08- Transaction Management- Intro

School of Computer Science
University of Windsor

Dr. Shafaq Khan

Announcements

- **Test 1 – Saturday, July 8th (9 am to 10 am); Location: ER1120**
- **Next week – Lab 4 (graded)**
- **Bonus marks for paper submission to a conference by Aug 7, 2023: 5 marks**

Agenda

- **Lecture**
  - Define Transaction
  - Consistency of Database
- **Lab 3**

Introductory Questions

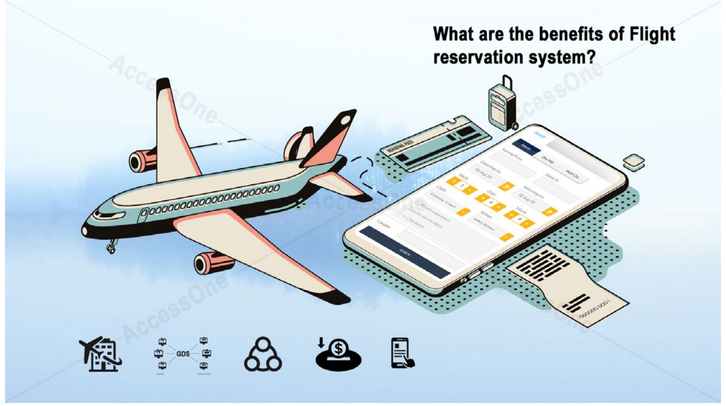
What do you mean by Transaction?

What is the purpose of concurrency control?

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

Transaction Processing Systems



Introduction to Transaction Processing (cont'd.)

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

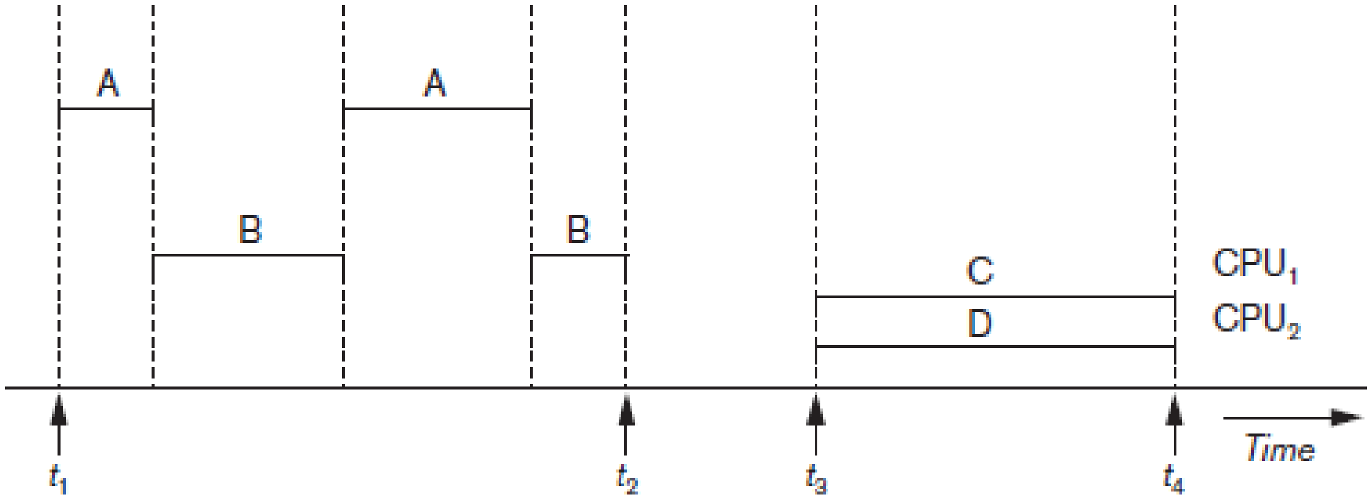


Figure 20.1 Interleaved processing versus parallel processing of concurrent transactions

Introduction

- Several users can potentially submit several **transactions** at the same time (**concurrently**)
- Transactions primarily consist of **read and write** operations of **Database objects**
- System has **interleaved operations** from various transactions so that performance is not jeopardized
- **Transaction Management** is one of the most critical and complex modules of a DBMS/DDBMS

Transactions

- A **transaction** is a logical unit of database processing.
- A transaction includes one or more database access operations
  - Insertion
  - Deletion
  - Modification
  - Retrieval
- Can either be embedded within an application program or can be specified via a high-level query language such as SQL.
- Transaction boundaries can be specified explicitly within an application program using **Begin transaction** and **End transaction**
- All operations between the two statements are considered <u>one transaction</u>
- A single application program may contain more than one transaction if it contains several **<u>transaction boundaries</u>**
- **Read-Only** Transactions – Do not update, but only retrieve
- **Read-Write** Transactions – Update

Example: Two relations from the instance of the *DreamHome* rental database

Staff (staffNo, fName, lName, position, sex, DOB, salary, branchNo)
PropertyForRent (propertyNo, street, city, postcode, type, rooms, rent, ownerNo, staffNo, branchNo)

Update the salary of a particular member of staff given the staff number, *x*	**Delete the member of staff with a given staff number *x***
	delete(**staffNo** = x)
	for all PropertyForRent records, pno
read(**staffNo** = x, salary)	begin
salary = salary * 1.1	read(**propertyNo** = pno, staffNo)
write(**staffNo** = x, salary)	if (staffNo = x) then
	begin
	staffNo = newStaffNo
	write(**property** No = pno, staffNo)
	end
	end

University of Windsor

Read Operation - Read_Item(X)

- Find the address of the disk block that contains item *X*.
- Copy that disk block into a buffer in main memory **(if that disk block is not already in some main memory buffer)**
- Copy item *X* from the buffer to the **program variable named *X***.

Write Operation: Write_Item(X)

- Find the address of the disk block that contains item *X*.
- Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
- Assign the **value of the program variable X** to the **database item X** in the buffer.
- Store the updated disk block from the buffer back to disk (either immediately

  or at some later point in time).

DBMS Buffers

- DBMS maintains a number of buffers
- Each buffer typically holds a block
- The DBMS tries to maintain the most active blocks at any given time
- If all the buffers are full and a **new block** has to be read onto the memory, an existing buffer has to make way for the new block

Review of ACID Properties

- ATOMICITY
- CONSISTENCY
- ISOLATION
- DURABILITY

Atomicity

- A transaction is an atomic unit of processing
  - It should be either performed in its entirety or not performed at all
  - **All or none** of the actions of the transactions

Example of Fund Transfer Transaction to transfer $50 from account A to account B:

Time	T_1
t_1	Begin_Transaction
t_2	read(A)
t_3	A = A – 50
t_4	write(A)
t_5	read(B)
t_6	B = B + 50
t_7	write(B)
t_8	commit

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

- Changes applied to a Database by a committed transaction must be permanent (or persist in the database)
- These changes must not be lost due to any failure (Other than the physical failure of the secondary storage medium)
- **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.

Time	T_1
t_1	Begin_Transaction
t_2	read(A)
t_3	A = A – 50
t_4	write(A)
t_5	read(B)
t_6	B = B + 50
t_7	write(B)
t_8	commit

Consistency

- Transactions should preserve the consistency of the database
  - If transactions are completely executed from the beginning to the end without logical interference from other transactions, they should **transition** the database from one **consistent state** to another

  - **Consistency requirement**: the sum of A and B is unchanged by the execution of the transaction.

Time	T_1	T_2
t_1	Begin_Transaction	
t_2	read(A)	
t_3	A = A – 50	Begin_Transaction
t_4	write(A)	read(A)
t_5	read(B)	A=A+200
t_6		write(A)
t_7	B = B + 50	commit
t_8	write(B)	
t_9	Rollback	

Isolation

- Even though actions from multiple transactions can be interleaved, the **net effect** of executing concurrent transactions must be equivalent to executing the transactions in **some serial order**
- **|t1|t2** should be equivalent to scheduling the transactions **serially** in one of the following order
  - **t1->t2**
  - **t2->t1**

Isolation requirement: if between steps 3 and 6, another transaction T_2 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).

Time	T_1	T_2
t_1	Begin_Transaction	
t_2	read(A)	
t_3	A = A – 50	Begin_Transaction
t_4	write(A)	read(A)
t_5	read(B)	A=A+200
t_6		write(A)
t_7	B = B + 50	commit
t_8	write(B)	
t_9	Rollback	

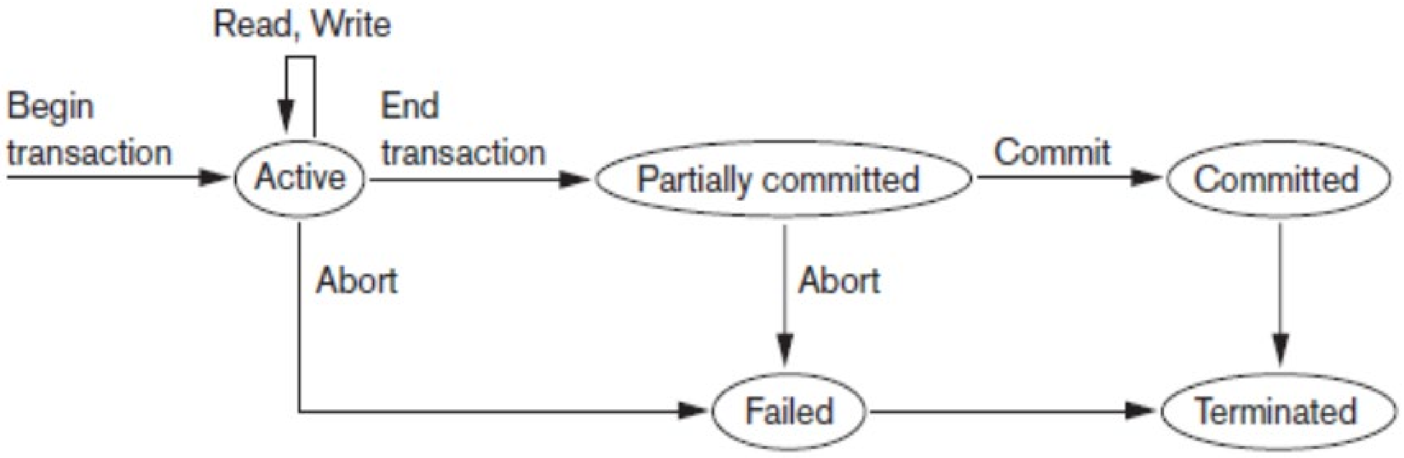
Transaction Support

- Transaction 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**.

- A committed transaction cannot be aborted.
  - If we decide that the committed transaction was a mistake, we must perform another compensating transaction to reverse its effects.

- Aborted transaction that is **rolled back** can be restarted later.
  - depending on the cause of the failure, may successfully execute and commit at that time.

State Transition Diagram



State transition diagram illustrating the states for transaction execution.

BEGIN_TRANSACTION: This marks the beginning of transaction execution.
READ or WRITE: These specify read or write operations on the database items that are executed as part of a transaction.
END_TRANSACTION: This specifies that READ and WRITE transaction operations have ended and marks the end of transaction execution. However, before completely committing, need to check for violations
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.
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**.

Concurrency Control

The process of **managing simultaneous operations** on the database without having them interfere with one another.

The Need for Concurrency Control:

A major objective in developing a database is to enable many users to access shared data concurrently.

- ✓ Relatively easy if all users are only reading data.
- ✓ When two or more users are accessing the database simultaneously and at least one is updating data, there may be interference that can result in inconsistencies.
- ✓ Although two transactions may be correct in themselves, interleaving of operations may produce an incorrect result.

Potential problems caused by concurrency

1. Lost update problem.
2. Uncommitted dependency problem.
3. Inconsistent analysis problem.

1. 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

Successfully completed update is overridden by another user.

Transaction T_1 is executing concurrently with transaction T_2

T_1 withdrawing \$10 from an account with bal_x, initially \$100.

T_2 depositing \$100 into same account.

If these transactions are executed serially, one after the other with no interleaving of operations final balance would be \$190.

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

2. Uncommitted Dependency Problem (dirty read)

Occurs when one transaction can see intermediate results of another transaction before it has committed.

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

Dirty data

- T_4 updates balx 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.

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

3. 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

Occurs when transaction reads several values but second transaction updates some of them during execution of first.

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).

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

Summary

We discussed the definition of transaction: with ACID and without ACID.

We defined the stages of Transaction Life Cycle.

We then discussed concurrency transactions and it’s problem.

Any Questions