University of Windsor

Transaction Management- Intro

Kalyani Selvarajah
School of Computer Science
University of Windsor

Advanced Database Topics
COMP 8157 01
Fall 2023

TODAY'S AGENDA

Transaction

Consistency of Database

Concurrency control

Demo: SQL server

https://domains.upperlink.ng/elementor-947/

PRE-ASSESSMENT

1. What do you mean by **Transaction**?
2. What are the potential problems of **Concurrency control**?
3. How to run a transaction in SQL server?

Introduction: Transaction

A **transaction** is an action, or series of actions, carried out by user or application, which reads or updates contents of database.

It is the basic unit of change in a DBMS.

Source: https://www.pinterest.com/pin/775533998318621246/

TRANSACTION EXAMPLE (1)

Transfer $200 from Tom’s account to Ann’s account.

Transaction:

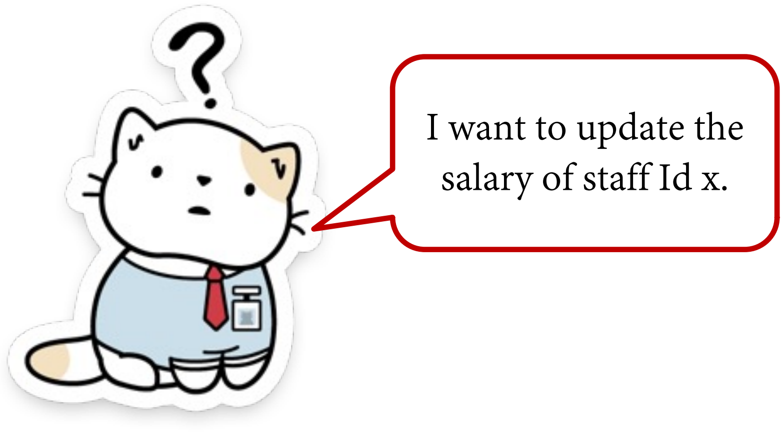
→ Deduct $200 from Tom’s account.

→ Add $200 to Ann’s account.

TRANSACTION EXAMPLE (2)

Staff (staffNo, fName, lName, position, sex, DOB, salary)

PropertyForRent (propertvNo, street, city, postcode, type, rooms, rent, staffNo)



Database operations

Non-database operations

TRANSACTION EXAMPLE (2)

Staff (staffNo, fName, lName, position, sex, DOB, salary)

PropertyForRent (propertvNo, street, city, postcode, type, rooms, rent, staffNo)

I want to delete a staff ID x because he resigned.

DELETE(staffNo = x)

Staff

StaffNo	fName	lName	Position	Sex	DOB	Salary
~~X~~	~~John~~	~~White~~	~~Manager~~	~~M~~	~~1-Oct-45~~	~~30000~~
Y	Ann	Beech	Assistant	F	10-Nov-60	12000
Z	David	Ford	Supervisor	M	24-Mar-58	18000

PropertyForRent

PropertyNo	Street	City	Postcode	Type	Rooms	Rent	StaffNo
PA14	16 Holhead	Aberdeen	AB7 5SU	House	6	SA9	Z
PL94	6 Argyll St	London	NW2	Flat	4	SL41	X

TRANSACTION EXAMPLE (2)

Staff (staffNo, fName, lName, position, sex, DOB, salary)

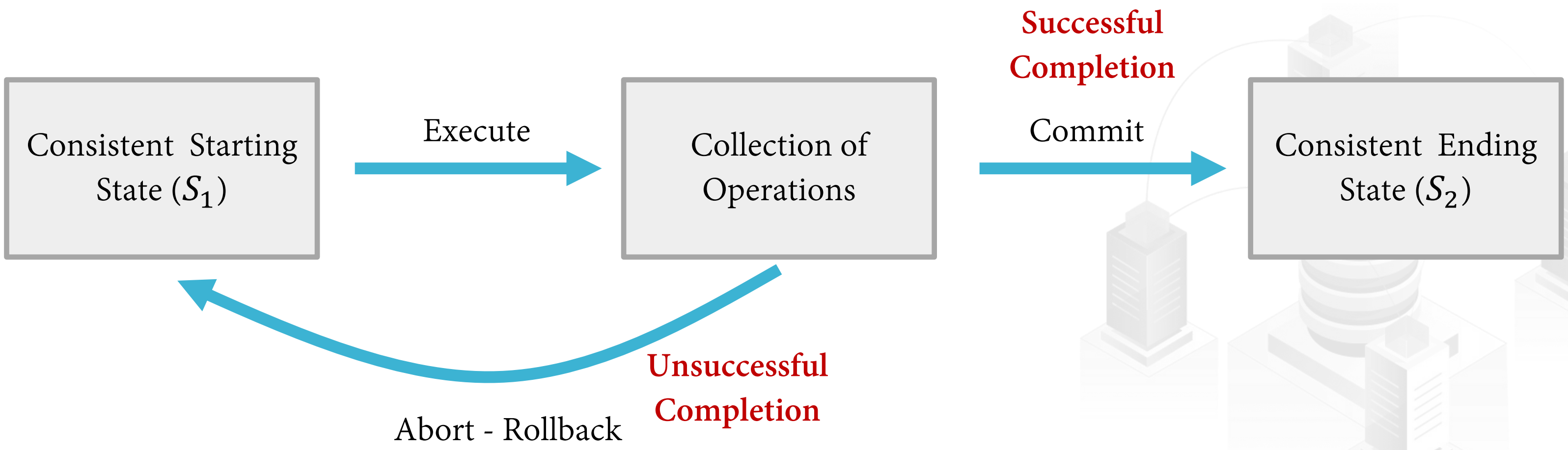
PropertyForRent (propertvNo, street, city, postcode, type, rooms, rent, staffNo)

I want to delete a staff ID x because he resigned.

```
DELETE(staffNo = x)
FOR ALL PropertyForRent records, pno
BEGIN
    READ(propertyNo = pno, staffNo)
    IF (staffNo = x) THEN
    BEGIN
        staffNo = newStaffNo
        WRITE(property No = pno, staffNo)
    END
END
```

TRANSACTION LIFE CYCLE

By Jim Gray

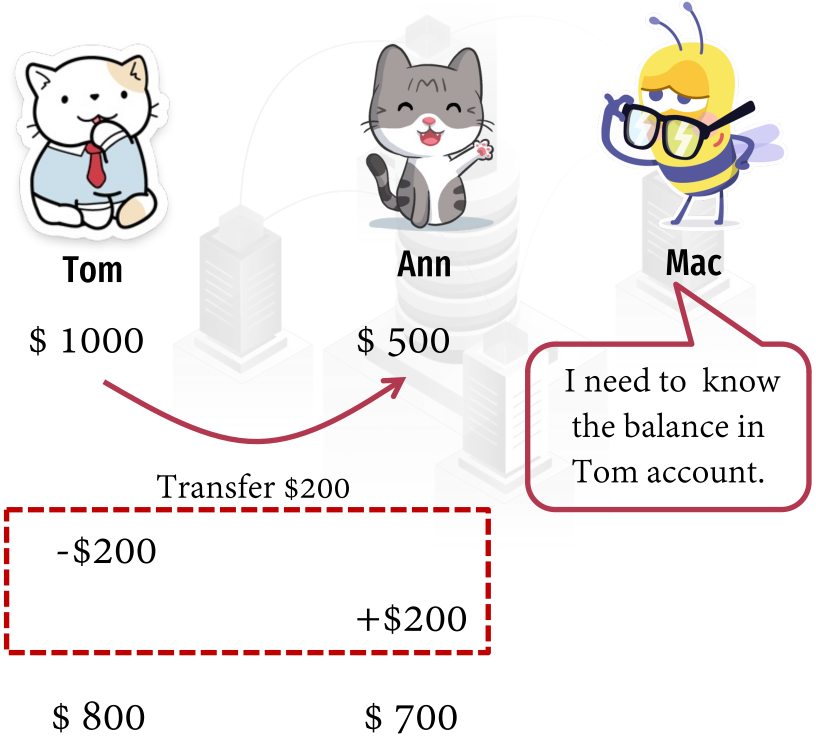


Success - transaction commits and database reaches a new consistent state.
Failure - transaction aborts, and database must be restored to consistent state before it started. Such a transaction is rolled back or undone.
NOTE: If we committed a transaction, it cannot be aborted. However, an aborted transaction that is rolled back can be restarted later.

PROPERTIES OF TRANSACTIONS (ACID)

ACID

- **Atomicity:** "All or nothing…"
- **Consistency:** “It looks correct to me..”
- **Isolation:** "All by myself…"
- **Durability:** "I will survive…"



A

B

CONCURRENCY CONTROL

Concurrent access of data X:

TIME

Time	T_1	T_2
t_1		BEGIN
t_2	BEGIN	READ(X)
t_3	READ(X)	X= X+100
t_4	X= X-10	WRITE(X)
t_5	WRITE(X)	COMMIT
t_6	COMMIT	

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

Concurrency Control is the protocol to allow transactions to access a database in a multi-programmed fashion while preserving the illusion that each of them is executing alone on a dedicated system.

POTENTIAL PROBLEMS OF CONCURRENCY

1. **Lost update problem:** Successfully completed update is overridden by another user.

2. **Uncommitted dependency problem:** Occurs when one transaction can see intermediate results of another transaction before it has committed.

3. **Inconsistent analysis problem:** Occurs when transaction reads several values but second transaction updates some of them during execution of first.

4. **Nonrepeatable read**: Occur when a transaction rereads a data item it has previously read but, in between, another transaction has modified it.

1. LOST UPDATE PROBLEM

Assume initially account X has \$100.

T_1 withdraws \$10 from account X.

T_2 deposits \$100 into same account X.

TIME

Time	T_1	T_2	X
t_1		BEGIN	100
t_2	BEGIN	READ(X)	100
t_3	READ(X)	X= X+100	100
t_4	X= X-10	WRITE(X)	200
t_5	WRITE(X)	COMMIT	90
t_6	COMMIT		90

The lost update problem.

2. UNCOMMITTED DEPENDENCY PROBLEM (DIRTY READ)

Assume initially account X has \$100.

T_1 withdraws \$10 from account X.

T_2 deposits \$100 into same account X.

TIME

Time	T_1	T_2	X
t_1		BEGIN	100
t_2		READ(X)	100
t_3		X= X+100	100
t_4	BEGIN	WRITE(X)	200
t_5	READ(X)	…	200
t_6	X= X-10	ROLLBACK	100
t_7	WRITE(X)		190
t_8	COMMIT		190

The uncommitted dependency problem.

3. INCONSISTENT ANALYSIS PROBLEM

Assume initially

→ account X has \$100.

→ account Y has \$50.

→ account Z has \$25.

T_1 transfers \$10 from account X to Z.

T_2 analyzes the total of accounts X, Y & Z.

TIME

Time	T_1	T_2	X	Y	Z	SUM
t_1		BEGIN	100	50	25	
t_2	BEGIN	SUM=0	100	50	25	0
t_3	READ(X)	READ(X)	100	50	25	0
t_4	X= X-10	SUM=SUM+X	100	50	25	100
t_5	WRITE(X)	READ(Y)	90	50	25	100
t_6	READ(Z)	SUM=SUM+Y	90	50	25	150
t_7	Z=Z+10		90	50	25	150
t_8	WRITE(Z)		90	50	35	150
t_9	COMMIT	READ(Z)	90	50	35	150
t_{10}		SUM=SUM+Z	90	50	35	185
t_{11}		COMMIT	90	50	35	185

The inconsistent analysis problem.

4. NONREPEATABLE READ (PHANTOM READ)

Assume initially account X has \$100.

T_1 withdraws \$10 from account X.

T_2 deposits \$100 into same account X.

TIME

Time	T_1	T_2	X
t_1		BEGIN	100
t_2	BEGIN	READ(X)	100
t_3	READ(X)	….	100
t_4	X= X-10		90
t_5	WRITE(X)		90
t_6	COMMIT	…..	90
		READ (X)	90
		COMMIT	

The unrepeatable read problem .

Serializability and Recoverability

Objective of a concurrency control protocol is to schedule transactions in such a way as to avoid any interference.

Obvious solution?

- allow only one transaction to execute at a time: one transaction is committed before the next transaction is allowed to begin.

Aim of a multi-user DBMS: to maximize the degree of concurrency or parallelism in the system.

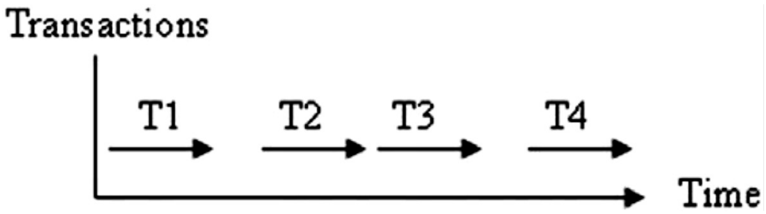
- transactions that can execute without interfering with one another can run in parallel.
- How?

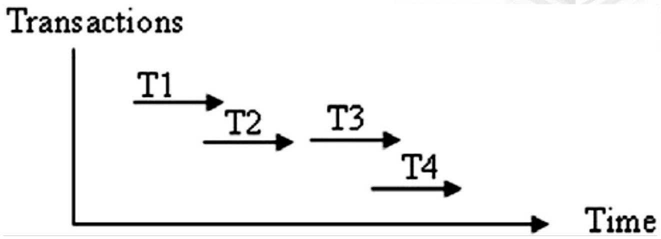
Schedule

Schedule: A sequence of the operations by a set of concurrent transactions that preserves the order of the operations in each of the individual transactions.

Serial Schedule: A schedule where the operations of each transaction are executed consecutively without any interleaved operations from other transactions.

Non serial schedule: A schedule where the operations from a set of concurrent transactions are interleaved.





Conflicts

A conflict occurs when two running transactions perform noncompatible operations on the same data item of the database.

A conflict occurs when one transaction writes an item that another transaction is reading or writing.

	READ T_2	WRITE T_2
READ T_1	No Conflict	Conflict
WRITE T_1	Conflict	Conflict

CONFLICT MATRIX.

Conflicts: Example

Conflict operations:

- ✓ the operations $r_1(X)$ and $w_2(X)$,
- ✓ the operations $r_2(X)$ and $w_1(X)$,
- ✓ and the operations $w_1(X)$ and $w_2(X)$

Do not Conflict:

- ✓ the operations $r_1(X)$ and $r_2(X)$ -- since they are both read operations.
- ✓ the operations $w_2(X)$ and $w_1(Y)$ -- they operate on distinct data items X and Y.
- ✓ operations $r_1(X)$ and $w_1(X)$ -- they belong to the same transaction.

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);	

Serializability

Objective of serializability is to find non-serial schedules that allow transactions to execute concurrently without interfering with one another.

In serializability, **ordering of read/writes** is important:

- ✓ If two transactions only read a data item, they do not conflict, and order is not important.
- ✓ If two transactions either read or write separate data items, they do not conflict, and order is not important.
- ✓ If one transaction writes a data item and another reads or writes same data item, order of execution is important

Serializability

Time	T_1	T_2
t_1	BEGIN	
t_2	READ(X)	
t_3	WRITE(X)	
t_4		BEGIN
t_5		READ(X)
t_6		WRITE(X)
t_7	READ(Y)	
t_8	WRITE(Y)	
t_9	COMMIT	
t_{10}		READ(Y)
t_{11}		WRITE(Y)
t_{12}		COMMIT

(a) Schedule S_1

T_1	T_2
BEGIN	
READ(X)	
WRITE(X)	
	BEGIN
	READ(X)
READ(Y)	
	WRITE(X)
WRITE(Y)	
COMMIT	
	READ(Y)
	WRITE(Y)
	COMMIT

(b) Schedule S_2

T_1	T_2
BEGIN	
READ(X)	
WRITE(X)	
READ(Y)	
WRITE(Y)	
COMMIT	
	BEGIN
	READ(X)
	WRITE(X)
	READ(Y)
	WRITE(Y)
	COMMIT

(c) Schedule S_3

TIME

Equivalent schedules: (a) nonserial schedule S_1; (b) nonserial schedule S_2 equivalent to S_1; (c) serial schedule S_3, equivalent to S_1 and S_2.

Conflict serializability

Conflict serializable schedule orders any **conflicting operations** in same way as some serial execution.

Testing for conflict serializability:

Under constrained write rule (transaction updates data item based on its old value, which is first read), use **precedence graph** to test for serializability.

Precedence Graph:

Create:

node for each transaction;

- ✓ a directed edge $T_i \longrightarrow T_j$, if T_j reads the value of an item written by T_i;
- ✓ a directed edge $T_i \longrightarrow T_j$, if T_j writes a value into an item after it has been read by T_i.
- ✓ a directed edge $T_i \longrightarrow T_j$, if T_j writes a value into an item after it has been written by T_i.

$R_i(A) \rightarrow W_j(A)$

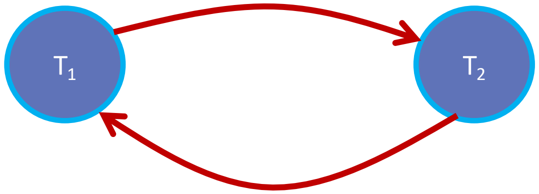
$W_i(A) \rightarrow R_j(A)$

$W_i(A) \rightarrow W_j(A)$

If there is a cycle in the precedence graph, schedule S is not (conflict) serializable; if there is no cycle, S is serializable.

Example - Non-conflict serializable schedule

T_1 is transferring \$100 from one account with balance X to another account with balance Y.
T_2 is increasing balance of these two accounts by 10%.



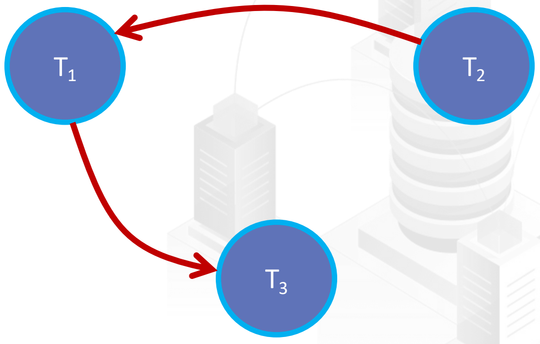
Precedence graph has a cycle and so is not serializable.

Time	T_1	T_2
t_1	BEGIN	
t_2	READ(X)	
t_3	X=X+100	
t_4	WRITE(X)	BEGIN
t_5		READ(X)
t_6		X=X*1.1
t_7		WRITE(X)
t_8		READ(Y)
t_9		Y=Y*1.1
t_{10}		WRITE(Y)
t_{11}	READ(Y)	COMMIT
t_{12}	Y=Y-100	
t_{13}	WRITE(Y)	
t_{14}	COMMIT	

Two concurrent update transactions that are not conflict serializable.

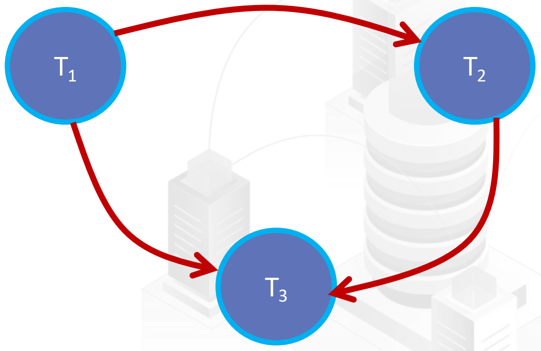
Example

Time	T_1	T_2	T_3
t_1	READ(A)		
t_2	WRITE(A)		
t_3			READ(A)
t_4			WRITE(A)
t_5			COMMIT
t_6		READ(B)	
t_7		WRITE(B)	
t_8		COMMIT	
t_9	READ(B)		
t_{10}	WRITE(B)		
t_{10}	COMMIT		



Example

Time	T_1	T_2	T_3
T_1	READ(A)		
T_2	READ(C)		
T_3	WRITE(A)		
T_4		READ(B)	
T_5	WRITE (C)		
T_6		READ(A)	
T_7			READ (C)
T_8		WRITE(B)	
T_9			READ (B)
T_{10}			WRITE(C)
T_{11}		WRITE(A)	
T_{12}			WRITE(B)



Precedence graph has no cycle that's why it is serializable.

View Serializability

There are several other types of serializability that offer less stringent definitions of schedule equivalence than that offered by conflict serializability.

One less restrictive definition is called **view serializability.**

Two schedules S_1 and S_2 consisting of the same operations from n transactions $T_1, T_2, \ldots, T_n$ are view equivalent if:

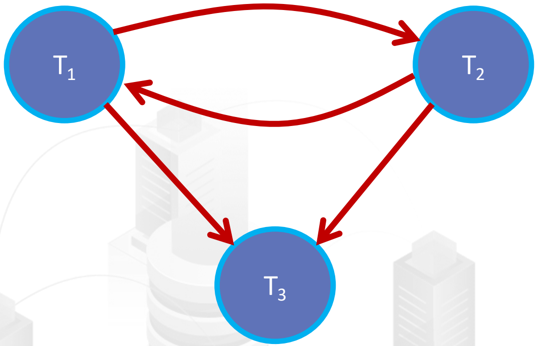
- ✓ For each data item x, if T_i reads initial value of x in S_1, T_i must also read initial value of x in S_2.
- ✓ For each read on x by T_i in S_1, if value read by T_i is written by T_j, T_i must also read value of x produced by T_j in S_2.
- ✓ For each data item x, if last write on x performed by T_i in S_1, same transaction must perform final write on x in S_2.

View Serializability

TIME

Time	T_1	T_2	T_3
t_1	BEGIN		
t_2	READ(X)		
t_3			
t_4		BEGIN	
t_5		WRITE(X)	
t_6	WRITE (X)	COMMIT	
t_7	COMMIT		
t_8			BEGIN
t_9			WRITE(X)
t_{10}			COMMIT

It is view serializable, as it is equivalent to the serial schedule T_1 followed by T_2 followed by T_3



This graph contains a cycle indicating that the schedule is not conflict serializable;

we can see that the edge $T_2 \longrightarrow T_1$ should not have been inserted into the graph, as the values of X written by T_1 and T_2 were never used by any other transaction because of the blind writes.

$$S1 = R1(X), W2(X), W1(X), W3(X)$$
$$S2 = R1(X), W1(X), W2(X), W3(X)$$

Schedules S1 and S2 are view equivalent