Database Management System

Module 2: Serializability

Salim Diwani
College of Informatics and Virtual Education
University of Dodoma

Module Outline:

- To understand the issues that arise when **two** or **more** transactions work concurrently.
- To introduce the notions of Serializability that ensure schedules for transactions that may run in **concurrent** fashion but still guarantee and **serial behavior**.
- To analyze the conditions, called **conflicts**, that need to be honored to attain Serializable schedules.

Serializability Concepts

- A **schedule** is a sequence of operations by a set of concurrent transaction 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 of the operations from a set of concurrent transaction are interleaved.

Serializability Concepts

Serializable Schedule: The objective of serializability is to find non-serial schedules that allow transactions to execute concurrently without interfering one another, and thereby produce a database state that could be produced by a serial execution.

That means a non-serial schedule is serializable and correct if it produces same results as some serial execution.

Serializability Concepts

- Basic Assumption – Each transaction preserves database **consistency.**
- Thus, **serial execution** of a set of transactions preserves database consistency.
- A (possibly concurrent) schedule is serializable if it is **equivalent** to a serial schedule.

Simplified view of transactions

- We ignore operations other than **read** and **write** instructions.
  - Other operations happen in memory (are temporary in nature) and (mostly) do not affect the state of the database.
  - This is a simplifying assumption for analysis.
- We assume that transaction may perform arbitrary computations on data in **local buffers** in between **reads** and **writes.**
- Our simplified schedules consist of only **read** and **write** instructions.

Conflicting Instructions

- Let I_i and I_j be two instructions T_i and T_j respectively. Instruction I_i and I_j **conflict** if and only if there exists some item **Q** accessed by both I_i and I_j, and atleast one of these instructions wrote **Q**.
  1. I_i = **read** (Q), I_j = **read** (Q). I_i and I_j **don't conflict**
  2. I_i = **read** (Q), I_j = **write** (Q). **They conflict**
  3. I_i = **write** (Q), I_j = **read** (Q). **They conflict**
  4. I_i = **write** (Q), I_j = **write** (Q). **They conflict**

Conflicting Instructions

- Intuiitively, a conflict between I_i and I_j forces a (logical) temporal order between them.
  - If I_i and I_j are consecutive in a schedule and they do not **conflict**, their results would remain the **same** even if they had been **interchanged** in the schedule.

Conflict Serializability

- If a schedule **S** can be transformed into a schedule **S'** by a series of **swaps** of **non-conflicting** instructions, we say that **S** and **S'** are **conflict equivalent.**
- We say that a schedule **S** is **conflict serializable** if it is **conflict** equivalent to a **serial schedule.**

Conflict Serializability (Cont.)

- **Schedule 3** can be transformed into **schedule 6** – a serial schedule where **T2** follows **T1**, by a series of swaps of **non-conflicting** instructions.
  - Swap **T1.read(B)** and **T2.write(A)**
  - Swap **T1.read(B)** and **T2.read(A)**
  - Swap **T1.write(B)** and **T2.write(A)**
  - Swap **T1.write(B)** and **T2.read(A)**
- Therefore, **schedule 3** is **conflict serializable:**

These swaps do not conflict as they work with different items (A or B) in different transactions.

Conflict Serializability (Cont.)

T1	T2
Read (A)	
Write ((A)	
	Read (A)
	Write (A)
Read (B)	
Write (B)	
	Read (B)
	Write (B)

T1	T2
Read (A)	
Write ((A)	
Read (B)	
Write (B)	
	Read (A)
	Write (A)
	Read (B)
	Write (B)

These swaps do not conflict as they work with different items (A or B) in different transactions

Conflict Serializability (Cont.)

- Example of a schedule that is not conflict serializable:

T3	T4
Read (Q)	
	Write (Q)
Write (Q)	

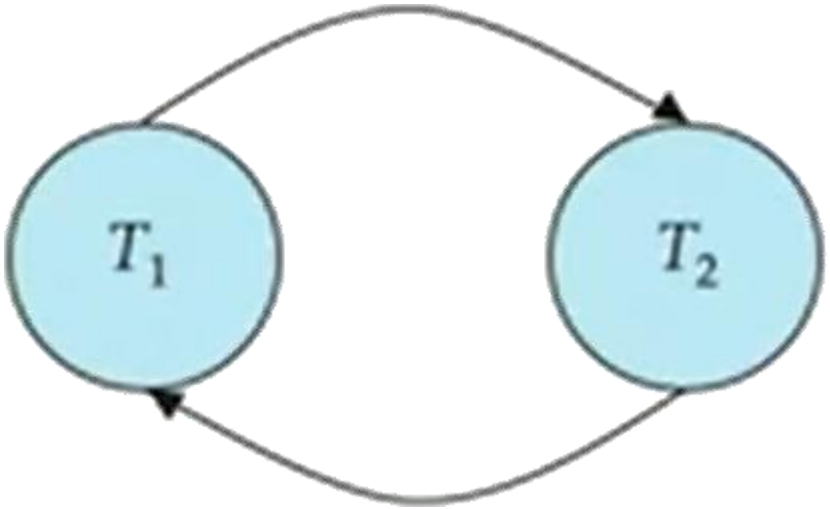
- We are unable to swap instructions in the above schedule to obtain either the serial schedule **<T3, T4>**, or the serial schedule **<T4, T3>**

Precedence Graph

- Consider some schedule of a set of transactions T_1, T_2, T_3,.....T_n
- Precedence graph
  - A direct graph where the vetrices are the transaction (names).
- We draw an arc from **Ti to Tj** if the two transactions **conflict**, and **Ti** accessed the data item on which the conflict arose earlier.
- We may label the arc by the item that was accessed.

Precedence Graph

- Example



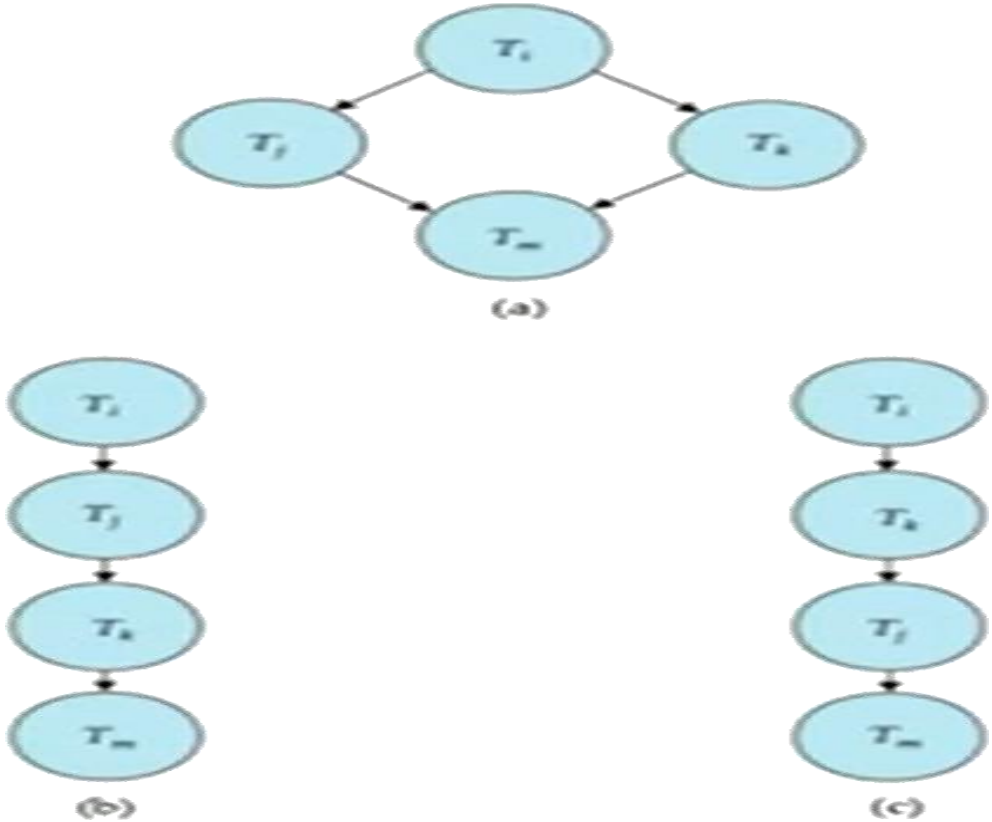
Testing for Conflict Serializability

- A schedule is conflict serializable if and only if its precedence graph is **acyclic.**
- Cycle detection algorithms exist which take order **n^2** time, where **n** is the number of vertices in the graph.
  - (better algorithms take order **n + e** where **e** is the number of edges).

Testing for Conflict Serializability

- If precedence graph is **acyclic**, the serializability order can be obtained by a **topological sorting** of the graph.
  - That is, a **linear order** consistent with the partial order of the graph.
  - For example, a serializability order for the schedule **(a)** would be one of either **(b)** or **(c).**

Testing for Conflict Serializability



Quiz

- consider the following schedule:
  - W1(A), R2(A), W1(B), W3(C), R2(C), R4 (B), W2 (D), W4 (E), R5(D), W5(E)

Than you...!