Assignment 6 - Solution

Problem 1

Suppose a transaction sets an intention-write lock on a file and later sets a write lock on a record of the file.
Is it safe for the transaction to release the intention-write lock before it commits? Why?

No it's not safe. Suppose record x is contained in file F. Consider the following execution:

$iwl_1[F]\ wl_1[x]\ w_1[x]\ iwu_1[F]$ **$rl_2[F]\ r_2[x]$** $w_1[x]\ wu_1[x]\ c_1$ **$ru_2[F]\ c_2$**

Transaction T_1 writes x twice, once before **$r_2[x]$** and once afterwards, so the result isn't serializable.

Problem 2

The multi-granularity locking protocol requires that if a transaction has a *w* or *iw* lock on a data item *x*, then it must have an *iw* lock on *x*'s parent.

2.A. Is it correct for a transaction to hold an *r* lock on *x*'s parent instead? Either explain why it's correct or give an example where it fails.

- No, it is incorrect.
- This would allow:
  - T_1 to read lock file F
    - (giving it permission to read every record in F),
  - T_2 to read lock F and write lock a record in F, such as x.
- Thus T_1 would implicitly have a read lock on x that conflicts with T_2's write lock on x.

2.B. Redo question (2.A), replacing "*r* lock" by "*w* lock".

Is it correct for a transaction to hold an w lock on *x*'s parent instead?

- Yes, it is correct.
- Assuming the given protocol is correct using *iw* locks, then *w* locks must work too
  - *w* lock is strictly stronger than an *iw* lock.
- By "stronger," we mean that any lock type that conflicts with an *iw* lock also conflicts with a *w* lock.
- You might think it's incorrect because it needlessly prevents certain operations from running.
  - This is a performance problem, but not incorrect, in the sense of breaking a conflict or an ACID property.

2.C. Assuming the lock graph is a tree, suggest a case where it would be useful to set such a *w* lock as in (2.B) (whether or not it's correct).

In cases where the data manager would escalate fine-grained *w* locks to coarse-grained locks, then using the coarse-grained *w* as parent will perform slightly better by avoiding the escalation cost and the cost of needlessly setting the fine-grained locks before escalating.

Problem 3

Consider the following database table, which supports a multiversion concurrency control.

TID	Prev TID	Account#	Balance
1	Null	10	100
3	1	10	200
1	Null	11	300
4	1	11	400
5	4	11	350
6	Null	12	500

Suppose the commit list contains {1,2,3,4,6} and there are no active transactions.

3.A. What is the state of the table after running the following transaction?:

TID=8: Increment the balance of account 10 by 100;

Delete account 12;

Insert account 13 with balance 700.

The database state after transaction 8 commits

TID	Prev TID	Account#	Balance
1	Null	10	100
3	1	10	200
1	Null	11	300
4	1	11	400
5	4	11	350
6	Null	12	500

TID	Prev TID	Account#	Balance
1	Null	10	100
3	1	10	200
8	**3**	**10**	**300**
1	Null	11	300
4	1	11	400
5	4	11	350
6	Null	12	500
8	**6**	**12**	**deleted**
8	**Null**	**13**	**700**

3.B. Suppose a read-only query with TID=7 reads all the accounts. It starts executing before executing transaction 8 starts executing and finishes after transaction 8 commits (same transaction 8 as part (a)). Which versions of which rows does it read?

3.B.

TID	Prev TID	Account#	Balance
1	Null	10	100
3	1	10	200
8	**3**	**10**	**300**
1	Null	11	300
4	1	11	400
5	4	11	350
6	Null	12	500
8	**6**	**12**	**deleted**
8	**Null**	**13**	**700**

- When it started executing, transaction 7 read the following commit list: {1,2,3,4,6}.
- It read the following:
  - version TID 3 of Account 10,
  - version TID 4 of Account 11 (because 5 didn't commit)
  - version TID 6 of account 12.
- It does not see any of transaction 8's updates because transaction 8 was not on the commit list when it started.

3.C. After transactions 7 and 8 have finished and no other transactions are active, suppose we garbage collect all of the versions that aren't needed. Assuming transaction ids increase monotonically with respect time, what does the table look like after the garbage collection step?

The garbage collector keeps the last committed update of each account:

TID	Prev TID	Account#	Balance
1	Null	10	100
3	1	10	200
8	**3**	**10**	**300**
1	Null	11	300
4	1	11	400
5	4	11	350
6	Null	12	500
8	**6**	**12**	**deleted**
8	**Null**	**13**	**700**

TID	Prev TID	Account#	Balance
8	Null	10	300
4	Null	11	400
8	Null	13	700

Problem 4

Suppose file *F* contains a sequence of fixed-length records, and *F*'s descriptor includes a *count* of the number of records in *F*, which is used to find the end of *F*. Consider the following two transactions:

- T_1:
  - Scan *F*, returning all the records in *F*
  - Read(*x*)
- T_2:
  - Insert a record into *F*
  - Write(*x*)

Data item *x* is not in F. Both transactions are two-phase locked (locking records in *F* and *x*), but neither transaction locks *count*.

4.A. Given an example of a non-serializable execution of T_1 and T_2. Explain why it's non- serializable.

1. T_1: Scan *F*, returning all the records in *F*
2. T_2: Insert a record into *F*
3. T_2: Write(*x*)
4. $Commit_2$
5. T_1: Read(*x*)

- The first two operations imply and T_1 precedes T_2, but since they don't lock count, the first operation doesn't cause the second one to be delayed.
- The third and fifth operations on *x* conflict, which imply that T_2 precedes T_1.
- So the execution isn't SR.

4.B. Explain why this is an example of the phantom problem.

- T_1:
  - Scan *F*, returning all the records in *F*
  - Read(*x*)
- T_2:
  - Insert a record into *F*
  - Write(*x*)

T_2's insertion into *F* is a phantom record. T_1's scan doesn't see the record, but T_1's Read(*x*) indirectly sees the result in data item *x* (assuming the value of *x* is a function of the records in *F*).