Unit 13
Concurrency Control

Content

- 13.1 Introduction
- 13.2 Serializability 可串聯的
- 13.3 Locking Technique
- 13.4 Two-Phase Locking

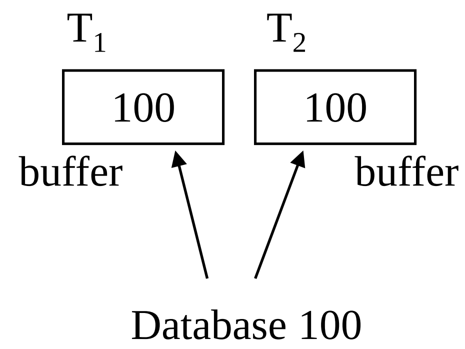
13.1 Introduction

Concurrency Control: Introduction

- **The Problem**
  - In a multiple-user DBMS, how to ensure **concurrent transactions** do not interfere with each other's operation?
- **Why concurrent transaction?**
  - minimize response time
  - maximize throughout
- **Concurrency control techniques**
  - Locking
    - 2PL (§13.4)
    - Tree protocol locking

      ⋮
  - Optimistic method
    - Timestamp ordering

      ⋮



Problem: Lost Update

- **The problem when works without concurrency control**
  1. Lost Update: Fig. 13.1
  2. Uncommitted Dependence: Fig. 13.2

<Example 13.1> Lost Update

Transaction A	**R=10** time	Transaction B
— —	|	— — — —
FETCH R R=10	t1	
— — —	t2	FETCH R R=10
UPDATE R R<--R+1=11	t3	— — —
— — —	***t4***	**UPDATE R** R<--R*2=20
	↓	

Fig. 13.1: Transaction A loses an update at time t4

if A--> B : R= (10+1) * 2 = 22

if B--> A : R= (10*2) + 1 = 21

Problems: Uncommitted Dependence

<Example 13.2> Uncommitted Dependence

Transaction A	R=10 time	Transaction B
—		—
—		—
—		
—	t1	UPDATE R R<--20
		—
FETCH R R=20	*t2*	—
—		—
—		—
S<--R+1=21	t3	**ROLLBACK** R=10
—	↓	

Fig. 13.2: Transaction A becomes <u>dependent on an uncommitted change</u> at time t2.

13.2 Serializability

可串聯的

Serializability: Definition

交錯並聯

- A formal criterion for correctness.
- Def : A given **interleaved execution (schedule)** of some set of transactions is said to be **serializable** iff it produces the same result as some serial execution of those transactions, for any given initial database state.
- Assumptions:
  - all transactions are individually correct.
  - Any serial execution of those transactions is also correct.
  - Transactions are all independent of one another.

Note: if **Tx A** does have to be run before **Tx B**, then user cannot submit **Tx B** until **Tx A** is committed.

Serializability: Example 13.3

- **<Example 13.3>**

  T1 = { r1(d), w1(d) }
  T2 = { r2(a), w2(a) }

  - **Serial execution** : $\underbrace{r1(d), w1(d)}_{T1}, \underbrace{r2(a), w2(a)}_{T2}$ T1 → T2

  交錯並聯 - **Interleaved execution** (schedule) : $\underbrace{r1(d)}_{T1}, \underbrace{r2(a)}_{T2}, \underbrace{w1(d)}_{T1}, \underbrace{w2(a)}_{T2}$

  - Serializable?

- Serialization: the serial execution that equivalent to the serializable execution, e.g. T1 - T2

Serializability: Example 13.4(a)

<Example 13.4>

Given : T1: Add 1 to A

T2: Double A

T3: Display A and set A to 1

Initial value: A = 0

<a> How many possible correct results ? 3! = 6.

T1 - T2 - T3 : A=1

T1 - T3 - T2 : A=2

T2 - T1 - T3 : A=1

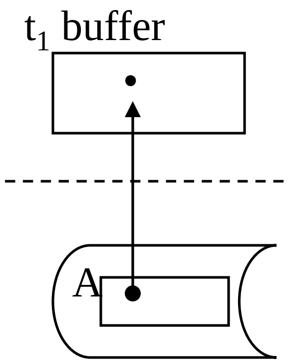
T2 - T3 - T1 : A=2

T3 - T1 - T2 : A=4

T3 - T2 - T1 : A=3

Serializability: Example 13.4(b)

<**b**> Suppose the internal structures of T1, T2, T3 are :



T1:	T2 :	T3 :
F1: Fetch A into t1	F2 : Fetch A into t2	F3 : Fetch A into t3
t1:= t1+1	t2 := t2*2	display t3
U1: Update A from t1	U2 : Update A from t2	U3 : Update A to 1

How many possible interleaved executions? 90

Fi- Fj- Fk- Up- Uq- Ur: 3*2*1*3*2*1 = 36

Fi- Fj- Up- Fk- Uq- Ur: 3*2*2*1*2*1 = 24

Fi- Fj- Up- Uq- Fk- Ur: 3*2*2*1*1*1 = 12

Fi- Up- Fj- Fk- Uq- Ur: 3*1*2*1*2*1 = 12

Fi- Up- Fj- Uq- Fk- Ur: 3*1*2*1*1*1 = 6

Fi- Up- Uq- Fj- Fk- Ur ?

Serializability: Example 13.4(c)

<c> **Is there any** interleaved executions **that produces "correct" result but is <u>not serializable</u> ?**

Consider the schedule (△):

△ : F1- F2- F3- U3- U2- U1 = 1 (if A=0) same as T1-T2-T3

But, Consider initial value of A is 10***:***

T1-T2-T3 : 1
T1-T3-T2 : 2
T2-T1-T3 : 1
T2-T3-T1 : 2
T3-T1-T2 : 4
T3-T2-T1 : 3

F1-F2-F3-U3-U2-U1=11 ≠ any serial execution
(10) (10) (10) (1)(20) (11)

∴ not serializable !

- An **interleaved execution (schedule)** of some set of transactions is considered to be **correct** iff it is **serializable** !

Testing for Serializability

- Two operations are said to be **conflict** if

  <1> they come from different transaction.

  <2> they operate on the same data element.

  <3> at least one of them is write operation.

  <e.g.> T1= { r1(a), w1(b) }

  T2= { r2(b), w2(c) }
  T3= { r3(c), w3(a) }

  - r1(a), w3(a) are conflict
  - r2(b), w1(b) are conflict
  - r3(c), w2(c) are conflict

Testing for Serializability (cont.)

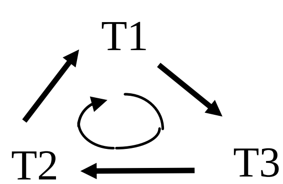
- Transaction Dependency Graph:
  - Nodes: transactions, e.g. T1, T2, T3
  - Arcs: dependence

T1 --> T2 if O1 and O2 are conflict, and O1 before O2 in a schedule **S.**

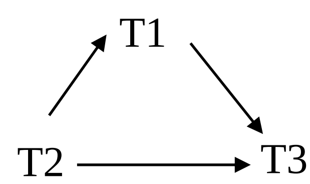
<e.g.> S1: r1(a), r3(c), r2(b), w1(b), w3(a), w2(c)

S2: r1(a), r2(b), w1(b), w2(c), r3(c), w3(a)

S1 :



S2 :



T2→T1→T3

- The Acyclicity Theorem
  - An interleaved transaction schedule is serialization iff its transaction dependency graph is acyclic.

13.3 Locking Technique

Locking Technique: Concept

- The effect of a lock is to lock other transaction out of the object.
- Two kinds of locks
  - Exclusive lock (X locks): for UPDATE
  - Shared lock (S locks): for RETRIEVE

- Compatibility matrix

A \ B	X	S	—
X	N	N	Y
S	N	Y	Y
—	Y	Y	Y

— : no lock
N : request not compatible
Y : request compatible

Locking Technique: Concept (cont.)

B / A	X	S	—
X	N	N	Y
S	N	Y	Y
—	Y	Y	Y

— : no lock
N : request not compatible
Y : request compatible

<e.g.> if transaction A holds a S lock on R, then

<1> a request from B for X lock on R

=> B goes into wait state.

<2> a request from B for S lock on R

=> B also hold the S lock on R

Ref: X locks and S locks are normally held until the next synchpoint. (Ref. p.12-12)

How locking solves the problems

- The lost Update Problem

Transaction A	R=10 time	Transaction B
——	|	——
——	|	——
FETCH R R=10	t1	——
(acquire S lock on R)	|	——
		——
		——
——	|	
——	t2	FETCH R R=10
——		(acquire S lock on R)
——	|	
——	|	——
UPDATE R R=10+1=11	t3	——
(request X lock on R)	|	——
wait	|	——
		——
wait	t4	UPDATE R R=10*2=20
wait	|	(request X lock on R)
wait	|	wait
wait	|	wait
wait	↓	wait

Fig. 13.4: No update is lost, but deadlock occurs at time t4.

How locking solves the problems (cont.)

- The Uncommitted Dependence Problem

Transaction A	time	Transaction B
—	|	—
—	|	—
—		
—	t1	UPDATE R R=20
—	|	(acquire X lock on R)
—	|	—
FETCH R	t2	—
(request S lock on R)	|	—
wait	|	—
wait	t3	synchpoint (ROLLBACK)
wait	|	(release X lock on R) R=10
resume : FETCH R	t4	
(acquire S lock on R)	↓	
—		

Fig 13.5: Transaction A is prevented from seeing an uncommitted change at time t2.

Deadlock Detection: Wait-for-Graph

- Deadlock Detection: **Wait-for-Graph**
  - **node**: transactions
  - **arc**: an edge from node Ti to Tj means Ti request a lock on an object that is hold by Tj.
  - the system draw an edge from Ti to Tj when the request is issued and erase that edge when Tj release the lock.
  - if there are edges from T1 to T2, T2 to T3, ..., Tn-1 to Tn, and Tn to T1 ==> T1, T2, ..., Tn are deadlocked.

Transaction A	time	Transaction B
—		—
—		—
—		—
request X lock on R1	t1	—
—		—
—	t2	request X lock on R2
—		—
request X lock on R2	t3	—
wait		—
wait	t4	request X lock on R1
wait		wait
wait		wait

Fig. 13.7: An example of deadlock.

Locking Protocol

- **Locking Protocol**: to ensure the Serializability
  1. Two-phase locking (2PL)
  2. Non-two-phase locking
     - Tree protocol locking
     - Directed acyclic graph protocol

Testing for Serializability

- **Testing for Serializability in Locking Protocol**
  - **Precedence Graph:**
    - **Node**: transactions
    - **Arc**: an **arc** from Ti to Tj (Ti $\longrightarrow$ Tj)
      if Oi $\in$ Ti, Oj $\in$ Tj and
      <1> Oi and Oj operates on the same data.
      <2> Oi is UNLOCK, Oj is LOCK.
      <3> Oi precedes Oj.
    - e.g. T1, T2

$T_1 \rightarrow T_2$	$T_2 \rightarrow T_1$
T_1 UNLOCK(d)	T_2 UNLOCK(d)
T_2 LOCK(d)	T_1 LOCK(d)

Theorem for Testing Serializability

- **Thm 13.1**: If the precedence graph of a schedule contains **no cycle** then schedule is **serializable**.

<Example 13.6>

Consider the following three transactions:

T_1: Lock A
Lock B
UNLock A
UNLock B

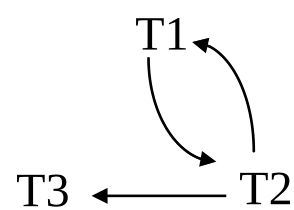
T_2: Lock B
Lock C
UNLock B
Lock A
UNLock C
UNLock A

T_3:

- A schedule for T1, T2, and T3:

(1) T1 : LOCK A
(2) T2 : LOCK B
(3) T2 : LOCK C
(4) T2 : UNLOCK B — From (4), (5) we have T2 $\longrightarrow$ T1
(5) T1 : LOCK B
(6) T1 : UNLOCK A — From (6), (7) we have T1 $\longrightarrow$ T2
(7) T2 : LOCK A
(8) T2 : UNLOCK C — From (8), (11) we have T2 $\longrightarrow$ T3
(9) T2 : UNLOCK A
(10) T3 : LOCK A
(11) T3 : LOCK C
(12) T1 : UNLOCK B
(13) T3 : UNLOCK C
(14) T3 : UNLOCK A

- The precedence graph of the schedule is:



not serializable !

13.4 Two-Phase Locking

Two-Phase Locking (2PL): Definition

- A protocol that guarantees **serializability**.
- A transaction obeying the **two-phase locking protocol (2PL)** if

  <a> before operating on any object, the transaction first acquires a lock on that object (the locking phase)

  <b> after releasing a lock, the transaction never acquires any more lock (the unlocking phase)

  i.e. in any transaction, all locks must precede all unlock.

2PL: Example

<e.g.>

T1:	T2:	T3:
LOCK A	LOCK A	LOCK B
LOCK B	LOCK C	LOCK C
UNLOCK A	UNLOCK C	UNLOCK B
UNLOCK B	UNLOCK A	LOCK A
		UNLOCK C
		UNLOCK A

T1 obey 2PL
T2 obey 2PL
T3 not obey 2PL

<e.g.1> T1, T2 any interleaved schedules are serializable.

<e.g.2> T1, T2 , T3 perhaps

Note: In practice, a lock releasing phase is often compressed into the single operation of COMMIT (or ROLLBACK) at end-of-transaction.

2PL: Theorm

- **Thm 13.2:** If all transactions obey the "2PL" protocol, then all possible interleaved schedules are serializable.

  [proof]: [by contradiction]

  Suppose not. Then by Thm 13.1, the **precedence Graph G** for **S** has a cycle, say

  $T_{i1} \Rightarrow T_{i2} \Rightarrow \ldots \Rightarrow T_{ip} \Rightarrow T_{i1}$.

  eg. $T1 \rightarrow T2 \rightarrow T3 \rightarrow T1$

  Then some LOCK by T_{i2} follows an UNLOCK by T_{i1};

  some LOCK by T_{i3} follows an UNLOCK by T_{i2};

  ⋮

  some LOCK by T_{i1} follows an UNLOCK by T_{ip};

  => A LOCK by T_{i1} follows an UNLOCK by T_{i1}

  => T_{i1} disobey of 2L protocol !! **Q.E.D. #**

UNLOCK LOCK

$Ti1 \rightarrow Ti2$

$Ti2 \rightarrow Ti3$

…

…

$Tip \rightarrow Ti1$

=>

One lock of Ti1 is after unlock!!

<e.g.1> T1, T2 any interleaved schedules are serializable.

<e.g.2> T1, T2 , T3 perhaps

2PL: Example 13.4(d)

<Example 13.4(d)>

<d> Is there any interleaved execution of T1, T2, T3 that is serializable but could not be produced if all three transactions obeyed the 2PL?

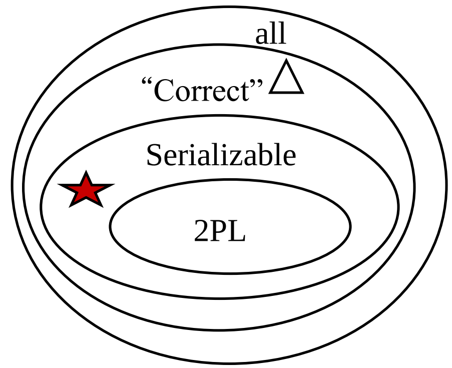
T1 :	T2 :	T3 :
F1 : Fetch A into t1	F2 : Fetch A into t2	F3 : Fetch A into t3
t1 := t1+1	t2 := t2*2	display t3
U1 : Update A from t1	U2 : Update A from t2	U3 : Update A from t1

2PL: Example 13.4(d) (cont.)

★ **Yes ! F1 - F3 - U1 - U3 - F2 - U2 = T1-T3-T2 is serializable**

SLOCK SLOCK wait until T3 terminates

∵ Operation U1 will not be able to proceed until that SLOCK by F3 has been released, and that will not happen until T3 terminates.



In fact, transaction T3 and T1 will deadlock when U3 is reached.

△ Ref. p.13-12

<Note> 2PL ≤ Serializable ≤ "Correct" ≤ All interleaved schedules

end of unit 13