



MC302 – DBMS: Transaction Processing Concepts

Goonjan Jain
Dept. of Applied Mathematics
Delhi Technological University

Overview

- Transaction Overview
- Concurrency
- Recovery

Motivation

- Lost Updates
(Concurrency control)  Two people change the same record (“Rajan”); how to avoid “race condition”?
- Durability
(Durability)  You transfer Rs. 100 from account -> processing; power failure – what happens?

DBMSs automatically handle both issues: **Transactions**

Transactions

- **Transaction** - execution of a sequence of one or more operations (e.g., SQL queries) on a shared database to perform some higher-level function.
- Basic unit of change in a DBMS:
 - Partial transactions are not allowed!
- Example: Transfer Rs. 10,000 from Rajan's account to his travel agent's account
- Transaction:
 - Check if Rajan has Rs.10,000 in his account
 - Deduct Rs. 10,000 from his account
 - Credit Rs. 10,000 to his travel agent's account

Approaches

- First – Strawman System
 - Execute transactions serially – one at a time
 - Make a local copy of DB
 - If the transaction completes successfully – overwrite the local DB with the new one
 - If transaction fails, remove the dirty copy
- Better - allow concurrent execution of independent transactions
- Why?
 - Higher utilization/throughput
 - Increased response time to users
- Essentials:
 - Correctness
 - Fairness

Transactions – Formal Definitions

- Database: A fixed set of named data objects (A, B, C, ...)
 - Data object – an entire DB, a set of tables, a table, a set of rows, a single row etc.
- Transaction: A sequence of read and write operations (R(A), W(B), ...)
 - DBMS's abstract view of a user program

Transactions in SQL

- A new transaction starts with the **begin** command.
- The transaction stops with either **commit** or **abort**:
 - If **commit**, all changes are saved.
 - If **abort**, all changes are undone so that it's like as if the transaction never executed at all.
- The transaction ends with a **end** command

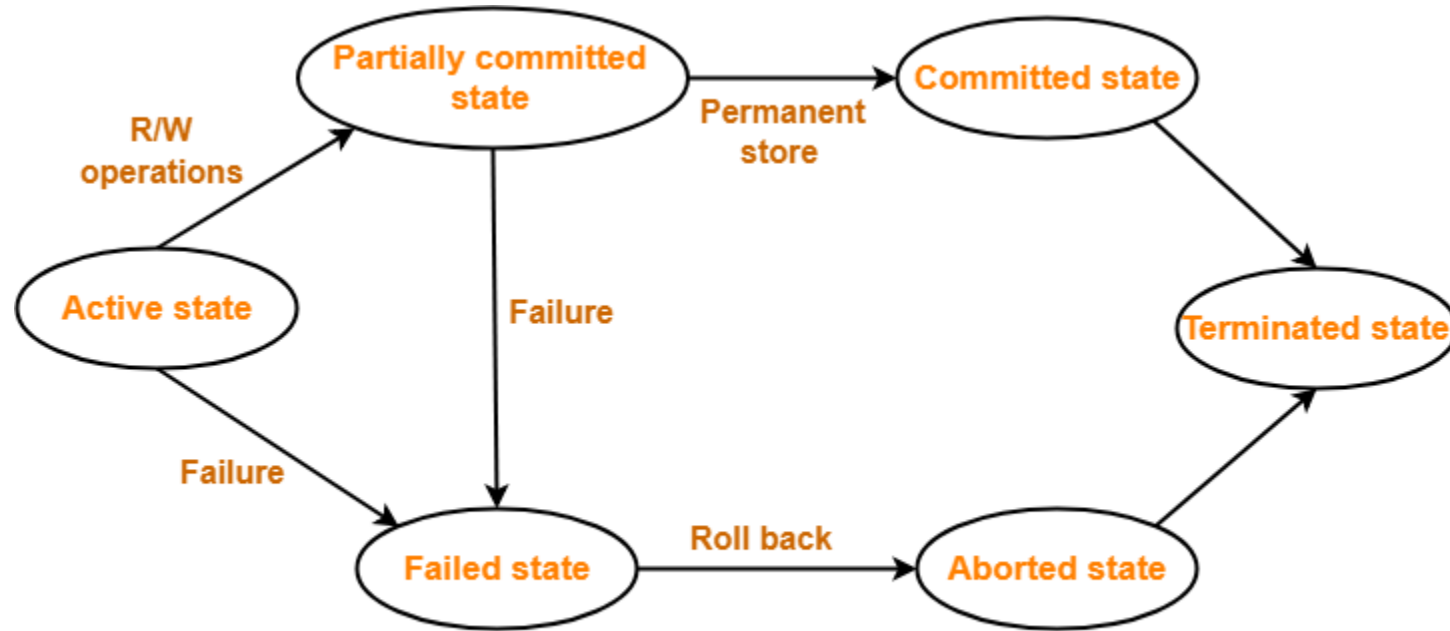
Correctness Criteria: ACID

- **A**tomicity: All actions in the transaction happen, or none happen.
“all or nothing”
- **C**onsistency: If each transaction is consistent and the DB starts consistent, then it ends up consistent.
“it looks correct”
- **I**solation: Execution of one transaction is isolated from that of other transactions.
“as if alone”
- **D**urability: If a transaction commits, its effects persist.
“survive failures”

Atomicity of Transactions

- DBMS guarantees that transactions are atomic.
 - From user's point of view: transaction always either executes all its actions, or executes no actions at all.
- We take Rs. 10,000 out of Rajan's account but then there is a power failure before we transfer it to his travel agent.
- When the database comes back on-line, what should be the correct state of Rajan's account?
- To ensure atomicity – **Logging**
 - DBMS logs all actions so that it can undo the actions of aborted transactions.

Transaction States



Transaction States in DBMS

Transaction States

- **Active** –while executing
- **Partially** committed – after final statement gets executed
- **Failed** – after discovery that cannot proceed with the normal execution
- **Aborted** – after transaction has been rolled back and the DB has been restored to its state prior to the start of the transaction
- **Committed** – after successful completion

Consistency

- **DB Consistency** - Data in the DBMS is accurate in modeling the real world and follows integrity constraints
- **Transaction Consistency** - if the database is consistent before the txn starts (running alone), it will be after also.
- In a distributed DBMS, the consistency level determines when other nodes see new data in the database:
 - **Strong**: Guaranteed to see all writes immediately, but transactions are slower.
 - **Weak/Eventual**: Will see writes at some later point in time, but transactions are faster.

Isolation of Transactions

- Users submit transactions, and each transaction executes *as if it was running by itself*.
- Concurrency is achieved by DBMS, which interleaves actions (reads/writes of DB objects) of various transactions.
- Two Main strategies:
 - **Pessimistic** – Don't let problems arise in the first place.
 - **Optimistic** – Assume conflicts are rare, deal with them after they happen.

Example

- Consider two transactions:
 - T1 – transfers Rs. 2000 from B's account to A's account
 - T2 – credits both accounts with 5% interest
- Assume that both have Rs. 10,000 each.
- What are the legal outcomes of running T1 and T2?

T1

```
BEGIN  
A = A+2000  
B = B-2000  
COMMIT
```

T2

```
BEGIN  
A = A*1.05  
B = B*1.05  
COMMIT
```

Example

- **Q:** What are the possible outcomes of running T1 and T2 together?

- **A:** Many! But A+B should be:

$$\text{Rs. } 20,000 * 1.05 = \text{Rs. } 21,000$$

- There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together. But, the net effect must be equivalent to these two transactions running **serially** in some order.
- Legal outcomes:
 - A=12,600, B=8,400 \longrightarrow 21,000
 - A=12,500, B=8,500 \longrightarrow 21,000
- The outcome depends on whether T1 executes before T2 or vice versa.

Example

- Legal outcomes – **S1**: $A=12,600$, $B=8,400$;
S2: $A=12,500$, $B=8,500$

S1	
T1	T2
$A = A + 2000$ $B = B - 2000$	
$A = A * 1.05$ $B = B * 1.05$	

S2	
T1	T2
$A = A + 2000$ $B = B - 2000$	
$A = A * 1.05$ $B = B * 1.05$	

Interleaving Transactions - Schedule

- Interleave the transactions in order to maximize concurrency.
 - Slow disk/network I/O.
 - Multi-core CPUs.
- Legal outcomes - A=12,600, B=8,400;

S1

T1	T2
A = A + 2000	
	A = A * 1.05
B = B - 2000	
	B = B * 1.05

Good

A=12,500, B=8,500

S2

T1	T2
A = A + 2000	
	A = A * 1.05
	B = B * 1.05
B = B - 2000	

Bad

Interleaving Transactions - Schedule

Schedule S

T1	T2
A = A + 2000	
	A = A * 1.05
	B = B * 1.05
B = B - 2000	

DBMS View of Schedule

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
R(B)	
W(B)	

Schedule

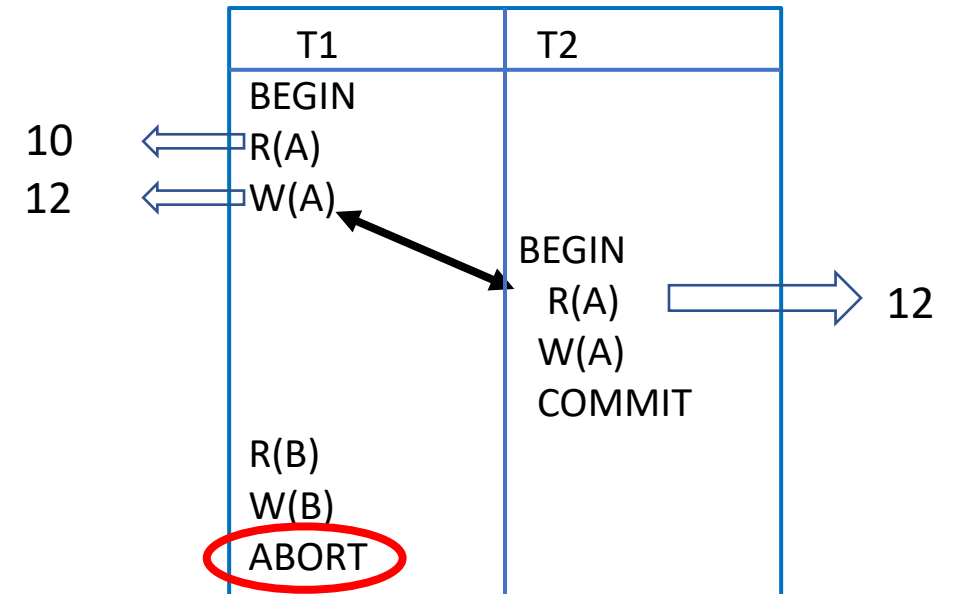
- Q: How would you judge that a schedule is 'correct'?
- A: if it is equivalent to **some** serial execution
- Formal Properties of Schedules
 - *Serial schedule*: does not interleave the actions of different transactions.
 - *Equivalent schedules*: For any database state, the effect of executing the first schedule is identical to the effect of executing the second schedule.
 - *Serializable schedule*: A schedule that is equivalent to some serial execution of the transactions.

Anomalies with Interleaved Execution

- Read-Write conflicts (R-W)
- Write-Read conflicts (W-R)
- Write-Write conflicts (W-W)
- Q – Why not Read-Read (R-R) conflicts?

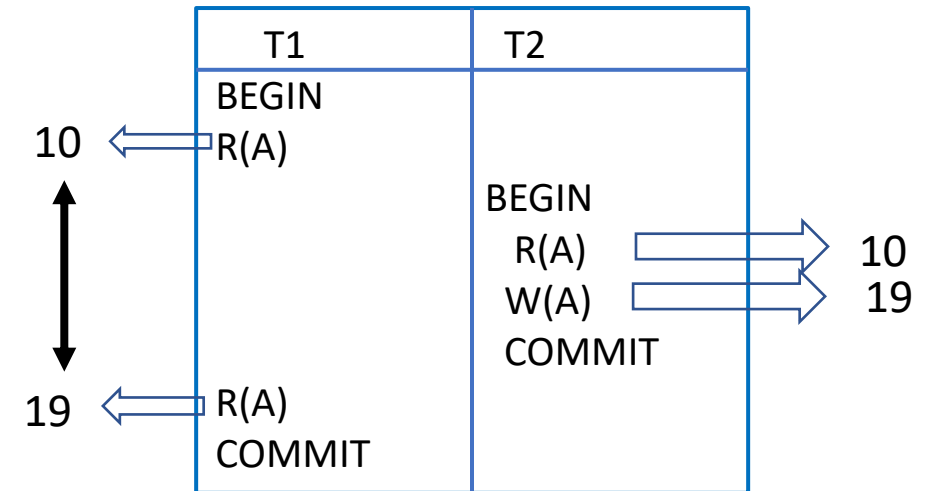
Write-Read conflicts

- Reading uncommitted data, “Dirty Reads”
- **Q:** Why is avoiding “dirty reads” important?
- **A:** If a transaction aborts, all actions must be undone. Any transaction that read modified data must also be aborted.



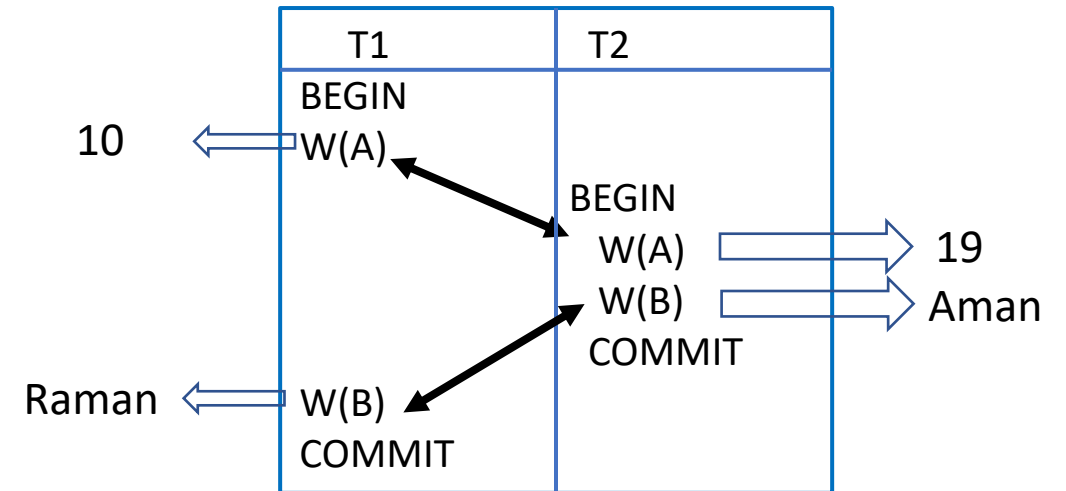
Read-Write conflicts

- “Unrepeatable Reads”



Write-Write conflicts

- “Lost update problem”
- Overwriting uncommitted data



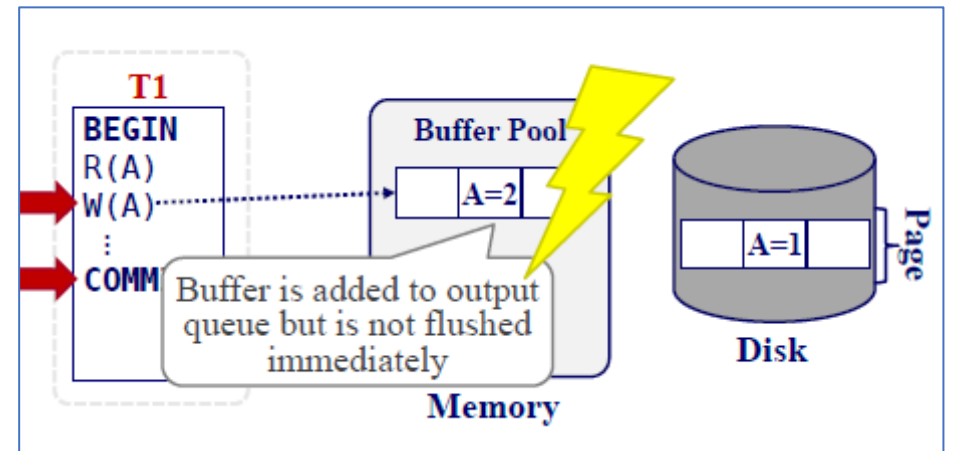
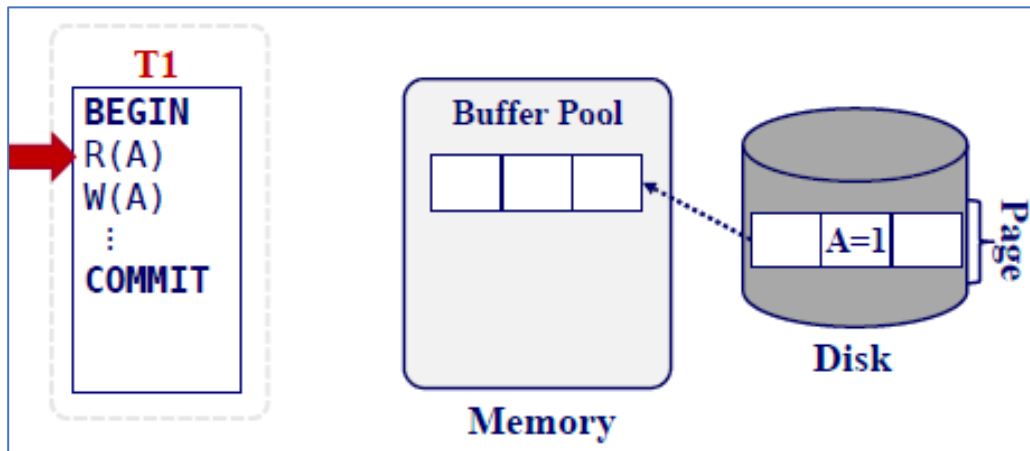
Solutions

- **Q:** How could you guarantee that all resulting schedules are correct (i.e., serializable)?
- **A:** Use locks! – part of the answer

Use locks; keep them until commit – full answer

Transaction Durability

- Records are stored on disk.
- For updates, they are copied into memory and flushed back to disk at the discretion of the O.S.





Write-Ahead Log

- Record the changes made to the database in a log ***before*** the change is made.
- Assume that the log is on stable storage.
- Log record format:
 - **<txnId, objectId, beforeValue, afterValue>**
 - Each transaction writes a log record first, before doing the change
- When a transaction finishes, the DBMS will:
 - Write a **<commit>** record on the log
 - Make sure that all log records are flushed before it returns an acknowledgement to application.
- After a failure, DBMS “replays” the log:
 - Undo uncommitted transactions
 - Redo the committed ones

Write-Ahead Log

<T1 start>  before
<T1, W, 1000, 2000>
<T1, Z, 5, 10>
<T1 commit>
 crash

REDO T1

<T1 start>  before
<T1, W, 1000, 2000>
<T1, Z, 5, 10>


UNDO T1