COMP9120

Week 9: Transaction Management

Semester 1, 2025

Professor Athman Bouguettaya
School of Computer Science

THE UNIVERSITY OF SYDNEY

Contents co-developed with Dr Lijun Chang

Warming up

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 Gadigal people of the Eora nation and pay my respects to their Elders, past, present and emerging.

Copyright warning

THE UNIVERSITY OF SYDNEY

Quiz information

- **Quiz** in Week 10 (next week)
  - Date: **Thursday, 8 May, starts at 5:30 PM Sharp.**
  - Mode: In-class
  - Each one of you will be assigned to a room for the quiz. This assignment has been announced on Ed and published on Canvas (see https://canvas.sydney.edu.au/courses/63042/groups#tab-42412). You **cannot change your room assignment**. The quiz will be held in 4 rooms (lecture theatres) across campus and 1 additional room for those on the Disability Academic Program.
  - You must arrive at **5:00pm sharp** at your **assigned quiz room** to have your id and room assignment checked before you are allowed to take the quiz. There will be no exception!
  - Duration → **90 minutes**
  - Pen-and-paper based closed book quiz. You are **not** allowed to bring anything besides your writing implements or a university approved dictionary. Everything else will be provided.
  - Covers week 1, 2, 3, 4, 5, 6, 8, 9 contents
    - **3 MCQ questions** (total of 3 marks) and
    - **6 essay questions** (total of 15 marks)

Note: **tutorials scheduled at 7pm on Thursday 8 May are cancelled**. If you are in one of these tutorials, please attend one of the alternative 8 pm sessions on Thursday or any session on Friday. **Week 10 Tutorials will be Q&A sessions.**

Outline

- Motivation for Transactions
- Required Properties of Transactions:
  - **A**tomicity, **C**onsistency, **I**solation, **D**urability (ACID)
    - Meaning of the ACID Properties (What?)
    - Importance of ACID Properties (Why?)
    - Strategies for Ensuring ACID Properties Hold (How?)

Motivation

Complex SQL statements

Not all database operations are *atomic, i.e., executed in one single operation*

Need to model *complex operations that consist of multiple steps* but should be **executed** as *one single logical operation:* **Transaction**

Example of transactions:

1. Purchasing a house
2. Bank transfer
3. Online transactions
4. etc.

No transactions?

What could happen if we *did not have* a *transaction*?

Transfer Operation

Transfer $100

from Account A

to Account B

Account A

Withdraw $100

Account B

Deposit $100

System fails

Account balance successfully updated for Account A

When the system recovers, the database state *no longer reflects the account transfer* (short of $100)!

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

one success
one fail

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

What is a transaction?

- A database program consisting of *one* or *more* SQL statements
  - *Executed* as an *atomic* unit

    means

    - Atomicity implies that the ***effect*** of the transaction is that it executes ***fully*** or ***not at all***.

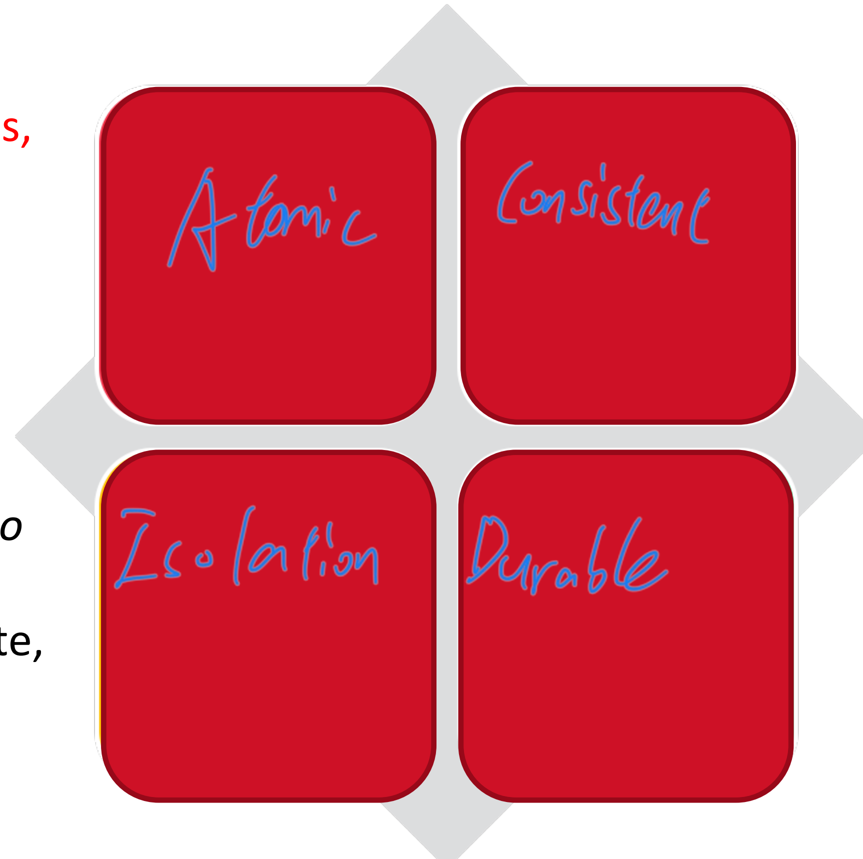
Another way to **describe** a transaction:

- *It is **a program** that 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 *complete*
- **Formal definition**: A ***transaction*** is a collection of *one* or *more* operations (consisting of ***reads*** and ***writes*** at the DBMS level) which reflect a *discrete* (i.e., *single*) unit of work.

Required properties of a transaction

Reliability and correctness of databases require transactions to conform to some strict expectations, called ***ACID*** *properties*.

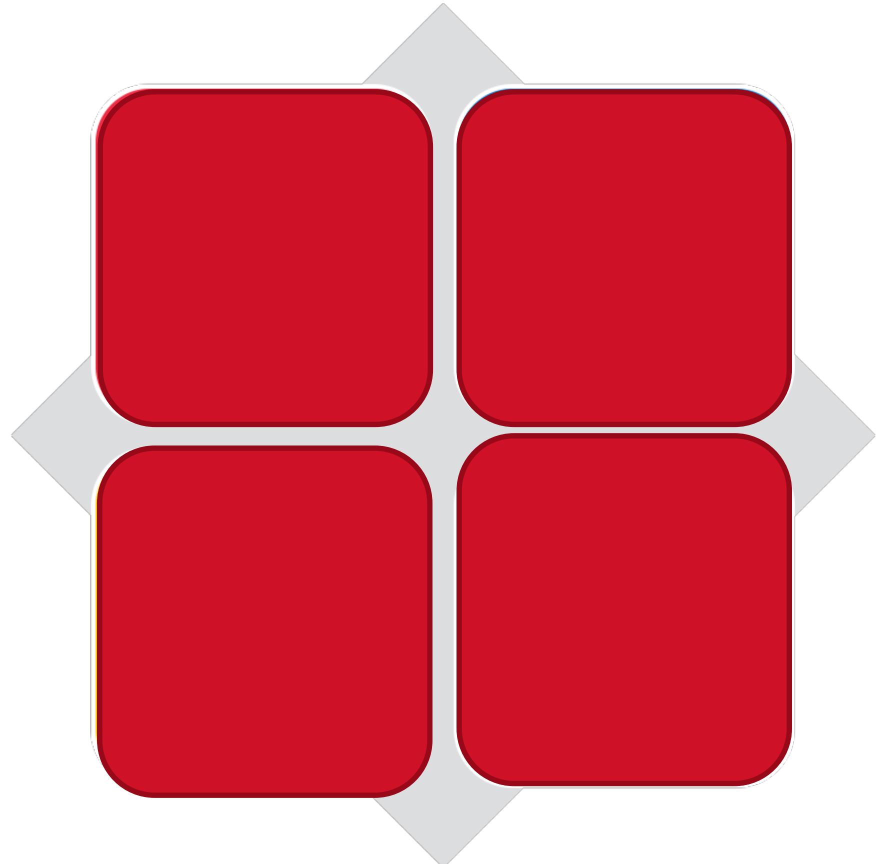
- *Atomicity*: A transaction is either performed *entirely* or *not performed at all*. The whole transaction is treated as *one atomic* operation.
- *Consistency*: A correct execution of a transaction must take a database *from one consistent state to another*: The transaction, if executed separately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



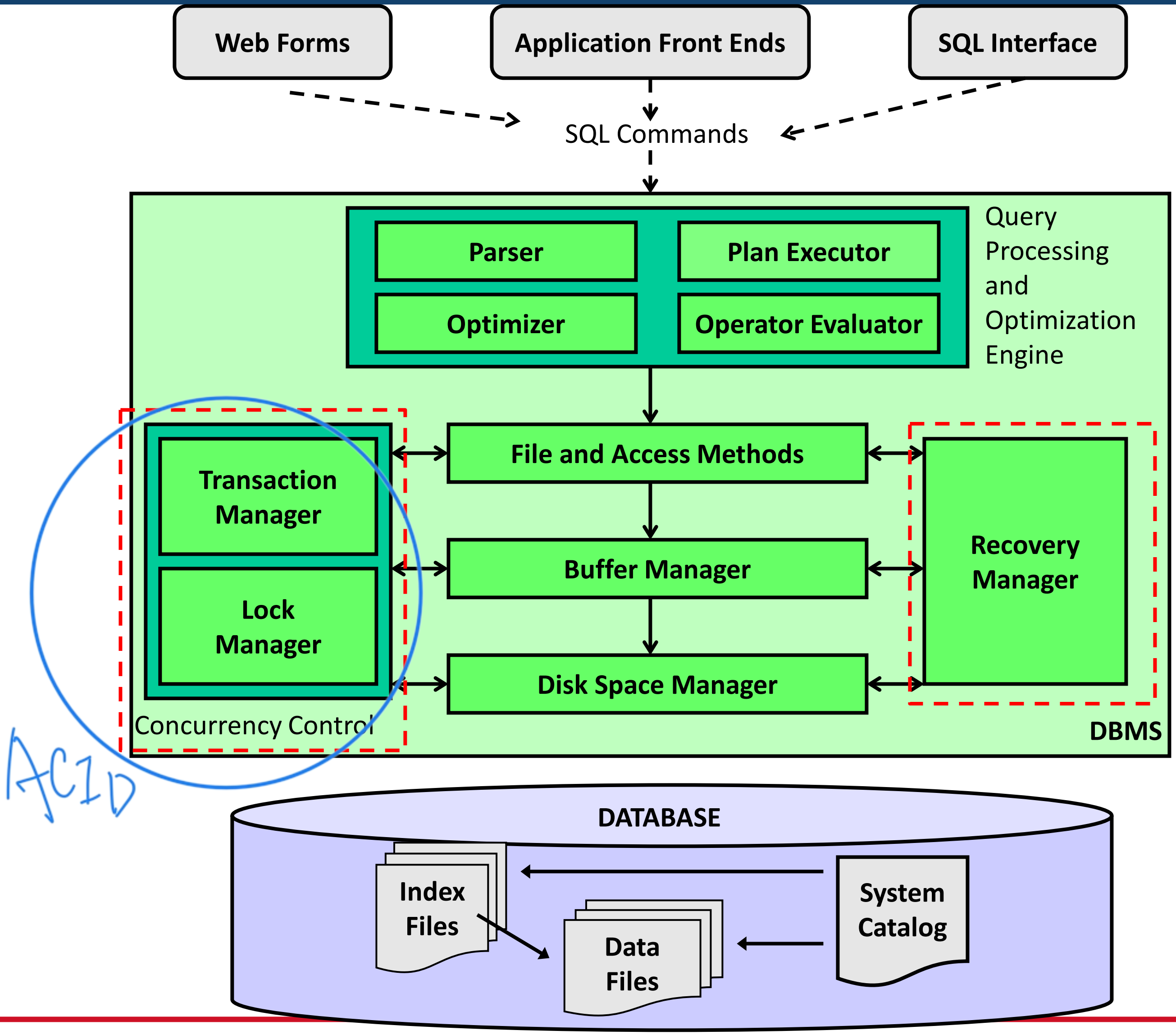
Required properties of a transaction (cont’d)

- *Isolation*: *Effect* of multiple transactions is the same as the transactions *running one after another*: For every two transactions running concurrently, the effect of the execution is as if the transactions 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*.



Internal Structure of a DBMS



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 consistent** if:

  1. *All database constraints* are satisfied
  2. The new database state *satisfies* the *specifications* of the transaction, i.e., *intended effects* 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* 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: Timing of Consistency Checking

Note: We can select **when** to **enforce** the database consistency in the transaction!

Example: We may *defer* the enforcement of 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 ;
COMMIT;
```

does not exist yet!

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
- **<u>A</u>tomicity**
  - What, why, and how?
- Durability
- Isolation

Transactions should be Atomic

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

- Solution: DBMS ***logs** all actions* that would need to be ***undone if*** the transaction is *aborted* (i.e., it is *incomplete*).
  - E.g., in case of a *failure*, all actions of *not-committed* transactions are *undone*.
  - In some cases, we can do a **redo, i.e.,** use the logs to copy over the data

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
  - Examples of reasons for **abort**:
    - System crash – e.g., power outage
    - Transaction aborted by system, e.g.,
      - Transaction or connection time-out,
      - Deadlocks,
      - Violation of constraints
    - Transaction explicit request to roll back

Writing a transaction in SQL

› 3 key relevant SQL commands to know:

Syntax

- **BEGIN**
- **COMMIT** *requests* to **commit** current transaction
- **ROLLBACK/ABORT** causes current transaction to **abort**.

sql statement

› Can also **SET AUTOCOMMIT OFF** or **SET AUTOCOMMIT ON** in pgadmin client
- 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?

uosCode	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?

<u>uosCode</u>	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

fail now, because lectureId is PK and should be unique

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

1. 1234
2. 3456 ✓
3. 4567

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*.

  - In the event of software or hardware malfunction, parts of the database may be erased or corrupted:
    - A database should *always* be *able* to *recover* to the last *consistent* state

Transaction logs

- Solution: use **stable storage** (e.g., hard disk) as a **log** to store a history of modifications made to the database.
- What part of the DBMS is responsible for this? **Recovery Manager**
- Mechanism:
  - Every transaction has a "log" associated with it.
  - Every time an **exclusive lock** on an item is granted, **any update** to that item is also **mirrored** in the log.
  - 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).

Outline

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

Purpose of Concurrency Control

› Note

交错的

- Transactions can run concurrently; meaning their operations can be interleaved. The interleaving is usually *decided* by the *host operating system* based on some scheduling algorithm.
- If there is *no intervention* from the transaction manager, the database may be left in an *incorrect* and *inconsistent* state because of

  problem

  - **Concurrent access** (i.e., interleaving of operations) involving ***updates*** to the database

› Therefore, there is a need to:

- **Control concurrent** *access* to the database to ensure not only **correctness** but also **efficiency**

Concurrent access issues

- We identify three (3) types of **problems** that can **compromise** database **correctness** in the presence of **concurrent** access:
  - **Lost update** problem
  - **Temporary update** problem
  - **Incorrect summary** problem

Interference issues

- **Lost Update Problem**:
  - Occurs when two transactions are *interleaved* in such a way that makes an *item's final value incorrect*. That is, a transaction *does not* see its *own update* but rather *sees the updates* of *other* transactions. This means that the ***update** of a transaction is **lost*** because **another** transaction has **updated** this value.

- **Temporary Update Problem**:
  - Occurs when a transaction *updates* an item and then *fails*. Another transaction that read the item is *unaware* it has been *changed back* to its *original* value.

    due to failure

- **Incorrect Summary Problem**:
  - Happens when a transaction is *updating* an *aggregate of items*. If another concurrent transaction is allowed, the *aggregating transaction* may potentially access a *mixture of old and new values*.

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.

› First case: Let us assume that the transferred amount in T1 is N=$50, and the amount deposited in T2 is M=$100. Assume that before we execute the schedule below, the accounts values for X=$60, Y=$30.

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)	

update here

read old (because before WRITE(x) in left)

TIME

	T_1	T_2	
X=60	READ (X)		
X=10	X= X-N		
		READ(X)	X=60
		X=X+M	X=160
X=10	WRITE(X)		
Y=30	READ(Y)		
		WRITE(X)	X=160
Y=80	Y=Y+N		
Y=80	WRITE(Y)		

item X has an **incorrect** value because its update by T1 is **lost**! X is now $160 (60+100) but should be $110 (60-50+100) at the end of this schedule!

Temporary update problem

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

› Consider two transactions T1 and T2 with the same initial values as in the previous example. A possible execution schedule is:

TIME

	T_1	T_2	
X=60	*READ (X)*		
X=10	*X= X-N*		
X=10	*WRITE(X)*		
		READ(X)	X=10
		X=X+M	X=110
		WRITE(X)	X=110
Y=30	*READ(Y)*		
	Y=Y+N		
	WRITE(Y)		

T1 fails here

. Read the new X
. but fail later so
new X rollback

› T1 **fails**: should change X back to its *original* value, i.e., X=\$60, but meanwhile T2 has read the *temporary incorrect value* of X=\$10!

Because T1 *failed*, all of T1 operations are *undone*. The issue is that T2 had *already* read the *updated value of X* from T1 *producing the value \$110* which *is no longer correct!*

Incorrect summary problem

› Example of the **incorrect summary problem**:

› Consider two transactions T1 and T2 and the following execution schedule. Assume that A=$80 and N=$50 and that initially X=$60, Y=$30

	T_1	T_2	
		SUM=0	
		READ(A)	A=80
		SUM=SUM+A	A=80
X=60	*READ(X)*		
X=10	*X=X-N*		
X=10	*WRITE(X)*		
		READ(X)	X=10
		SUM=SUM+X	SUM=90
		READ(Y)	Y=30
		SUM=SUM+Y	SUM=120
Y=30	*READ(Y)*		
Y=80	*Y=Y+N*		
Y=80	*WRITE(Y)*		

TIME

› 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 SUM=$170 after the completion of the two transactions!

Transactions in SQL standard

› There are four ***isolation levels*** in the SQL standard:

- ***read uncommitted***: A transaction can *read data* written by a concurrent *uncommitted* transaction. This is a called **dirty read**.
- ***read committed***: the database *will not read* any of the *uncommitted* values, i.e., **no dirty reads.**
- ***repeatable read***: A transaction *only sees data committed* **before** the transaction **began**; it *never sees* either *uncommitted* data or *changes committed* by concurrent transactions while it is executing.
- ***serializable***: *highest level* of isolation - *serializable execution* is defined to be an execution of concurrently executing transactions which produce the *same effect* as some *serial execution* of these same transactions.

- **PostgreSQL** does **not** implement **read uncommitted** and requests for this type of isolation is defaulted to **read committed.**

Transactions SQL standard mapping to update problems*

Mapping of *isolation levels* to *update problems*:

- **read uncommitted** ➡ allows dirty reads
- **read committed** ➡ addresses temporary update problem
- **repeatable read** ➡ addresses incorrect summary problems
- **serializable** ➡ addresses the lost update problem

fit each problem respectively

How to set the transaction isolation level for the current transaction block

- SET TRANSACTION ISOLATION LEVEL

  { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED };

Syntax

Short break

please stand up and stretch

Let us also menti….

Outline

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

Transactions should be Isolated

- Transactions should be *isolated* from the effects of *other* concurrent transactions.
- 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!

THE UNIVERSITY OF SYDNEY

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

Change order would affect meaning

- The transactions can execute in another order…Remember that 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 execution.

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

Isolation through Serializability

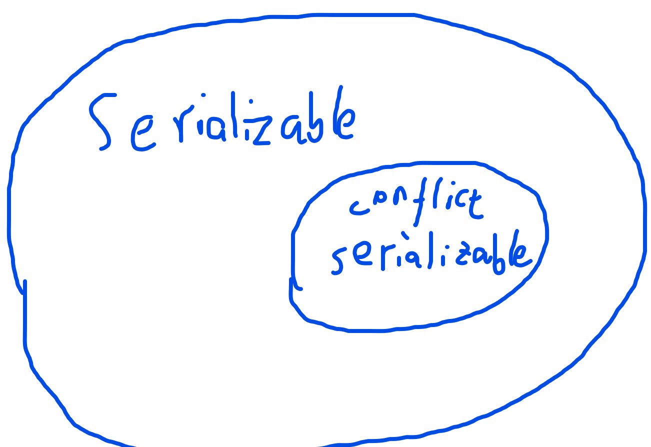
- **Serial Schedule:** A schedule in which all transactions are executed from start to finish, *without any interleaving*, i.e., *one after the other*.
  - In *serial execution*, each transaction is **isolated** from all others

- However, **Interleaving** (concurrent execution) **improves performance** and **response time**:
  - 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***.
  - A **schedule** is **serializable** if and only if it is **equivalent** to *some* **serial** schedule
  - Two schedules S1 and S2 are **equivalent** if, ***for any initial database state***, the ***effect*** on the database of executing S1 **is identical to** the *effect* of executing S2



Example of a Serializable Schedule

› Consider the following **interleaved** execution (called a **schedule**)

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

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

serializable

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

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

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

› Note that there is another serial schedule.

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

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

Do not care because we already find one

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
```

$A_F = (A_i\text{-}100)*1.05,\ B_F = B_i*1.05+100$

› It is not serializable: the result of the above interleaved execution is *not the same* to *either* of the following *two serial* executions.

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

$A_F = 1.05*(A_i\text{-}100),\ B_F = 1.05*(B_i+100)!$

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

$A_F = (1.05*A_i)\ \text{-}100!,\ B_F = (1.05*B_i)+100$

DBMS's View of a Schedule

› Serializability is *expensive* to *check*

- We need to *check* the *effect* of the schedule on *all* initially consistent databases

› Let us see how we can *analyze* schedules *without* executing them.

› Assume the following schedule:

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

› To do this, we need to see the *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 (or R_1): reading by transaction T1
  - R2 (or R_2): reading by transaction T2
- **W: writing** the content of an object into the database
  - W1, W2 are similarly defined

Conflict serializability

- One type of serializability is ***conflict serializability***: A schedule is conflict serializable if it is ***conflict equivalent*** to a **serial** schedule.
- **Conflicts:**
  - Two transactions can ***read two different items in any order: no conflict***
  - Two transactions can ***read*** **the same item in any order:** ***no conflict****.*
  - Two transactions can ***read/write*** **different data items in any order:** ***no conflict*****.**
  - In the event we are ***reading/writing*** **the** ***same data item***, we define the cases when conflicts may occur:
    - 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*****.**

Isolation through Conflict Serializability

read or write

- More formally, two operations a_i and a_j of transactions T_i and T_j **conflict** if:

  (1) *they access the same data X,*
  (2) they come from different transactions, and
  (3) at least one of them writes X.
  In this case, (a_i,a_j) is called a **conflict pair**.

  1. $a_i=R(X),a_j=R(X)$ No Conflict
  2. $a_i=R(X),a_j=W(X)$ Conflict
  3. $a_i=W(X),a_j=R(X)$ Conflict
  4. $a_i=W(X),a_j=W(X)$ Conflict

> **Note**
> With SQL:
> **SELECT** corresponds to a ***Read***
> **INSERT** corresponds to a ***Write***
> **DELETE**, **UPDATE** correspond to
> a ***Read*** followed by ***Write***

- A schedule is **conflict serializable** if it is ***conflict equivalent*** to ***some serial schedule***
  - Two schedules are ***conflict equivalent*** if:
    - They involve the *same set of operations* of the *same transactions*
    - They order *every pair of conflicting operations the same way*

Checking conflict-serializability

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

在detail PPT上有动画

Time

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

1

2

3

- 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*. This means the above schedule is *conflict-serializable*

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 one 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 ***any cycle*** *within the graph,* then the schedule is ***not*** *conflict serializable*. why?

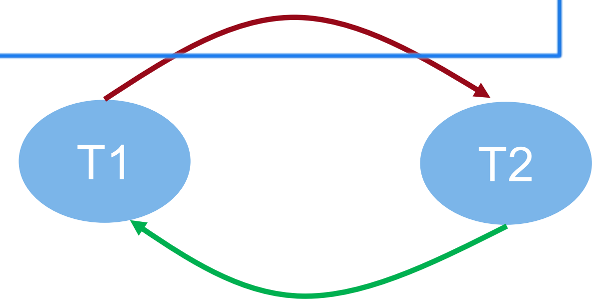
Checking conflict-serializability

- if there is an edge from a transaction T1 to T2, then in the *equivalent serial schedule*, **T1 should come before T2**. A cycle, **however**, means that:
  - 1. T1 should *come before* T2
  - 2. T2 should *come before* T1
- Which is ***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 witn the **partial order of the precedence graph**.
- **Main issue** with the **SGT** approach:
  - ***expensive*** to maintain SGT graphs: **high overhead** in letting **schedules go unchecked** until a **non-serializable** schedule is **detected**!

Testing for Conflict Serializability

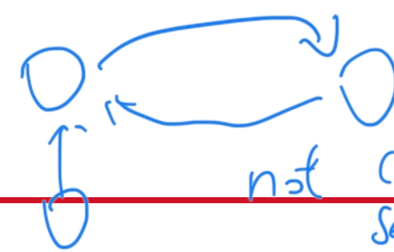
- Consider some schedule of a set of transactions $T_1, T_2, \ldots, T_n$
- **Precedence graph:**
  - direct graph where the vertices are the transactions.
  - edge from T_i to T_j if the two transactions: 1. conflict, and 2. T_i accessed the data item before T_j.
- Central Theorem:
  - A schedule is ***conflict serializable*** *if and only if* its ***precedence graph is acyclic***, **i**.e., there is no cycle.
- Example:

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



  - T1 and T2 have 3 conflict pairs:

    (R1(A),W2(A)), (R2(A),W1(A)), (W2(A),W1(A))



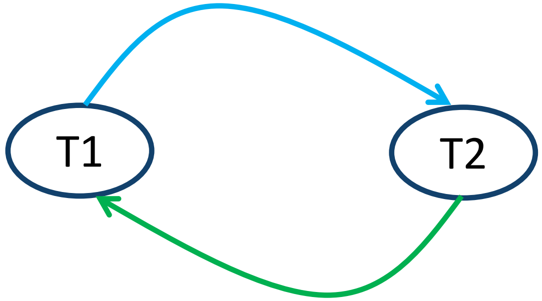
not conflict serializable

Exercise

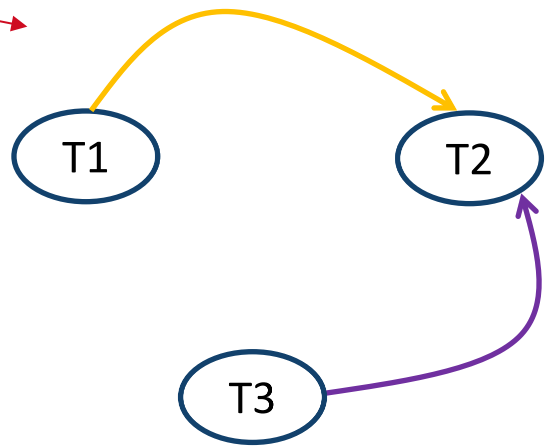
Determine whether each of the following schedules are conflict serializable; justify your answer by drawing the precedence graph. If a schedule is conflict serializable, please give a conflict equivalent serial schedule.

a) R1(x),R2(y),R1(z),R3(z),R2(x),R1(y)

b) R1(x),W2(y),R1(z),W2(x),R1(y)



c) R1(x),W2(y),R1(z),R3(x),W2(x),R2(y)



Solution:

a) All Reads – no conflicts – hence ***conflict serializable***

b) No: It is ***not* conflict serializable**

c) It ***is* conflict serializable** and **<u>equivalent</u>** to **(T1, T3, T2)** or **(T3, T1, T2)**

THE UNIVERSITY OF SYDNEY

swap is not true

Serializability vs Conflict Serializability

› If a schedule is ***conflict serializable***, then ***it must also be serializable***. However, the **converse is not true**! Consider the following schedule S:

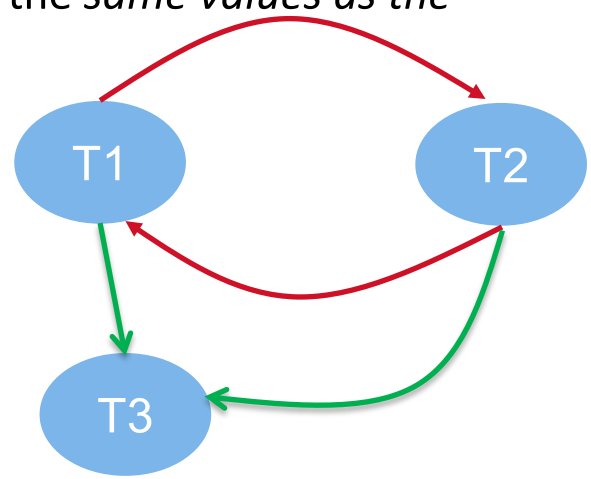
```
T1: W1(B)                W1(A)
T2:           W2(B) W2(A)
T3:                           W3(A)
```

serializable

conflict serializable

- In this schedule, the final value of *A is the value written by T3* and the final value of *B is the value written by T2*.
- Note that the *serial schedule* (T1, T2, T3) leaves A and B with the *same values as the above schedule*. Schedule S is therefore ***serializable.***

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



- However, schedule S is ***not conflict serializable****:*
  - It is ***not conflict equivalent*** to ***any serial schedule***.

  Proof: there is **a cycle** in the **precedence graph**.

Possible Anomalies for *Non-conflict Serializable Schedules*

- Reading Uncommitted Data ("**dirty reads**")

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

System Crash

- **Unrepeatable Reads**: may not read same value twice

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

- Overwriting Data produced (written) by another transaction ("**Lost Update**"):

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

Outline

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

Lock-based Concurrency Control

- So far, we have been optimistic about the *rise of conflicts* in schedules, i.e., we focused on the detection of conflicts.
- **Lock-based** protocol: an implementation scheduler that is part of the family of pessimistic protocols
- Issues:
  - Need a notion of ***locking*** (to prevent conflict) to lock an item *before we 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**
  - Problem: determining the **granularity** of **locks**
    - **Large**: (too **coarse**) - **no effective concurrency**
    - **Small**: (too **fine**) - lock **overhead high**

Lock Compatibility Matrix

Note: Can we have *more concurrency* by *differentiating* between *Read* and *Write* locks? YES!

Read locks: "**Shared**" lock (S)
Write locks: "**Exclusive**" lock (X)

Held by T1 / T2 Requests	**Shared** read	**Exclusive** write
Shared read	OK	T2 wait on T1
Exclusive write	T2 wait on T1	T2 wait on T1

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 have used that item
  - However, this may leave database in an *inconsistent* state.
- Example: Consider an airline reservation system where two airline agents are trying to make a booking on the same flight. A schedule could look like this:

too early

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)

- This schedule would result in making one single reservation instead of 2!

Two-Phase Locking

- **Basic Two-Phase Locking (2PL)**
- algorithm:
  - for every transaction
    - *obtain locks*
    - *perform computations*
    - *release locks* and *commit*
- **Idea** behind **2PL**: 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 only increase*** but ***not decrease***)
  - **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* 2PL**: 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*.
  - Assuming *no* deadlocks and failures, 2PL implements *conflict-serializability* and therefore *ensures serializability*.

- Example: Is this a two-phase locking schedule?
  - No!

after release lock you cannot get lock

TIME

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 with two-phase locking

› Consider the following two transactions:

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

› A schedule with locks might be:

```
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 here?

- T1 waiting on T2 to release lock on B
- T2 waiting on T1 to release lock on A

DEADLOCK!!

Deadlock definition

› **Deadlock** occurs whenever a transaction

- T1 holds a lock on an item A and is requesting a (conflicting) lock on an item B and
- T2 holds a lock on item B and is requesting a (conflicting) lock on item A. (Note: item A and item B could be the same item!).

› **In two phase locking, deadlocks may occur.**

Two ways of dealing with deadlocks:

2 solution for dead lock

› Deadlock **prevention**

- ***Static* 2-phase locking**:
  - Each transaction pre-declares its *readset* (shared locks) and *writeset* (exclusive locks) and gets ***all locks or none***.

› Deadlock **detection**

› A transaction in the **waiting cycle** must be **aborted** by DBMS

- DBMS uses deadlock detection algorithms/timeout to deal with this issue

Example

› Consider the following table, named ***Offerings***:

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

› Two (2) transactions, T1 and T2

- Each row is an object
- Statements interleaved as below.

› Consider the following schedule:

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.year +2 **FROM** yr **WHERE** uosCode = 'INFO2120'
T1	**COMMIT**
T2	**COMMIT**

Example

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

- Consider the previous schedule of two transactions, T1 and T2
  - Each row is an object

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.year+2 **FROM** yr **WHERE** uosCode = 'INFO2120'
T1	**COMMIT**
T2	**COMMIT**

R1(A),R1(B)

R2(A)

R1(A),W1(A)

R2(B) W2(B)

Time

Example

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

- Assume **strict 2PL** and **row-level locking** is used.
  - How would the following **schedule** be **affected**?
  - Convert **Reads** and **Writes** into **S** and **X** locks:

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=year+1 **WHERE** lecturerId = 4711 **AND** uosCode = 'COMP5138'	S1 (A) Request X1(A), wait
T2	**UPDATE** Offerings **SET** year=yr.year+2 **FROM** yr **WHERE** uosCode = 'INFO2120'	S2(B) Request X2(B), 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 database 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 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 are not enforced

› Explain whether an execution schedule is conflict serializable

› Explain how locking provides isolation.

3 types

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/

In two weeks

- Storage and Indexing
  - Storing data in a database
  - Retrieving records from a database
  - B^+ Tree index
- Kifer/Bernstein/Lewis
  - Chapter 9 (9.1-9.5)
- Ramakrishnan/Gehrke
  - Chapter 8
- Ullman/Widom
  - Chapter 8 (8.3 onwards)
- Silberschatz/Korth/Sudarshan (5th ed)
  - Chapter 11 and 12

See you during the quiz and the best of luck!