Distributed Database System (CS-600)

Mr. Nauman Iqbal email id: nauman@biit.edu.pk Whatsapp# 0321-5821151
Mr. Ahsan email id: ahsan@biit.edu.pk, Whatsapp# 0333-5189656

(Week 9) Lecture 17-18 (Chapter 20 of Book)

Assignemnt#7 (given at End)

1. **Submission Date: Sunday 26th-April-2020 before 11:59PM.**
2. **You must prepare handwritten Assignment and send it to respective course teacher (after scanning it) for assessment by email only.**

Assignment submission through Email:

Email Subject must be in correct format otherwise assignment will not be checked.

Email Subject Format: 4digitReg-DDS-section-ASG3

Example: 1234-DDS-CS6A-ASG3 Correct

2017-ARID-1234-DDS-CS6A-ASG3 Incorrect

Objectives: Learning objectives of this lecture are

- **Concurrency Control**
- **Need of Concurrency Control**
- **Concurrency Problems**
- **Schedule**
- **Types of Schedule**
- **Conflict Serializability Graph**
- **Recoverability**

Distributed Database System (CS-600)

Mr. Nauman Iqbal **email id: nauman@biit.edu.pk Whatsapp# 0321-5821151**
Mr. Ahsan **email id: ahsan@biit.edu.pk, Whatsapp# 0333-5189656**

Overview:

In this lecture, we are going to learn "Concurrency Control" in transaction. What are concurrent transactions? Why do we need concurrency control for transaction? Problems in concurrent transaction and how can we solve those problems. How to detect concurrent transaction that my cause a conflict?

Concurrency control:

- The process of managing simultaneous operations on the database without having them interferes with one another.
- Prevents interference when two or more users are accessing database simultaneously and at least one is updating data.
- Although two transactions may be correct in themselves, interleaving of operations may produce an incorrect result.

Concurrent Transaction:

Multiple transaction that executes in DBMS at the same time are known as concurrent transactions.

Need for Concurrency Control:

A major objective in developing a database is to enable many users to access shared data concurrently. Concurrent access is relatively easy if all users are only reading data, as there is no way that they can interfere with one another. However, when two or more users are accessing the database simultaneously and at least one is updating data, there may be interference that can result in inconsistencies.

There are different problems which may occur due to concurrent execution of transactions on a DBMS as:

- **Lost update problem.**
- **Uncommitted dependency problem.**
- **Inconsistent analysis problem**

Lost Update Problem

An apparently successfully completed update operation by one user can be overridden by another user. This is known as the **lost update problem.**

Time	T1	T2	balx
t1		begin-transaction	100
t2	begin-transaction	read(balx)	100
t3	read(balx)	balx = balx + 100	100
t4	balx = balx -10	write(balx)	200
t5	write(balx)	commit	90
t6	commit		90

Example:

In above example, we have a database value balx (balance x), whose initial value is "100". This data item is read by both transactions T1 and T2. As we know, whenever READ operation is carried out during transaction, it reads data from database. So, when both transaction read balx, at that time value of balx was 100. So both read value 100 against "balx". Now transaction T2 added "100" and wrote "balx". Which becomes "200" now. But problem occurs here, transaction T1 read "100" and

subtracted "10" from it which becomes "90" and wrote it. Now transaction T1 overwrote value written by T2 and the final values is "90" now. But logically, value should be 190. So transaction T2 wrote a value but transaction T1 overwrite it with an incorrect value which is incorrect.

Uncommitted dependency (or dirty read) problem:

The uncommitted dependency problem occurs when one transaction is allowed to see the intermediate results of another transaction before it has committed.

Example:

Time	T1	T2	balx
t1		**begin-transaction**	**100**
t2		**read(balx)**	**100**
t3		**balx = balx + 100**	**100**
t4	**begin-transaction**	**write(balx)**	**200**
t5	**read(balx)**	**⋮**	**200**
t6	**balx = balx -10**	**rollback**	**100**
t7	**write(balx)**		**190**
t8	**commit**		**190**

In above example, transaction T2 writes value "200" to database successfully and then this value is read by transaction T1. But afterwards transaction T2 rollbacks due to any reason. We all know whenever transaction executes rollback operations it reverts all its operations. That is why value of balx is reverted back to 100 from 200. But unfortunately, transaction T1 has read written value which was 200 before rollback operations. Now transaction T1 will apply all operations of its read value which is 200. Now transaction T1 subtracts 10 from balx and value becomes 190 and writes it successfully. Here value of bal should be 90 because transaction T2 has issued rollback statement which leaves no changes in database but due to such sequence of operations, uncommitted value of transaction T2 has been read by transaction T1 which created this problem.

Inconsistent analysis problem:

The problem of inconsistent analysis occurs when a transaction reads several values from the database but a second transaction updates some of them during the execution of the first.

Example:

Time	T5	T6	balx	baly	balz	sum
t1		begin-transaction	100	50	25	
t2	begin-transaction	sum=0	100	50	25	0
t3	read(balx)	read(balx)	100	50	25	0
t4	balx = balx – 10	sum = sum + balx	100	50	25	100
t5	write(balx)	read(baly)	90	50	25	100
t6	read(balz)	sum = sum + baly	90	50	25	150
t7	balz = balz + 10	.	90	50	25	150
t8	write(balz)	.	90	50	35	150
t9	commit	read(balz)	90	50	35	150
t10		sum = sum + balx	90	50	35	185
t11		commit	90	50	35	185

In above example, we can see that transaction T5 is reading balx and balz, and updating changed values in database. In the meanwhile, transaction T6 only reads values of balx, baly and balz. Transaction T6 is adding values of all data items into a local variable "sum" for some analysis. Here we can see, "sum"

variable has a value "185" but if we add final values of balx, baly and balz then answer is "175". Which means analysis performed by transaction T6 is incorrect.

We have gone through problems of concurrent transactions. Before moving at the next step, we will understand some concepts given below.

Schedule: A set of transactions that appears in the system for execution at the same time is known as schedule.

Serial schedule: A schedule where the operations of each transaction are executed consecutively without any interleaved operations from other transactions.

Which means each transaction must be executed individually without any interference of any other transaction. If execution of transaction has started, then all of its operations will be executed first then other transaction will be executed. A transaction has to wait to be executed until previous transaction is completed.

Example:

T1	T2
	begin-transaction
begin-transaction	read(balx)
read(balx)	balx = balx + 100
balx = balx -10	write(balx)
write(balx)	commit
commit	

Here is a schedule that included 2 transaction. If above schedule is to be executed as "Serial Schedule" then it will look like.

Time	T1
t0	begin-transaction
t1	read(balx)
t2	balx = balx -10
t3	write(balx)
t4	commit
t5	T2
t6	begin-transaction
t7	read(balx)
t8	balx = balx + 100
t9	write(balx)
t10	commit

Here we can see transaction T2 is executed when transaction T1 is completed. In the meanwhile, T2 has to wait.

NOTE: Result of Serial Schedule is always correct. (Because transaction does not interfere in other transaction)

Non-serial schedule: A schedule where the operations from a set of concurrent transactions are interleaved.

Which means no waiting for other transaction, transactions will be executed in such a way operations will be interleaved.

Distributed Database System (CS-600)

Mr. Nauman Iqbal **email id: nauman@biit.edu.pk Whatsapp# 0321-5821151**
Mr. Ahsan **email id: ahsan@biit.edu.pk, Whatsapp# 0333-5189656**

Example:

T1	**T2**
	begin-transaction
begin-transaction	**read(balx)**
read(balx)	**balx = balx + 100**
balx = balx -10	**write(balx)**
write(balx)	**commit**
commit	

Here is a schedule that included 2 transaction. If above schedule is to be executed as "Non-Serial Schedule" then it will look like.

Transaction	Operation
T2	**begin-transaction**
T1	**begin-transaction**
T2	**read(balx)**
T1	**read(balx)**
T2	**balx = balx + 100**
T1	**balx = balx – 10**
T2	**write(balx)**
T1	**write(balx)**
T2	**commit**
T1	**commit**

Here we can see that operations of each transaction are executed one by one. In this way, none of transaction has to wait for other to be executed. But problem is here that may end with incorrect values in database.

Conclusion:

Serial Schedule	**Non-Serial**
Always correct data	**May corrupt database by incorrect values**
Practically impossible to implement by very long waiting intervals.	**No waiting time.**

Serializable Schedule:

A non-serial schedule which produces same result as same serial execution will be known as "Serializable" schedule. In this case database will be correct and there will be no waiting as well which is our desire. To prevent inconsistency from transactions interfering with one another, it is essential to guarantee Serializability of concurrent transactions.

In Serializability, the ordering of read and write operations is important:

- If two transactions only read a data item, they do not conflict and order is not important.
- If two transactions either read or write completely separate data items, they do not conflict and order is not important.

Mr. Nauman Iqbal **email id: nauman@biit.edu.pk Whatsapp# 0321-5821151**
Mr. Ahsan **email id: ahsan@biit.edu.pk, Whatsapp# 0333-5189656**

- If one transaction writes a data item and another either reads or writes the same data item, the order of execution is important.

Conflict Serializability Graph:

Conflict serializibility graph tell whether given schedule has conflicting statements or not. If it does have conflicting statement then system can re-arrange operations to avoid any sort of conflict. Given are steps for creating "Conflict Serialzablity" graph.

- Create a node for each transaction.
- Create a directed edge T*i* → T*j*, if T*j* reads the value of an item written by T*i*.
- Create a directed edge T*i* → T*j*, if T*j* writes a value into an item after it has been read by T*i*.
- Create a directed edge T*i* → T*j*, if T*j* writes a value into an item after it has been written by T*i*.

If graph contains a "Cycle" then it means schedule is "NOT Serializable".
If graph does not contain a "Cycle" then it means schedule is "Serialziable".

Example 1:

Time	T1	T2
t1		begin-transaction
t2	begin-transaction	read(balx)
t3	read(balx)	balx = balx + 100
t4	balx = balx -10	write(balx)
t5	write(balx)	commit
t6	commit	

Steps:

1- Create Node for each transaction

2- Here we can see T1 write(x) which is previously written by T2. So arrow from T2 to T1.

3- Now we can see that T2 write a data item which is previously read by T1. So arrow from T1 to T2.

Mr. Nauman Iqbal **email id: nauman@biit.edu.pk Whatsapp# 0321-5821151**
Mr. Ahsan **email id: ahsan@biit.edu.pk, Whatsapp# 0333-5189656**

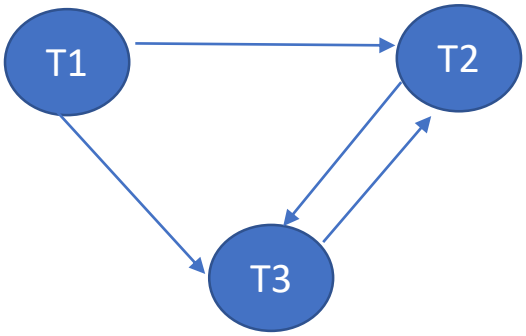
NOTE: if an arrow already exists from one node to another, then no need to duplicate arrow.

Conclusion:

- As there is a cycle in above graph so we will declare that above schedule is **"NOT Serializable".**

Example 2:

Time	T1	T2	T3
t1		begin-transaction	
t2	begin-transaction	read(baly)	
t3	read(balx)	baly = baly + 100	begin-transction
t4	balx = balx -10	write(baly)	read(baly)
t5	write(balx)	read(balx)	read(balx)
t6	commit	baly = balx + 50	baly = baly + balx
t7		write(baly)	write(baly)
t8		commit	commit



As there is a cycles between T2 and T3 in above graph, so schedule is **"NOT Serializable".**

Recoverability: Serializability identifies schedules that maintain the consistency of the database, assuming that none of the transactions in the schedule fails. An alternative perspective examines the *recoverability* of transactions within a schedule. If a transaction fails, the atomicity property requires that we undo the effects of the transaction. In addition, the durability property states that once a transaction commits, its changes cannot be undone (without running another, compensating, transaction).

Recoverable schedule: A schedule where, for each pair of transactions T*i* and T*j*, if T*j* reads a data item previously written by T*i*, then the commit operation of T*i* precedes the commit operation of T*j*.

Distributed Database System (CS-600)

Mr. Nauman Iqbal **email id: nauman@biit.edu.pk Whatsapp# 0321-5821151**
Mr. Ahsan **email id: ahsan@biit.edu.pk, Whatsapp# 0333-5189656**

Example:

Time	T1	T2
t1		begin-transaction
t2		read(balx)
t3		balx = balx + 100
t4	begin-transaction	write(balx)
t5	read(balx)	⋮
t6	balx = balx -10	⋮
t7	write(balx)	⋮
t8	commit	⋮
		commit

Above schedule is **"Not Recoverable"** because according to definition of recoverable schedule, if a transaction reads a data item previously written by other transaction. Then transaction which writes will commit first. Here T2 writes first so commit operations must be before commit operation of T1.

Distributed Database System (CS-600)

Mr. Nauman Iqbal **email id: nauman@biit.edu.pk Whatsapp# 0321-5821151**
Mr. Ahsan **email id: ahsan@biit.edu.pk, Whatsapp# 0333-5189656**

Assignment #5

Question#1:

Draw "Conflict Serializability" graph for given schedule and tell if schedule is serializable or not.

time	T1	T2	T3	T5
t_0				
t_{01}	Read(x)		Read(y)	Read(z)
t_{02}	x=x+10	Read(t)	y=y*2	z=z+50
t_{03}	write(x)	t=t-20	y=y+100	write(z)
t_{04}				
t_{05}	read(a)	read(b)	Read(c)	
t_{06}	a=a+50	b=b+t		read(y)
t_{07}	commit	read(e)	Read(z)	y=y*5
t_{08}		e=e+80	z=z+c	Write(y)
t_{09}		Write(e)	write(z)	
t_{10}		Write(b)	commit	commit
t_{11}		commit		

Question#2:

Execute the following schedules as "serial" and "non-serial" schedule and display final values of data items for each.

i- DB Values: balx=60

Time	T1	T2
t1		**begin-transaction**
t2	**begin-transaction**	**read(balx)**
t3	**read(balx)**	**balx = balx + 100**
t4	**balx = balx -10**	**write(balx)**
t5	**write(balx)**	**commit**
t6	**commit**	

Mr. Nauman Iqbal **email id: nauman@biit.edu.pk Whatsapp# 0321-5821151**
Mr. Ahsan **email id: ahsan@biit.edu.pk, Whatsapp# 0333-5189656**

---

ii- DB Values: balx=50, baly=60, balz=100

Time	T5	T6
t1		begin-transaction
t2	begin-transaction	sum=0
t3	read(balx)	read(balx)
t4	balx = balx – 10	sum = sum + balx
t5	write(balx)	read(baly)
t6	read(balz)	sum = sum + baly
t7	balz = balz + 10	.
t8	write(balz)	.
t9	commit	read(balz)
t10		sum = sum + balx
t11		commit

Question#3:

Solve Lab Manual given with lecture notes.