Problem Set 3

Part 1

Brendan Murphy

Problem 1: Conflict Serializability

Schedule 1:

w1(A); r2(A); w2(B); r3(B); w3(C); r1(C)

Precedence Graph for Schedule 1:

T1	T2	T3
w(A)		
	r(A)	
	w(B)	
		r(B)
		w(C)
r(C)		

T1 reads the value that T3 writes for C. T2 reads the value that T1 writes for A. T3 reads the value that T2 writes for B. T1`s write of A is the final one. T2`s write of B is the final one. T3`s write of C is the final one. The action of reading C in T1 after T3 has written to it makes it not consistent with a serial schedule. The precedence graph above indicates that Schedule 1 is <u>not conflict serializable</u> because there is no way to turn it into an equivalent serial schedule by swapping pairs of consecutive actions that do not conflict.

Schedule 2:

r1(C); w1(A); r3(A); w2(B); r3(B); w3(C)

Precedence Graph for Schedule 2:

T1	T2	T3
r(C)		
w(A)		
		r(A)
	w2(B)	
		r(B)
		w(C)

T3 reads the value that T2 writes for B which means T2 must come before T3. T3 reads the value that T1 writes for A. which means that T1 must come before T3. The precedence graph above indicates that Schedule 2 is conflict serializable because there is a way to turn it into an equivalent serial schedule by swapping pairs of consecutive actions that do not conflict. The schedule is equivalent to serial ordering T1;T2;T3;

Problem 2: Two-phase locking and isolation

Consider these two transactions:

```
T1: r(B); r(C); r(A); w(C); c
T2: r(B); r(A); r(C); w(A); c
```

Complete a schedule that follows two-phase locking rules:

Recoverable

T1	T2
sl(b)	
R(b)	
	Sl(b)
	r(b)
	Sl(a)
	R(a)
Sl(c)	
R(c)	
	Sl(c)
	R(c)
Sl(a)	
R(a)	
	Xl(a)
	W(a)
	U(b)
	u(a)
	U(c)
	commit
xl(c)	
W(c)	
U(b)	
U(c)	
U(a)	
commit	

The schedule observes rigorous locking because all locks are held until they are committed. The schedule above is recoverable because neither transaction is reading the others transaction's write of a data-item before the other's commit. The schedule is cascadeless because there are no dirty reads.

Complete a schedule that follows two-phase locking rules:

Unrecoverable

T1	T2
sl(b)	
R(b)	
	Sl(b)
	r(b)
	Sl(a)
	R(a)
Sl(c)	
R(c)	
	Sl(c)
	R(c)
	Xl(a)
	W(a)
	U(b)
	u(a)
	U(c)
Sl(a)	
R(a)	
W(c)	
U(b)	
U(c)	
U(a)	
commit	
	commit

The schedule does not observe rigorous locking because all of the locks are not held until they are committed. The schedule above is not recoverable because T1 reads T2's write of A, and then T1 commits before T2. This is a dirty read and classifies the schedule as *not* cascadeless.

Problem 3: Lock Modes

```
xl2(A); w2(A); sl1(B); r1(B); ul3(C);
```

1. ul1(A): This lock request is denied because T2 holds an exclusive lock on item A.
2. sl3(B): This lock request is granted because there are no exclusive locks on item B.
3. ul2(B): This lock request is granted because no transaction holds an exclusive lock on item B.
4. sl1(C): This lock request is denied because T3 holds an update lock on item C.
5. xl3(C): This lock request is denied because T3 holds an update lock on item C.
6. ul2(C): This lock request is denied because T3 holds an update lock on item C.

Problem 4: Deadlock detection

Sequence 1:

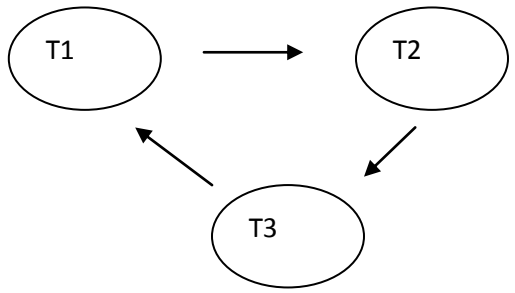
```
r2(C); r1(D); w1(C); r3(C); w2(C); w3(D); w1(D)
```

In this sequence deadlock will occur under rigorous two-phase locking because there is a cycle in the waits-for graph.

Schedule for sequence 1:

T1	**T2**	**T3**
	Sl(c); R2(c)	
Sl(d); R1(d)		
Xl(d); W1(c) denied; wait for T2		
		Sl(c); R3(c)
	Xl(c); W2(c) denied; wait for T3	
		Xl(d); W3(d) denied; wait for T1

Waits-for graph Sequence 1:



Sequence 2:

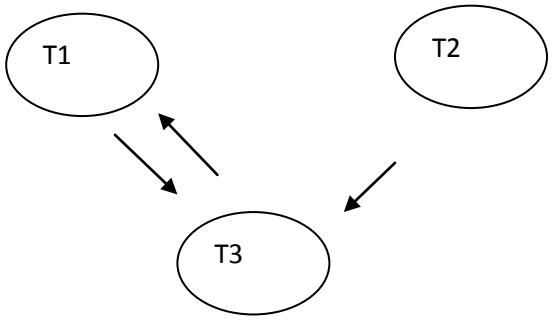
```
w3(A); r2(A); w2(A); r1(B); r3(B); w3(B); w1(B)
```

Schedule for sequence 2:

T1	**T2**	**T3**
		Xl(a); W3(a)
	Sl(a); R2(a) denied; wait for T3	
Sl(b); R1(b)		
		Sl(b); R3(b)
		Xl(b); W3(b) denied; wait for T1
Xl(b); W1(b) denied; wait for T3		

In this sequence deadlock will occur under rigorous two-phase locking because there is a cycle in the waits-for graph.

Waits-for graph sequence 2:



Problem 5: Timestamps and multiple versions

Consider the following sequence of operations:

```
s1; s2; s3; s4; s5; w3(A); w1(A); w4(A); r5(A); r2(A); c1; c2; c3; c4; c5
```

1. In response to the sequence above, a DBMS using regular timestamp-based concurrency *without* commit bits would behave as follows:

   s1;
   T1 begins and is assigned a timestamp TS1 = 1;
   s2;
   T2 begins and is assigned a timestamp TS2 = 2;
   s3;
   T3 begins and is assigned a timestamp TS3 = 3;
   s4;
   T4 begins and is assigned a timestamp TS4 = 4;
   s5;
   T5 begins and is assigned a timestamp TS5 = 5;

   w3(A);

   -T3 is allowed to write, because A has a WTS of zero
   -WTS(A) = 3

   w1(A);

   -ignored, because A has a WTS of 3 (Thomas Wright Rule)

   w4(A);

   -allowed, because A has a WTS of 3
   -WTS(A) = 4

   r5(A);

   -allowed, because A has a RTS of zero but this is a dirty read.
   -RTS(A) = 5

r2(A);

-denied, because A has a RTS of 5
-abort T2 and restart it sometime later with a new timestamp

c1;
c2;
c3;
c4;
c5;

2. In response to the sequence above, a DBMS using regular timestamp-based concurrency *with* commit bits would behave as follows:

s1;
T1 begins and is assigned a timestamp TS1 = 1;
s2;
T2 begins and is assigned a timestamp TS2 = 2;
s3;
T3 begins and is assigned a timestamp TS3 = 3;
s4;
T4 begins and is assigned a timestamp TS4 = 4;
s5;
T5 begins and is assigned a timestamp TS5 = 5;

w3(A);

-T3 is allowed to write, because A has a WTS of zero
-WTS(A) = 3
-A.commit = false

w1(A);

-denied: wait

w4(A);

-allowed, because A has a WTS of 3
-WTS(A) = 4
-A.commit = false

r5(A);

-allowed, because A has a RTS of zero but this is a dirty read.
-RTS(A) = 5

r2(A);

-denied, because A has a RTS of 5
-abort T2 and restart it sometime later with a new timestamp

c1;
c2;
c3;
c4;
A.commit = true
c5;

3. In response to the sequence above, a DBMS using *multiversion* timestamp-based concurrency *without* commit bits would behave as follows:
   s1;
   T1 begins and is assigned a timestamp TS1 = 1;
   s2;
   T2 begins and is assigned a timestamp TS2 = 2;
   s3;
   T3 begins and is assigned a timestamp TS3 = 3;
   s4;
   T4 begins and is assigned a timestamp TS4 = 4;
   s5;
   T5 begins and is assigned a timestamp TS5 = 5;

   w3(A);

   -T3 is allowed to write, a new copy of A(3) is created with:
   -WTS(A) = 3

   w1(A);

   -T1 is allowed to write, a new copy of A(1) is created with:
   -WTS(A) = 1

w4(A);

-T4 is allowed to write, a new copy of A(4) is created with:
-WTS(A) = 4

r5(A);

-allowed, because A has a RTS of zero but this is a dirty read.
-A(4) is the version read
-RTS(A) = 5

r2(A);

-denied, because A has a RTS of 5
-abort T2 and restart it sometime later with a new timestamp

c1;
c2;
c3;
c4;
c5;

Problem 6: A schema for an XML database

```
<!ELEMENT author-data ((author | book |  wrote)*)>
        <!ELEMENT author (name, dob?))>
                <!ATTLIST author_id ID #REQUIRED
                                book IDREFS #REQUIRED>
                <!ELEMENT name (#PCDATA)>
                <!ELEMENT dob (#PCDATA)>
        <!ELEMENT book ( title, publisher, num_pages))>
                <!ATTLIST book isbn ID #REQUIRED
                                genre #PCDATA “fiction” >
                <!ELEMENT title (#PCDATA)>
                <!ELEMENT publisher (#PCDATA)>
                <!ELEMENT num_pages (#PCDATA)>
        <!ELEMENT wrote EMPTY>
                <!ATTLIST wrote author_id IDREF #REQUIRED
                                Isbn IDREF #REQUIRED>
```

Note: a DTD cannot fully specify a foreign key restrain because there is no way to specify the type of elements that an IDREF of IDREFS attribute refers to.