COMP9120

Week 8: Transaction Management

Semester 2, 2022

Professor Athman Bouguettaya
School of Computer Science

THE UNIVERSITY OF SYDNEY

Contents developed by Dr Lijun Chang

THE UNIVERSITY OF SYDNEY

Acknowledgement of Country

I would like to acknowledge the Traditional Owners of Australia and recognise their continuing connection to land, water and culture. I am currently on the land of the Darug people and pay my respects to their Elders, past, present and emerging.

I further acknowledge the Traditional Owners of the country on which you are on and pay respects to their Elders, past, present and future.

THE UNIVERSITY OF SYDNEY

Copyright warning

Outline

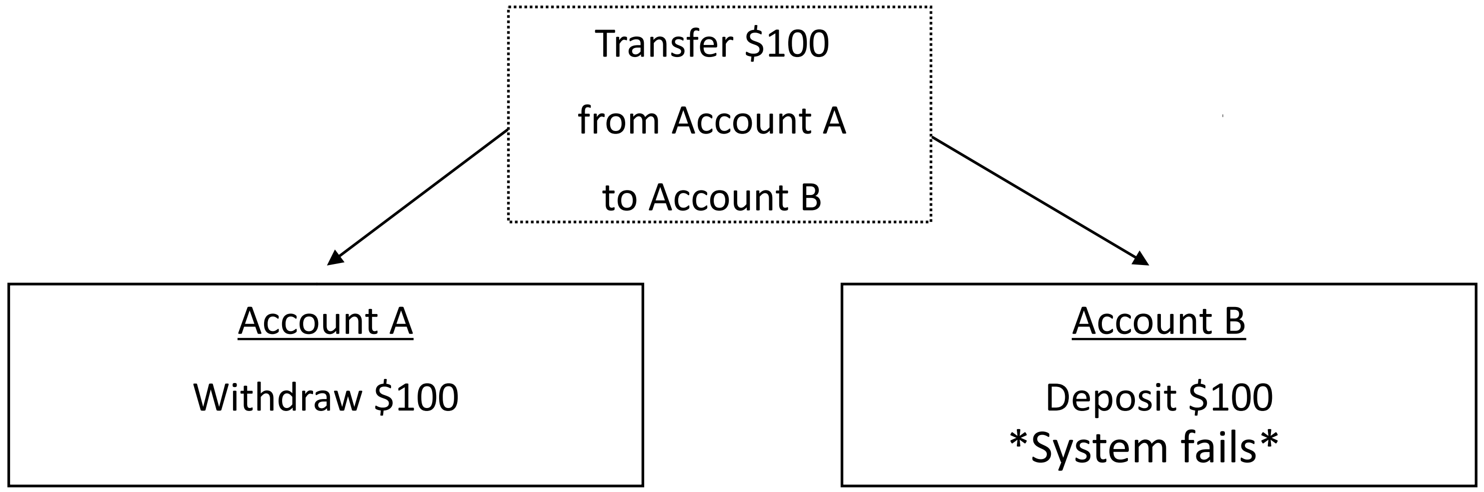
- What is a transaction?

- Four desirable properties of transaction:
  - **A**tomicity, **C**onsistency, **I**solation, **D**urability

What is the meaning of these properties? Why do we need these properties? How are these properties ensured to hold?

Motivation

Transfer Operation



Account balance successfully updated for Account A

System recovers but database state no longer reflects the account transfer (short $100)

Should group withdraw & deposit operations together – so that either they *both succeed*, or *not happen at all*

```
BEGIN;
Withdraw $100 from Account A;
Deposit $100 into Account B;
COMMIT;
```

What is a transaction?

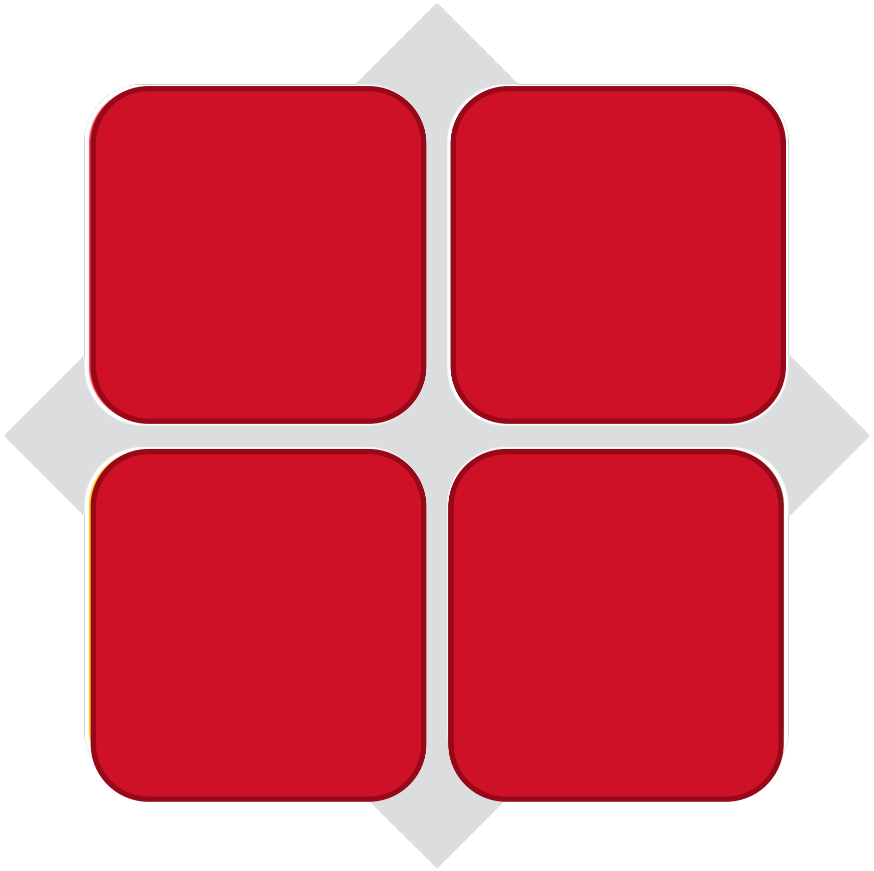
- a database program
- an atomic unit
  - Atomicity implies that the effect of the transaction is that it executes either *fully* or *not at all*.

In other words:

- ***A program** is executed to change the database state in a correct way*
  - e.g., Bank balance must be updated on both accounts when a transfer is made
- Such an execution of a program is often modelled as a ***transaction***:
  a collection of one or more operations (consisting of reads and writes at the DBMS level) which reflect a discrete unit of work
  - Transactions should conform to some property requirements (***ACID** properties*).

Required properties of a transaction

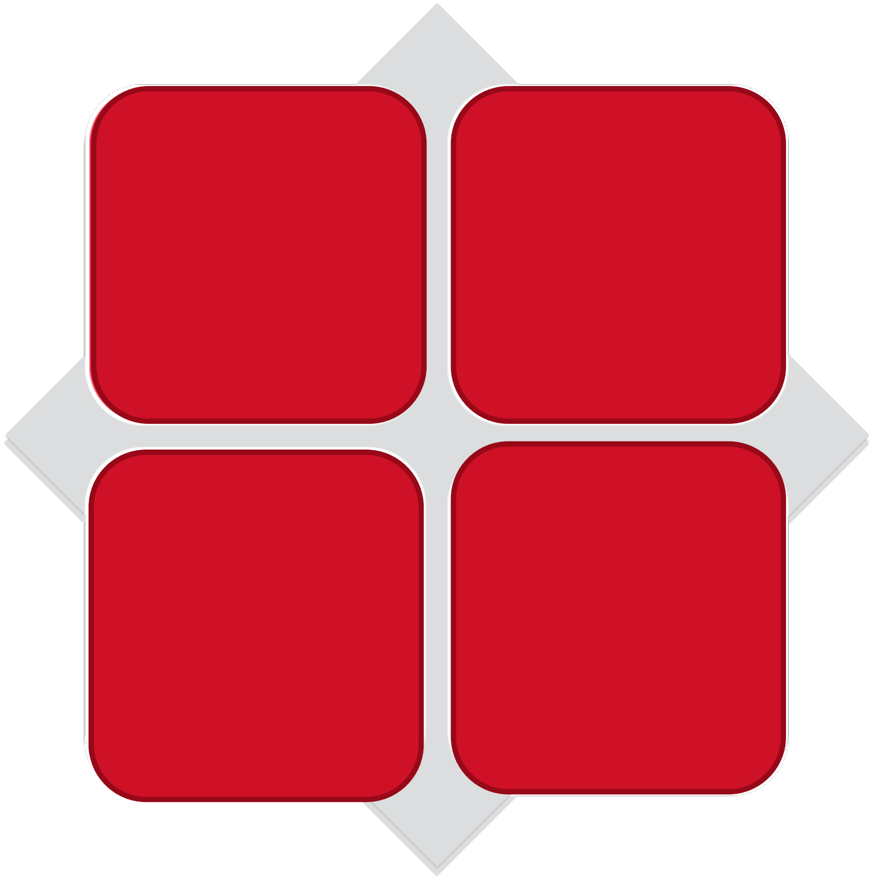
- ***A*tomicity**: A transaction is either performed entirely or not performed at all. The whole transaction is treated as one atomic operation.
- ***C*onsistency**: A correct execution of a transaction must take a database from one consistent state to another. The transaction, if executed separately from others, leaves the database in a correct state, i.e., *all of its integrity constraints* are satisfied.



Required properties of a transaction (cont'd)

- *Isolation:* Effect of multiple transactions is the same as these transactions running one after another. For every two transactions running concurrently, the effect of the execution is such that it is as if they are running sequentially, i.e., a transaction is unaware of other transactions that may be running concurrently.

- *Durability:* Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failures. This means that once a transaction completes successfully, the results will survive even if there is a system failure.



Outline

- **Consistency**
  - What, why, and how?
- **Atomicity**
- **Durability**
- **Isolation**

Transaction consistency

- Assuming the database is in a consistent state initially (satisfying all constraints), when the transaction completes, it is said that the **transaction is consistent** if:
  1. *All database constraints* are satisfied
  2. The new database state satisfies the specifications of the transaction

Consistency refers to the requirement that any given transaction can only modify data in allowed ways. Therefore, any data written to the database must agree with all defined rules, including constraints, cascades, and triggers.

- Each transaction should preserve the consistency of the database. Note that this is mainly the responsibility of the application developer!
  - Database cannot 'fix' the correctness of a badly coded transaction
  - Example of a bad transaction for a bank transfer:
    - Withdraw $100 from account A, but only deposit $90 into account B.

Transaction: Deferring Integrity Constraints

```
CREATE TABLE UnitOfStudy (
  uos_code          VARCHAR(8),
  title             VARCHAR(20),
  lecturer_id       INTEGER,
  credit_points     INTEGER,
  CONSTRAINT UnitOfStudy_PK PRIMARY KEY (uos_code),
  CONSTRAINT UnitOfStudy_FK FOREIGN KEY (lecturer_id)
     REFERENCES Lecturer DEFERRABLE INITIALLY IMMEDIATE
);

BEGIN;
   SET CONSTRAINTS UnitOfStudy_FK DEFERRED;
   INSERT INTO UnitOfStudy VALUES('INFO1000', 'Graphics', 3, 6);
   INSERT INTO Lecturer VALUES(3, 'Steve', CSE);
   SET CONSTRAINTS UnitOfStudy_FK IMMEDIATE ; -- Optional

COMMIT;
```

UnitOfStudy

uos_code	**title**	**lecturer_id**	**credit_points**
COMP9120	DBMS	1	6
COMP9007	Algorithm	2	6

Lecturer

Lecturer_id	**name**	**department**
1	Adam	CSE
2	Lily	IT

Outline

- Consistency
- **Atomicity**
  - What, why, and how?
- Durability
- Isolation

Transactions should be Atomic

- A DBMS user can think of a transaction as always executing all of its operations in *one single step*, or *not* executing any operations at all.
  - Every transaction should act as an *atomic* operation.

- A real-world transaction is expected to *either happens* or *not happen at all* (e.g., for bank transfer, either both withdrawal + deposit occur, or neither occurs).
  - *Partially completed* transaction can lead to an incorrect database state.

- DBMS ***logs*** all actions so that would need to be ***undone*** if the transaction *aborted* (i.e., it is *incomplete*).
  - E.g., in case of a failure, all actions of *not-committed* transactions are *undone*.

Commit and Abort

- If the transaction successfully completes, it is said to have **committed**
  - The DBMS is responsible for ensuring that all changes to the database have been saved
- If the transaction does not successfully complete, it is said to have been **aborted**
  - The DBMS is responsible for undoing, i.e., **rolling back**, all changes in the database that the transaction had made
  - Possible reasons for **abort**:
    - System crash
    - Transaction aborted by system, e.g.,
      - Transaction or connection time-out,
      - Violation of constraints
    - Transaction explicit request to roll back

Writing a transaction in SQL

- 3 key relevant SQL commands to know:
  - **BEGIN**
  - **COMMIT** *requests* to **commit** current transaction
  - **ROLLBACK** causes current transaction to **abort**.

- Can also **SET AUTOCOMMIT OFF** or **SET AUTOCOMMIT ON**
  - With *auto-commit on*, each statement is *its own transaction* and 'auto-commits'
  - With *auto-commit off*, statements form part of a larger transaction delimited by the keywords discussed above.
  - Different clients have different defaults for auto-commit.

What value should be returned?

<u>uosCode</u>	lecturerId
COMP5138	3456
COMP5338	4567

```
BEGIN;
UPDATE Course SET lecturerId=1234 WHERE uosCode='COMP5138';
COMMIT;
SELECT lecturerId FROM Course WHERE uosCode='COMP5138';
```

1. 1234 ✓
2. 3456
3. 4567

What value should be returned?

uosCode	lecturerId
COMP5138	3456
COMP5338	4567

```
BEGIN;
UPDATE Course SET lecturerId=1234 WHERE uosCode='COMP5138';
ROLLBACK;
SELECT lecturerId FROM Course WHERE uosCode='COMP5138';
```

1. 1234
2. [✓ 3456]
3. 4567

What value should be returned?

uosCode	lecturerId
COMP5138	3456
COMP5338	4567

```
BEGIN;
UPDATE Course SET lecturerId=4567 WHERE uosCode='COMP5138';
COMMIT;
SELECT lecturerId FROM Course WHERE uosCode='COMP5138';
```

1. 1234
2. 3456 ✓
3. 4567

Writing a transaction with JDBC

- APIs, like JDBC, often provide explicit functions for controlling the transaction semantics

- By default, transactions are in **AutoCommit** mode, i.e., it is **on** by default.
  - Each SQL statement is considered its own transaction.
  - No explicit commit, no transactions with more than one statement…

- 3 new methods to know (all for JDBC connection class):
  - **setAutoCommit(false)**
  - **commit**()
  - **rollback**()

Transactions with Python DB-API

› DB-API provides transaction control via connection object methods

› According to DB-API spec, db connections initially start with **auto-commit mode** switched **off**

  - Can use autocommit attribute of connection to set auto-commit mode

› Transaction control

  - **conn.commit()** successfully finishes (commits) current transaction
  - **conn.rollback()** aborts current transaction

Outline

- Consistency
- Atomicity
- **Durability**
  - What, why, and how?
- Isolation

Transactions should be Durable

- Once a transaction is committed, its effects should persist in a database, and these effects should be permanent even if the system crashes.
  - A database should always be able to recover to the last consistent state

- Problems with recovery from failures:

  1. Non-erasable writes (e.g. printers). how do we recover?
     - Solution: do not write until transaction commits.
  2. Long-lived transactions: typically hours and days of computations.
     - Need to see partial results, thus violating atomicity.

Transaction failure

- Scenario after failure of a transaction:
- Assuming that partial updates have already been written back to disk, the alternatives are:
  1. Either re-execute transaction but... we may leave the database inconsistent because we did not complete the previous transaction execution.
  2. Or do not do anything. this also may leave the database inconsistent because of the partial execution.
- Example: bank transfer transaction.

Withdraw \$100 from Account A; ***Failure***
Deposit \$100 into Account B; $\longleftarrow$

Transaction logs

- Solution: use stable storage (e.g., hard disk) as a log to store a history of modifications made to the database.
- Mechanism:

1. Every transaction has a “log” associated with it.
2. Every time an exclusive lock on an item is granted, any update to an item is also mirrored in the log .
3. If a transaction aborts, depending on the recovery protocol, use the log to undo/redo the transaction.

- Undo operation: bring back an item to its initial value (i.e., before the transaction started execution).
- Redo operation: copy the log value of an item from stable storage to disk (making the modification now persistent/permanent).

Hardware solution

› Hardware implementation of durability:
  - Database is stored redundantly on mass storage devices to protect against media failure (e.g., RAID)

Outline

- Consistency
- Atomicity
- Durability
- Isolation
  - What, why, and how?
  - Isolation through conflict serializability
  - Lock-based concurrency control

Purpose of Concurrency Control

- Control concurrent access to the database to ensure correctness and efficiency
- Problems arise when concurrent access involves *updates* to the database

Concurrent access issues

- Types of problems that can compromise database correctness in the presence of concurrent access. There are three main issues:
  - **Lost update** problem
  - **Temporary update** problem
  - **Incorrect summary** problem

Interference issues

- Interference of concurrent operations

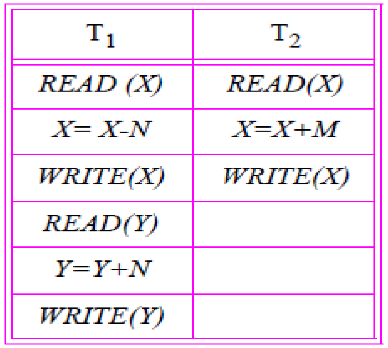
  - **Lost update problem**: occurs when two transactions are interleaved in such a way that makes an item final value incorrect. That is, a transaction does not see its update but the updates of other transactions. The update of a transactions is lost because another transaction has updated this value.
  - **Temporary update problem**: this occurs when a transaction updates an item and then fails. Another transaction reads that item unaware it has been changed back to its original value.
  - **Incorrect summary problem**: this happens when a transaction is accessing/updating an aggregate of records. If a concurrent transaction is allowed, it may potentially access a mixture of old and new values.

THE UNIVERSITY OF SYDNEY

Lost update problem

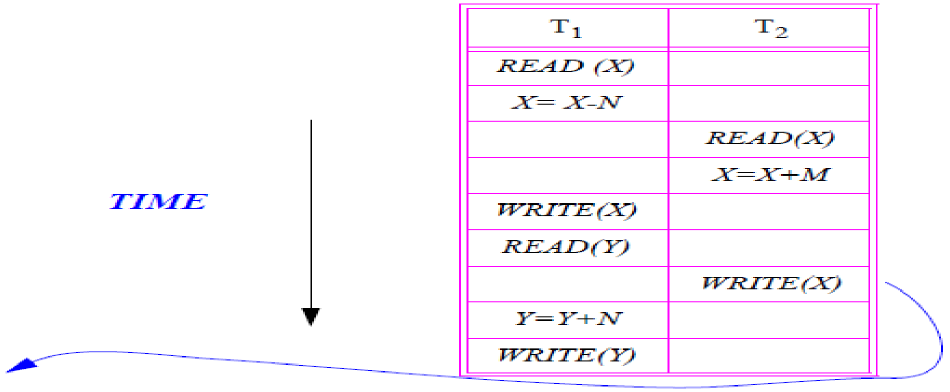
› Example of the **lost update problem**:

› Consider 2 concurrent transactions T1 and T2. T1 is a bank account transfer. T2 is a bank account deposit. X and Y are two different accounts.



T_1	T_2
READ (X)	READ(X)
X= X-N	X=X+M
WRITE(X)	WRITE(X)
READ(Y)	
Y=Y+N	
WRITE(Y)	

› First case: assume that the amount transferred is N=$50, and the amount deposited is M=$100 and that initially X=$60, Y=$30. One possible schedule of execution is:



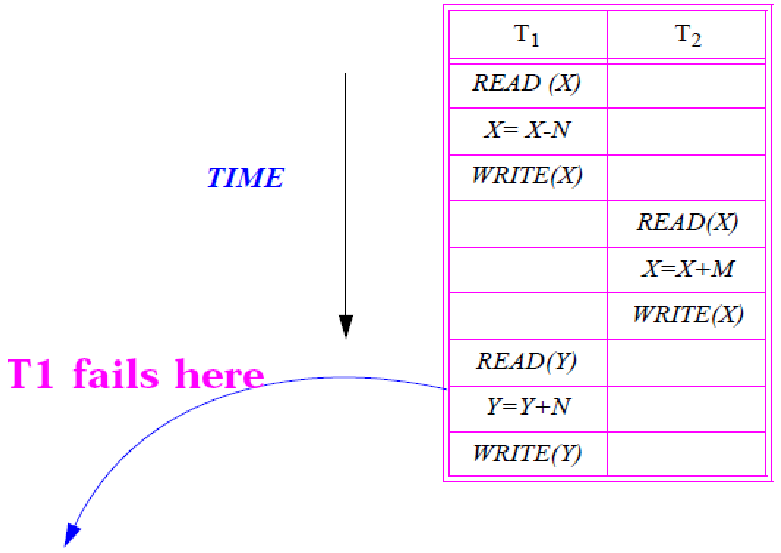
T_1	T_2
READ (X)	
X= X-N	
	READ(X)
	X=X+M
WRITE(X)	
READ(Y)	
	WRITE(X)
Y=Y+N	
WRITE(Y)	

› item X has an incorrect value because its update by T1 is lost: X=$160 (60+100) but should have been $110 (60-50+100)!

THE UNIVERSITY OF SYDNEY

Temporary update problem

- Example of the **temporary update problem**:
- A possible execution schedule is:


TIME
T1 fails here

T_1	T_2
READ (X)	
X= X-N	
WRITE(X)	
	READ(X)
	X=X+M
	WRITE(X)
READ(Y)	
Y=Y+N	
WRITE(Y)	

- T1 fails: should change X back to its original value but meanwhile T2 has read the temporary incorrect value of X. Using the same values as in the first case, X is equal to $110 after the completion of the two transactions. Because T1 failed, all of its operations are undone. However, T2 has already read the value of X which is no longer correct!

Incorrect summary problem

- Example of the **incorrect summary problem**:
- Assume that A= \$80 and N=\$50 and that initially X=\$60, Y=\$30
- Another execution schedule is

TIME ↓

T_1	T_2
	SUM=0
	READ(A)
	SUM=SUM+A
READ(X)	
X=X-N	
WRITE(X)	
	READ(X)
	SUM=SUM+X
	READ(Y)
	SUM=SUM+Y
READ(Y)	
Y=Y+N	
WRITE(Y)	

- T2 reads X after N is subtracted and reads Y before N is added: an incorrect summary is obtained. It consists of new and old values. In this case, SUM= \$120 while it should be \$170 after the completion of the two transactions!

Transactions should be Isolated

› Transactions should be isolated from the effects of other concurrent transactions.

- Note that a DBMS usually handles many transactions concurrently.

› Let us consider two transactions that are run concurrently

- Transaction T1 is transferring $100 from account A to account B.
- T2 credits both accounts with a 5% interest payment.

```
T1:  BEGIN   A=A-100,        B=B+100       COMMIT
T2:  BEGIN   A=1.05*A,       B=1.05*B      COMMIT
```

We assume that all transactions commit, there is no aborted transaction!

Example Executions of Two Transactions

› **Serial execution**: we can look at the transactions in a timeline view

```
T1: A=A-100, B=B+100
T2:                        A=1.05*A, B=1.05*B
```

Time

T1 transfers $100 from account A to account B

T2 credits both accounts with a 5% interest payment

› The transactions could occur in another order… **DBMS allows it**!

```
T1:                        A=A-100, B=B+100
T2: A=1.05*A, B=1.05*B
```

Time

T2 credits both accounts with a 5% interest payment

T1 transfers $100 from account A to account B

› DBMS can also **interleave** the transactions.

```
T1: A=A-100,              B=B+100
T2:           A=1.05*A,             B=1.05*B
```

Isolation through Serializability

- **Serial Schedule** – A schedule in which all transactions are executed from start to finish, without interleaving, one after the other.
  - In serial execution, each transaction is **isolated** from all others
- However, **interleaving** (concurrent execution) improves performance
  - Some transactions may be *slow* and long-running – don’t want to block other transactions!
  - Disk access may be *slow* – let some transactions use CPUs while others access disk!

Isolation through Serializability

- Though individual transactions, running separately from others, yield correct database states, their concurrent execution may yield incorrect states: Thus to ensure database correctness, we need to ensure transaction *Isolation*: *serializability*.
- **Serializability**:
  A schedule is **serializable** if it is equivalent to *some* serial schedule
  - Two schedules S1 and S2 are **equivalent** if, ***for any database state***, the effect on the database of executing S1 **is identical to** the effect of executing S2
  - Let a set of transactions on a database occur concurrently. *A schedule is serializable iff it is equivalent to some serial execution.*

Example of a Serializable Schedule

› Consider the following **interleaved** execution

```
T1: A=A-100,                   B=B+100
T2:              A=1.05*A,                   B=1.05*B
```

$A_F = 1.05*(A_i-100), B_F = 1.05*(B_i+100)$

› It is serializable, as the result of the above interleaved execution is the same as that of the following serial execution of T1, T2

```
T1: A=A-100, B=B+100
T2:                           A=1.05*A, B=1.05*B
```

$A_F = 1.05*(A_i-100), B_F = 1.05*(B_i+100)$

```
T1:                            A=A-100, B=B+100
T2: A=1.05*A, B=1.05*B
```

$A_F = (1.05*A_i)-100, B_F = (1.05*B_i)+100$

Example of a Non-Serializable Schedule

› Consider the following **interleaved** execution

```
T1: A=A-100,                          B=B+100
T2:           A=1.05*A, B=1.05*B
```

$A_F = (A_i-100)*1.05$, $B_F = B_i*1.05+100$

› It is not serializable, as the result of the above interleaved execution is not the same as that of either of the following two serial executions.

```
T1: A=A-100, B=B+100
T2:                       A=1.05*A, B=1.05*B
```

$A_F = 1.05*(A_i-100)$, $B_F = 1.05*(B_i+100)$

```
T1:                       A=A-100, B=B+100
T2: A=1.05*A, B=1.05*B
```

$A_F = (1.05*A_i) -100$, $B_F = (1.05*B_i)+100$

Short 5 mn break:

please stand up, stretch, and move around

THE UNIVERSITY OF SYDNEY

Outline

- Consistency
- Atomicity
- Durability
- Isolation
  - What, why, and how?
  - Isolation through conflict serializability
  - Lock-based concurrency control

Conflict serializability

› One type of serializability is *conflict serializability*: A schedule is conflict serializable if it is *conflict equivalent* to a serial schedule.

› Conflicts:

- *No conflict* can arise when reading/writing different data items.
- Two transactions can read the same item in any order: *no conflict.*
- In the event we are reading/writing the same data item, we define the cases when conflicts may arise:
    - A read of a transaction T1 followed by a *write* of a transaction T2 is not semantically the same as a *write* of a transaction T2 followed by a *read* of a transaction T1: *conflict*.
    - The order of two *writes* of two transactions does matter. The last value will depend on which *write* comes last: *conflict*.

› In summary: *whenever the order matters, there is a conflict*. In the case of 2 *reads*, the order does not matter and hence there is no conflict.

DBMS’s View of a Schedule

- Serializability is expensive to check
  - It needs to check the effect of the schedule on all consistent databases
- Let’s see how to analyze schedules without executing them.

```
T1: A=A-100,                         B=B+100
T2:            A=1.05*A, B=1.05*B
```

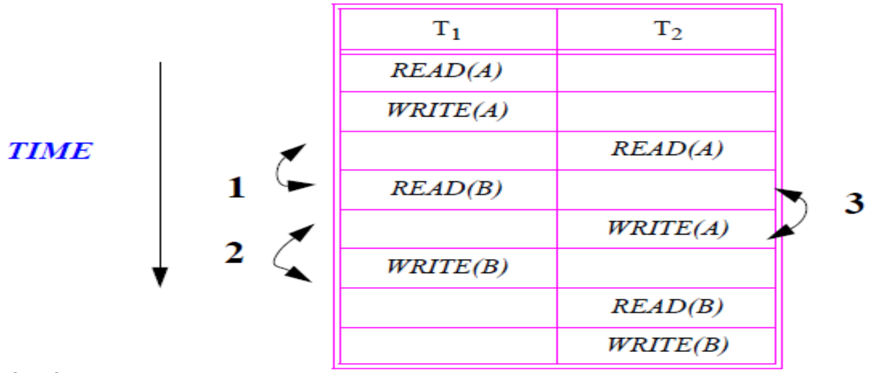
- To do this, we need to see DBMS’s view of a schedule

```
T1: R1(A),W1(A),                             R1(B),W1(B)
T2:               R2(A),W2(A),R2(B),W2(B)
```

  - R: reading the content of an object from the database
    - R1: reading by transaction T1 (also written as R_1)
    - R2: reading by transaction T2 (also written as R_2)
  - W: writing the content of an object into the database
    - W1, W2 are similarly defined

Checking conflict-serializability

- How do we check for conflict serializability?
- Since the order among non-conflicting operations does not matter, use *non conflicting swappings*:
- Example: check if this schedule is conflict serializable

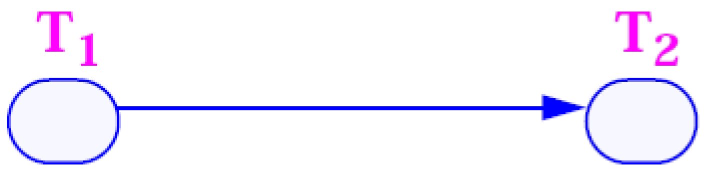


T1	T2
READ(A)	
WRITE(A)	
	READ(A)
READ(B)	
	WRITE(A)
WRITE(B)	
	READ(B)
	WRITE(B)

- Swap READ(B) of T1 with READ(A) of T2
- Swap WRITE(B) of T1 with WRITE(A) of T2
- Swap WRITE(B) of T1 with READ(A) of T2
- The resulting schedule is serial (T1, T2). Therefore, the two schedules are *conflict equivalent*.

Checking conflict-serializability

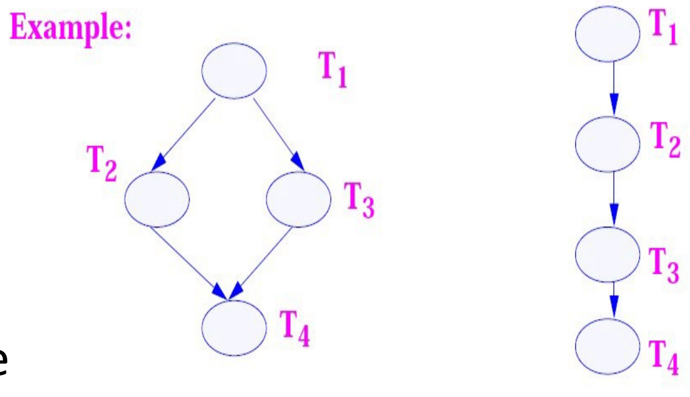
- Is there another way to test conflict serializability? YES
- Use a *Precedence Graph* (also called the *Serialization Graph Testing* or *SGT*).



- The above edge corresponds to the one or more of the following cases
  - T1 executes write(A) before T2 executes read(A).
  - T1 executes read(A) before T2 executes write(A).
  - T1 executes write(A) before T2 executes write(A).
- Algorithm: Check for cycles. If there is a *cycle,* then the schedule is *not conflict serializable*. why?

Checking conflict-serializability

- if there is an edge between a transaction T1 and another T2 (pointing to T2) then in the equivalent serial schedule, T1 should come before T2. A cycle would mean that:
  - 1. T1 should come before T2
  - 2. T2 should come before T1
- Therefore impossible → not conflict serializable.
- If the SGT graph is *acyclic* then there is a serial schedule obtained from a *topological sorting*. This would determine a *linear order* consistent with the partial order of the precedence graph.
- Problems with the SGT approach: *expensive* to maintain SGT graphs. High overhead in letting schedules go unchecked until a non-serializable schedule is detected.



Which of these schedules are conflict serializable?

- R1(x),R2(y),R1(z),R3(z),R2(x),R1(y)
  - all reads – no conflicts – hence conflict serializable

- R1(x),W2(y),R1(z),R3(z),W2(x),R1(y)
  - non-conflict serializable:
    - If we say the equivalent serial schedule is (T1,T2) then there is a conflict (W2(y),R1(y)) which indicates that T2 should come before T1.
    - If we say the equivalent serial schedule is (T2,T1) then there is a conflict (R1(x),W2(x)) which indicates that T1 should come before T2.

- R1(x),W2(y),R1(z),R3(x),W2(x),R2(y)
  - conflict serializable: conflict (R1(x),W2(x)) indicates that T1 should come before T2. Conflict (R3(x),W2(x)) indicates that T3 should come before T2. No conflicts between T1 and T3. Therefore the above schedule is conflict serializable and equivalent to (T1,T3,T2).

Serializability vs Conflict Serializability

› If a schedule is conflict serializable, then it must also be serializable, but not vice versa!

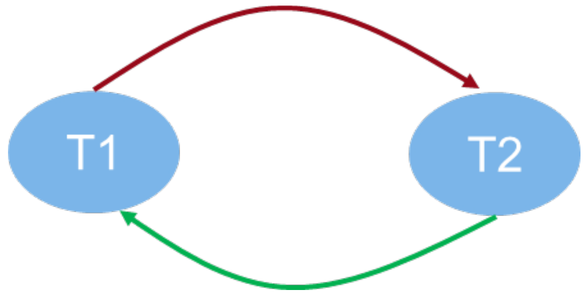
```
T1:  W1(B)                   W1(A)
T2:            W2(B)  W2(A)
T3:                               W3(A)
```

- This schedule leaves A with the value written by T3 and B with the value written by T2.
- Note also that the serial schedule (T1, T2, T3) leaves A and B with the same values.

```
T1:  W1(B)W1(A)
T2:              W2(B)  W2(A)
T3:                            W3(A)
```

- This schedule is serializable, but not conflict serializable
  - It is equivalent to the serial schedule S1 = T1, T2, T3
  - It is not conflict equivalent to any serial schedule. Why?

  Proof: there is a cycle in the precedence graph.



Exercise

Determine whether each of the following schedules are conflict serializable; justify your answer. If a schedule is conflict serializable, please give a conflict equivalent serial schedule.

a) R1(z), R2(x), W2(z), R1(x), W1(y)

b) R1(c), R2(c), W2(c), W1(c)

Outline

- Consistency
- Atomicity
- Durability
- Isolation
  - What, why, and how?
  - Isolation through conflict serializability
  - Lock-based concurrency control

Lock-based Concurrency Control

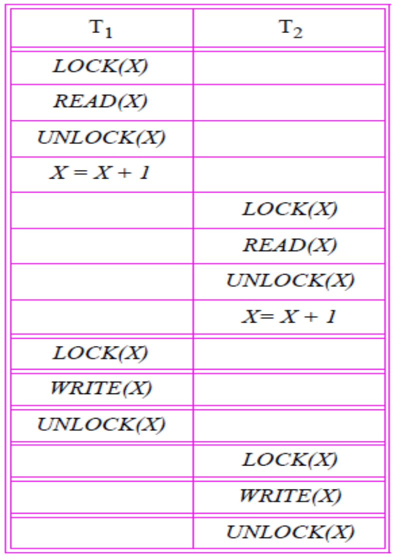
- Lock-based protocols: Issues:
  - Need a notion of locking (to prevent conflict) lock an item before you use it
    - If another transaction has a lock, the second transaction requesting that same item will have to wait
    - Lock manager maintains a lock table
  - Granularity of locks?
    - Large: (too coarse) - no effective concurrency
    - Small: (too fine) - lock overhead high
- Note: Can have more concurrency by differentiating between locks:
- *read* locks: “shared” lock (S)
- *write* locks: “exclusive” lock (X)

Compatibility matrix

	S	X
S	*T*	*F*
X	*F*	*F*

Issues with unlocking

- Problem: *unlocking* data items
- *When* should we release a lock on an item?
  - One way is to do it is as soon as we are thru with an item
  - However, this may leave database in an inconsistent state.
- Example of an airline reservation system: If two airline agents are trying to make a booking on the same flight, a schedule could look like this:



T_1	T_2
LOCK(X)	
READ(X)	
UNLOCK(X)	
$X = X + 1$	
	LOCK(X)
	READ(X)
	UNLOCK(X)
	$X = X + 1$
LOCK(X)	
WRITE(X)	
UNLOCK(X)	
	LOCK(X)
	WRITE(X)
	UNLOCK(X)

- Would result in making one single reservation instead of 2!

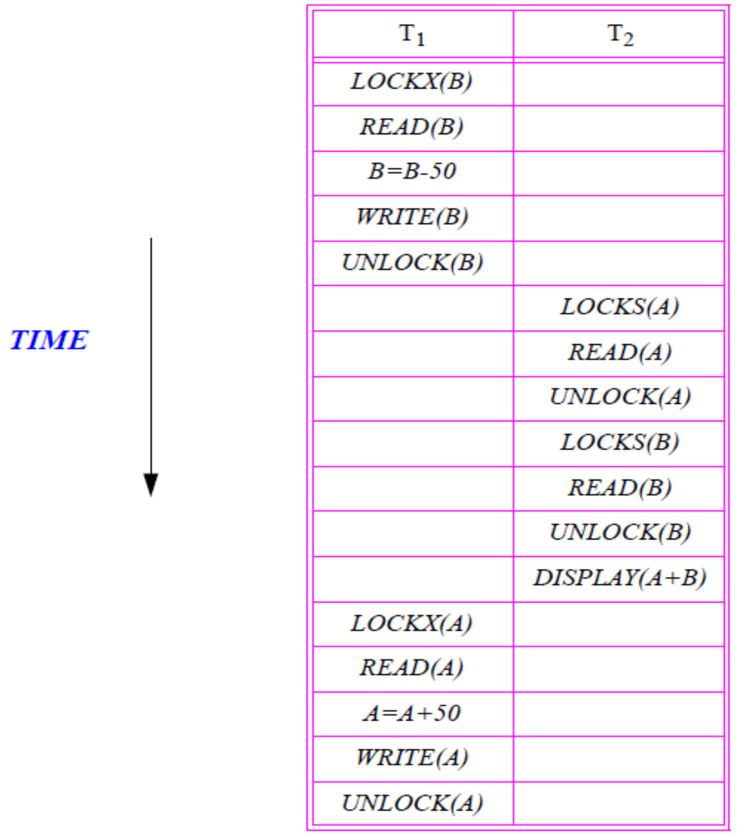
Two-Phase Locking

- **Basic two-phase locking -** idea: for every transaction;
  - get locks
  - perform computations
  - release locks and commit
- insist that all locks be granted before any are released; in essence the two-phase locking consists of:
  - a **growing** phase (the number of locks may increase but not decrease) and
  - a **shrinking** phase (once a lock has been released, the number of locks can only decrease until no more locks exist).
  - **commit** the changes to the database
- ***Strict*** **two-phase locking**: all locks are released *after* commit.

Goals of two-Phase Locking

› Goals of two-phase locking

- Prevents partial results from being seen (i.e., used) by some other transactions to prevent *dirty reads*. This will actually depend on when the commit occurs.
- Supports *serializability*.
- Example: Is this a two phase locking schedule?



T_1	T_2
LOCKX(B)	
READ(B)	
B=B-50	
WRITE(B)	
UNLOCK(B)	
	LOCKS(A)
	READ(A)
	UNLOCK(A)
	LOCKS(B)
	READ(B)
	UNLOCK(B)
	DISPLAY(A+B)
LOCKX(A)	
READ(A)	
A=A+50	
WRITE(A)	
UNLOCK(A)	

Deadlock

› Consider:

```
T1:  R(A),W(A),R(B),W(B)
T2:  R(B),W(B),R(A),W(A)
```

› Schedule with locking might start as:

```
T1:  S(A),R(A), X(A),W(A),                                 S(B)?
T2:                         S(B),R(B),X(B),W(B),  S(A)?
```

› What is happening?

- T1 waiting on T2 to release lock on B
- T2 waiting on T1 to release lock on A
- ***DEADLOCK!***

Deadlock definition

- **Deadlock**:
  - Deadlock occurs whenever a transaction T1 holds a lock on an item A and is requesting a (conflicting) lock on an item B and a transaction T2 holds a lock on item B and is requesting a (conflicting) lock on item A. (A and B could be the same item!).
- In two phase locking, deadlocks may occur.

Two ways of dealing with deadlocks:

- Deadlock prevention
  - *Static* 2-phase locking:
    - Each transaction pre-declares its readset (shared locks) and writeset (exclusive locks) and get all locks or none.
- Deadlock detection
  - A transaction in the cycle must be aborted by DBMS (since transactions will wait forever)
  - DBMS uses deadlock detection algorithms or timeout to deal with it

Exercise

uosCode	year	semester	lecturerId
COMP5138	2012	S1	4711
INFO2120	2011	S2	4711

- 2 transactions, T1 & T2
- Assume no concurrency control, each row is an object
- Statements interleaved as below.

T1	**SELECT** * **FROM** Offerings **WHERE** lecturerId = 4711
T2	**SELECT** year **INTO** :yr **FROM** Offerings **WHERE** uosCode = 'COMP5138'
T1	**UPDATE** Offerings **SET** year=year+1 **WHERE** lecturerId = 4711 **AND** uosCode = 'COMP5138'
T2	**UPDATE** Offerings **SET** year=:yr+2 **WHERE** uosCode = ‘INFO2120'
T1	**COMMIT**
T2	**COMMIT**

Exercise

uosCode	year	semester	lecturerId	
COMP5138	2012	S1	4711	A
INFO2120	2011	S2	4711	B

- 2 transactions, T1 & T2
- Assume no concurrency control, each row is an object
- Statements interleaved as below.

T1	**SELECT** * **FROM** Offerings **WHERE** lecturerId = 4711	R1(A),R1(B)
T2	**SELECT** year **INTO** :yr **FROM** Offerings **WHERE** uosCode = ‘COMP5138'	R2(A)
T1	**UPDATE** Offerings **SET** year=year+1 **WHERE** lecturerId = 4711 **AND** uosCode = 'COMP5138'	R1(A),W1(A)
T2	**UPDATE** Offerings **SET** year=:yr+2 **WHERE** uosCode = ‘INFO2120'	R2(B) W2(B)
T1	**COMMIT**	
T2	**COMMIT**	

Time

Exercise

uosCode	year	semester	lecturerId	
COMP5138	2012	S1	4711	A
INFO2120	2011	S2	4711	B

- 2 transactions, T1 & T2
- Assume strict 2PL with row-level locking is used.
- How would the following schedule be affected?

T1	**SELECT** * **FROM** Offerings **WHERE** lecturerId = 4711	S1(A),S1(B)
T2	**SELECT** year **INTO** :yr **FROM** Offerings **WHERE** uosCode = 'COMP5138'	S2(A)
T1	**UPDATE** Offerings **SET** year=yr+1 **WHERE** lecturerId = 4711 **AND** uosCode = 'COMP5138'	Request X1(A), wait
T2	**UPDATE** Offerings **SET** year=:yr+2 **WHERE** uosCode = ‘INFO2120'	Request X2(A), wait
T1	**COMMIT**	
T2	**COMMIT**	

We have a deadlock!

Summary

- Let us return to our two transactions:
  - Transaction T1 is transferring $100 from account *A* to account *B*.
  - T2 credits both accounts with a 5% interest payment.

```
T1:  BEGIN   A=A-100,   B=B+100    COMMIT
T2:  BEGIN   A=1.05*A,  B=1.05*B   COMMIT
```

- *Atomicity requirement*
  - *all updates* of a transaction are reflected in the db or none.
- *Consistency requirement*
  - T1 *does not change* the total sum of *A* and *B*, and after T2, *this total sum* is 5% higher.
- *Isolation requirement*
  - There is no guarantee that T1 will execute before T2, if both are submitted together. However, the actions of T1 *should not partially affect* those of T2, or vice-versa.
- *Durability requirement*
  - once a transaction has completed, the updates to the database by this transaction must persist *despite failures*

Key Skills

You should be able to:

- Explain how ACID properties define correct transaction behaviour
- Identify update anomalies when ACID properties aren’t enforced
- Explain whether an execution schedule is conflict serializable
- Explain how locking provides isolation.

References

- **Ramakrishnan /Gehrke – Chapter 16, details in Ch. 17 & 18**
- Kifer/Bernstein/Lewis – Chapter 18
- Ullman/Widom – Chapter 6.6
- Transactions & JDBC – [JDBC] JDBC documentation
  - Docs for java.sql.connection (with commit, rollback and setAutoCommit) http://docs.oracle.com/javase/6/docs/api/java/sql/Connection.html
  - See also tutorial http://docs.oracle.com/javase/tutorial/jdbc/basics/transactions.html
- Transactions & DB-API:
  - Python DB-API: https://www.python.org/dev/peps/pep-0249/

Next Week

- Schema Normalization
  - Functional Dependencies
  - Schema Normal Forms
  - Schema Normalization
- Readings:
  - **Ramakrishnan/Gehrke (Cow book), Chapter 19**
  - Kifer/Bernstein/Lewis book, Chapter 6
  - Ullman/Widom, Chapter 3 (up-to 3.5)

See you next week!