Transactions

UNIT IV

Overview

- Two phase commit
- SQL support
- Schedules
- Serializability
- Characterizing Schedules based on Recoverability

Two-Phase Commit

- Required for distributed or heterogeneous environments, so that correctness is maintained in case of failure during a multi-part COMMIT
- Prepare phase has all local resource managers force logs to a persistent log, local managers reply ok or not
- Commit phase – if all replies are ok, the coordinator commits, and orders the local managers to complete the process; otherwise all are ordered to ROLLBACK

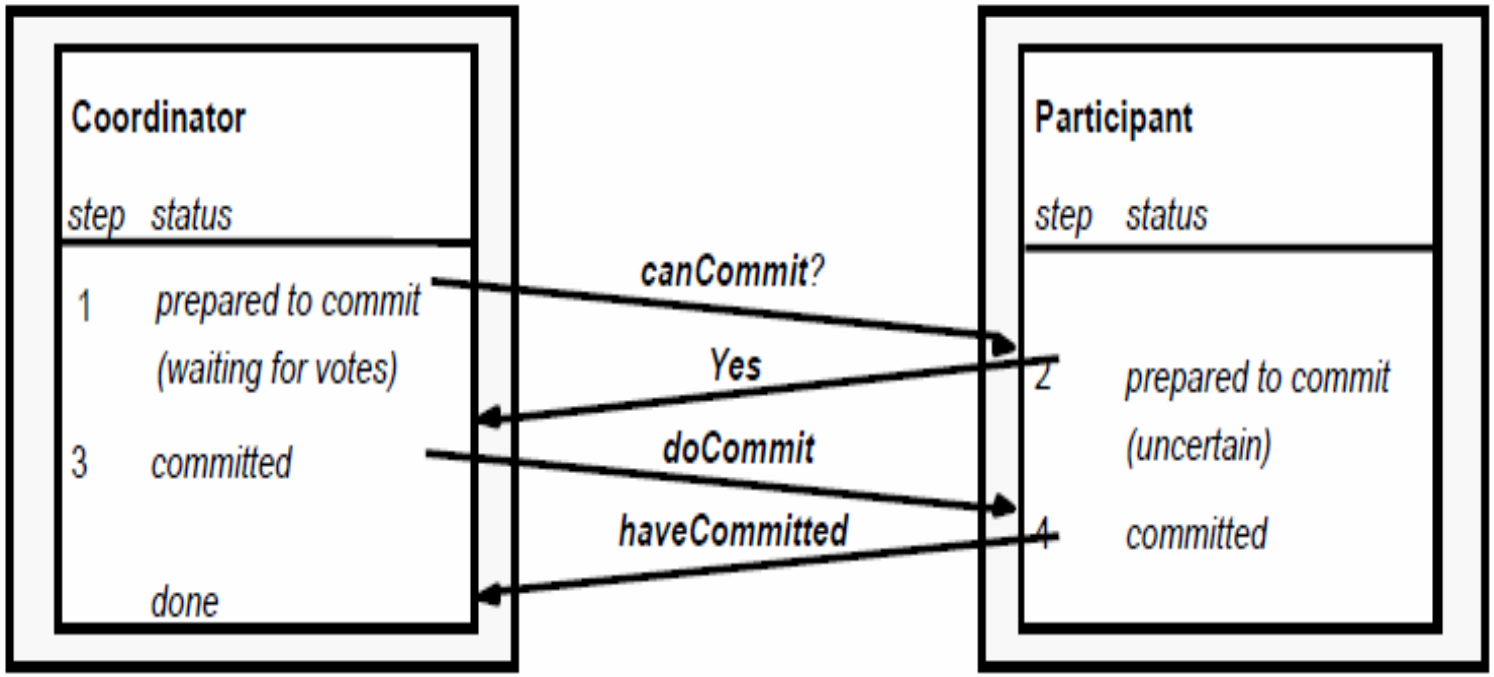
Two-Phase Commit

- The two-phase commit protocol ensures that all participating resources (database servers) receive and implement the same action (either to commit or to roll back a transaction).
- Every global transaction has a coordinator and one or more participants, defined as follows:
  - The coordinator directs the resolution of the global transaction. It decides whether the global transaction should be committed or stopped.
  - The two-phase commit protocol always assigns the role of coordinator to the current database server.
  - The role of coordinator cannot change during a single transaction.

Two-Phase Commit

- PHASE – I [Prepare]
  - The coordinator instructs all participants to get ready on the transaction
    - must force all log records out to its own physical log.
  - If forced write is successful, the participants now replies 'YES' to the coordinator; otherwise 'NO'.
- PHASE – II [Commit]
  - Coordinator forces a record to its own physical log, recording its decision.
  - If all replies were 'YES', decision is 'COMMIT'; if any reply was 'NO', the decision is 'ROLLBACK'.
  - Coordinator informs each participant of its decision, and each participant must then commit / rollback the transaction locally, as instructed.

Communication in 2PC



SQL Facilities

- START TRANSACTION

  < option commalist > ;
- The option commalist specifies an access point, an isolation level, or both
- Access mode can be READ ONLY or READ WRITE
- Isolation level sets isolation from other transactions
- SAVEPOINT establishes a point within a transaction to which you can ROLLBACK

Schedules

- **Schedule** – a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed
  - a schedule for a set of transactions must consist of all instructions of those transactions
  - must preserve the order in which the instructions appear in each individual transaction.
- A transaction that successfully completes its execution will have a commit instructions as the last statement
  - by default transaction assumed to execute commit instruction as its last step
- A transaction that fails to successfully complete its execution will have an abort instruction as the last statement

Schedule 1

- Let T_1 transfer \$50 from A to B, and T_2 transfer 10% of the balance from A to B.
- A serial schedule in which T_1 is followed by T_2 :

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

Schedule 2

- A serial schedule where T2 is followed by T1

T_1	T_2
	read(A) $temp := A * 0.1$ $A := A - temp$ write(A) read(B) $B := B + temp$ write(B)
read(A) $A := A - 50$ write(A) read(B) $B := B + 50$ write(B)	

Schedule 3

- Let T_1 and T_2 be the transactions defined previously. The following schedule is not a serial schedule, but it is *equivalent* to Schedule 1.

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

In Schedules 1, 2 and 3, the sum A + B is preserved.

Schedule 4

- The following concurrent schedule does not preserve the value of $(A + B)$.

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + temp$
	write(B)

Serial Schedule

- A *schedule* is a sequence of the operations by a set of concurrent transactions that preserves the order of operations in each of the individual transactions
- A *serial* schedule is a schedule where operations of each transaction are executed consecutively without any interleaved operations from other transactions (each transaction commits before the next one is allowed to begin)

Serial schedules

- Serial schedules are guaranteed to avoid interference and keep the database consistent
- However databases need concurrent access which means interleaving operations from different transactions

Serializability

- Two schedules are *equivalent* if they always have the same effect.
- A schedule is *serializable* if it is equivalent to some serial schedule.
- For example:
  - if two transactions only read some data items, then the order in which they do it is not important
  - If T1 reads and updates X and T2 reads and updates a different data item Y, then again they can be scheduled in any order.

Serializability

- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:
  1. **conflict serializability**
  2. **view serializability**
- *Simplified view of transactions*
  - Our simplified schedules consist of only **read** and **write** instructions.

Serial and Serialisable

Interleaved Schedule

T1 Read(X)
T2 Read(X)
T2 Read(Y)
T1 Read(Z)
T1 Read(Y)
T2 Read(Z)

Serial Schedule

T2 Read(X)
T2 Read(Y)
T2 Read(Z)

T1 Read(X)
T1 Read(Z)
T1 Read(Y)

This schedule is serializable:

Conflicting Instructions

- Instructions I_i and I_j of transactions T_i and T_j respectively, **conflict** if and only if there exists some item Q accessed by both I_i and I_j, and at least 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
- Intuitively, 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.

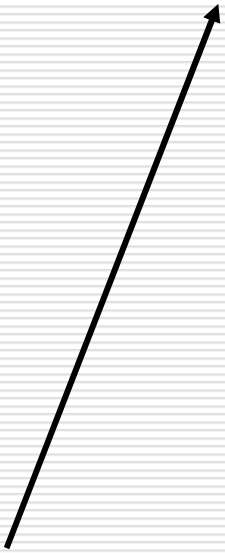
Conflict Serializability

- 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 Serializable Schedule

Interleaved Schedule

T1 Read(X)
T1 Write(X)
T2 Read(X)
T2 Write(X)
T1 Read(Y)
T1 Write(Y)
T2 Read(Y)
T2 Write(Y)



Serial Schedule

T1 Read(X)
T1 Write(X)
T1 Read(Y)
T1 Write(Y)

T2 Read(X)
T2 Write(X)
T2 Read(Y)
T2 Write(Y)

This schedule is serialisable, even though T1 and T2 read and write the same resources X and Y: they have a conflict

Conflict Serializability

- Schedule 3 can be transformed into Schedule 6, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions.
  - Therefore Schedule 3 is conflict serializable.

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Schedule 3

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Schedule 6

Conflict Serializability

- Example of a schedule that is not conflict serializable:

T_3	T_4
read(Q)	
	write(Q)
write(Q)	

- We are unable to swap instructions in the above schedule to obtain either the serial schedule < T_3, T_4 >, or the serial schedule < T_4, T_3 >.

Conflict Serialisability

- Two transactions have a conflict:
  - NO If they refer to different resources
  - NO If they are reads
  - YES If at least one is a write and they use the same resource
- A schedule is *conflict serialisable* if transactions in the schedule have a conflict but the schedule is still serialisable

Conflict Serialisability

- Conflict serialisable schedules are the main focus of concurrency control
- They allow for interleaving and at the same time they are guaranteed to behave as a serial schedule

View Serializability

- Let *S* and *S´* be two schedules with the same set of transactions. *S* and *S´* are **view equivalent** if the following three conditions are met, for each data item *Q*,
  1. If in schedule S, transaction T_i reads the initial value of *Q*, then in schedule *S´* also transaction T_i must read the initial value of *Q.*
  2. If in schedule S transaction T_i executes **read**(*Q)*, and that value was produced by transaction T_j (if any), then in schedule *S´* also transaction T_i must read the value of *Q* that was produced by the same **write**(Q) operation of transaction T_j .
  3. The transaction (if any) that performs the final **write**(*Q*) operation in schedule *S* must also perform the final **write**(*Q*) operation in schedule *S´.*

View Serializability

- View Equivalence leads to another definition of serializability called ***view serializability***.
- A schedule *S* is **view serializable** if it is view equivalent to a serial schedule.
- As long as each read operation of a transaction reads the result of *the same write operation* in both schedules, the write operations of each transaction must produce the same results.
- **"The view"**: the read operations are said to see the *the same view* in both schedules. Every conflict serializable schedule is also view serializable.
- Below is a schedule which is view-serializable but *not* conflict serializable.

T_3	T_4	T_6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

Conflict Serialisability

- Important questions: how to determine whether a schedule is conflict serialisable
- How to construct conflict serialisable schedules

Recoverable Schedules

Need to address the effect of transaction failures on concurrently running transactions.

- **Recoverable schedule** — if a transaction T_j reads a data item previously written by a transaction T_i , then the commit operation of T_i appears before the commit operation of T_j.
- The following schedule (Schedule 11) is not recoverable if T_9 commits immediately after the read

T_8	T_9
read(A)	
write(A)	
	read(A)
read(B)	

Cascading Rollbacks

- **Cascading rollback** – a single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)

T_{10}	T_{11}	T_{12}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)

  If T_{10} fails, T_{11} and T_{12} must also be rolled back.

- Can lead to the undoing of a significant amount of work

Cascadeless Schedules

- **Cascadeless schedules** — cascading rollbacks cannot occur; for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j.
- Every cascadeless schedule is also recoverable
- It is desirable to restrict the schedules to those that are cascadeless

Precedence Graphs

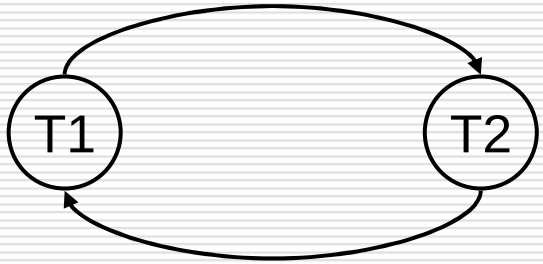
- To determine if a schedule is conflict serialisable we use a precedence graph
  - Transactions are vertices of the graph
  - There is an edge from T1 to T2 if T1 must happen before T2 in any equivalent serial schedule
- Edge T1 $\rightarrow$ T2 if in the schedule we have:
  - T1 Read(R) followed by T2 Write(R) for the same resource R
  - T1 Write(R) followed by T2 Read(R)
  - T1 Write(R) followed by T2 Write(R)
- The schedule is serialisable if there are no cycles

Precedence Graph Example

- The lost update schedule has the precedence graph:

T1	T2
Read(X)	
X = X - 5	
	Read(X)
	X = X + 5
Write(X)	
	Write(X)
COMMIT	
	COMMIT

T1 Write(X) followed by T2 Write(X)

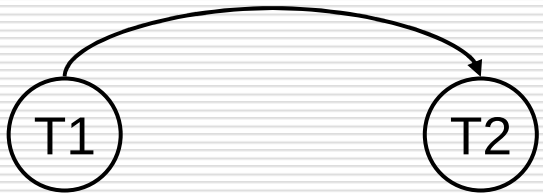


T2 Read(X) followed by T1 Write(X)

Precedence Graph Example

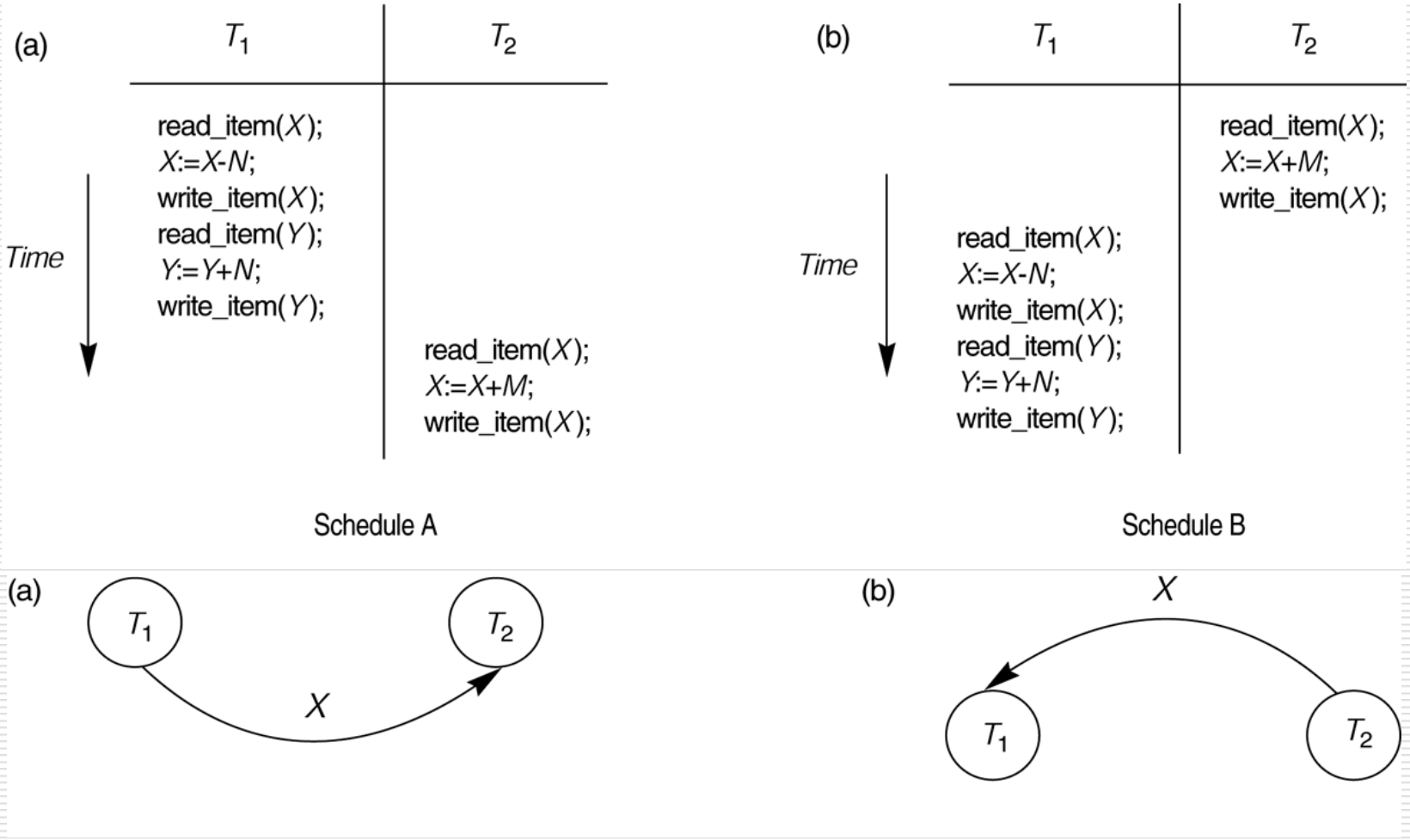
- No cycles: conflict serialisable schedule

T1 reads X before T2 writes X and
T1 writes X before T2 reads X and
T1 writes X before T2 writes X



T1	T2
Read(X)	
Write(X)	
	Read(X)
	Write(X)

Precedence Graph Example



Precedence Graph Example