Advanced Databases

Transactions – Concurrency

Dr. George Mertzios
Michaelmas Term

george.mertzios@durham.ac.uk
Room 2066, MCS Building
Tel: 42 429

Course Outline

- Enhanced Entity-Relationship (EER) Model
- Semistructured Databases - XML
- XML Data Manipulation - XPath, XQuery
- **Transactions** and Concurrency Control
- Distributed Transactions
- Distributed Concurrency Control

Transactions

- <u>DB Management System (DBMS):</u>

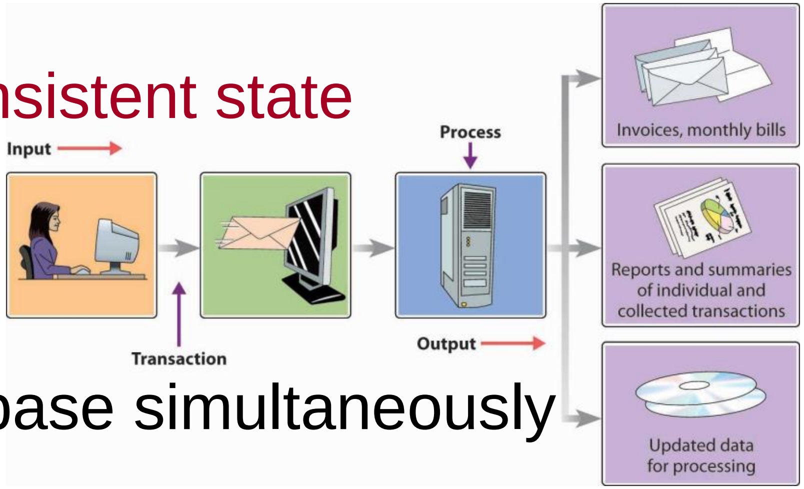
  a software that allows to manage efficiently a DB (i.e. define / create / maintain / control access)

- We need to *trust* a DBMS

$\Rightarrow$ mechanisms to ensure that the database:

  - is reliable
  - always remains in a consistent state

- Especially when:
  - software / hardware failures
  - multiple users access the database simultaneously
    - e.g.: a bank account



Transactions

- Concurrency control protocols:
  prevent database accesses to interfere with each other

- Database recovery: the process of restoring a database to a correct state after a failure

Central notion:

- Transaction: an action (or series of actions) carried out by a single user / program, which reads / updates the database
  - one *logical unit of work*: “one action” in the real world,
    e.g.: move £100 from an account to another

Transactions

Simple examples:

```
read(staffNo = x, salary)
salary = salary * 1.1
write(staffNo = x, salary)
```

update the salary of the staff who has staff number = x

```
delete(staffNo = x)
for all PropertyForRent records, pno
begin
    read(propertyNo = pno, staffNo)
    if (staffNo = x) then
    begin
        staffNo = newStaffNo
        write(propertyNo = pno, staffNo)
    end
end
```

1. remove the staff with staff number = x
2. in all properties x supervised, replace x by the staff with staff number = newStaffNo

Transactions

- At the end of a transaction:
  - database again in consistent state
  - valid integrity / referential constraints (primary / foreign keys)
- During the execution of a transaction:
  - maybe in an inconsistent state, i.e. constraints may be violated !
- A transaction can have two outcomes:
  - committed
    - when it completes successfully
  - rolled back
    - when it does *not* completes successfully

Properties of transactions

All transactions must have the ACID properties:

- Atomicity: the "all-or-nothing" property
  - a transaction is either performed entirely, or it is not performed at all
  - *Who is responsible?*
    the recovery subsystem of the DBMS

Properties of transactions

All transactions must have the ACID properties:

- Consistency: a transaction must transform the database from a consistent state to another consistent state
  - *Who is responsible?*
    both the *DBMS* and the *application developers*
- Example:
  - DBMS can enforce integrity / referential constraints
  - but: the programmer may make an error in the transaction logic and credits the wrong account

    $\Longrightarrow$ again inconsistent state !

Properties of transactions

All transactions must have the ACID properties:

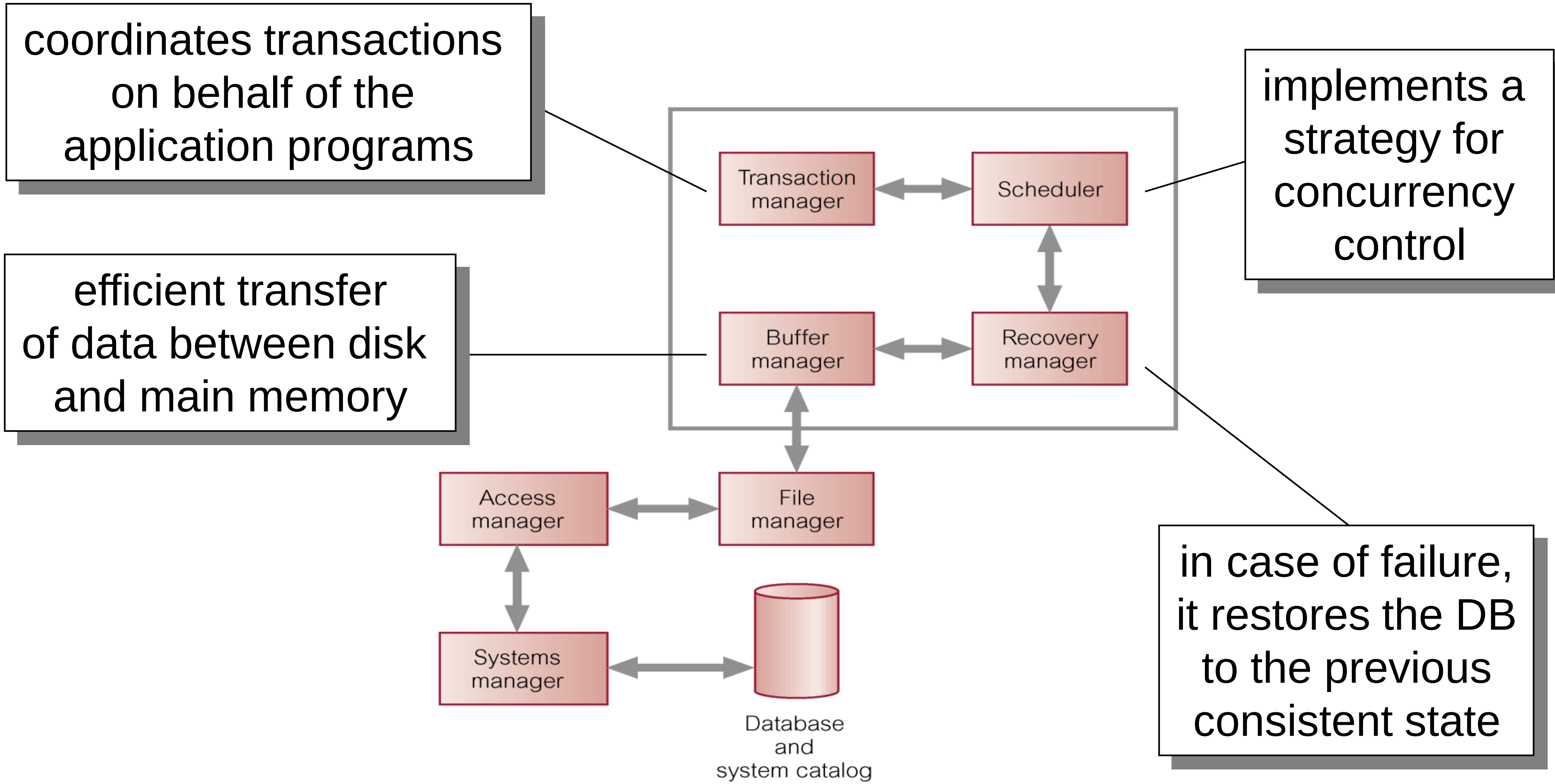
- Isolation: transactions execute independently
  - the partial effects of incomplete transactions should not be visible to other transactions
  - *Who is responsible?*

    the concurrency control system of the DBMS

- Durability: the effects of a committed transaction are permanently recorded (in the disk)
  - they should be never lost because of a failure
  - *Who is responsible?*

    the recovery subsystem of the DBMS

Database Architecture



Aims of the scheduler (or lock manager):

1. Efficiency: maximize concurrency
2. Correctness: do not allow executing transactions to interfere

Concurrency control

- Concurrency control: the process of managing simultaneous operations on the DB, without having them interfere with each other

- Main purpose: when many users access the DB

- Very different from multi-user Operating Systems:
  - an OS allows two people to *edit* a document at the same time
  - if both write, then one's changes get lost
  - not in a DBMS !

Concurrency control

- Two transactions may be:
  - both correct by themselves, but
  - when they are executed simultaneously,
    they may cause inconsistency of the database

- Three types of problems by interleaving transactions:
  - lost update problem
  - uncommitted dependency problem
  - inconsistent analysis problem

Lost update problem

- Lost update: "Override by mistake"
  - an (apparently) successfully completed update operation by one user is overridden by another user

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 can be avoided:

- by preventing T_1 from reading bal_x until after update

Uncommitted dependency problem

- Uncommitted dependency (or "dirty data"):
  - a transaction is allowed to see the 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

T_3 reads "dirty data"

abort for some reason

Reading "dirty data" can be avoided:

- prevent T_3 from reading bal_x until T_4 commits / aborts

Inconsistent analysis problem

- Inconsistent analysis:
  - a transaction reads some values, while they are being updated by another transaction

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

Solution: prevent T_6 from reading bal_x and bal_z until T_5 completed the updates

Concurrency control

- An obvious solution to all the above problems:
  - allow only one transaction at a time, i.e. one transaction is committed and then the next one can start
- However:
  - we want to maximize concurrency, i.e. parallelism
- Therefore:
  - we need mechanisms that are guaranteed to ensure consistency with concurrency

Schedules

- Schedule: a sequence of operations from a set of n concurrent transactions $T_1, T_2, \ldots, T_n$ such that:
  - the *order* of the operations in each transaction T_i is preserved in the schedule

- Serial schedule: a schedule where the operations of any two transactions are not interleaved

  - Note: the order of the transactions in a serial schedule matters !
  - Example (bank account): interest is calculated before / after
    a large deposit is made

- Non-serial schedule: a schedule where the operations of some transactions are interleaved

Serializable schedules

- Any serial schedule always leaves the database in a consistent state
  - although different schedules lead to different states
- A non-serial schedule is serializable if:
  - it produces a database state that can be produced by some serial execution of the same transactions
- How to find an equivalent serial schedule?
  - in serializability, the *order* of *read / write* operations is important

Serializable schedules

How to find an equivalent serial schedule?

- The following pairs of operations are **not in conflict:**
  - when two transactions only read some data item
  - when two transactions read or write completely separate data items
- The following pairs of operations **are in conflict:**
  - when one transaction writes a data item and another one either reads or writes the same data item
- In serializability, the ordering matters only for operations that are in conflict
  - all other pairs of operations can have any order we want !

Serializable schedules

Time	T_7	T_8	T_7	T_8	T_7	T_8
t_1	begin_transaction		begin_transaction		begin_transaction	
t_2	read(bal_x)		read(bal_x)		read(bal_x)	
t_3	write(bal_x)		write(bal_x)		write(bal_x)	
t_4		begin_transaction		begin_transaction	read(bal_y)	
t_5		read(bal_x)		read(bal_x)	write(bal_y)	
t_6		write(bal_x)	read(bal_y)		commit	
t_7	read(bal_y)			write(bal_x)		begin_transaction
t_8	write(bal_y)		write(bal_y)			read(bal_x)
t_9	commit		commit			write(bal_x)
t_{10}		read(bal_y)		read(bal_y)		read(bal_y)
t_{11}		write(bal_y)		write(bal_y)		write(bal_y)
t_{12}		commit		commit		commit
	serializable schedule		serializable schedule		serial schedule	

This type of serializability is called "conflict serializability"

Testing conflict serializability

To check whether a given non-serial schedule is (conflict) serializable or not, we construct the precedence graph (or serialization graph):

- A directed graph $G = (N, E)$ with a set of nodes N and a set of directed edges E, with:
  - a node for each transaction
  - a directed edge $T_i \rightarrow T_j$ whenever:
    - T_j reads a value of an item written by T_i, or
    - T_j writes a value into an item after it has been read or written by T_i

Testing conflict serializability

In the precedence graph, an edge $T_i \rightarrow T_j$ means that in any equivalent serial schedule, T_i appears before T_j

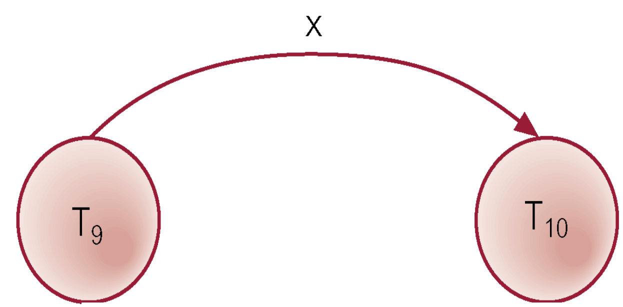
- It can be proved:
  - A schedule is (conflict) serializable if and only if its precedence graph has no directed cycle

$\Rightarrow$ efficient (polynomial-time) algorithm for checking serializability !

Testing conflict serializability

In the precedence graph, an edge $T_i \rightarrow T_j$ means that in any equivalent serial schedule, T_i appears before T_j

Time	T_9	T_{10}
t_1	begin_transaction	
t_2	read(bal_x)	
t_3	$bal_x = bal_x + 100$	
t_4	write(bal_x)	begin_transaction
t_5		read(bal_x)
t_6		$bal_x = bal_x$ *1.1
t_7		write(bal_x)
t_8		read(bal_y)
t_9		$bal_y = bal_y$ *1.1
t_{10}		write(bal_y)
t_{11}	read(bal_y)	commit
t_{12}	$bal_y = bal_y - 100$	
t_{13}	write(bal_y)	
t_{14}	commit	



A non-serializable schedule

Other types of serializability

Two schedules S_1 and S_2 are view equivalent, if:

- for each data item x, if transaction T_i reads the initial value of x in S_1, then T_i reads the initial value of x also in S_2
- for each data item x, if the last write operation on x in S_1 was done by transaction T_i, then T_i must perform the last write operation on x also in S_2
- for a read operation on data item x by transaction T_i in S_1, if the value of x read by T_i was written by transaction T_j, then T_i must also read the value of x produced by T_j in S_2

In other words: S_1 and S_2 are view equivalent if they return the same results

Other types of serializability

- A non-serial schedule is view serializable if:
  - it is view equivalent to a serial schedule

- Conflict serializable $\Longrightarrow$ View serializable
- The converse is not true!

- It can be proved:
  - testing for view serializability is NP-complete, i.e. most probably not efficient
  - every view serializable schedule which is not conflict serializable has one or more blind writes

Other types of serializability

Example of a view serializable schedule:
(but not conflict serializable)

Time	T_{11}	T_{12}	T_{13}
t_1	begin_transaction		
t_2	read(**bal$_x$**)		
t_3		begin_transaction	
t_4		write(**bal$_x$**)	
t_5		commit	
t_6	write(**bal$_x$**)		
t_7	commit		
t_8			begin_transaction
t_9			write(**bal$_x$**)
t_{10}			commit

blind writes
(without read)

Summary of the Lecture

- Transactions:
  - committed
  - rolled back
  - ACID properties
- Concurrency control:
  - lost update problem
  - uncommitted dependency problem (dirty data)
  - inconsistent analysis problem
  - (conflict) serializability
  - view serializability