Database Systems

Instructor: Bilal Khalid Dar

Agenda

- Transaction Management

Introduction to Transaction Processing

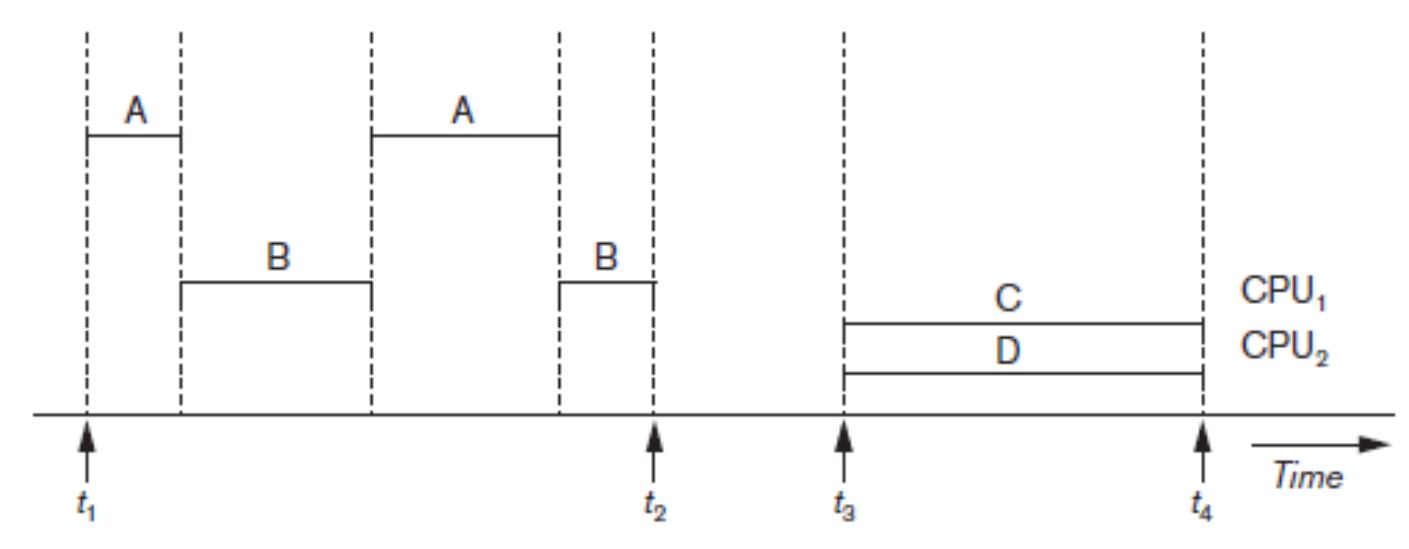
- Single-user DBMS
  - At most one user at a time can use the system
  - Example: home computer
- Multiuser DBMS
  - Many users can access the system (database) concurrently
  - Example: airline reservations system

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



Interleaved processing versus parallel processing of concurrent transactions

Database Items

- Database represented as collection of named data items
- Size of a data item called its granularity
- Data item
  - Record
  - Disk block
  - Attribute value of a record
- Transaction processing concepts independent of item granularity

Transactions

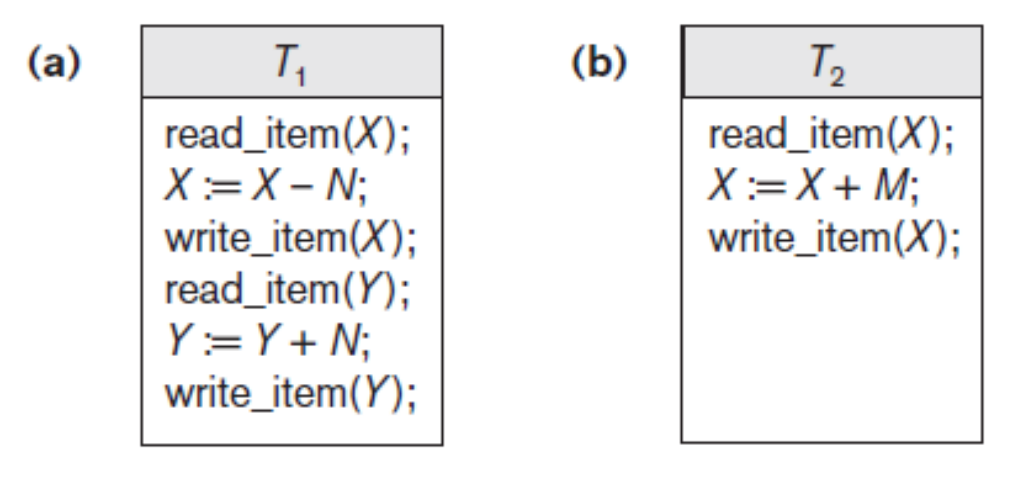
- Transaction: an executing program
  - Forms logical unit of database processing
- Begin and end transaction statements
  - Specify transaction boundaries
- Read-only transaction
- Read-write transaction

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

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



Two sample transactions (a) Transaction *T*1 (b) Transaction *T*2

Transaction support

Transaction

Action, or series of actions, carried out by user or application, which reads or updates contents of database.

- Logical unit of work on the database.
- Application program is series of transactions with non-database processing in between.
- Transforms database from one consistent state to another, although consistency may be violated during transaction.

Example transaction

(a)	(b)	
		Database consistent
read(bal_x)	read(bal_x)	
$bal_x = bal_x + 10$	$bal_x = bal_x - 10$	
write(bal_x)	write(bal_x)	Database inconsistent
commit	read(bal_y)	
	$bal_y = bal_y + 10$	
	write(bal_y)	
	commit	Database consistent

Transaction support

- Can have one of two outcomes:
  - Success - transaction *commits* and database reaches a new consistent state.
  - Failure - transaction *aborts*, and database must be restored to consistent state before it started.
  - Such a transaction is *rolled back* or *undone*.
- Committed transaction cannot be aborted.
- Aborted transaction that is rolled back can be restarted later.

Desirable Properties of a transaction

- Four basic *(ACID)* properties of a transaction are:

Atomicity ‘All or nothing’ property.

Consistency Must transform database from one consistent state to another.

Isolation Partial effects of incomplete transactions should not be visible to other transactions.

Durability Effects of a committed transaction are permanent and must not be lost because of later failure.

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

Concurrency control

Process of managing simultaneous operations on the database without having them interfere with one another.

- Prevents interference when two or more users are accessing database simultaneously and at least one is updating data.
- Although two transactions may be correct in themselves, interleaving of operations may produce an incorrect result.

Conflict

Order-sensitivity of operations

- Two operations of different transactions, but on the same data item, *conflict* if
  - Their mutual order is significant, i.e., determines at least one of the following:
    - The final value of that item read by future transactions
    - The value of the item as read by present transactions

Need for concurrency control

- Three examples of potential problems caused by concurrency:
  - Lost update problem.
  - Uncommitted dependency problem.
  - Inconsistent analysis problem.

Lost update problem

- Successfully completed update is overridden by another user.
- T_1 withdrawing $10 from an account with bal_x, initially $100.
- T_2 depositing $100 into same account.
- Serially, final balance would be $190.

Lost update problem

Time	T_1	T_2	bal_x
t_1		begin_transaction	100
t_2	begin transaction	read(bal_x)	100
t_3	read(bal_x)	$bal_x = bal_x + 100$	100
t_4	$bal_x = bal_x - 10$	write(bal_x)	200
t_5	write(bal_x)	commit	90
t_6	commit		90

- Loss of T_2's update avoided by preventing T_1 from reading bal_x until after update.

Uncommitted dependency problem

- Occurs when one transaction can see intermediate results of another transaction before it has committed.
- T_4 updates bal_x to \$200 but it aborts, so bal_x should be back at original value of \$100.
- T_3 has read new value of bal_x (\$200) and uses value as basis of \$10 reduction, giving a new balance of \$190, instead of \$90.

Uncommitted dependency problem

Time	T_3	T_4	bal_x
t_1		begin_transaction	100
t_2		read(bal_x)	100
t_3		$bal_x = bal_x + 100$	100
t_4	begin_transaction	write(bal_x)	200
t_5	read(bal_x)	⋮	200
t_6	$bal_x = bal_x - 10$	rollback	100
t_7	write(bal_x)		190
t_8	commit		190

- Problem avoided by preventing T_3 from reading bal_x until after T_4 commits or aborts.

Inconsistent analysis problem

- Occurs when transaction reads several values but second transaction updates some of them during execution of first.
- Sometimes referred to as *dirty read* or *unrepeatable read*.
- T_6 is totaling balances of account x ($100), account y ($50), and account z ($25).
- Meantime, T_5 has transferred $10 from bal_x to bal_z, so T_6 now has wrong result ($10 too high).

Inconsistent analysis problem

Time	T_5	T_6	bal_x	bal_y	bal_z	sum
t_1		begin_transaction	100	50	25	
t_2	begin_transaction	sum = 0	100	50	25	0
t_3	read(bal_x)	read(bal_x)	100	50	25	0
t_4	$bal_x = bal_x - 10$	sum = sum + bal_x	100	50	25	100
t_5	write(bal_x)	read(bal_y)	90	50	25	100
t_6	read(bal_z)	sum = sum + bal_y	90	50	25	150
t_7	$bal_z = bal_z + 10$		90	50	25	150
t_8	write(bal_z)		90	50	35	150
t_9	commit	read(bal_z)	90	50	35	150
t_{10}		sum = sum + bal_z	90	50	35	185
t_{11}		commit	90	50	35	185

- Problem avoided by preventing T_6 from reading bal_x and bal_z until after T_5 completed updates.

Serializability

- Objective of a concurrency control protocol is to schedule transactions in such a way as to avoid any interference.
- Could run transactions serially, but this limits degree of concurrency or parallelism in system.
- Serializability identifies those executions of transactions guaranteed to ensure consistency.

Serializability

Schedule

Sequence of reads/writes by set of concurrent transactions.

Serial Schedule

Schedule where operations of each transaction are executed consecutively without any interleaved operations from other transactions.

- No guarantee that results of all serial executions of a given set of transactions will be identical.

Serializability

- Schedule where operations from set of concurrent transactions are interleaved.
- Objective of serializability is to find nonserial schedules that allow transactions to execute concurrently without interfering with one another.
- In other words, want to find nonserial schedules that are equivalent to *some* serial schedule. Such a schedule is called *serializable*.

Concurrency control protocols

- Two basic concurrency control techniques:
  - Locking,
  - Timestamping.
- Both are conservative approaches: delay transactions in case they conflict with other transactions.
- Optimistic methods assume conflict is rare and only check for conflicts at commit.

Locking methods

Transaction uses locks to deny access to other transactions and so prevent incorrect updates.

- Most widely used approach to ensure serializability.
- Generally, a transaction must claim a *shared* (*read*) or *exclusive* (*write*) lock on a data item before read or write.
- Lock prevents another transaction from modifying item or even reading it, in the case of a write lock.

Locking – basic rules

- If transaction has shared lock on item, can read but not update item.
- If transaction has exclusive lock on item, can both read and update item.
- Reads cannot conflict, so more than one transaction can hold shared locks simultaneously on same item.
- Exclusive lock gives transaction exclusive access to that item.

Incorrect locking schedule

Time	T_7	T_8	bal_x	bal_y
t_1	write_lock(bal_x)		100	400
t_2	read(bal_x)		"	"
t_3	$bal_x = bal_x + 100$		"	"
t_4	write(bal_x)		200	"
t_5	unlock(bal_x)		"	"
t_6		write_lock(bal_x)	"	"
t_7		read(bal_x)	"	"
t_8		$bal_x = bal_x * 1.1$	"	"
t_9		write(bal_x)	220	"
t_{10}		unlock(bal_x)	"	"
t_{11}		write_lock(bal_y)	"	"
t_{12}		read(bal_y)	"	"
t_{13}		$bal_y = bal_y * 1.1$	"	"
t_{14}		write(bal_y)	"	440
t_{15}		unlock(bal_y)	"	"
t_{16}		commit	"	"
t_{17}	write_lock(bal_y)		"	"
t_{18}	read(bal_y)		"	"
t_{19}	$bal_y = bal_y - 100$		"	"
t_{20}	write(bal_y)		"	340
t_{21}	unlock(bal_y)		"	"
t_{22}	commit		220	340

Incorrect locking schedule

- If at start, $bal_x = 100$, $bal_y = 400$, result should be:
  - $bal_x = 220$, $bal_y = 330$, if T_9 executes before T_{10}, or
  - $bal_x = 210$, $bal_y = 340$, if T_{10} executes before T_9.
- However, result gives $bal_x = 220$ and $bal_y = 340$.
- S is *not* a serializable schedule.
- Problem is that transactions release locks too soon, resulting in loss of total isolation and atomicity.
- To guarantee serializability, need an additional protocol concerning the positioning of lock and unlock operations in every transaction.

Two- phase locking

Transaction follows 2PL protocol if all locking operations precede first unlock operation in the transaction.

- Two phases for transaction:
  - Growing phase - acquires all locks but cannot release any locks.
  - Shrinking phase - releases locks but cannot acquire any new locks.

Preventing lost update problem

Time	T_1	T_2	bal_x
t_1		begin_transaction	100
t_2	begin_transaction	write_lock(bal_x)	100
t_3	write_lock(bal_x)	read(bal_x)	100
t_4	WAIT	$bal_x = bal_x + 100$	100
t_5	WAIT	write(bal_x)	200
t_6	WAIT	commit/unlock(bal_x)	200
t_7	read(bal_x)		200
t_8	$bal_x = bal_x - 10$		200
t_9	write(bal_x)		190
t_{10}	commit/unlock(bal_x)		190

Preventing uncommitted dependency problem

Time	T_3	T_4	bal_x
t_1		begin_transaction	100
t_2		write_lock(bal_x)	100
t_3		read(bal_x)	100
t_4	begin_transaction	$bal_x = bal_x + 100$	100
t_5	write_lock(bal_x)	write(bal_x)	200
t_6	WAIT	rollback/unlock(bal_x)	100
t_7	read(bal_x)		100
t_8	$bal_x = bal_x - 10$		100
t_9	write(bal_x)		90
t_{10}	commit/unlock(bal_x)		90

Preventing inconsistent analysis problem

Time	T_5	T_6	bal_x	bal_y	bal_z	sum
t_1		begin_transaction	100	50	25	
t_2	begin_transaction	sum = 0	100	50	25	0
t_3	write_lock(bal_x)		100	50	25	0
t_4	read(bal_x)	read_lock(bal_x)	100	50	25	0
t_5	$bal_x = bal_x - 10$	WAIT	100	50	25	0
t_6	write(bal_x)	WAIT	90	50	25	0
t_7	write_lock(bal_z)	WAIT	90	50	25	0
t_8	read(bal_z)	WAIT	90	50	25	0
t_9	$bal_z = bal_z + 10$	WAIT	90	50	25	0
t_{10}	write(bal_z)	WAIT	90	50	35	0
t_{11}	commit/unlock(bal_x, bal_z)	WAIT	90	50	35	0
t_{12}		read(bal_x)	90	50	35	0
t_{13}		sum = sum + bal_x	90	50	35	90
t_{14}		read_lock(bal_y)	90	50	35	90
t_{15}		read(bal_y)	90	50	35	90
t_{16}		sum = sum + bal_y	90	50	35	140
t_{17}		read_lock(bal_z)	90	50	35	140
t_{18}		read(bal_z)	90	50	35	140
t_{19}		sum = sum + bal_z	90	50	35	175
t_{20}		commit/unlock(bal_x, bal_y, bal_z)	90	50	35	175

Deadlock

An impasse that may result when two (or more) transactions are each waiting for locks held by the other to be released.

Time	T_9	T_{10}
t_1	begin_transaction	
t_2	write_lock(bal_x)	begin_transaction
t_3	read(bal_x)	write_lock(bal_y)
t_4	$bal_x = bal_x - 10$	read(bal_y)
t_5	write(bal_x)	$bal_y = bal_y + 100$
t_6	write_lock(bal_y)	write(bal_y)
t_7	WAIT	write_lock(bal_x)
t_8	WAIT	WAIT
t_9	WAIT	WAIT
t_{10}	⋮	WAIT
t_{11}	⋮	⋮

Deadlock

- Only one way to break deadlock: abort one or more of the transactions.
- Deadlock should be transparent to user, so DBMS should restart transaction(s).
- Three general techniques for handling deadlock:
  - Timeouts.
  - Deadlock prevention.
  - Deadlock detection and recovery.

Timestamp methods

Transactions ordered globally so that older transactions, transactions with *smaller* timestamps, get priority in the event of conflict.

- Conflict is resolved by rolling back and restarting transaction.
- No locks so no deadlock.

Timestamp methods

Timestamp

A unique identifier created by DBMS that indicates relative starting time of a transaction.

- Can be generated by using system clock at time transaction started, or by incrementing a logical counter every time a new transaction starts.
- Read/write proceeds only if *last update on that data item* was carried out by an older transaction.
- Otherwise, transaction requesting read/write is restarted and given a new timestamp.

Optimistic methods

- Based on assumption that conflict is rare and more efficient to let transactions proceed without delays to ensure serializability.
- At commit, check is made to determine whether conflict has occurred.
- If there is a conflict, transaction must be rolled back and restarted.
- Potentially allows greater concurrency than traditional protocols.

Optimistic methods

- Three phases:
  - **Read**: from start to just before commit; read from database into local variables and update local data.
  - **Validation**: check to ensure serializability is not violated; if violated transaction aborted and restarted.
  - **Write** (for update transactions): updates made to local variables then applied to database.

Recovery

Process of restoring database to a correct state in the event of a failure.

- Need for Recovery Control
  - Two types of storage: volatile (main memory) and nonvolatile.
  - Volatile storage does not survive system crashes.
  - Stable storage represents information that has been replicated in several nonvolatile storage media with independent failure modes.

Types of Failure

- System crashes, resulting in loss of main memory.
- Media failures, resulting in loss of parts of secondary storage.
- Application software errors.
- Natural physical disasters.
- Carelessness or unintentional destruction of data or facilities.
- Sabotage.

Transactions and recovery

- Transactions represent basic unit of recovery.
- Recovery manager responsible for **atomicity** and **durability**.
- If failure occurs between commit and database buffers being flushed to secondary storage then, to ensure durability, recovery manager has to *redo* (*rollforward*) transaction's updates.
- If transaction had not committed at failure time, recovery manager has to *undo* (*rollback*) any effects of that transaction for atomicity.
- Partial undo - only one transaction has to be undone.
- Global undo - all transactions have to be undone.

Recovery facilities

- DBMS should provide following facilities to assist with recovery:
  - Backup mechanism, which makes periodic backup copies of database.
  - Logging facilities, which keep track of current state of transactions and database changes.
  - Checkpoint facility, which enables updates to database in progress to be made permanent.
  - Recovery manager, which allows DBMS to restore database to consistent state following a failure.

Log file

- Contains information about all updates to database:
  - Transaction records.
  - Checkpoint records.
- Often used for other purposes (for example, auditing).

Log file

- Transaction records contain:
  - Transaction identifier.
  - Type of log record, (transaction start, insert, update, delete, abort, commit).
  - Identifier of data item affected by database action (insert, delete, and update operations).
  - Before-image of data item.
- After-image of data item.
  - Log management information.

Log file

Tid	Time	Operation	Object	Before image	After image	pPtr	nPtr
T1	10.22	START				0	2
T1	10.23	UPDATE	STAFF S1500	(old value)	(new value)	1	8
T2	10.24	START				0	4
T2	10.26	INSERT	STAFF S0003		(new value)	3	5
T2	10.27	DELETE	STAFF S3250	(old value)		4	6
T2	10.27	UPDATE	MEMBER M250178	(old value)	(new value)	5	9
T3	10.28	START				0	11
T1	10.28	COMMIT				2	0
	10.29	CHECKPOINT	T2, T3				
T2	10.29	COMMIT				6	0
T3	10.30	INSERT	MEMBER M166884		(new value)	7	12
T3	10.31	COMMIT				11	0

Log file

- Log file may be duplexed or triplexed.
- Log file sometimes split into two separate random-access files.
- Potential bottleneck; critical in determining overall performance.

Checkpointing

Checkpoint

Point of synchronization between database and log file. All buffers are force-written to secondary storage.

- Checkpoint record is created containing identifiers of all active transactions.
- When failure occurs, redo all transactions that committed since the checkpoint and undo all transactions active at time of crash.

Checkpointing

- In previous example, with checkpoint at time t_c, changes made by T_2 and T_3 have been written to secondary storage.
- Thus:
  - only redo T_4 and T_5,
  - undo transactions T_1 and T6.

Recovery techniques

- If database has been damaged:
  - Need to restore last backup copy of database and reapply updates of committed transactions using log file.
- If database is only inconsistent:
  - Need to undo changes that caused inconsistency. May also need to redo some transactions to ensure updates reach secondary storage.
  - Do not need backup, but can restore database using before- and after-images in the log file.

Immediate update

- Updates are applied to database as they occur.
- Need to redo updates of committed transactions following a failure.
- May need to undo effects of transactions that had not committed at time of failure.
- Essential that log records are written before write to database. *Write-ahead log protocol*.

Immediate update

- If no "*transaction commit*" record in log, then that transaction was active at failure and must be undone.
- Undo operations are performed *in reverse order in which they were written to log*.


Thank
You