Database Concept and Systems

CIT 314 (2 Units)

Bilkisu Muhammad-Bello (Mrs.)

Concurrent Transactions Processing

Concepts

Introduction

- If control of concurrent execution is left entirely to the operating system, many possible schedules, including ones that leave the database in an inconsistent state, are possible.
- It is the job of the database system to ensure that any schedule that gets executed will leave the database in a consistent state. The **concurrency-control component** carries out this task.
- We can ensure consistency of the database under concurrent execution by making sure that any schedule that executed has the same effect as a schedule that could have occurred without any concurrent execution. That is, the schedule should, in some sense, be equivalent to a serial schedule.
- In this unit, we shall illustrate concurrent execution of transactions with some examples to explain how a database could be in an inconsistent state, if not controlled. You will also be introduced to some new concepts of how to schedules transactions to ensure database consistency as well as how to initiate a transaction using SQL.

Serializability

- When two or more transactions are executed concurrently on a database, their effect should be the same as if they had executed serially.
- It is therefore important that we know which schedule will lead to database consistency and which will not.
- The execution sequences of two or more transactions are called **schedules**.
- Most schedule schemes lead to the concept of ***Conflict serializability*** and ***View serializability*** as discussed below.

Conflict Serializability (1 of 4)

- Consider a schedule, ***S*** with two consecutive instructions I_i and I_j in transactions T_i and T_j respectively ($i \neq j$).
- If I_i and I_{j} refer to different data items, then we can swap I_i and I_j without affecting the results of any instructions in the schedule.
- However, if I_i and I_j refer to the same data item Q, then the order of the two steps may matter. For the moment, let's deal with only **Read** and **Write** instructions since they are the only significant operations of a transaction, from a scheduling point of view

Schedule 3

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)

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

Conflict Serializability (2 of 4)

- Now, the four cases that may arise are:
  - *i.* I_i = read (Q), I_j = read (Q). The order of instructions I_i and I_j does not matter, since the two instructions both read the same value of Q, from two different transactions T_i and T_j.
  - *ii.* I_i = read (Q), I_j = write (Q). If I_i comes before I_j, then T_i does not read the value of Q that is written by T_j in instruction I_j. If I_j comes before I_i then T_i read the value of Q written by T_j. Thus the order I_i and I_j matters.
  - *iii.* I_i = write (Q), I_j = read (Q). For the same reason as in ii) above, the order of the two transactions matter.
  - *iv.* I_i = write (Q), I_j = write (Q). Since they both are writing, the order of the two transactions matter and the next read(Q) instruction will have the value of the last write instruction.

Conflict Serializability (3 of 4)

- We therefore say that the execution of two instructions ***conflict*** if they are operations by different transactions on the same data item, and at least one of these instructions is a **write** operation.
- To illustrate the concept of conflicting instructions, we consider schedule 3, in in the previous lesson. The write(A) instruction of T_1 conflicts with the read(A) instruction of T_2.
- However, the write(A) instruction of T_2 does not conflict with the read(B) instruction of T_1, because the two instructions access different data items.

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)

Conflict Serializability (4 of 4)

- 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***.
- In our previous examples, schedule 1 is not conflict equivalent to schedule 2. However, schedule 1 is conflict equivalent to schedule 3, because the read(B) and write(B) instruction of T1 can be swapped with the read(A) and write(A) instruction of T2.
- We say that a schedule *S is* ***conflict serializable*** if it is conflict equivalent to a serial schedule.
- Thus, schedule 3 is conflict serializable, since it is conflict equivalent to the serial schedule 1.
- It is however possible to have two schedules that produce the same outcome, but that are not conflict equivalent.

View Serializability (1 of 3)

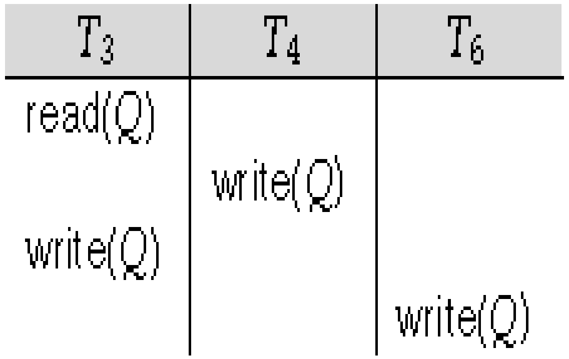
- It is a form of equivalence that is less stringent than conflict equivalence, but that, like conflict equivalence, is based on only the read and write operations of transactions.
- Consider two schedules *S* and *S*', where the same set of transactions participates in both schedules. The schedules *S* and *S*' are said to be view equivalent if three conditions are met:
  1. For each data item Q, if transaction T_i reads the initial value of Q in schedule *S,* then transaction T_i must, in schedule *S*', also read the initial value of Q.
  2. For each data item Q, if transaction T_i executes read(Q) in schedule *S*, and if that value was produced by a write(Q) operation executed by transaction T_j , then the read(Q) operation of transaction T_i must, in schedule *S*', also read the value of Q that was produced by the same write(Q) operation of transaction T_j.
  3. For each data item Q, the transaction (if any) that performs the final write(Q) operation in schedule *S* must perform the final write(Q) operation in schedule *S*'.

View Serializability (2 of 3)

- Conditions 1 and 2 above ensure that each transaction reads the same values in both schedules and, therefore, performs the same computation.
- Condition 3, coupled with conditions 1 and 2, ensures that both schedules result in the same final system state.
- In our previous examples, schedule 1 is not view equivalent to schedule 2, since, in schedule 1, the value of account A read by transaction T_2 was produced by T_1, whereas this case does not hold in schedule 2.
- However, schedule 1 is view equivalent to schedule 3, because the values of account A and B read by transaction T_2 were produced by T_1 in both schedules.
- The concept of view equivalence leads to the concept of view serializability. We say that a schedule *S* **is view serializable** if it is view equivalent to a serial schedule.
- Every conflict-serializable schedule is also view serializable, but there are view serializable schedules that are not conflict serializable.
- Transactions that perform write operations without having performed a read operation are called blind writes.
- Blind writes appear in any view-serializable schedule that is not conflict serializable.

View Serializability (3 of 3)

- The schedule in the figure is view serializable, since one read(Q) instruction reads the initial value of Q in both transactions and T_6 performs the final write of Q in both schedules.
- Every conflict-serializable schedule is also view serializable, but there are view serializable schedules that are not conflict serializable.
- Transactions that perform write operations without having performed a read operation are called blind writes.
- Blind writes appear in any view-serializable schedule that is not conflict serializable.



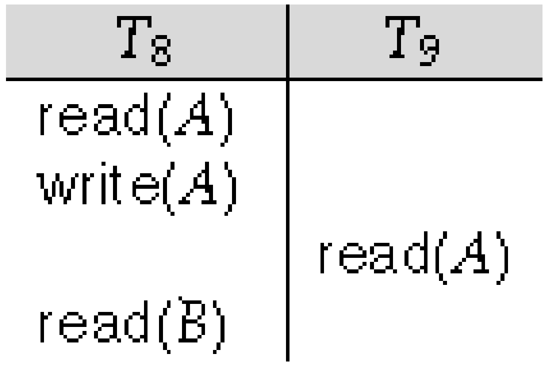
T_3	T_4	T_6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

Recoverability

- So far, we have studied what schedules are acceptable from the viewpoint of consistency of the database, assuming implicitly that there are no transaction failures.
- We now address the effect of transaction failures during concurrent execution.
- If a transaction Ti fails, for whatever reason, we need to undo the effect of this transaction to ensure the atomicity property of the transaction. In a system that allows concurrent execution, it is necessary also to ensure that any transaction Tj that is dependent on Ti (that is, Tj has read data written by Ti) is also aborted.
- To achieve this, we need to place restrictions on the type of schedules permitted in the system. Therefore, we need to address the issue of what schedules are acceptable from the viewpoint of recovery from transaction failure.

Recoverable Schedules

- A **recoverable schedule** is one where, 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 commit operation of T_j .
- Suppose that the system allows T_9 to commit immediately after executing the read(*A)* instruction in this example*.* Thus, T_9 commits before T_8 does.
- Now suppose that T8 fails before it commits. Since T_9 has read the value of data item A written by T_8, we must abort T_9 to ensure transaction atomicity.
- However, T_9 has already committed and cannot be aborted. Thus, we have a situation where it is impossible to recover correctly from the failure of T_8*.*
- This is an example of a ***non-recoverable*** schedule, which should not be allowed. Most database system require that all schedules be *recoverable.*



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

Cascadeless Schedules

- Even if a schedule is recoverable, to recover correctly from the failure of a transaction *T_i*, we may have to roll back several transactions. Such situations occur if transactions have read data written by *T_i*. For example:
  - If *T_{10}* fails, *T_{10}* must be rolled back. Since *T_{11}* is dependent on *T_{10}*, *T_{11}* must be rolled back. Since *T_{12}* is dependent on *T_{11}*, *T_{12}* must be rolled back.
- This phenomenon, in which a single transaction failure leads to a series of transaction rollbacks, is called **cascading rollback.**
- Cascading rollback is undesirable, since it leads to the undoing of a significant amount of work. It is desirable to restrict the schedules to those where cascading rollbacks cannot occur. Such schedules are called cascadeless schedules.
- A **cascadeless schedule** is one where, 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* . It is easy to verify that every cascadeless schedule is also recoverable.

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

Transaction Definition in SQL

- The SQL standard specifies that a transaction begins implicitly. Transactions are ended by one of these SQL statements:
  - ***Commit work***
  - ***Rollback work***
- The keyword **work** is optional in both the statements.
- The **Commit** statement is used in SQL to signal successful end of all updates in a transaction and it tells the DBMS to save all the changes to the database and terminate the current transaction.
- On the other hand, **Rollback** statement is used in SQL to abort all updates within the current transaction and also instructs the database to revert back to its original state before the transaction commenced.
- If a program terminates without either of these commands, the updates are either committed or rolled back (which of the two happens is not specified by the standard and depends on the implementation).
- The standard also specifies that the system must ensure both serializability and freedom from cascading rollback. The definition of serializability used by the standard is that a schedule must have the *same effect* as would some serial schedule. Thus, conflict and view serializability are both acceptable.

Example

- **Begin transaction**
- **insert into** orders **values** (“ÁCB234”, ’29-Jan-2013”, “S113”);
- **insert into** orders **values** (“ÁCB234”, ’28-Jan-2013”, “S176”);
- **….**
- **update** stock;
- **set** stock_qty = stock_qty - 25
- **where** prdtcode = “S560”;
- **update** stock
- **set** stock_qty = stock_qty -25
- **where** prdtcode = “K552”
- **….**
- **Commit**
- **End transaction**