Homework # 9

21.1. What is the two-phase locking protocol? How does it guarantee serializability?

A transaction is said to follow the two-phase locking protocol if all locking operations (read_lock, write_lock) precede the first unlock operation in the transaction. Such a transaction can be divided into two phases: an expanding or growing (first) phase, during which new locks on items can be acquired but none can be released; and a shrinking (second) phase, during which existing locks can be released but no new locks can be acquired. If lock conversion is allowed, then upgrading of locks (from read-locked to write-locked) must be done during the expanding phase and downgrading of locks (from write-locked to read-locked) must be done in the shrinking phase.

Serializability occurs when multiple read and write operation is perform simultaneously on same database. The locking of database, by enforcing two-phase locking rules enforces serializability. Hence, multiple read and write operation on same database is prevented. So, if every transaction in a schedule follows the two-phase locking protocol, the schedule is guaranteed to be serializable.

21.2. What are some variations of the two-phase locking protocol? Why is strict or rigorous two-phase locking often preferred?

There are a number of variations of two-phase locking (2PL).

1: **Conservative 2PL** (or **static 2PL**): It requires a transaction to lock all the items it accesses before the transaction begins execution, by **predeclaring** it read-set and write-set. The **read-set** of a transaction is the set of all items that the transaction reads, and the **write-set** is the set of all items that it writes. If any of the predeclared items needed cannot be locked, the transaction does not lock any item; instead, it waits until all the items are available for locking. Conservative 2PL is a deadlock-free protocol.

2: **Strict 2PL**: It which guarantees strict schedules. In this variation, a transaction T does not release any of its exclusive (write) locks until after it commits or aborts. Hence, no other transaction can read or write an item that is written by T unless T has committed, leading to a strict schedule for recoverability. Strict 2PL is not deadlock-free.

3: **Rigorous 2PL**: A more restrictive variation of strict 2PL is **rigorous 2PL**, which also guarantees strict schedules. In this variation, a transaction T does not release any of its locks (exclusive or shared) until after it commits or aborts, and so it is easier to implement than strict 2PL.

Conservative 2PL is difficult to use in practice because of the need to predeclare the read-set and write- set, which is not possible in some situations. Hence, Strict 2PL or Rigorous is mostly preferred.

21.3. Discuss the problems of deadlock and starvation, and the different approaches to dealing with these problems.

Deadlock occurs when each transaction T in a set of two or more transactions is waiting for some item that is locked by some other transaction T in the set. Hence, each transaction in the set is in a waiting queue, waiting for one of the other trans- actions in the set to release the lock on an item. But because the other transaction is also waiting, it will never release the lock.

Starvation occurs when a transaction cannot proceed for an indefinite period of time while other transactions in the system continue normally. This may occur if the waiting scheme for locked items is unfair in that it gives priority to some transactions over others.

Deadlock prevention protocol:
Conservative two-phase locking: It requires that every transaction lock all the items it needs in advance (which is generally not a practical assumption) if any of the items cannot be obtained, none of the items are locked. Rather, the transaction waits and then tries again to lock all the items it needs. This solution further limits concurrency.

Ordering all the items: A second protocol, which also limits concurrency, involves ordering all the items in the database and making sure that a transaction that needs several items will lock them according to that order. This requires that the programmer (or the system) is aware of the chosen order of the items, which is also not practical in the database context.

Deadlock detection: Here the system checks if a state of deadlock actually exists. This solution is attractive if we know there will be little interference among the transactions—that is, if different transactions will rarely access the same items at the same time. This can happen if the transactions are short and each transaction locks only a few items, or if the transaction load is light.

Timeouts: Another simple scheme to deal with deadlock is the use of **timeouts**. This method is practical because of its low overhead and simplicity. In this method, if a transaction waits for a period longer than a system-defined timeout period, the system assumes that the transaction may be deadlocked and aborts it regardless of whether a deadlock actually exists.

Solution for starvation is to have a fair waiting scheme, such as using a **first-come-first-served** queue; transactions are enabled to lock an item in the order in which they originally requested the lock. Another scheme allows some transactions to have priority over others but increases the priority of a transaction the longer it waits, until it eventually gets the highest priority and proceeds. wait-die and wound-wait schemes avoid starvation, because they restart a transaction that has been aborted with its same original timestamp, so the possibility that the same transaction is aborted repeatedly is slim.

21.4. Compare binary locks to exclusive/shared locks. Why is the latter type of locks preferable?

Binary Locks:
It can have two **states** or **values**: locked and unlocked (or 1 and 0, for simplicity). A distinct lock is associated with each database item X. If the value of the lock on X is 1, item X cannot be accessed by a database operation that requests the item. If the value of the lock on X is 0, the item can be accessed when requested, and the lock value is changed to 1. We refer to the current value (or state) of the lock associated with item X as **lock(X)**.

Two operations, lock_item and unlock_item, are used with binary locking. A transaction requests access to an item X by first issuing a **lock_item(X)** operation. If LOCK(X) = 1, the transaction is forced to wait. If LOCK(X) = 0, it is set to 1 (the transaction **locks** the item) and the transaction is allowed to access item X. When the transaction is through using the item, it issues an **unlock_item(**X**)** operation, which sets LOCK(X) back to 0 (**unlocks** the item) so that X may be accessed by other transactions. Hence, a binary lock enforces **mutual exclusion** on the data item.

Exclusive/shared locks:
There are three locking operations in shared locks: read_lock(X), write_lock(X), and unlock(X). A lock associated with an item X, LOCK(X), now has three possible states: read-locked, write-locked, or unlocked. A **read-locked item** is also called **share-locked** because other transactions are allowed to read the item, whereas a **write-locked item** is called **exclusive-locked** because a single transaction exclusively holds the lock on the item. If LOCK(X) = write-locked, the value of locking_transaction(s) is a single transaction that holds the exclusive (write) lock on X. If LOCK(X)=read-locked, the value of locking transaction(s) is a list of one or more transactions that hold the shared (read) lock on X. Following are the rules followed by this protocol.

- A transaction T must issue the operation read_lock(X) or write_lock(X) before any read_item(X) operation is performed in T.
- A transaction T must issue the operation write_lock(X) before any write_item(X) operation is performed in T.
- A transaction T must issue the operation unlock(X) after all read_item(X) and write_item(X) operations are completed in T.
- A transaction T will not issue a read_lock(X) operation if it already holds a read (shared) lock or a write (exclusive) lock on item X.
- A transaction T will not issue a write_lock(X) operation if it already holds a read (shared) lock or write (exclusive) lock on item X.
- A transaction T will not issue an unlock(X) operation unless it already holds a read (shared) lock or a write (exclusive) lock on item X.

Binary locks, which are simple but are also too restrictive for database concurrency control purposes and so are not used much. on the other hand, shared/exclusive locks or read/write locks which provide more general locking capabilities and are used in database locking schemes.

21.6. Describe the cautious waiting, no waiting, and timeout protocols for deadlock prevention.

Suppose that transaction T_i tries to lock an item X but is not able to because X is locked by some other transaction T_j with a conflicting lock. The rules followed by these schemes are:

Cautious waiting: If T_j is not blocked (not waiting for some other locked item), then T_i is blocked and allowed to wait; otherwise abort T_i.

No waiting algorithm: If a transaction is unable to obtain a lock, it is immediately aborted and then restarted after a certain time delay without checking whether a deadlock will actually occur or not. In this case, no transaction ever waits, so no deadlock will occur. However, this scheme can cause transactions to abort and restart needlessly

Timeouts: This method is practical because of its low overhead and simplicity. In this method, if a transaction waits for a period longer than a system-defined timeout period, the system assumes that the transaction may be deadlocked and aborts it regardless of whether a deadlock actually exists.

21.14. What is multiple granularity locking? Under what circumstances is it used?

Multiple granularity level 2PL protocol, with shared/exclusive locking modes, is a protocol where a lock can be requested at any level.

The **multiple granularity locking (MGL)** protocol consists of the following rules:

- The lock compatibility must be adhered to.
- The root of the tree must be locked first, in any mode.
- A node N can be locked by a transaction T in S or IS mode only if the parent node N is already locked by transaction T in either IS or IX mode.
- A node N can be locked by a transaction T in X, IX, or SIX mode only if the parent of node N is already locked by transaction T in either IX or SIX mode.
- A transaction T can lock a node only if it has not unlocked any node (to enforce the 2PL protocol).
- A transaction T can unlock a node, N, only if none of the children of node N are currently locked by T.

The multiple granularity level protocol is especially suited when processing a mix of transactions that include,

- Short transactions that access only a few items (records or fields).
- Long transactions that access entire files. In this environment, less transaction blocking and less locking overhead are incurred by such a protocol when compared to a single-level granularity locking approach.

Question 1: The wait-die and the wound-wait deadlock prevention methods each have two possible scenarios which can occur. Provide a clear and descriptive example of all 4 scenarios. Also describe how locking data in a specific order prevents deadlocks with an example.

Suppose that transaction T_i tries to lock an item X but is not able to because X is locked by some other transaction T_j with a conflicting lock. The rules followed by these schemes are:

Wait-die: If TS(Ti) < TS(Tj), then (Ti older than Tj) Ti is allowed to wait; otherwise (Ti younger than Tj) abort Ti (Ti dies) and restart it later with the same timestamp.

Example for Wait-die:
Assume 3 transactions Ta, Tb, Tc and timestamps for Ta = 2, Tb = 4 and Tc = 10. If Ta requests a data item hold by Tb, then Ta will have to wait. If Tc requests a data item that has been hold by Tb, then this will cause the Tc to be rolled back**.**

Wound wait: If TS(Ti) < TS(Tj), then (Ti older than Tj) abort Tj (Ti wounds Tj) and restart it later with the same timestamp; otherwise (Ti younger than Tj) Ti is allowed to wait.

Example for Wound-wait:
Assume 3 transactions Ta, Tb, Tc and timestamps for Ta = 2, Tb = 4 and Tc = 10. If Ta requests a data item hold by Tb, then data will be released from Tb. This will cause Tc to rolled back. On the other hand, if Tc requests a data item that has been hold by Tb, then Tc will have to wait**.**

Question 2: Suppose you are the DBA for a heavily used database where many transactions are being concurrently applied throughout the day. At 10AM you noticed that no transactions are being committed. Being the local expert, you quickly deduce the problem to a culprit transaction which is hoarding locks. It seems that the intern is inappropriately executing a huge transaction by locking all the rows within a critical Human Resources table. Describe how to use the techniques recently covered such as checkpointing, REDO, UNDO etc. to resolve the current dilemma. Make sure to explain how your DBMS is affected by being configured as an immediate update or deferred update.

Checkpoint: A record is written into the log periodically at that point when the system writes out to the database on disk all DBMS buffers that have been modified. As a consequence of this, all transactions that have their [commit, T] entries in the log before a [checkpoint] entry do not need to have their WRITE operations redone in case of a system crash, since all their updates will be recorded in the database on disk during checkpointing. As part of checkpointing, the list of transaction ids for active transactions at the time of the checkpoint is included in the checkpoint record, so that these transactions can be easily identified during recovery.

Deferred update: In this technique do not physically update the database on disk until after a transaction commits; then the updates are recorded in the database. REDO only possible

Immediate update: In this technique, the database may be updated by some operations of a transaction before the transaction reaches its commit point. REDO/UNDO possible.

Scenario 1: (If system was configured on Deferred Update)

- If all the transactions are committed before system failure then there will be a commit command and also checkpoint in the log file, so in case of a system failure a REDO operation will be performed from this checkpoint on all the transactions and no ROLLBACK operation will be performed.

- If all the transactions are not committed before system failure then there will not be a commit command in the log file, so in case of system failure NO-REDO operation will be performed but instead a ROLLBACK operation will be performed that is in the log file.

Log entry	Log written to disk	Changes written to database buffer	Changes written on disk
start_transaction(T)	No	N/A	N/A
read_item(T, x)	No	N/A	N/A
write_item(T, x)	No	No	No
commit(T)	Yes	Yes	*Yes
checkpoint	Yes	Undefined	Yes(of committed Ts)

*Yes: writing back to disk may occur not immediately.

Scenario 2: (If system was configured on Immediate Update)

- If all the transactions are committed before system failure then there will be a commit command and also a checkpoint in the log file, so in case of a system crash, UNDO command will be performed from that checkpoint onwards.

- If a transaction fails after recording some changes in the database on disk but before reaching its commit point, the effect of its operations on the database must be undone that means UNDO operation must be performed, that is, the transaction must be rolled back.

Log entry	Log written to disk	Changes written to database buffer	Changes written on disk
start_transaction(T)	No	N/A	N/A
read_item(T, x)	No	N/A	N/A
write_item(T, x)	Yes	Yes	*Yes
commit(T)	Yes	Undefined	Undefined
Checkpoint	Yes	Undefined	Yes(of committed Ts)

*Yes: writing back to disk may not occur immediately

Question 3: From our textbook 22.4, describe the shadow paging recovery technique. Under what circumstances does it not require a log?

This recovery scheme does not require the use of a log in a single-user environment. Shadow paging considers the database to be made up of a number of fixed- size disk pages (or disk blocks) say, n for recovery purposes.

- A directory with n entries is constructed. The directory is kept in main memory if it is not too large, and all references reads or writes to database pages on disk go through it.

- When a transaction begins executing, the current directory whose entries point to the most recent or current database pages on disk is copied into a shadow directory.

- The shadow directory is then saved on disk while the current directory is used by the transaction.

- During transaction execution, the shadow directory is never modified.

- When a write item operation is performed, a new copy of the modified database page is created, but the old copy of that page is not overwritten. Instead, the new page is written elsewhere on some previously unused disk block.

- The current directory entry is modified to point to the new disk block, whereas the shadow directory is not modified and continues to point to the old unmodified disk block.

- For pages updated by the transaction, two versions are kept. The old version is referenced by the shadow directory and the new version by the current directory.

Failure Recovery:

- To recover from a failure during transaction execution, it is sufficient to free the modified database pages and to discard the current directory.

- The state of the data base before transaction execution is available through the shadow directory, and that state is recovered by reinstating the shadow directory.

- The database thus is returned to its state prior to the transaction that was executing when the crash occurred, and any modified pages are discarded.

- Committing a transaction corresponds to discarding the previous shadow directory. Since recovery involves neither undoing nor redoing data items, this technique can be categorized as a NO-UNDO/NO-REDO technique for recovery.

Disadvantage of Shadow Paging:

- One disadvantage of shadow paging is that the updated database pages change location on disk. This makes it difficult to keep related database pages close together on disk without complex storage management strategies.

- If the directory is large, the overhead of writing shadow directories to disk as transactions commit is significant.

- Another complication is how to handle garbage collection when a transaction commit.

- The old pages referenced by the shadow directory that have been updated must be released and added to a list of free pages for future use. These pages are no longer needed after the transaction commits.

- Another issue is that the operation to migrate between current and shadow directories must be implemented as an atomic operation.

Question 4: In your own words, using atomic operations, describe how two concurrently executed Bank ATM transactions can ensure data integrity.

Atomicity. A transaction is an atomic unit of processing; it should either be performed in its entirety or not performed at all.

Scenerio1:
Where a simultaneous Bank ATM transaction is performed for one transaction it should not affect another transaction. Each transaction must be done individually. If the transaction failed, it should not affect another transaction and should not change the state of the data. The method to avoid conflicts where two transactions are performed simultaneously, it uses a lock system. With the concept of locking, find and lock the resource before using it.

Scenerio2:
When performing an ATM withdrawal service, the bank will check our account to make sure there are required funds available. If no wallet was available, they could have stopped the transaction without touching any data. Once the fund is available in the account continue the purchase process. Then pay the amount from the account requested. Then generate a receipt and submit the amount and record to the user. Work will only be successful once all these steps have been completed.

Atomic operations are performed without interference from other functions. Here all processes are different, meaning that the process in one trade is not reflected in another function. Which provides data security. Once the work is completed, its results are permanently stored in the database. Ensures that data is not lost. Therefore, ensure data integrity.

Question 5: Describe how Serialization is analogous to credit card companies providing credit to customers.

A serial schedule is always a serializable schedule because any transaction only starts its execution when another transaction has already completed its execution. However, a non-serial schedule of transactions needs to be checked for Serializability.

The credit card company provides credit card to lots of customer and doesn't check their background before providing the credit card. This may cause them a problem on a short-term basis but on a long-term basis it is beneficial. As there are lots of customer there could be a problem with serialization when they do transaction and it is needed to be fixed by credit customer to avoid loses. Now the things is, it is very tedious and expensive job to check the sterilization for every customer and even though this may help them for short term, but for longer term it will cause them more harm than good. Checking the serialization for every customer will put a Big load on the entire system that checks for serialization. Hence, the credit card company assumes than in 1000 customer there will 50 customer who will have this serializations issues and fixed this issue only when there is problem of serializations. This makes sure than customer is satisfied and there is no problem with transaction or any serialization issues. So, credit card company provides credit card to number of people and when these people do a transaction concurrently then at the same time company maintain and correct the transaction issues only if they occur, this is how the serialization is analogous to credit card company.

Question 6: Given the following figure below whose transactions are T1 : R(A), R(D), W(D), T2 : R(B), W(B), R(D), W(D), and T3 : R(C), W(B), R(A), W(A), which transaction(s) is affected by a cascade rollback and why?

Log	A	B	C	D
	30	15	40	20
start_transaction(T_3)				
read_item(T_3, C)				
*write_item(T_3, B, 15, 12)		12		
start_transaction(T_2)				
read_item(T_2, B)				
**write_item(T_2, B, 12, 18)		18		
start_transaction(T_1)				
read_item(T_1, A)				
read_item(T_1, D)				
write_item(T_1, D, 20, 25)				25
read_item(T_2, D)				
**write_item(T_2, D, 25, 26)				26
read_item(T_3, A)				
System crash				

- The transaction T2 reads the value B which was written by T3
- T3 is rolled back because it did not reach the commit point.
- Because T3 is rolled back, T2 must also be rolled back.
- The write operations of T2, marked by ** in the log, are the ones that are undone.
- Only write operations are undone during transaction rollback.
- Write_item operations need to be undone during transaction rollback.

- Read_item operations are recorded in the log only to determine whether cascading rollback of additional transactions is necessary.