**201433707 이형욱**

**14.4** Justify the following statement: Concurrent execution of transactions is more important when data must be fetched from (slow) disk or when transactions are long, and is less important when data are inmemory and transactions are very short.

**Answer:**

If a transaction is very long or when it fetches data from a slow disk, it takes a long time to

complete. In absence of concurrency, other transactions will have to wait for longer period of

time. Average responce timewill increase.Alsowhen the transaction is readingdata from disk,

CPU is idle. So resources are not properly utilized. Hence concurrent execution becomes

important in this case.However,when the transactions are short or the data is available in

memory, these problems do not occur.

**14.6** Consider the precedence graph of Figure 14.16. Is the corresponding

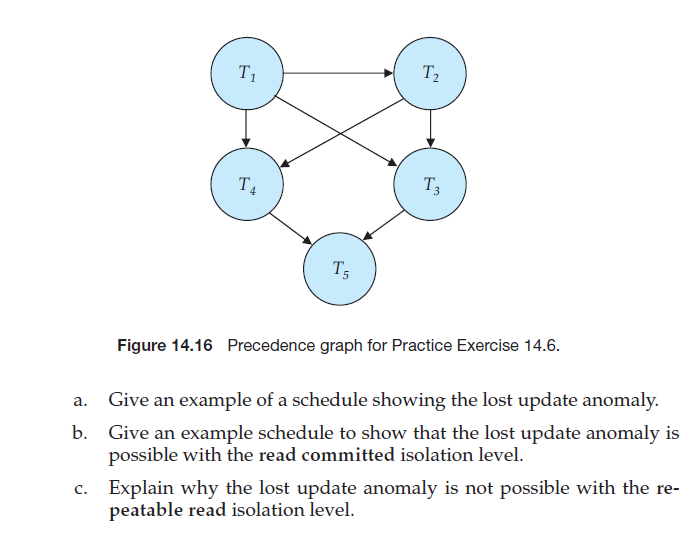
schedule conflict serializable? Explain your answer.

**Answer:**

There is a serializable schedule corresponding to the precedence graph below, since the graph

is acyclic. A possible schedule is obtained by doing a topological sort, that is, *T*1, *T*2, *T*3, *T*4,

*T*5.

**14.8** The **lost update** anomaly is said to occur if a transaction *Tj* reads a data item, then another transaction *Tk* writes the data item (possibly based on a previous read), afterwhich *Tj* writes the data item.The update performed by *Tk* has been lost, since the update done by *Tj* ignored the value written by *Tk* . 

a. A schedule showing the Lost Update Anomaly:

T1 | T2

------------------------------------

Read(A) |

| Read(A)

| write(A)

Write(A) |

In the above schedule, the value written by the transaction *T*2 is lost because of the write of

the transaction *T*1.

b. Lost Update Anomaly in Read Committed Isolation Level

T1 | T2

------------------------------------

Lock-S(A) |

Read(A) |

Unlock(A) |

| lock-X(A)

| read(A)

| write(A)

| unlock(A)

| commit

Lock-X(A) |

Write(A) |

Unlock(A) |

Commit |

c. Lost Update Anomaly is not possible in Repeatable Read isolation level. In repeatable read

isolation level, a transaction *T*1 reading a data item *X*, holds a shared lock on *X* till the end.

This makes it impossible for a newer transaction *T*2 to write the value of *X* (which requires

X-lock) until *T*1 finishes. This forces the serialization order *T*1, *T*2 and thus the value written

by *T*2 is not lost.

**14.14** Explain the distinction between the terms *serial schedule* and *serializable*

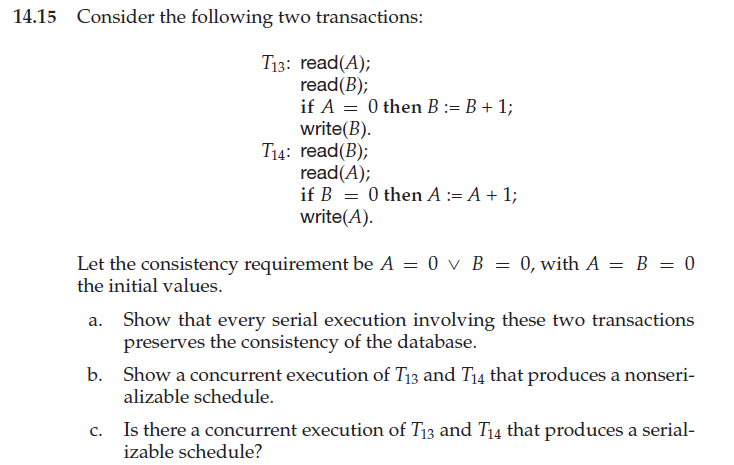
*schedule*.

**Answer:**

A schedule inwhich all the instructions belonging to one single transaction appear together is

called a *serial schedule*. A *serializable schedule* has a weaker restriction that it should be *equivalent* to some serial schedule. There are two definitions of schedule equivalence –

conflict equivalence and view equivalence. Both of these are described in the chapter.



**a.** There are two possible executions: *T*1 *T*2 and *T*2 *T*1

|  |  |
| --- | --- |
| 0  0  0 | 0  1  1 |

Case 1: A B

Initially

After t1

After t2

Consistency met : A=0ⅴB=0 ≡ TⅴF=T

Case 2: A B

|  |  |
| --- | --- |
| 0  1  1 | 0  0  0 |

Initially

After t2

After t1

Consistency met : A=0ⅴB ≡ FⅴT=T

**b.** Any interleaving of *T*1 and *T*2 results in a non-serializable schedule.



T1 | T2

------------------------------------

read(A) |

| read(B)

| read(A)

Read(B) |

If A=0 then B=B+1 | |

| if B=0 then A=A+1

| write(A)

Write(B) |

c. There is no parallel execution resulting in a serializable schedule. From part a. we know that a serializable schedule results in A = 0 ∨ B = 0. Suppose we start with T1 read(A). Then when the schedule ends, no matter when we run the steps of T2, B = 1. Now suppose we start executing T2 prior to completion of T1. Then T2 read(B) will give B a value of 0. So when T2 completes, A = 1. Thus B = 1 ∧ A = 1 → ¬ (A = 0 ∨ B = 0). Similarly for starting with T2 read(B).

**14.16** Give an example of a serializable schedule with two transactions such

that the order in which the transactions commit is different from the

serialization order.

**Answer:**

Let *r*, *s* and *t* be three relations. Consider a materialized view on these defined by (*r* 1 *s* 1 *t*).

Suppose relation *r* does not have any attributes common to *s* or *t*, while *s* and *t* have foreign

key relationship. Each of them have 1000 tuples and 100 tuples are added to *r*. Then

recomputation is better because (*s* 1 *t*) can be computed first which will have 1000 tuples. It

can then be joined with *t*. In incremental view maintenance, the increment in *t* will first be

joined with either *s* or *t* which will have 100000 tuples (cartesian product). This huge relation

will then be joined with *t* which will be very expensive. However, if 100 tuples are added to *s*

instead of *r* in the above situation, incremental view maintenance will obviously be better as

increment in *s* can be joined with *t* to get a relation of size 100 which can then be joined with

*r*.