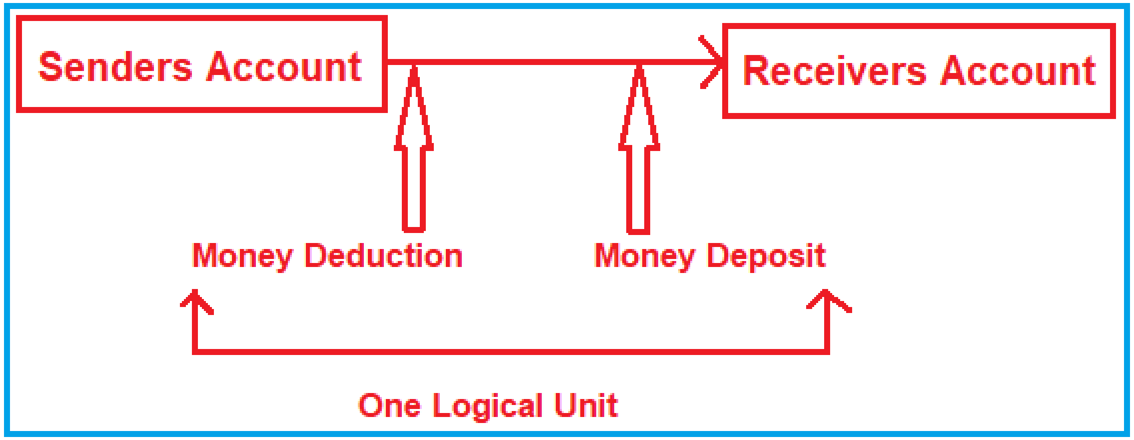
Introduction to Transaction Processing (cont'd.)

- Interleaved processing
- Parallel processing
    - Processes C and D in figure below



Transactions

- Transaction: an executing program
  - Forms logical unit of database processing



- Begin and end transaction statements
  - Specify transaction boundaries
- Read-only transaction
- Read-write transaction

Read and Write Operations

- **read_item(X)**
  - Reads a database item named X into a program variable named X
  - Process includes finding the address of the disk block, and copying to and from a memory buffer
- **write_item(X)**
  - Writes the value of program variable X into the database item named X
  - Process includes finding the address of the disk block, copying to and from a memory buffer, and storing the updated disk block back to disk

Implementation Details of Transaction

1. Begin the transaction
2. Process database commands
3. Check for errors

   If error occurs

       Roll back the transaction

   Else

       Commit the transaction

Implementation Details

```sql
--Create Product table
CREATE TABLE Product
(
 ProductID INT PRIMARY KEY,
 Name VARCHAR(40),
 Price INT,
 Quantity INT
 )
 GO

 -- Populate Product Table with test data
 INSERT INTO Product VALUES(101, 'Product-1', 100, 10)
 INSERT INTO Product VALUES(102, 'Product-2', 200, 15)
 INSERT INTO Product VALUES(103, 'Product-3', 300, 20)
 INSERT INTO Product VALUES(104, 'Product-4', 400, 25)
```

ProductID	Name	Price	Quantity
101	Product-1	100	10
102	Product-2	200	15
103	Product-3	300	20
104	Product-4	400	25

Example of COMMIT transaction with DML statements

```
BEGIN TRANSACTION
 INSERT INTO Product VALUES(105,'Product-5',500, 30)
 UPDATE Product SET Price =350 WHERE ProductID = 103
 DELETE FROM Product WHERE ProductID = 103
COMMIT TRANSACTION
```

ProductID	Name	Price	Quantity
101	Product-1	100	10
102	Product-2	200	15
104	Product-4	400	25
105	Product-5	500	30

Rollback Transaction

```
BEGIN TRANSACTION
INSERT INTO Product VALUES(110,'Product-10',600, 30)
INSERT INTO Product VALUES(110,'Product-10',600, 30)

IF(@@ERROR > 0)
BEGIN
    Rollback Transaction
END
ELSE
BEGIN
   Commit Transaction
END
```

Read and Write Operations (cont'd.)

- Read set of a transaction
  - Set of all items read
- Write set of a transaction
  - Set of all items written

(a)

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

(b)

T_2
read_item(X);
$X := X + M$;
write_item(X);

Concurrency Control

- Transactions submitted by various users may execute concurrently
    - Access and update the same database items
    - Some form of concurrency control is needed
- The lost update problem
    - Occurs when two transactions that access the same database items have operations interleaved
    - Results in incorrect value of some database items

The Lost Update Problem

(a)

Time

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);	

Item X has an incorrect value because its update by T_1 is *lost* (overwritten).

The Temporary Update Problem

(b)

Time

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);	

Transaction T_1 fails and must change the value of X back to its old value; meanwhile T_2 has read the *temporary* incorrect value of X.

The Incorrect Summary Problem

(c)

T_1	T_3

	sum := 0;
	read_item(A);
	sum := *sum* + A;
	⋮

read_item(X);
$X := X - N$;
write_item(X);	

	read_item(X);
	sum := *sum* + X;
	read_item(Y);
	sum := *sum* + Y;

read_item(Y);
$Y := Y + N$;
write_item(Y);	

T_3 reads X after N is subtracted and reads Y before N is added; a wrong summary is the result (off by N).

The Unrepeatable Read Problem

- Transaction T reads the same item twice
- Value is changed by another transaction T' between the two reads
- T receives different values for the two reads of the same item

Transaction T1	Transaction T2
R (X)	
	R (X)
W (X)	
	R (X) // Unrepeated Read

Why Recovery is Needed?

- Committed transaction
  - Effect recorded permanently in the database
- Aborted transaction
  - Does not affect the database
- Types of transaction failures
  - Computer failure (system crash)
  - Transaction or system error
  - Local errors or exception conditions detected by the transaction

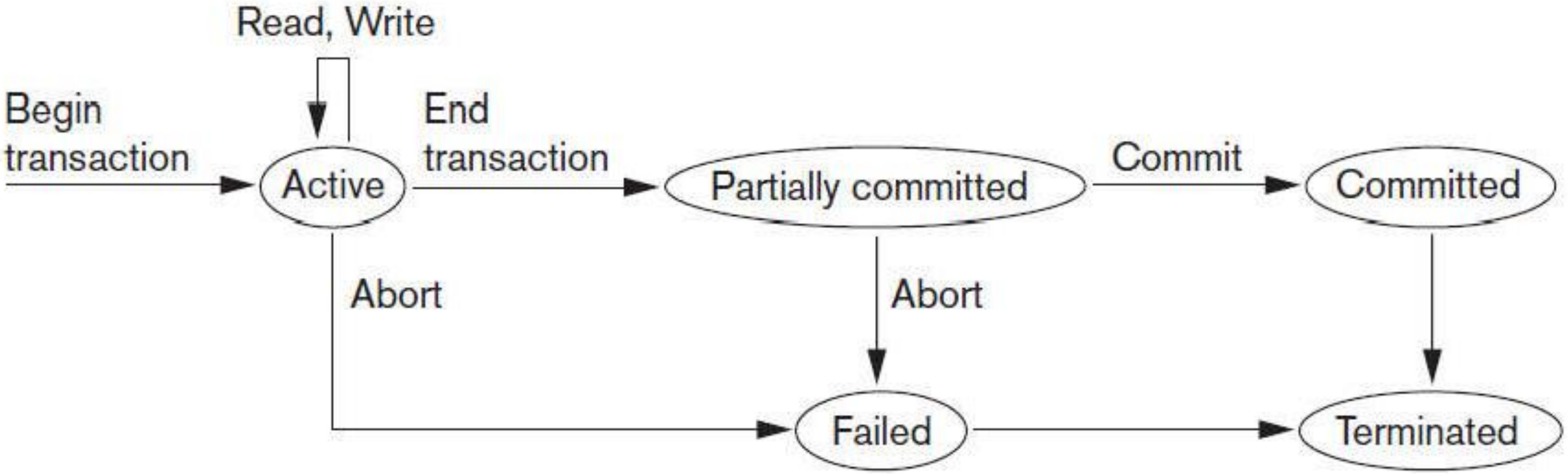
Why Recovery is Needed (cont'd.)

- Types of transaction failures (cont'd.)
  - Concurrency control enforcement
  - Disk failure
  - Physical problems or catastrophes
- System must keep sufficient information to recover quickly from the failure
  - Disk failure or other catastrophes have long recovery times

20.2 Transaction and System Concepts

- System must keep track of when each transaction starts, terminates, commits, and/or aborts
  - BEGIN_TRANSACTION
  - READ or WRITE
  - END_TRANSACTION
  - COMMIT_TRANSACTION
  - ROLLBACK (or ABORT)

Transaction and System Concepts (cont'd.)



The System Log

- System log keeps track of transaction operations
- Sequential, append-only file
- Not affected by failure (except disk or catastrophic failure)
- Log buffer
  - Main memory buffer
  - When full, appended to end of log file on disk
- Log file is backed up periodically
- Undo and redo operations based on log possible

Commit Point of a Transaction

- Occurs when all operations that access the database have completed successfully
  - And effect of operations recorded in the log
- Transaction writes a commit record into the log
  - If system failure occurs, can search for transactions with recorded start_transaction but no commit record
- Force-writing the log buffer to disk
  - Writing log buffer to disk before transaction reaches commit point

20.3 Desirable Properties of Transactions

- ACID properties
  - **Atomicity**
    - Transaction performed in its entirety or not at all
  - **Consistency preservation**
    - Takes database from one consistent state to another
  - **Isolation**
    - Not interfered with by other transactions
  - **Durability or permanency**
    - Changes must persist in the database

Desirable Properties of Transactions (cont'd.)

- Levels of isolation
  - Level 0 isolation does not overwrite the dirty reads of higher-level transactions
  - Level 1 isolation has no lost updates
  - Level 2 isolation has no lost updates and no dirty reads
  - Level 3 (true) isolation has repeatable reads
    - In addition to level 2 properties
  - Snapshot isolation

20.4 Characterizing Schedules Based on Recoverability

- Schedule or history
  - Order of execution of operations from all transactions
  - Operations from different transactions can be interleaved in the schedule
- Total ordering of operations in a schedule
  - For any two operations in the schedule, one must occur before the other

Characterizing Schedules Based on Recoverability (cont'd.)

- Two conflicting operations in a schedule
  - Operations belong to different transactions
  - Operations access the same item X
  - At least one of the operations is a write_item(X)
- Two operations conflict if changing their order results in a different outcome
- Read-write conflict
- Write-write conflict

Characterizing Schedules Based on Recoverability (cont'd.)

- Recoverable schedules
  - Recovery is possible
- Non recoverable schedules should not be permitted by the DBMS
- No committed transaction ever needs to be rolled back
- Cascading rollback may occur in some recoverable schedules
  - Uncommitted transaction may need to be rolled back

Characterizing Schedules Based on Recoverability (cont'd.)

- Cascadeless schedule
  - Avoids cascading rollback
- Strict schedule
  - Transactions can neither read nor write an item X until the last transaction that wrote X has committed or aborted
  - Simpler recovery process
    - Restore the before image

20.5 Characterizing Schedules Based on Serializability

- Serializable schedules
  - Always considered to be correct when concurrent transactions are executing
  - Places simultaneous transactions in series
    - Transaction T_1 before T_2, or vice versa

(a)

Time

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

Schedule A

(b)

Time

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

Schedule B

(c)

Time

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);	

Schedule C

Time

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);	
$Y := Y + N$;	
write_item(Y);	

Schedule D

Characterizing Schedules Based on Serializability (cont'd.)

- Problem with serial schedules
  - Limit concurrency by prohibiting interleaving of operations
  - Unacceptable in practice
  - Solution: determine which schedules are equivalent to a serial schedule and allow those to occur
- Serializable schedule of *n* transactions
  - Equivalent to some serial schedule of same *n* transactions

Characterizing Schedules Based on Serializability (cont'd.)

- Result equivalent schedules
  - Produce the same final state of the database
    - May be accidental
  - Cannot be used alone to define equivalence of schedules

S_1	S_2
read_item(X);	read_item(X);
$X := X + 10$;	$X := X * 1.1$;
write_item(X);	write_item (X);

Characterizing Schedules Based on Serializability (cont'd.)

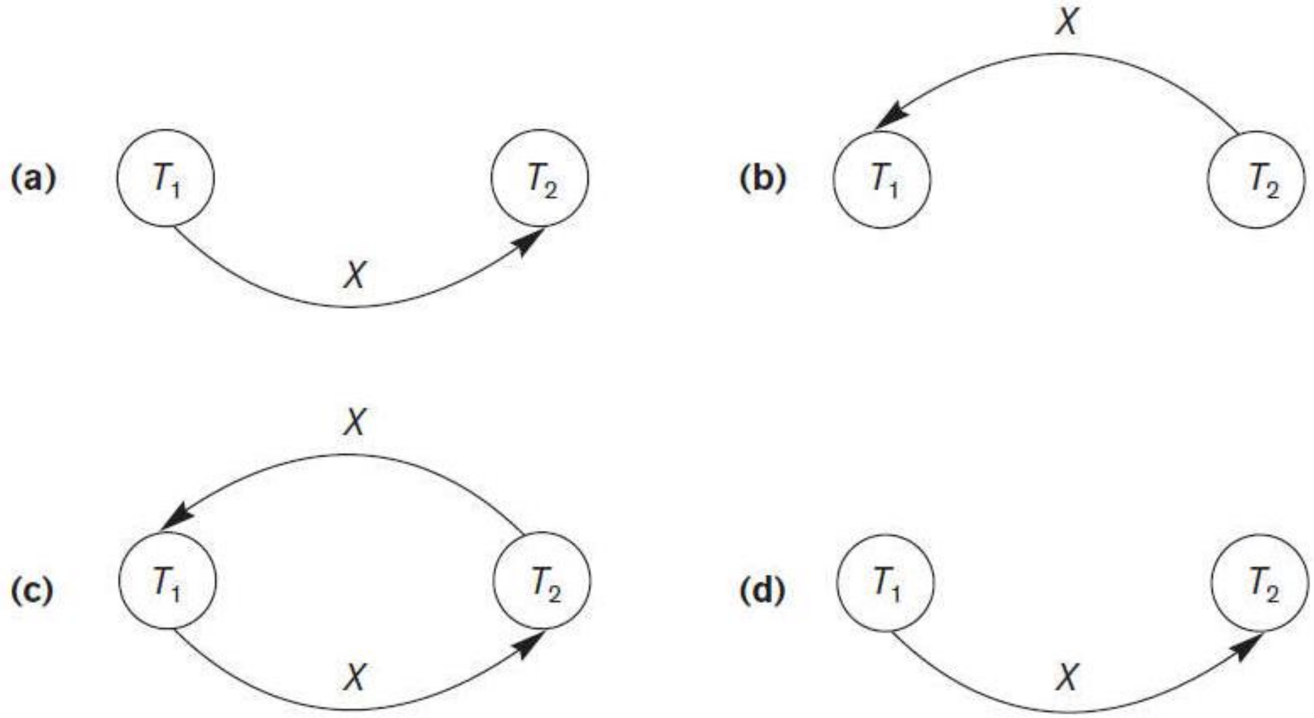
- Conflict equivalence
  - Relative order of any two conflicting operations is the same in both schedules
- Serializable schedules
  - Schedule S is serializable if it is conflict equivalent to some serial schedule S'.

Characterizing Schedules Based on Serializability (cont'd.)

- Testing for serializability of a schedule

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.

Characterizing Schedules Based on Serializability (cont'd.)



How Serializability is Used for Concurrency Control

- Being serializable is different from being serial
- Serializable schedule gives benefit of concurrent execution
    - Without giving up any correctness
- Difficult to test for serializability in practice
    - Factors such as system load, time of transaction submission, and process priority affect ordering of operations
- DBMS enforces protocols
    - Set of rules to ensure serializability

View Equivalence and View Serializability

- View equivalence of two schedules
  - As long as each read operation of a transaction reads the result of the same write operation in both schedules, the write operations of each transaction must produce the same results
  - Read operations said to see the same view in both schedules
- View serializable schedule
  - View equivalent to a serial schedule

View Equivalence and View Serializability (cont'd.)

- Conflict serializability similar to view serializability if constrained write assumption (no blind writes) applies
- Unconstrained write assumption
  - Value written by an operation can be independent of its old value
- Debit-credit transactions
  - Less-stringent conditions than conflict serializability or view serializability

20.6 Transaction Support in SQL

- No explicit Begin_Transaction statement
- Every transaction must have an explicit end statement
  - COMMIT
  - ROLLBACK
- Access mode is READ ONLY or READ WRITE
- Diagnostic area size option
  - Integer value indicating number of conditions held simultaneously in the diagnostic area

Transaction Support in SQL (cont'd.)

- Isolation level option
  - Dirty read
  - Nonrepeatable read
  - Phantoms

	Type of Violation		
Isolation Level	Dirty Read	Nonrepeatable Read	Phantom
READ UNCOMMITTED	Yes	Yes	Yes
READ COMMITTED	No	Yes	Yes
REPEATABLE READ	No	No	Yes
SERIALIZABLE	No	No	No

Transaction Support in SQL (cont'd.)

- Snapshot isolation
  - Used in some commercial DBMSs
  - Transaction sees data items that it reads based on the committed values of the items in the database snapshot when transaction starts
  - Ensures phantom record problem will not occur

20.7 Summary

- Single and multiuser database transactions
- Uncontrolled execution of concurrent transactions
- System log
- Failure recovery
- Committed transaction
- Schedule (history) defines execution sequence
  - Schedule recoverability
  - Schedule equivalence
- Serializability of schedules